**Q. What is thread in selenium?**

In Selenium, a thread refers to a separate path of execution within a program. In the context of Selenium, threading is often used to run multiple browser instances in parallel, which can be useful for parallel testing or to speed up test execution.

Key Points about Threads in Selenium:

1. Parallel Execution:
   * Threads allow you to run multiple tests simultaneously. For example, if you have 10 test cases, you can execute them in parallel across multiple browser instances, reducing the overall test execution time.
2. Concurrency in Tests:
   * When running tests in parallel using threads, each thread operates independently. This means that each test case can interact with its own browser instance without interference from other threads.
3. Thread Safety:
   * When using threads in Selenium, it’s crucial to ensure that the WebDriver instance is thread-safe. This means each thread should have its own instance of WebDriver to avoid conflicts or unexpected behaviors.
4. Frameworks Supporting Multi-Threading:
   * Testing frameworks like TestNG and JUnit support parallel execution and allow you to configure how many threads to use for running tests in parallel. In TestNG, for example, you can define the number of threads in the test suite XML file.

**How would you decide which test cases to automate for a streaming website?**

Answer: Not all test cases are suitable for automation. Here’s how you can decide:

* **Repeatability:** Automate tests that need to be run frequently, such as regression tests or smoke tests.
* **Business Criticality:** Focus on automating test cases that cover critical user journeys, such as logging in, searching for content, and playing videos.
* **Complexity:** Automate scenarios that are complex or tedious to test manually, such as testing across multiple devices or handling different user roles.
* **Stability:** Prioritize test cases that are stable and less likely to change frequently.
* **Data-Driven Scenarios:** Automate scenarios that require running the same set of actions with multiple sets of data, such as testing different user subscriptions or content libraries.

**Explain how you would automate the testing of a streaming service across different devices and browsers.**

Answer: Testing across different devices and browsers ensures that the streaming service provides a consistent user experience regardless of how users access it.

* **Cross-Browser Testing Tools:** Use tools like Selenium Grid, BrowserStack, or Sauce Labs to run automated tests on different browsers and devices.
* **Responsive Design Testing:** Automate tests that check the UI layout and functionality on various screen sizes (mobile, tablet, desktop).
* **Device-Specific Scenarios:** Automate tests for device-specific features, like touch gestures on mobile devices or video playback on smart TVs.
* **Parallel Execution:** Set up the automation framework to execute tests in parallel across multiple environments, reducing the overall test execution time.

**#Annoation in TestNG**

Annotations in TestNG are used to control the flow of test execution and to configure and manage test methods. They provide a way to define different methods with specific purposes, such as setup, teardown, or actual test cases, and help in organizing and managing test cases effectively.

1. @Test
2. @BeforeSuite
3. @AfterSuit
4. @Listeners
5. @BeforeMethods and many more

**Q. How can you skip a test method in TestNG?**

You can skip a test method by setting the enabled attribute of the @Test annotation to false.

**Q. What is the purpose of the @DataProvider annotation?**The @DataProvider annotation in TestNG is used to create a method that provides data to a test method. The primary purpose of @DataProvider is to allow the execution of the same test method multiple times with different sets of data.

**Key Features and Purpose of @DataProvider:**

1. **Data-Driven Testing:**
   * @DataProvider allows you to run a test method multiple times with different data. This is ideal for scenarios where the test logic remains the same, but you want to validate it against various inputs.
2. **Parameterization:**
   * It enables parameterization of test methods, making tests more flexible and reducing redundancy. Instead of writing multiple test methods for different input values, you can write a single test method and pass different data sets to it.
3. **Custom Data Sets:**
   * You can define custom data sets in the @DataProvider method. This can include hard-coded values, values retrieved from external sources (like databases or files), or dynamically generated values.
4. **Multi-dimensional Data:**
   * @DataProvider supports multi-dimensional arrays, allowing you to pass complex data structures to your test methods.

**Benefits:**

* **Reduces Code Duplication:**
  + Instead of writing separate test methods for each data set, you can consolidate them into a single method with a @DataProvider.
* **Enhances Maintainability:**
  + Centralizing data in one place makes it easier to manage and modify test inputs without altering the test logic.
* **Improves Test Coverage:**
  + By easily running tests with multiple data sets, you can improve the coverage and reliability of your tests.

**Q. Explain the concept of TestNG Suite ?**

A TestNG suite is a central concept in TestNG that allows you to define and execute a group of test cases together. A suite in TestNG is essentially a collection of tests, which can include multiple test classes, test methods, groups, and configurations. The suite is defined in an XML file (commonly known as the TestNG XML file), which provides a way to manage and organize test execution in a flexible and hierarchical manner.

**Key Concepts of TestNG Suites:**

1. **Test Suite Structure:**
   * A TestNG suite is defined using an XML file (testng.xml). The XML file outlines the structure of the suite, including which test classes to run, how to configure them, and in what order they should be executed.
2. **Grouping and Hierarchy:**
   * TestNG suites allow you to organize tests into different groups and sub-groups. You can run specific groups of tests or exclude certain groups, providing fine-grained control over test execution.
3. **Parallel Execution:**
   * TestNG suites support parallel execution of tests. You can configure the suite to run tests in parallel at different levels (suites, tests, methods) to reduce the overall execution time.
4. **Dependency Management:**
   * TestNG suites enable you to define dependencies between different tests. You can specify that certain tests should only run if others have passed, providing a way to manage complex test scenarios.
5. **Custom Configuration:**
   * The XML file allows for custom configuration, such as setting up listeners, defining parameters, and including/excluding specific test methods or classes.
6. **Multi-Module Testing:**
   * In larger projects, where tests might be spread across different modules, TestNG suites allow you to bring together tests from various modules into a single suite for comprehensive testing.

**Q. How can you pass parameters to a TestNG test method?**

1. @Parameters Annotation: Use this when you need to pass parameters from the TestNG XML file to your test methods.
2. @DataProvider Annotation: Use this when you want to run the same test with multiple sets of data. It’s ideal for data-driven testing.
3. @Optional:Provides a default value when a parameter is not specified in the TestNG XML file.

**Q. What is the purpose of the alwaysRun attribute in TestNG?**The alwaysRun attribute in TestNG is used to ensure that a particular test method, configuration method, or dependent method always gets executed, regardless of the outcome of other test methods or their dependencies. This is particularly useful in scenarios where certain cleanup operations, setup tasks, or dependent methods must be executed even if some tests fail or are skipped.

**Key Use Cases for alwaysRun:**

1. Configuration Methods (@BeforeMethod, @AfterMethod, etc.):
   * When used with configuration methods (like @BeforeMethod, @AfterMethod, @BeforeClass, @AfterClass, etc.), setting alwaysRun=true ensures that these methods will execute even if a test method fails or is skipped.
2. Dependent Methods (dependsOnMethods):
   * If you have a test method that depends on the execution of another test method (using dependsOnMethods), setting alwaysRun=true on the dependent method ensures it runs even if the methods it depends on fail or are skipped.
3. Cleanup Tasks:
   * For methods that perform essential cleanup tasks (like closing a browser, releasing resources, or resetting a database), alwaysRun=true ensures that these tasks are executed to maintain test environment integrity, regardless of the outcome of the test cases.

**Q. What is the purpose of the timeOut attribute in TestNG?**

The timeOut attribute in TestNG is an important tool for managing and controlling the execution time of your tests. It helps in:

* Preventing tests from running indefinitely due to bugs or unexpected conditions.
* Controlling the overall test suite execution time, ensuring tests complete within reasonable limits.
* Enforcing consistency in test execution times across different environments, particularly in CI pipelines.

**Q. Explain the purpose of the preserveOrder attribute in TestNG.**

The preserveOrder attribute, though primarily configured through TestNG XML or by using method priorities, serves the purpose of ensuring that test methods are executed in a specific sequence. This is useful for:

* Maintaining dependencies between test methods.
* Debugging and troubleshooting by ensuring predictable test execution order.
* Managing complex test scenarios where the order of execution is critical.

**Q1: What is Selenium?**

Selenium is an open-source testing tool used for automating web browsers. It provides a suite of tools for testing web applications across various platforms, browsers, and programming languages.

**Q2: What are the components of Selenium?**

Selenium consists of four main components: Selenium IDE, Selenium WebDriver, Selenium Grid, and Selenium RC.

**Q3: What is Selenium WebDriver?**

Selenium WebDriver is a tool used for automating web applications. It provides a programming interface to create and execute test cases using various programming languages.

**Q4: What are the advantages of using Selenium WebDriver?**

The advantages of using Selenium WebDriver include cross-browser compatibility, support for multiple programming languages, easy integration with other testing frameworks, and the ability to automate complex web application workflows.

**Q5: What programming languages are supported by Selenium WebDriver?**

Selenium WebDriver supports multiple programming languages, including Java, Python, C#, Ruby, and JavaScript.

**Q6: What are the basic steps to create a Selenium WebDriver test case?**

The basic steps to create a Selenium WebDriver test case include:

1. Launching a web browser
2. Navigating to a web page
3. Interacting with web elements
4. Performing actions on web elements
5. Verifying expected results

**Q7: What is a WebElement?**

A WebElement is an object in a web page that can be interacted with using Selenium WebDriver. Examples of web elements include buttons, links, text fields, and dropdown menus.

**Q8: How do you locate a WebElement using Selenium WebDriver?**

You can locate a WebElement using various methods, such as ID, name, class name, tag name, link text, and partial link text.

**Q9: What is the difference between findElement() and findElements() in Selenium WebDriver?**

findElement() returns a single WebElement that matches the specified locator, while findElements() returns a list of all WebElements that match the specified locator.

**Q10: What is the difference between getText() and getAttribute() in Selenium WebDriver?**

getText() returns the visible text of a WebElement, while getAttribute() returns the value of the specified attribute of a WebElement.

**Q11: Why use Java with Selenium WebDriver?**

Java is a popular programming language that is widely used in the industry. It provides a robust set of libraries and frameworks that can be used to build powerful test automation frameworks with Selenium WebDriver.

**Q12: What are the prerequisites for using Java with Selenium WebDriver?**

The prerequisites for using Java with Selenium WebDriver include:

1. Java Development Kit (JDK) installed on your machine
2. A Java IDE, such as Eclipse or IntelliJ IDEA
3. The Selenium WebDriver Java bindings
4. The WebDriver executable for the browser you want to test

**Q13: How do you set up a Java project for Selenium WebDriver?**

You can set up a Java project for Selenium WebDriver by following these steps:

1. Create a new Java project in your IDE
2. Add the Selenium WebDriver Java bindings to your project’s build path
3. Download the WebDriver executable for the browser you want to test
4. Create a new WebDriver instance and start automating your tests

**Q14: How do you create a WebDriver instance in Java?**

You can create a WebDriver instance in Java using the following code:

                  WebDriver driver = new ChromeDriver();

**Q15: What is Selenium WebDriver with Java?**

Selenium WebDriver with Java is an open-source tool that enables automated testing of web applications in Java programming language. It provides a set of APIs for interacting with web browsers and performing actions on web elements.

**Q16: What is Test Automation in Selenium WebDriver with Java?**

Test automation in Selenium WebDriver with Java refers to automating the process of testing web applications using the Selenium WebDriver API and the Java programming language. It involves creating test scripts that simulate user interactions with the web application and verify the expected results.

**Q17: What are the benefits of Test Automation in Selenium WebDriver with Java?**

The benefits of test automation in Selenium WebDriver with Java include:

1. Improved testing efficiency and speed
2. Reduced testing costs and time-to-market
3. Increased test coverage and accuracy
4. Improved software quality and reliability

**Q19: What are the different components of Selenium WebDriver with Java?**

The different components of Selenium WebDriver with Java include:

WebDriver API: A set of APIs for interacting with web browsers

Selenium Grid: A tool for distributing tests across multiple machines and browsers

TestNG: A testing framework for test execution and reporting

Maven: A build automation tool for managing dependencies and building projects

**Q20: What are the different types of locators in Selenium WebDriver with Java?**

The different types of locators in Selenium WebDriver with Java include:

ID                Name                           Class Name              Tag Name

Link Text                    Partial Link Text             CSS Selector      XPath

**Q21: What is a WebElement in Selenium WebDriver with Java?**

A WebElement in Selenium WebDriver with Java is a representation of a web element on a web page, such as a text box, button, or link. It provides methods for interacting with the element, such as clicking, sending keys, or getting its text.

**Q22: How do you find a WebElement in Selenium WebDriver with Java?**

You can find a WebElement in Selenium WebDriver with Java by using one of the locator strategies provided by the WebDriver API, such as findElement(By.id(“elementId”)). This returns a WebElement instance that represents the located element on the web page.

**Q23: What is the difference between implicit and explicit waits in Selenium WebDriver with Java?**

Implicit waits in Selenium WebDriver with Java are set globally for the WebDriver instance and specify a maximum time to wait for an element to be found or an action to be performed before throwing a NoSuchElementException or a TimeoutException. Explicit waits, on the other hand, are applied to specific elements and actions and use the WebDriverWait class to wait for a specific condition to be true before proceeding with the execution of the test.

**Q24: What are the different types of waits in Selenium WebDriver with Java? Provide code examples.**

The different types of waits in Selenium WebDriver with Java include:

1. **Implicit waits**
2. **Explicit waits**
3. **Fluent waits Implicit wait**

**Example:**

WebDriver driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

**Explicit wait example:**

WebDriverWait wait = new WebDriverWait(driver, 10);

WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(“myElement”)));

**Q25: What is a fluent wait in Selenium WebDriver with Java? Provide a code example.**

A fluent wait in Selenium WebDriver with Java is a type of explicit wait that waits for a particular condition to be met with a specific frequency. It provides a flexible way to wait for a certain element to be present, clickable, or invisible.

**Q26: What is the Page Object Model (POM) in Selenium WebDriver with Java? Provide a code example.**

The Page Object Model (POM) is a design pattern used to create object-oriented classes that serve as an interface to a webpage or a part of it. This pattern helps in maintaining the test scripts by separating the test logic from the page structure, making it easier to manage and more reusable.

**Page Object Model (POM) example:**

public class LoginPage { private WebDriver driver;

public LoginPage(WebDriver driver) { this.driver = driver;

}

@FindBy(id = “username”)

private WebElement usernameField; @FindBy(id = “password”)

private WebElement passwordField; @FindBy(id = “loginButton”)

private WebElement loginButton;

public void setUsername(String username) { usernameField.sendKeys(username);

}

public void setPassword(String password) { passwordField.sendKeys(password);

}

public void clickLoginButton() { loginButton.click();

}

**Q27:  What is the advantage of using the Page Object Model (POM) in Selenium WebDriver with Java? Provide a code example.**

The advantages of using the Page Object Model (POM) in Selenium WebDriver with Java include:

Improved code maintainability and readability

Reduced code duplication and test script size

Increased reusability of code and web elements

Easy modification of the web pages and elements

**Q28: What is a TestNG listener in Selenium WebDriver with Java? Provide a code example.**

A TestNG listener in Selenium WebDriver with Java is a class that implements the TestNG listener interface and can be used to customize the behavior of the TestNG test execution. It provides a way to listen to different events that occur during the test execution, such as test start, test failure, test success, etc.

**Q29: What is a DataProvider in Selenium WebDriver with Java?**

A DataProvider in Selenium WebDriver with Java is a TestNG feature that allows you to run the same test method multiple times with different sets of data. It is useful when you want to test the same functionality with different input data.

**Q30: What is a WebDriverEventListener in Selenium WebDriver with Java?**

A WebDriverEventListener in Selenium WebDriver with Java is an interface that defines methods that are called by WebDriver when certain events occur, such as before clicking an element, after clicking an element, before navigating to a new page, after navigating to a new page, etc. It can be used to customize the behavior of WebDriver.

**Q31: What is the difference between getWindowHandle() and getWindowHandles() in Selenium WebDriver with Java?**

getWindowHandle() method in Selenium WebDriver with Java returns a string handle of the current window or tab, while getWindowHandles() method returns a set of string handles of all open windows or tabs. The getWindowHandle() method is useful when you want to perform an action on the current window or tab, while the getWindowHandles() method is useful when you want to switch to a different window or tab.

**Q32: What is an Action in Selenium WebDriver with Java?**

An Action in Selenium WebDriver with Java is a class that provides methods to perform complex user interactions, such as dragging and dropping an element, double-clicking an element, hovering over an element, etc. It is useful when you want to simulate user interactions with the web page.

**Q33: How do you handle dropdown/select elements using Selenium WebDriver?**

// Find the dropdown/select element

WebElement dropdown = driver.findElement(By.id(“dropdown-id”));

// Create a Select object

Select select = new Select(dropdown);

// Select by visible text

select.selectByVisibleText(“Option 1”);

// Select by value

select.selectByValue(“option-2-value”);

// Select by index

select.selectByIndex(2);

// Deselect all options

select.deselectAll();

// Perform further actions on the dropdown

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

**Q34: How do you perform mouse hover actions using Selenium WebDriver?**

// Find the element to hover over

WebElement element = driver.findElement(By.id(“element-id”));

// Create an Actions object

Actions actions = new Actions(driver);

// Perform mouse hover action actions.moveToElement(element).perform();

// Perform further actions after the mouse hover

// …

// Close the browser driver.quit();

}

**Q35: How do you capture screenshots using Selenium WebDriver?**

// Capture the screenshot TakesScreenshot screenshot = (TakesScreenshot) driver;

File srcFile = screenshot.getScreenshotAs(OutputType.FILE);

// Save the screenshot to a specific location

File destFile = new File(“path/to/save/screenshot.png”); FileUtils.copyFile(srcFile, destFile);

**Q36. How do you perform file uploads using Selenium WebDriver?**

// Find the file upload input element

WebElement fileInput = driver.findElement(By.id(“file-input-id”));

// Provide the file path to upload

String filePath = “path/to/file.txt”; fileInput.sendKeys(filePath);

**Q37. How do you handle frames/iframe elements using Selenium WebDriver?**

// Switch to a frame by index

driver.switchTo().frame(0);

// Switch to a frame by name or ID

driver.switchTo().frame(“frame-name”);

// Switch to a frame by WebElement

WebElement frameElement = driver.findElement(By.id(“frame-id”)); driver.switchTo().frame(frameElement);

// Switch back to the default content

driver.switchTo().defaultContent();

**Q38. How do you perform scrolling actions using Selenium WebDriver?**

// Scroll vertically by pixel

JavascriptExecutor js = (JavascriptExecutor) driver; js.executeScript(“window.scrollBy(0, 500)”);

// Scroll vertically to an element

WebElement element = driver.findElement(By.id(“element-id”)); js.executeScript(“arguments[0].scrollIntoView();”, element);

// Scroll horizontally by pixel

js.executeScript(“window.scrollBy(500, 0)”);

**Q39. How do you handle multiple windows/tabs using Selenium WebDriver?**

// Get the window handles

Set<String> windowHandles = driver.getWindowHandles();

// Switch to a specific window/tab

String mainWindowHandle = driver.getWindowHandle();

for (String windowHandle : windowHandles) {

if (!windowHandle.equals(mainWindowHandle)) { driver.switchTo().window(windowHandle); break;

**Q40. How do you perform keyboard actions (e.g., pressing Enter, typing special characters) using Selenium WebDriver?**

import org.openqa.selenium.Keys;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class KeyboardActions {

public static void main(String[] args) {

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Find an input field element

WebElement inputField = driver.findElement(By.id(“input-field-id”));

// Type text with keyboard actions inputField.sendKeys(“Text to type”);

// Press Enter key

inputField.sendKeys(Keys.ENTER);

// Type special characters with keyboard actions inputField.sendKeys(Keys.CONTROL, “a”); inputField.sendKeys(Keys.BACK\_SPACE);

// Perform further actions after keyboard actions

// …

// Close the browser driver.quit();

}

}

**Q41. How do you handle JavaScript alerts, confirmations, and prompts using Selenium WebDriver?**

import org.openqa.selenium.Alert;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class JavaScriptAlerts {

public static void main(String[] args) {

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Navigate to a webpage with a JavaScript alert driver.get(“https://example.com”)

// Click a button that triggers a JavaScript alert

driver.findElement(By.id(“alert-button”)).click();

// Switch to the alert

Alert alert = driver.switchTo().alert();

// Get the text of the alert

String alertText = alert.getText();

// Accept the alert

alert.accept();

// Dismiss the alert

alert.dismiss();

// Perform further actions after handling the alert

// Close the browser driver.quit();

}

}

**Q42. How do you handle synchronization/wait conditions in Selenium WebDriver?**

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

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

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

public class Synchronization {

public static void main(String[] args) {

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Set the maximum wait time in seconds int waitTime = 10;

// Navigate to a webpage driver.get(“https:example.com”);

// Wait for an element to be visible

WebDriverWait wait = new WebDriverWait(driver, waitTime);

WebElement element =

wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(“element-id”)));

// Perform further actions after synchronization

// …

// Close the browser

driver.quit();

**Q43. How do you handle checkboxes and radio buttons using Selenium WebDriver?**

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Find a checkbox element

WebElement checkbox = driver.findElement(By.id(“checkbox-id”));

// Check the checkbox if it is not selected if (!checkbox.isSelected()) {

checkbox.click();

// Find a radio button element

WebElement radioButton = driver.findElement(By.id(“radio-button-id”));

// Select the radio button if it is not selected if (!radioButton.isSelected()) {

radioButton.click();

**Q44. How do you handle pop-up windows and child windows using Selenium WebDriver?**

public class WindowHandling {

public static void main(String[] args) {

System.setProperty(“webdriver.chrome.driver”, “path/to/chromedriver”); WebDriver driver = new ChromeDriver(); driver.get(“https://example.com”);

// Click a link/button that opens a new window/pop-up

driver.findElement(By.id(“new-window-button”)).click()

// Get the window handles

Set<String> windowHandles = driver.getWindowHandles();

// Switch to the new window/pop-up

for (String windowHandle : windowHandles) {

driver.switchTo().window(windowHandle);

if (driver.getTitle().equals(“New Window”)) {

break;

}

}

// Perform actions on the new window/pop-up

// Close the new window/pop-up driver.close();

// Switch back to the main window

String mainWindowHandle = driver.getWindowHandle(); driver.switchTo().window(mainWindowHandle);

// Perform further actions on the main window

// Close the browser

driver.quit();

}

}

**Q45. How do you handle cookies using Selenium WebDriver?**

import org.openqa.selenium.Cookie;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class CookieHandling {

public static void main(String[] args) {

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Navigate to a webpage driver.get(“https://www.example.com”);

// Add a cookie

Cookie cookie = new Cookie(“cookie-name”, “cookie-value”); driver.manage().addCookie(cookie);

// Get all cookies

Set<Cookie> cookies = driver.manage().getCookies();

// Delete a cookie

driver.manage().deleteCookie(cookie);

// Delete all cookies

driver.manage().deleteAllCookies();

// Perform further actions after cookie handling

// …

// Close the browser

driver.quit();

}

}

**Q48. How do you handle browser notifications using Selenium WebDriver?**

// Create ChromeOptions instance

ChromeOptions options = new ChromeOptions();

// Disable browser notifications

options.addArguments(“–disable-notifications”);

// Launch Chrome browser with options

WebDriver driver = new ChromeDriver(options);

// Perform actions after disabling notifications

// …

// Close the browser

driver.quit();

}

**Q52. How do you handle drag-and-drop actions using Selenium WebDriver?**

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Navigate to a webpage

driver.get(“https:example.com”);

// Find the source and target elements for drag and drop

WebElement sourceElement = driver.findElement(By.id(“source-element”)); WebElement targetElement = driver.findElement(By.id(“target-element”));

// Perform drag and drop action

Actions actions = new Actions(driver); actions.dragAndDrop(sourceElement, targetElement).build().perform();

// Perform further actions after drag and drop

// …

// Close the browser

driver.quit();

}

}

**Q53. How do you handle dynamic dropdowns and select options using Selenium WebDriver?**

// Launch Chrome browser

WebDriver driver = new ChromeDriver();

// Navigate to a webpage with a dropdown/select element driver.get(“https://example.com”);

// Find the dropdown/select element

WebElement dropdown = driver.findElement(By.id(“dropdown-id”));

// Create Select object

Select select = new Select(dropdown);

// Select an option by visible text select.selectByVisibleText(“Option 1”);

// Select an option by value select.selectByValue(“option1”);

// Select an option by index select.selectByIndex(0);

// Perform further actions after dropdown handling

// …

// Close the browser

Driver.quit();

**Q54. What is Cross browser testing?**

Cross Browser Testing is a type of functional test to check that your web application works as expected in different browsers. A web application can be opened in any browser by the end user. For example, some people prefer to open https://x.com in Firefox browser, while others can be using Chrome browser or IE.

So we need to ensure that the web application will work as expected in all popular browsers so that more people can access and use it. This motive can be fulfilled with Cross Browser Testing of the product.

**Q55. What is Fluent Wait In Selenium WebDriver?**

FluentWait can define the maximum amount of time to wait for a specific condition and frequency with which to check the condition before throwing an “ElementNotVisibleException” exception.

**Q56. What is the difference between driver.close() and driver.quit() methods?**

The purpose of these two methods (driver.close and driver.quit) is almost same. Both allow us to close a browser but still, there is a difference.

**Q57. Where will you maintain information like URL, login, and password?**

These type of information which are used very often and change as per the authorization and environment should always be maintained in a separate file. In case the user hardcore it into the code system, then the user has to change it in every file which has its reference.

**Q58. What is DOM?**

The document object represents the whole HTML document.

When the HTML document is loaded in the browser, it becomes a document object. It is the root element that represents the HTML document. It has properties and methods. With the help of document objects, we can add dynamic content to our web page.

**Q59. Explain what is an assertion in Selenium and what are the types of assertions.**

The actual result is compared with the expected result with the help of Assertion. It means verification is done to check if the state of the application is the same as what we are expecting or not. For creating assertion we are going to use the Assert class provided by TestNG.

There are two types of Assertion:-  Hard Assertions and Soft assertions.

**Q60. What are XPath and its types?**

XPath stands for XML(eXtensible Markup Language) Path. Using XPath we can navigate to any element in an XML document. Since XML is a component of HTML, XPaths can be used to find web elements on any web page.  There are two types of XPath: 1. Relative XPath 2. Absolute XPath

**Q61. What is a Framework?**

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

**Q62. What methods can be used to locate the frame?**

Basically, we can switch over the elements and handle frames in Selenium using 3 ways.

By Index

By Name or Id

By Web Element

**Q63. Using a web driver how you can store a value that is a text box?**

You can use the following command to store a value which is a text box using the web driver:

driver.findElement(By.id(“your Textbox”)).sendKeys(“your keyword”);

**Q65. Can I navigate back and forth in a browser in Selenium WebDriver?**

We use Navigate interface to navigate back and forth in a browser. It has methods to move back, forward as well as refresh a page:

driver.navigate().forward(); – to navigate to the next web page with reference to the browser’s history

driver.navigate().back(); – takes back to the previous webpage with reference to the browser’s history

driver.navigate().refresh(); – to refresh the current web page thereby reloading all the web elements

driver.navigate().to(“url”); – to launch a new web browser window and navigate to the specified URL.

**Q66. What are the ways to refresh a browser using Selenium WebDriver?**

There are multiple ways to refresh a page in Selenium:

1. Using driver.navigate().refresh() command as mentioned in the question 45

2. Actions action = new Actions(driver);

action.keyDown(Keys.CONTROL).sendKeys(Keys.F5).keyUp(Keys.CONTROL).perform();

**Q67. How To Resize Browser Window Using Selenium WebDriver?**

To resize the browser window to particular dimensions, we use the ‘Dimension’ class to resize the browser window or driver.manage().window().maximize();

**Q68. How to find out whether an element is displayed on the web page?**

WebDriver facilitates the user with the following methods to check the visibility of the web elements. These web elements can be buttons, drop boxes, checkboxes, radio buttons, labels, etc.

isDisplayed()

boolean elePresent = driver.findElement(By.xpath(“xpath”)).isDisplayed();

isSelected()

boolean eleSelected= driver.findElement(By.xpath(“xpath”)).isSelected();

isEnabled()

boolean eleEnabled= driver.findElement(By.xpath(“xpath”)).isEnabled();

**Q69. How can you find Broken Links on a page using Selenium WebDriver?**

**Q71. List some scenarios which we cannot automate using Selenium WebDriver?**

Bitmap comparison is not possible using Selenium WebDriver

Automating Captcha is not possible using Selenium WebDriver

We can not read bar code using Selenium WebDriver

**Q72. Explain how you can switch back from a frame.**

To switch back from a frame use method defaultContent()

Syntax-driver.switchTo().defaultContent();

**Q73. How can we get the text of a web element?**

Using getText();

**Q74. How to press the ENTER key on the text box In Selenium WebDriver?**

To press the ENTER key using Selenium WebDriver, We need to use Selenium Enum Keys with its constant ENTER.

driver.findElement(By.xpath(“xpath”)).sendKeys(Keys.ENTER);

**Q75. Write steps to automate primary “login” functionality test cases for an application.**

Step 1) Understand the project requirement.

Step 2) Identify the Test scenarios

Step 3) Prepare a data input file with the data corresponding to each scenario

Step 4) Launch the tool from the program.

Step 5) Identify the username, password, and login buttons.

Step 6) Verify that the error message for negative scenarios is the same as the success message for positive test -scenarios.

**Q77. What is TestNg?**

TestNG is a testing framework designed to simplify a broad range of testing needs, from unit testing to integration testing.

**Q78. What are the advantages of TestNG?**

TestNG has the following advantages:

1) Firstly, TestNG is capable of producing reports automatically with all the necessary information such as failed tests, passed tests, test execution times, etc.

2) Secondly, TestNG makes use of annotations such as @BeforeMethod, @Test, etc., which are easily understandable as their naming is after their working.

3) Thirdly, TestNG provides a grouping of methods by which we can group multiple methods as one unit. In other words, Grouping performs operations on all the tests in a group at once rather than individually.

4) Fourthly, TestNG provides a test method parameterization, which means we can provide parameters in the TestNG and call the function repeatedly with different values. Moreover, parameterization helps in data-driven testing in TestNG.

5) Fifthly, TestNG provides the prioritization of methods. In other words, by defining the priorities of the methods in TestNG, we can alter the default execution sequence of the test methods according to our wish.

6) In addition to the above, TestNG allows parallel testing, which increases efficiency and improves the overall running time of test methods. With the TestNG framework, you can easily integrate with other tools such as Maven, Jenkins, etc. Moreover, TestNG provides a feature to run multiple test methods on various browsers to test for cross-browser compatibility issues on your website. It is cross-browser testing.

Additionally, TestNG allows us to run the tests separately. So, if you run the tests and only one test failed, you can run this test independently in the next execution.

Moreover, TestNG allows the test methods to depend on each other. It’s also called Test Dependency in TestNG.

Lastly, TestNG provides a bunch of assertion methods for testing more efficiently.

Subsequently, you can learn more about the benefits of the TestNG framework here.

**Q79. How do you run the TestNG script?**

TestNG script is run by right-clicking on the TestNG class -> Run As -> TestNG Test.

**Q80. How to create and run TestNG.xml?**

In TestNG framework, we need to create a TestNG XML file to create and handle multiple test classes. We do configure our test run, set test dependency, include or exclude any test, method, class or package and set priority etc in the XML file.

**Q81. How can we create a data-driven framework using TestNG?**

By using @DataProvider annotation,  we can create a Data Driven Testing Framework.

@DataProvider(name=”getData”)

public Object[][] getData(){

//Object [][] data = new Object [rowCount][colCount];

Object [][] data = new Object [2][2];

data [0][0] = “FirstUid”;

data [0][1] = “FirstPWD”;

data[1][0] = “SecondUid”;

data[1][1] = “SecondPWD”;

return data;

}

**Q82. What is Parameterized testing in TestNG?**

Parameterized tests allow developers to run the same test over and over again using different values.

There are two ways to set these parameters:

with testng.xml –

<parameter name=“url” value=“https://www.guvi.in/”></parameter>

<parameter name=“username” value=“abc@guvigeek.com”></parameter>

<parameter name=“password” value=“guvigeek”></parameter>

with Data Providers  –

@DataProvider(name=”ReadExcel”)

public String[][] getData() throws IOException {

String[][] readData = ReadExcel.readExcelSheet(excelfile);

return readData;}

@Test(dataProvider = “ReadExcel”)

**Q83.  How to set test case priority in TestNG?**

We use priority attribute to the @Test annotations. In case priority is not set then the test scripts execute in alphabetical order.

package TestNG;

import org.testng.annotations.\*;

public class PriorityTestCase{

@Test(priority=0)

public void testCase1() {

system.out.println(“Test Case 1”);

}

@Test(priority=1)

public void testCase2() {

system.out.println(“Test Case 2”);

}

}

**Q84. What are the annotations available in TestNG?**

@BeforeSuite : The annotated method will be run before all tests in this suite have run.

@BeforeTest : The annotated method will be run before any test method belonging to the classes inside the <test> tag is run.

@BeforeClass : The annotated method will be run before the first test method in the current class is invoked.

@DataProvider : Marks a method as supplying data for a test method.

@BeforeMethod : The annotated method will be run before each test method.

@Test: Marks a class or a method as part of the test.

@AfterMethod: The annotated method will be run after each test method.

@AfterClass : The annotated method will be run after all the test methods in the current class have been run.

@AfterTest : The annotated method will be run after all the test methods belonging to the classes inside the <test> tag have run.

@AfterSuite : The annotated method will be run after all tests in this suite have run.

@Parameters : To inject static datas into the testcase

@Parameters({“url”,”username”,”password”})

**Q85. What is TestNG Assert and list out common TestNG Assertions?**

TestNG Asserts help us to verify the condition of the test in the middle of the test run. Based on the TestNG Assertions, we will consider a successful test only if it is completed the test run without throwing any exception.

Some of the common assertions supported by TestNG are:

assertEqual(String actual,String expected)

assertEqual(String actual,String expected, String message)

assertEquals(boolean actual,boolean expected)

assertTrue(condition)

assertTrue(condition, message)

assertFalse(condition)

assertFalse(condition, message)

**Q87. How to disable a test case in TestNG?**

To disable the test case we use the parameter enabled = false to the @Test annotation.

@Test(enabled = false)

**Q88. How to skip a @Test method from execution in TestNG?**

By using throw new SkipException()

Once SkipException() thrown, remaining part of that test method will not be executed and control will goes directly to next test method execution.

throw new SkipException(“Skipping – This is not ready for testing “);

**Q89. How to Ignore a test case in TestNG?**

To ignore the test case we use the parameter enabled = false to the @Test annotation.

@Test(enabled = false)

**Q90. How TestNG allows to state dependencies?**

TestNG allows two ways to declare the dependencies.

Using attributes dependsOnMethods in @Test annotations.

Using attributes dependsOnGroups in @Test annotations.

**Q91. What are the different ways to produce reports for TestNG results?**

TestNG offers two ways to produce a report.

Listeners implement the interface org.testng.ITestListener and are notified in real-time of when a test starts, passes, fails, etc…

Reporters implement the interface org.testng.IReporter and are notified when all the suites have been run by TestNG. The IReporter instance receives a list of objects that describe the entire test run.

**Q92. What is the use of @Listener annotation in TestNG?**

TestNG listeners are used to configure reports and logging. One of the most widely used listeners in testNG is ITestListener interface. It has methods like onTestStart, onTestSuccess, onTestFailure, onTestSkipped etc. We should implement this interface creating a listener class of our own. Next, we should add the listener’s annotation (@Listeners) in the Class which was created.

**Q93. How to write regular expression In testng.xml file to search @Test methods containing “smoke” keyword.**

Regular expression to find @Test methods containing keyword “smoke” is as mentioned below.

<methods>

     <include name=”.\*smoke.\*”/>

</methods>

**Q94. What is the time unit we specify in test suites and test cases?**

We specify the time unit in test suites and test cases in milliseconds

**Q96. What is the use of @Test(invocationCount=x)?**

The invocationcount attribute tells how many times TestNG should run a test method

@Test(invocationCount = 10)

public void testCase1(){

In this example, the method testCase1 will be invoked ten times

**Q97. What is the use of @Test(threadPoolSize=x)?**

The threadPoolSize attribute tells to form a thread pool to run the test method through multiple threads.

Note: This attribute is ignored if invocationCount is not specified

@Test(threadPoolSize = 3, invocationCount = 10) public void testCase1(){

In this example, the method testCase1 will be invoked from three different threads

**Q98. What does the test timeout mean in TestNG?**

The maximum number of milliseconds a test case should take.

@Test(threadPoolSize = 3, invocationCount = 10,  timeOut = 10000)

public void testCase1(){

In this example, the function testCase1 will be invoked ten times from three different threads. Additionally, a time-out of ten seconds guarantees that none of the threads will block on this thread forever.

**Q99. What are @Factory and @DataProvider annotations?**

@Factory: A factory will execute all the test methods present inside a test class using a separate instance of the respective class with a different set of data.

@DataProvider: A test method that uses DataProvider will execute the specific methods multiple times based on the data provided by the DataProvider. The test method will be executed using the same instance of the test class to which the test method belongs.

**Q100. What is the Page Object Model in Selenium?**

The Page Object Model is a Design Pattern that has become popular in Selenium Test Automation. It is a widely used design pattern in Selenium for enhancing test maintenance and reducing code duplication.

Page object model (POM) can be used in any kind of framework such as modular, data-driven, keyword-driven, hybrid framework, etc.

A page object is an object-oriented class that serves as an interface to a page of your Application Under Test(AUT). The tests then use the methods of this page object class whenever they need to interact with the User Interface (UI) of that page.

The benefit is that if the UI changes for the page, the tests themselves don’t need to change, only the code within the page object needs to change. Subsequently, all changes to support that new UI is located in one place

**Q101. What is Page Factory?**

We have seen that the ‘Page Object Model’ is a way of representing an application in a test framework. For every ‘page’ in the application, we create a Page Object to reference the ‘page’ whereas a ‘Page Factory’ is one way of implementing the ‘Page Object Model’.

**Q102. What is the difference between a Page Object Model (POM) and a Page Factory?**

Page Object is a class that represents a web page and holds the functionality and members. Page Factory is a way to initialize the web elements you want to interact with within the page object when you create an instance of it.

**Q103. What are the advantages of the Page Object Model Framework?**

Code reusability – We could achieve code reusability by writing the code once and using it in different tests.

Code maintainability – There is a clean separation between test code and page-specific code such as locators and layout which makes it very easy to maintain code. Code changes only on Page Object Classes when a UI change occurs. It enhances test maintenance and reduces code duplication.

Object Repository – Each page will be defined as a Java class. All the fields on the page will be defined in an interface as members. The class will then implement the interface.

Readability – Improves readability due to clean separation between test code and page-specific code

**What are the four parameters of Selenium?**

There are four conditions (parameters) for Selenium to pass a test. These are as follows: URL, host, browser, and port number.

**What are the 5 methods in Selenium?**

There are five Selenium Method Categories in all:

1. Browser Methods

2. WebElement Methods

3. Navigation Methods

4. Wait Methods

5. Switch Methods

**Q Explain the difference between findElement and findElements.**

* **findElement**: This method is used to find a single web element on a web page. If the element is found, it returns a WebElement object representing the first matching element. If no element is found, it throws a NoSuchElementException.
* **findElements**: This method is used to find multiple web elements on a web page. It returns a list of WebElement objects representing all matching elements. If no elements are found, it returns an empty list.

**Question**: **If you click a link that opens a new window, how do you switch to that window and perform actions on it?**

|  |
| --- |
| String mainWindowHandle = driver.getWindowHandle();  Set<String> allWindowHandles = driver.getWindowHandles();  for (String handle : allWindowHandles) {      if (!handle.equals(mainWindowHandle)) {          driver.switchTo().window(handle);          // Perform actions in the new window          driver.close(); // Close the new window          break;      }  }  driver.switchTo().window(mainWindowHandle); // Switch back to the main window |

**Question**: **How do you handle a scenario where multiple windows are opened, and you need to switch to a specific one based on its title?**

|  |
| --- |
| String desiredWindowTitle = "Target Window Title";  Set<String> allWindowHandles = driver.getWindowHandles();  for (String handle : allWindowHandles) {      driver.switchTo().window(handle);      if (driver.getTitle().equals(desiredWindowTitle)) {          // Perform actions in the desired window          break;      }  } |

**Scenario: Closing All Windows Except the Main Window**

**Question**: **How do you close all windows except the main window?**

|  |
| --- |
| String mainWindowHandle = driver.getWindowHandle();  Set<String> allWindowHandles = driver.getWindowHandles();  for (String handle : allWindowHandles) {      if (!handle.equals(mainWindowHandle)) {          driver.switchTo().window(handle);          driver.close();      }  }  driver.switchTo().window(mainWindowHandle); // Back to the main window |

**Question: A pop-up window appears after clicking a button. How do you interact with this pop-up window and then return to the main window?**

|  |
| --- |
| String mainWindowHandle = driver.getWindowHandle();  Set<String> allWindowHandles = driver.getWindowHandles();  for (String handle : allWindowHandles) {      if (!handle.equals(mainWindowHandle)) {          driver.switchTo().window(handle);          // Interact with the pop-up window          driver.close(); // Close the pop-up          break;      }  }  driver.switchTo().window(mainWindowHandle); // Return to the main window |

**Question: After performing certain actions, how would you verify that a specific number of windows are open?**

|  |
| --- |
| int expectedWindowsCount = 3; // For example  Set<String> allWindowHandles = driver.getWindowHandles();  if (allWindowHandles.size() == expectedWindowsCount) {      System.out.println("Correct number of windows are open.");  } else {      System.out.println("Incorrect number of windows are open.");  } |

**Question**: How would you handle a scenario where you need to switch back and forth between two windows?

|  |
| --- |
| String firstWindowHandle = driver.getWindowHandle();  driver.findElement(By.linkText("Open new window")).click(); // Opens a new window  Set<String> allWindowHandles = driver.getWindowHandles();  String secondWindowHandle = null;  for (String handle : allWindowHandles) {      if (!handle.equals(firstWindowHandle)) {          secondWindowHandle = handle;          driver.switchTo().window(secondWindowHandle);          // Perform actions in the second window          break;      }  }  // Switch back to the first window  driver.switchTo().window(firstWindowHandle);  // Perform actions in the first window  // Switch again to the second window  driver.switchTo().window(secondWindowHandle);  // Perform more actions in the second window |

**Question. If 5 tabs are open, one opened google, one facebook, one CNN, one twitch, one gmail in same given order, now I want to close 3 tab , what to do**

|  |
| --- |
| import org.openqa.selenium.WebDriver;  import org.openqa.selenium.chrome.ChromeDriver;  import java.util.ArrayList;  import java.util.List;  import java.util.Set;  public class CloseMultipleTabs {      public static void main(String[] args) {          // Open the five websites in different tabs          driver.get("https://www.google.com");          driver.switchTo().newWindow(WindowType.TAB).get("https://www.facebook.com");          driver.switchTo().newWindow(WindowType.TAB).get("https://www.cnn.com");          driver.switchTo().newWindow(WindowType.TAB).get("https://www.twitch.tv");          driver.switchTo().newWindow(WindowType.TAB).get("https://mail.google.com");          // Get all window handles          Set<String> allWindowHandles = driver.getWindowHandles();          List<String> allHandlesList = new ArrayList<>(allWindowHandles);          // Close the first three tabs          for (int i = 0; i < 3; i++) {              driver.switchTo().window(allHandlesList.get(i));              driver.close();            // Switch to the remaining open tab (Twitch or Gmail, depending on the order)          driver.switchTo().window(allHandlesList.get(3));          // Continue your test logic here...          // At the end, quit the WebDriver session          driver.quit();      } |

**Explain this line?**

|  |
| --- |
| Set<String> allWindowHandles = driver.getWindowHandles();  List<String> allHandlesList = new ArrayList<>(allWindowHandles); |

Explanation:

1. Set<String> allWindowHandles = driver.getWindowHandles();:
   * driver.getWindowHandles() is a method provided by Selenium WebDriver that returns a Set<String> containing the unique identifiers (window handles) of all the currently opened browser windows or tabs in the session.
   * A window handle is a unique string identifier that the browser assigns to each open window or tab during a WebDriver session.
   * The returned Set<String> contains all these handles. Since Set does not maintain the order of elements, the window handles might not be in the order the tabs were opened.
2. List<String> allHandlesList = new ArrayList<>(allWindowHandles);:
   * This line converts the Set<String> of window handles into a List<String>.
   * new ArrayList<>(allWindowHandles) creates an ArrayList and initializes it with all elements from the Set.
   * The purpose of converting the Set to a List is to allow for indexed access to the window handles. In other words, you can now access the handles by their position (index) in the list, such as allHandlesList.get(0) for the first handle, allHandlesList.get(1) for the second handle, and so on.

Why Convert from Set to List?

* Indexed Access: Unlike a Set, a List allows you to access elements by their index (e.g., the third tab's handle can be accessed using allHandlesList.get(2)).
* Order Preservation: While the Set itself doesn't guarantee order, the order in the List corresponds to the order of iteration over the Set—this is often consistent with the order the tabs were opened in many implementations, though not guaranteed.

**ERROR AND EXCEPTION IN SELENIUM**

In Selenium WebDriver, errors and exceptions are common, especially when dealing with various elements on a webpage, browser interactions, and different environments. Understanding these exceptions helps in debugging and writing more robust and resilient automation scripts.

**Common Selenium WebDriver Exceptions**

1. **NoSuchElementException**
   * **Cause**: Raised when WebDriver cannot find an element on the web page using the given locator.
   * **Solution**: Ensure the locator used is correct, wait for the element to be present using waits, or handle the exception properly.
2. **TimeoutException**
   * **Cause**: Thrown when a command does not complete within the specified time. This often happens with explicit waits.
   * **Solution**: Adjust the timeout period or verify that the element is loading as expected within the given time.
3. **NoSuchWindowException**
   * **Cause**: Occurs when trying to switch to a window that does not exist.
   * **Solution**: Ensure the window handle is correct and exists, or catch the exception to handle it gracefully.
4. **NoSuchFrameException**
   * **Solution**: Verify that the frame exists and is correctly identified, or handle the exception if the frame is not always present.
5. **StaleElementReferenceException**
   * **Cause**: Raised when an element is no longer attached to the DOM. This typically happens when the page is refreshed or elements are dynamically loaded.
   * **Solution**: Re-locate the element after the DOM has changed or handle the exception to retry the action.
6. **ElementNotVisibleException**
   * **Cause**: Occurs when an element is present in the DOM but is not visible, and an **Solution**: Ensure the element is visible using WebDriverWait or check the visibility condition before interacting with it.
7. **ElementNotInteractableException**
   * **Cause**: Thrown when an element is visible but not interactable (e.g., disabled input field).
   * **Solution**: Wait until the element is interactable or ensure that no overlays or modals are obstructing the element.

1. **InvalidSelectorException**
   * **Cause**: Occurs when the selector used to find an element is invalid, often due to incorrect syntax.
   * **Solution**: Ensure the selector is correctly written and follows the syntax for the locator type being used.
2. **WebDriverException**
   * **Cause**: A generic exception indicating an issue with WebDriver itself. This can happen due to problems with the WebDriver installation, browser compatibility, or unexpected browser closure.
   * **Solution**: Verify the WebDriver and browser versions are compatible, check the WebDriver path, and ensure no issues with the browser instance.
3. **InvalidElementStateException**
   * **Cause**: Thrown when an element is in a state that does not allow interaction (e.g., attempting to type into a read-only input field).
   * **Solution**: Ensure the element's state allows interaction before performing actions on it.
4. **MoveTargetOutOfBoundsException**
   * **Cause**: Thrown when trying to move the mouse to an element that is out of the browser's viewport.
   * **Solution**: Scroll to the element to bring it into the viewport before performing actions on it.
5. **SessionNotFoundException**
   * **Cause**: Occurs when trying to interact with a WebDriver session that no longer exists (e.g., browser has been closed).
   * **Solution**: Ensure the session is active before attempting any interaction, or handle the exception to recreate the session.
6. **TimeoutException**
   * **Cause**: Thrown when a command takes too long to complete, often with waits like WebDriverWait.
   * **Solution**: Increase the timeout duration, or handle the exception gracefully by providing a fallback.

**Handling Exceptions in Selenium**

You can handle exceptions in Selenium using try-catch blocks in your code to make it more robust:

|  |
| --- |
| try {      WebElement element = driver.findElement(By.id("some-id"));      element.click();  } catch (NoSuchElementException e) {      System.out.println("Element not found: " + e.getMessage());  } catch (TimeoutException e) {      System.out.println("Operation timed out: " + e.getMessage());  } catch (Exception e) {      System.out.println("An unexpected error occurred: " + e.getMessage()); |

**When to Use Each:**

* **Use driver.close()**:
  + When you want to close a specific tab or window but continue working with other open windows/tabs.
  + For example, if your test scenario involves opening multiple tabs, performing actions in each, and closing them one by one, you would use driver.close().
* **Use driver.quit()**:
  + When your test has completed, and you want to close all browser instances and clean up the WebDriver session.
  + It is typically used at the end of the test to ensure that all browser instances are closed and no background processes related to WebDriver are left running.

**Scope**: driver.close() closes only one window or tab, while driver.quit() closes all windows/tabs and ends the WebDriver session.

**Session Management**: driver.close() keeps the session alive if there are other open windows or tabs, while driver.quit() completely ends the session.

.

**What are static and dynamic variable in selenium and how to handle dynamic variable ?**

**Static Variables**

Static variables are those that remain constant and do not change with each test execution or page load. These can be identifiers like:

* Fixed IDs: Elements that always have the same id.
* Static Class Names: Elements that always have the same class attribute.
* Tag Names: Elements identified by a consistent tag.
* XPath/CSS Selectors: Locators that point to elements with a fixed structure.

**Handling Static Variables:**

* You can directly use static locators in your Selenium code without worrying about changes

|  |
| --- |
| WebElement element = driver.findElement(By.id("fixedId")); |

**Dynamic Variables**

**Dynamic variables** refer to elements or attributes that can change with each page load, session, or user interaction. These could include:

* **Dynamic IDs**: IDs that include a changing part, like a timestamp or session ID.
* **Dynamically Generated Elements**: Elements whose attributes or even existence depends on user actions or specific conditions.
* **Elements with Changing Attributes**: Elements that might change class, name, or other attributes dynamically.

**Handling Dynamic Variables**:

1. **XPath Functions**: Utilize XPath functions like contains(), starts-with(), and text() to create locators that can match elements based on partial values.

|  |
| --- |
| WebElement element = driver.findElement(By.xpath("//div[contains(@id, 'partialId')]")); |

See this:  <https://www.youtube.com/watch?v=H1_rmdsZ4gI>

1. **Use CSS Selectors**:

* CSS selectors can be designed to match patterns, making them useful for dynamic elements.

|  |
| --- |
| WebElement element = driver.findElement(By.cssSelector("div[id\*='partialId']")); |

1. **Use Wildcards**:
   * Wildcards in locators can match parts of dynamic attributes.

|  |
| --- |
| WebElement element = driver.findElement(By.xpath("//div[starts-with(@id, 'prefixId')]")); |

1. **Find Element by Other Static Attributes**:
   * Sometimes, even if an id is dynamic, other attributes like class or tag may remain constant.

|  |
| --- |
| WebElement element = driver.findElement(By.className("staticClassName")); |

1. **JavaScript Executor**:
   * In complex cases, you can use JavaScript to find elements.

|  |
| --- |
| WebElement element = (WebElement) ((JavascriptExecutor) driver).executeScript("return document.querySelector('[id^=\"prefix\"]')"); |

1. **Wait for Elements**:

Dynamic elements might not be present immediately. Using Selenium's wait mechanisms (like WebDriverWait) can help manage this.

|  |
| --- |
| WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));  WebElement element = wait.until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicId"))); |

**Xpath and css locators in selenium**

In Selenium, both XPath and CSS selectors are commonly used to locate elements on a web page. Here's a comparison of the two:

**XPath**

XPath is a powerful language that allows you to navigate through the elements and attributes in an XML document or HTML DOM structure.

**Advantages of XPath:**

* **Flexible Selection:** You can traverse both forward and backward in the DOM, allowing for complex queries.
* **Access to All Nodes:** XPath can be used to access elements based on attributes, text, and even relationships between elements.
* **Conditional Matching:** XPath supports conditional matching, like selecting elements based on certain attributes or their values.

**Common XPath Syntax:**

* **Absolute XPath:**html/body/div[1]/div[2]/input  
  (Starts from the root and leads to the target element.)
* **Relative XPath:**//input[@id='username']  
  (Starts from anywhere in the DOM structure and locates elements based on attributes.)
* **Using Attributes:**//button[@type='submit']
* **Using Text:**//a[text()='Login']
* **Contains Function:**//input[contains(@class, 'form-control')]
* **Following Sibling:**//label[@for='username']/following-sibling::input
* **Parent Node:**//input[@id='password']/parent::div

**CSS Selectors**

CSS Selectors are another way to locate elements. They are generally faster and more concise than XPath but less powerful for certain types of complex queries.

**Advantages of CSS Selectors:**

* **Performance:** CSS selectors are usually faster in most modern browsers.
* **Simplicity:** For straightforward element selection, CSS selectors are simpler and more concise.
* **Supported by Web Development:** Since they are also used in web development, many developers are already familiar with them.

**Common CSS Selector Syntax:**

* **ID Selector:**#username  
  (Selects an element with a specific ID.)
* **Class Selector:**.form-control  
  (Selects all elements with a specific class.)
* **Element Selector:**input  
  (Selects all <input> elements.)
* **Attribute Selector:**input[type='text']  
  (Selects elements based on an attribute.)
* **Child Selector:**div > input  
  (Selects direct child elements.)
* **Descendant Selector:**div input  
  (Selects all input elements inside a div, no matter how deeply nested.)
* **Pseudo-Classes:**a:hover  
  (Selects elements when hovered over.)

**Example Using XPath:**

|  |
| --- |
| **WebElement username = driver.findElement(By.xpath("//input[@id='username']"));** |

**Example Using CSS:**

|  |
| --- |
| **WebElement username = driver.findElement(By.cssSelector("#username"));** |

The **Robot** class and the **Actions** class in Selenium WebDriver are both used to handle advanced interactions with web pages, but they serve different purposes and are suited for different use cases. Here's a comparison of the two:

**1. Robot Class**

The Robot class is part of **Java AWT** and is designed for simulating low-level keyboard and mouse events at the operating system level. It is not specific to Selenium and works beyond the browser context.

**Key Features**

* Works at the OS level, meaning it can interact with non-browser applications as well.
* Useful for handling scenarios like:
  + Keyboard inputs (e.g., pressing special keys like ALT, SHIFT, etc.)
  + Mouse movements and clicks outside the browser.
  + Handling file upload/download windows (native popups).
  + Taking screenshots.

**Limitations:** Cannot directly interact with web elements (you need to rely on coordinates or key combinations).

* Platform-dependent and sometimes slower compared to Actions.

**Example**

|  |
| --- |
| import java.awt.Robot;  import java.awt.event.KeyEvent;  public class RobotExample {      public static void main(String[] args) throws Exception {          Robot robot = new Robot();            // Simulate pressing the TAB key          robot.keyPress(KeyEvent.VK\_TAB);          robot.keyRelease(KeyEvent.VK\_TAB);            // Simulate pressing and releasing ENTER          robot.keyPress(KeyEvent.VK\_ENTER);          robot.keyRelease(KeyEvent.VK\_ENTER);      } |

**2. Actions Class:** The Actions class is part of **Selenium WebDriver** and is specifically designed for advanced user interactions with web elements in a browser.

**Key Features**

* Works directly within the browser context.
* Supports complex interactions like:
  + Hovering over elements.
  + Drag-and-drop actions.
  + Clicking, double-clicking, or right-clicking.
  + Keyboard inputs specific to browser elements.
  + Handling sliders and resizing elements.

**Limitations:** Limited to the browser and does not work outside it.and Cannot handle native OS-level dialogs like file uploads.

|  |
| --- |
| import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.WebElement;  import org.openqa.selenium.chrome.ChromeDriver;  import org.openqa.selenium.interactions.Actions;  public class ActionsExample {      public static void main(String[] args) {          WebDriver driver = new ChromeDriver();          driver.get("https://example.com");          // Locate the element to hover over          WebElement element = driver.findElement(By.id("menu"));          // Create an Actions object          Actions actions = new Actions(driver);          // Perform a hover action          actions.moveToElement(element).perform();            // Perform a drag-and-drop action (example)          WebElement source = driver.findElement(By.id("source"));          WebElement target = driver.findElement(By.id("target"));          actions.dragAndDrop(source, target).perform();          driver.quit(); |

**When to Use**

* **Robot Class**: Use for tasks outside the browser context, such as handling OS-level dialogs or automating non-browser applications.
* **Actions Class**: Use for tasks within the browser, like simulating mouse hover, drag-and-drop, or complex key combinations.

**What’s your process for building a custom test automation framework?**

When building a custom test automation framework, I follow these steps:

Define Requirements: Understand the testing needs and the application under test.

Choose a Framework Type: Decide between Data-Driven, Keyword-Driven, or Hybrid frameworks based on the project requirements

.

Set Up Project Structure: Organize the project into directories for tests, utilities, and resources.

Implement Core Components: Develop reusable components for test execution, reporting, and logging.

Integrate Testing Tools: Use tools like TestNG for test management and reporting.

Continuous Integration: Set up CI/CD pipelines to automate the execution of tests.

***Comparison of Explicit, Implicit, and Fluent Waits in Selenium***

**Implicit Wait**

* **Definition**: Waits for a specified amount of time for all elements to appear before throwing a NoSuchElementException.
* **Scope**: Global (applied to all WebDriver interactions).
* **Best Use Case**: When all elements on a page load at the same pace and no complex conditions are needed.

// Set implicit wait of 10 seconds

driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(0));

* **Disadvantage**: It applies to every element lookup, which may lead to unnecessary delays if elements load faster.

**2. Explicit Wait**

* **Definition**: Waits for a specific condition to be true for a particular element before throwing an exception.
* **Scope**: Local (applied only to a specific element or condition).
* **Best Use Case**: For elements that may take varying times to load or require specific conditions to be met (e.g., element becomes clickable).

**Example (Java using WebDriverWait)**

// Wait for a specific element to be visible for up to 15 seconds

WebDriverWait wait = new WebDriverWait(driver, 15);

WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("submitButton")));

* **Advantages**: More flexible as it allows different wait conditions (e.g., visibility, clickability).
* **Disadvantage**: Needs to be applied for every specific condition manually.

**3. Fluent Wait**

* **Definition**: A more advanced version of Explicit Wait that allows you to define the polling interval (how frequently it checks the condition) and exception handling.
* **Scope**: Local (applied to specific elements).
* **Best Use Case**: For situations where elements may load with unpredictable delays, or intermittent failures occur due to short-lived issues.

Wait wait = new FluentWait<>(driver) .withTimeout(Duration.ofSeconds(30))

    .pollingEvery(Duration.ofSeconds(5))

    .ignoring(NoSuchElementException.class);

* **Advantages**: Offers more control with polling frequency and exception handling.

**Disadvantages**: More complex to configure than Implicit or Explicit Waits.

**4. Thread.sleep() in Selenium**

Thread.sleep() is a Java method that **pauses execution** for a specified amount of time. It is commonly used in Selenium tests to add delays.

**Why Is Thread.sleep() Not Recommended?**

🚫 **Fixed Delay**: It waits for the full duration, even if the element appears earlier.  
🚫 **Increases Execution Time**: Slows down tests unnecessarily.  
🚫 **Not Dynamic**: Does not adjust based on element availability.  
🚫 **Can Cause Flakiness**: If an element takes longer than expected, the test may fail.

**Difference Between POM (Page Object Model) and Page Factory in Selenium**

Both **Page Object Model (POM)** and **Page Factory** are design patterns that help improve the readability, maintainability, and scalability of Selenium test scripts

**Page Object Model (POM)**

* **Definition**:  
  A design pattern where each web page or section of a web application is represented as a separate class. Each class contains web elements and the actions that can be performed on the page.
* **How It Works**:  
  You define the web elements using By locators (e.g., By.id, By.xpath) and create methods to interact with those elements.

**Advantages**:

* Simple and easy to implement.
* Clear separation of web element locators and actions.
* Easy to maintain and modify.

**Disadvantages**:

* Repeated calls to driver.findElement for each interaction can lead to slower performance.
* No automatic lazy loading of web elements.

**Page Factory**

* **Definition**:  
  An extension of POM that uses the @FindBy annotation to locate web elements. It leverages the PageFactory class to initialize the elements and supports lazy loading (elements are only loaded when they are accessed).
* **How It Works**:  
  You annotate web elements with @FindBy and use PageFactory.initElements to initialize them.

**Advantages**:

* Cleaner syntax with annotations (@FindBy).
* Elements are initialized once using PageFactory.initElements, which can improve performance.
* Supports lazy loading, which can avoid StaleElementReferenceException in some cases.

**Disadvantages**:

* More complex to implement than standard POM.
* Difficult to debug issues when elements are not found or initialized properly.

**When to Use TestNG vs. Selenium Grid for Parallel Testing**

* **TestNG Parallel Execution**:  
  Suitable for running tests on a single machine across multiple browser instances. It’s easy to set up and works well for smaller test suites.
* **Selenium Grid Parallel Execution**:  
  Ideal for large-scale, distributed testing across multiple machines, operating systems, or browser versions. It helps ensure cross-browser compatibility and faster test execution.

**Differentiate between Assertions and Verifications in Selenium.**

**1. Assertions**

* Definition:  
  Assertions are conditions that must be true for the test to proceed. If an assertion fails, the test is immediately terminated (stops execution).
* Use Case:  
  Assertions are used when a condition is critical for continuing the test (e.g., checking that a user is successfully logged in before proceeding).
* Implementation in TestNG:  
  TestNG provides the Assert class for assertions.

**Output on Failure**: The test will immediately stop and report a failure if the assertion condition is not met

**Types of Assertions in Selenium**

Assertions in Selenium are primarily used with **TestNG** or **JUnit**.

**1. Hard Assertions (Assert)**

* **Stops test execution immediately if failed**.

import org.testng.Assert;

import org.testng.annotations.Test;

public class HardAssertionExample {

@Test

public void testHardAssertion() {

System.out.println("Step 1 executed");

**Assert.assertEquals(5, 10, "Numbers do not match");**

System.out.println("Step 2 executed"); // This won't execute if assertion fails

}

}

**Soft Assertions (SoftAssert)**

* **Does not stop execution immediately**, even if an assertion fails..

import org.testng.annotations.Test;

import org.testng.asserts.SoftAssert;

public class SoftAssertionExample {

@Test

public void testSoftAssertion() {

SoftAssert softAssert = new SoftAssert();

System.out.println("Step 1 executed");

softAssert.assertEquals(5, 10, "First check failed");

System.out.println("Step 2 executed");

softAssert.assertTrue(false, "Second check failed");

softAssert.assertAll(); // Reports all failures at the end

}

}

**2. Verifications**

* **Definition**:  
  Verifications are conditions that **log the failure but allow the test to continue**. This ensures that all verification points are checked even if some conditions fail.
* **Use Case**:  
  Verifications are used when it’s acceptable to continue the test even if a condition is not met (e.g., checking multiple UI elements on a page).
* **Implementation in TestNG**:  
  You can achieve verification by using SoftAssert in TestNG, which collects all failures and reports them at the end.

**Output on Failure**: All verifications are executed, and any failures are reported at the end of the test.

**When to Use Assertions vs. Verifications**

* **Assertions**:  
  Use assertions when the test cannot continue if the condition is not met (e.g., verifying the login was successful before performing further actions).
* **Verifications**:  
  Use verifications when multiple checks need to be performed, and you want to gather all the results before deciding the test outcome (e.g., validating multiple UI elements on a page).

**Approaches to Handling Frames, Pop-ups, and Alerts in Selenium:**

1. Handling Frames in Selenium

Frames are used to embed another HTML document within the main document. Selenium needs to switch its context to the frame to interact with the elements inside it.

Common Methods for Frames:

* driver.switchTo().frame(int index) – Switches by frame index (starting from 0).
* driver.switchTo().frame(String nameOrId) – Switches by frame name or ID.
* driver.switchTo().frame(WebElement element) – Switches using a frame WebElement.
* driver.switchTo().defaultContent() – Switches back to the main page.

**2. Handling Pop-ups (New Browser Windows or Tabs)**

Pop-ups often appear as new browser windows or tabs. Selenium allows switching between multiple windows or tabs.

Common Methods:

* driver.getWindowHandle() – Gets the current window handle.
* driver.getWindowHandles() – Gets all open window handles.
* driver.switchTo().window(String windowHandle) – Switches to a specific window or tab.

**3. Handling JavaScript Alerts in Selenium**

JavaScript alerts can be of three types:

1. **Simple Alerts** – Displays a message with an "OK" button.
2. **Confirmation Alerts** – Displays a message with "OK" and "Cancel" buttons.
3. **Prompt Alerts** – Displays a message with an input field and "OK" and "Cancel" buttons.

**Common Methods for Alerts**:

* driver.switchTo().alert() – Switches to the alert.
* alert.accept() – Clicks the "OK" button.
* alert.dismiss() – Clicks the "Cancel" button.
* alert.getText() – Retrieves the alert message text.
* alert.sendKeys(String text) – Enters text in a prompt alert.

**How do you initialize WebDriver?**

|  |
| --- |
| WebDriver driver = new ChromeDriver(); |

**How would you locate an element if it does not have attributes like ID or Class?**

Using JavaScriptExecutor in Selenium

JavaScriptExecutor is an interface in Selenium that allows you to execute JavaScript code within the context of the browser. This can be useful when standard WebDriver methods are not sufficient, such as when manipulating DOM elements, scrolling, handling alerts, or executing custom JavaScript.

How to Use JavaScriptExecutor in Selenium

1. Casting WebDriver to JavaScriptExecutor: Since JavaScriptExecutor is an interface, you'll need to cast your WebDriver instance to JavascriptExecutor to use its methods.

|  |
| --- |
| JavascriptExecutor js = (JavascriptExecutor) driver; |

**What are the new changes in Selenium 4 (W3C architecture)?**

1. W3C WebDriver Standard: Selenium now follows the W3C WebDriver standard, improving cross-browser consistency and stability.
2. Selenium Grid 4: Revamped with easier setup, better UI, and support for Docker and distributed execution.
3. Relative Locators: New locator strategies like above(), below(), toLeftOf(), toRightOf() for easier element identification based on relative positions.
4. Improved DevTools Protocol: Better integration with Chrome DevTools for tasks like network manipulation and performance metrics.
5. Mobile Testing: Enhanced integration with Appium for mobile web and app testing.
6. Deprecation of DesiredCapabilities: Replaced by more consistent Options (e.g., ChromeOptions, FirefoxOptions).
7. Shadow DOM Support: Improved handling of Shadow DOM elements in modern web applications.
8. New WebDriver APIs: Enhanced timeout, window handling, and cookie management.

**How do you switch between multiple windows in Selenium?**

Steps to Switch Between Multiple Windows

1. Get the Current Window Handle:
   * The current window handle can be retrieved using getWindowHandle().  
     String currentWindow = driver.getWindowHandle();
2. Get All Window Handles:
   * You can get all the window handles using getWindowHandles(), which returns a set of all the open window handles.  
     Set<String> allWindows = driver.getWindowHandles();
3. Switch to Another Window:
   * Iterate over the window handles to switch to a different window.

|  |
| --- |
| for (String windowHandle : allWindows) {      if (!windowHandle.equals(currentWindow)) {          driver.switchTo().window(windowHandle);          break;  // Once you switch, you can exit the loop      } |

**What are the exceptions in selenium?**

**NoSuchElementException**

Thrown when an element is not found on the web page. This usually happens when findElement() or findElements() is used and the element cannot be located.

**NoSuchWindowException**Thrown when the window you're trying to switch to is not found or no longer exists.

**NoSuchFrameException**

Thrown when switching to a frame that doesn’t exist or is unavailable.

**ElementNotInteractableException**

This occurs when an element is present on the page but cannot be interacted with (e.g., it's hidden, disabled, or covered by another element).

**TimeoutException**

Occurs when an operation (like waiting for an element to appear) exceeds the timeout limit set in WebDriverWait.

In **TestNG**, annotations are used to define and control the flow of test execution. They allow you to specify methods that should run before or after a test, or define test groups, dependencies, and more. TestNG annotations make your test scripts flexible, readable, and easy to maintain.

Here are the most common TestNG annotations:

**1. @Test**

* Marks a method as a test method.
* You can specify attributes like priority, groups, dependsOnMethods, and are enabled to customize the behavior.

**2. @BeforeMethod**

* Runs before each test method. It is used for setting up preconditions for the tests (e.g., initializing WebDriver, opening a browser).

**3. @AfterMethod**

* Runs after each test method. It is used for cleanup activities (e.g., closing a browser, clearing resources).

**4. @BeforeClass**

* Runs once before the first method of the current class is invoked. It is used for a setup that is needed only once for the entire class (e.g., initializing shared resources).

**5. @AfterClass**

* Runs once after all methods of the current class have been invoked. It is used for cleanup tasks that only need to be done once after all tests in a class have completed.

**6. @BeforeSuite**

* Runs before any tests in the test suite. This is useful for global setup (e.g., starting a server or setting global configurations).

**7. @AfterSuite**

* Runs after all tests in the test suite. It is used for global cleanup (e.g., stopping a server or closing resources after the entire suite has run).

**8. @BeforeTest**

* Runs before any test methods in the <test> tag of the TestNG XML file. This allows you to perform setup tasks before the tests of a particular suite start running.

**9. @AfterTest:** Runs after all test methods in the <test> tag of the TestNG XML file. It’s used for performing cleanup after a specific group of tests.

**10. @DataProvider̤:** Provides data to test methods. This annotation is used to create a method that supplies data for a test method. The test method is run once for each data set.

**11. @Parameters:** Used to pass parameters from the TestNG XML file to test methods. This allows for flexible test configuration.

**Why do we use waits in selenium?**

In Selenium, waits are used to handle synchronization issues between the test script execution and the loading of web elements on a webpage. Without proper waits, the script may try to interact with elements that haven't fully loaded yet, leading to NoSuchElementException, ElementNotInteractableException, or StaleElementReferenceException.

**What Are Synchronization Issues in Selenium?**

Synchronization issues occur when the **test script execution speed** does not match the **loading time of web elements** on a webpage. This can lead to situations where Selenium tries to interact with an element before it is fully available, causing exceptions.

**Types of Synchronization Issues**

1. **Page Load Delay**
   * The test script executes faster than the webpage loads, causing elements to be unavailable.
2. **Element Not Visible Yet:** The element is present in the DOM but not visible or interactable.

**CUCUMBER: BDD**

The Cucumber framework is a Behavior-Driven Development (BDD) testing tool used for writing test cases in a human-readable format. It enables collaboration between developers, testers, and business stakeholders by using Gherkin syntax (Given-When-Then).

**Key Components of Cucumber Framework**

1. Feature Files (.feature) – Contains test scenarios written in Gherkin.
2. Step Definitions – Maps Gherkin steps to Java (or other languages) methods.
3. Test Runner – Executes Cucumber tests using JUnit or TestNG.
4. Hooks – Code that runs before/after scenarios (setup/teardown).
5. Data Tables & Parameterization – Used to handle multiple test data inputs.

**What is BDD (Behavior-Driven Development)?**

Behavior-Driven Development (BDD) is a software development approach that enhances collaboration between developers, testers, and business stakeholders. It ensures that all stakeholders have a clear understanding of the application's behavior before development begins.

BDD uses plain English statements written in Gherkin syntax (Given-When-Then) to describe application behavior, making it easy to read and understand for both technical and non-technical members.

**Difference: Traditional Development vs. BDD**

| **Feature** | **Traditional Development** | **Behavior-Driven Development (BDD)** |
| --- | --- | --- |
| **Approach** | Code-driven, focuses on implementation first | Behavior-driven, focuses on user expectations first |
| **Requirement Format** | Written in technical documentation (may not be easily understood by business stakeholders) | Written in Gherkin syntax (easily readable by everyone) |
| **Stakeholder Involvement** | Limited collaboration between developers, testers, and business teams | Strong collaboration between developers, testers, and business stakeholders |
| **Test Writing** | Unit tests & automation scripts written after development | Tests (scenarios) are written before development starts |
| **Testing Framework** | Uses JUnit, TestNG, Selenium, etc. | Uses Cucumber, SpecFlow, Behave, etc. |
| **Example Requirement** | "Implement a login function with authentication." | "Given the user is on the login page, When they enter valid credentials, Then they should be logged in successfully." |
| **Flexibility** | Code-focused, harder to change when requirements shift | Requirement-focused, adapts easily to changing business needs |
| **Communication** | Gap between business and technical teams | Improved communication between teams |

**Example of BDD in Action**

Traditional Development Requirement:

* "Create a login functionality with a database check."
* Developers implement logic first, and testers write test cases later.

**BDD Requirement Using Gherkin (Feature File):**

Feature: Login functionality

Scenario: Successful login with valid credentials

Given User is on the login page

When User enters "admin" and "password123"

Then User should be logged in successfully

* Developers and testers collaborate on defining these scenarios before coding starts.
* Step definitions in Selenium/Java/Python execute these scenarios.

**Why Use BDD?**

✔ Ensures clear communication between business and technical teams  
✔ Prevents misinterpretation of requirements  
✔ Encourages test-first development (reduces bugs later)  
✔ Increases automation efficiency

**Feature File in Cucumber (BDD):** A feature file is a core part of the Cucumber framework where test scenarios are written in a human-readable format using Gherkin syntax.

It describes the behavior of an application from a user's perspective in Given-When-Then format.

**1. Structure of a Feature File**

A feature file has the following structure:  
1️. Feature: Describes the functionality under test.  
2️. Scenario: Represents a specific test case.  
3️. Given: Defines the initial state (preconditions).  
4️. When: Defines an action performed by the user.  
5️. Then: Defines the expected outcome.  
6️. And/But: Used for additional steps.  
7️. Background: Common preconditions for multiple scenarios.  
8️. Scenario Outline: Used for data-driven testing.

**Where to Store Feature Files?**

Feature files are usually stored under:  
📂 src/test/resources/features/ (for Maven projects)  
📂 src/test/java/features/ (for some Gradle setups)

**Who Writes Feature Files in BDD?**

Feature files in Cucumber (BDD) are usually written collaboratively by multiple stakeholders to ensure that the expected behavior of the application is well understood. The key contributors are:

**1. Business Analysts (BAs) / Product Owners (POs)**

* Define business requirements in plain English.
* Ensure the feature file aligns with user stories.
* Collaborate with developers and testers to refine scenarios.

**2. QA Engineers / Testers**

* Convert business requirements into test scenarios using Gherkin syntax.
* Ensure completeness by covering all edge cases.
* Validate that the feature file is clear and testable.

**3. Developers**

* Review the feature files to understand expected behavior.
* Implement the step definitions in code.
* Ensure the application behaves as described.

**4. Scrum Master / Agile Coach (Optional)**

* Facilitates collaboration between teams.
* Ensures that BDD principles are followed correctly.

**Example of a Basic Feature File:**

**Feature: Login Functionality**

**Scenario: Successful login with valid credentials**

**Given User is on the login page**

**When User enters "admin" and "password123"**

**Then User should be logged in successfully**

**Detailed Breakdown of Feature File Components**

**🔹 Feature:**

* Describes what the file is testing.

Feature: Login Functionality

**🔹 Scenario:**

* A single test case written in Given-When-Then format.

Scenario: Successful login with valid credentials

**🔹 Given:**

* Defines the **precondition** before the action.

Given User is on the login page

**🔹 When:**

* Specifies the **action** performed by the user.

When User enters "admin" and "password123"

**🔹 Then:**

* Specifies the **expected result**.

Then User should be logged in successfully

**🔹 And / But:**

* Used for additional steps in a scenario.

Given User is on the login page

And The login form is displayed

**Using Background (Common Precondition)**

If multiple scenarios share the **same setup**, use Background instead of repeating **Given** steps.

Feature: Login Functionality

Background:

Given User is on the login page

Scenario: Successful login

When User enters "admin" and "password123"

Then User should be logged in successfully

Scenario: Unsuccessful login with invalid password

When User enters "admin" and "wrongpassword"

Then User should see an error message

👉 The **Background** step runs **before every scenario** in the file.

**Scenario Outline (Data-Driven Testing)**

If you want to **test multiple sets of data**, use **Scenario Outline** with **Examples**.

Feature: Login Functionality

Scenario Outline: Login with multiple credentials

Given User is on the login page

When User enters "<username>" and "<password>"

Then User should see "<message>"

Examples:

| username | password | message |

| admin | password123 | Login successful |

| user1 | wrongpass | Invalid credentials |

| test | testpass | Login successful |

**What is Gherkin in BDD?**

**Gherkin** is a **domain-specific language (DSL)** used in **Behavior-Driven Development (BDD)** to write test cases in a human-readable format. It acts as a bridge between **business requirements** and **automation tests**.

📌 **Key Features:**  
✔ Uses simple, **plain English** syntax.  
✔ Written in .feature files.  
✔ Follows **Given-When-Then** structure.  
✔ Supports **multiple languages** (over 60+

Here are some key **Gherkin language keywords**:

**Main Keywords:**

1. **Feature** – Defines a feature of the application.
2. **Scenario** – Describes a specific test case.
3. **Scenario Outline** – Used for parameterized scenarios.
4. **Examples** – Provides data sets for a Scenario Outline.
5. **Background** – Defines preconditions for all scenarios in a feature.
6. **Given** – Describes the initial state (preconditions).
7. **When** – Specifies the action or event.
8. **Then** – Defines the expected outcome.
9. **And** – Used to add multiple Given/When/Then steps.
10. **But** – Used for negative conditions or variations.

**Other Keywords:**

* **Rule** – Defines business rules for a feature.
* **Doc Strings (""" or ''')** – Used for multi-line text inputs.
* **Tags (@tagname)** – Used to organize and filter scenarios.
* **Comments (#)** – Used to add comments in the file.

**The "Three Amigos" is a collaborative approach in Behavior-Driven Development (BDD) and Agile teams. It involves three key perspectives:**

1. **Business (Product Owner/BA)** – Defines what the feature should achieve.
2. **Development (Developer/Engineer)** – Ensures technical feasibility and implementation.
3. **Testing (QA/Tester)** – Defines how the feature will be tested.

**Purpose of the Three Amigos Meeting:**

* Align understanding before development starts.
* Discuss acceptance criteria using **Gherkin** (Given-When-Then).
* Identify edge cases and clarify business rules.
* Reduce rework and misunderstandings.

**Example Process in a Meeting:**

1. **Business**: "Users should be able to reset their password via email."
2. **Developer**: "We’ll create an API that sends a password reset link."
3. **Tester**: "What happens if the email is incorrect? Should we show a generic message?"

In **Cucumber with Selenium**, **Step Definitions** are Java methods that define how each step in a **Gherkin feature file** should be executed. Below are the steps to create **Step Definitions** and run **Feature Files** in a Cucumber project.

**1. Create Feature File**

A **Feature File** describes the test scenarios using **Gherkin syntax** (.feature file extension).

📌 **Example: Login.feature**

Feature: Login Functionality

Scenario: Successful Login

Given User is on the login page

When User enters valid username and password

Then User should be redirected to the homepage

**2. Create Step Definitions**

Step Definitions contain **Java methods** that execute the actions mentioned in the feature file.

📌 **Example: LoginSteps.java**

import io.cucumber.java.en.\*;

import org.junit.Assert;

public class LoginSteps {

@Given("User is on the login page")

public void user\_is\_on\_the\_login\_page() {

System.out.println("User navigates to login page");

}

@When("User enters valid username and password")

public void user\_enters\_valid\_username\_and\_password() {

System.out.println("User enters credentials");

}

@Then("User should be redirected to the homepage")

public void user\_should\_be\_redirected\_to\_the\_homepage() {

System.out.println("User successfully logged in");

}

}

📌 **Notes:**

* Each step in the feature file must have a corresponding method in the Step Definition file.
* The **annotations** (@Given, @When, @Then) map the steps to Java methods.
* The method names can be anything, but the **text inside the annotation** must match the feature file step.

**Key Takeaways**

✅ You **can** use a **single step definition file** for **multiple feature files** if the step texts match.  
✅ **Cucumber searches for step definitions in the package** specified in glue = "stepDefinitions".  
✅ If a feature file contains a **new step** not defined in the step definitions, you must add a new method.

Would you like to see how to organize step definitions into **multiple files** for better structure? 🚀

**Why Use a Test Runner?**

A **Cucumber Test Runner** (TestRunner.java) is used to:  
✅ Run all **feature files** automatically  
✅ Define configurations like **reporting**, **glue path**, and **tags**  
✅ Avoid running feature files manually one by one

**Can It Be Executed Without a Test Runner?**

Yes, you can execute **Cucumber tests without a Test Runner**, but it requires an alternative approach.

**1️. Running Feature Files Directly in IntelliJ or Eclipse**

Most IDEs allow running .feature files directly without a Test Runner:

* **Right-click** the .feature file
* Click **Run ‘FeatureName’**
* The test will execute using the **default Cucumber engine**

⚠️ **Limitation:**

* You can only run **one feature file at a time**
* No custom **report generation** or **filtering**

**So, Do You Really Need a Test Runner?**

✅ **YES** → If you want to run multiple feature files, generate reports, and filter scenarios  
✅ **NO** → If you just want to run a single .feature file manually

Would you like to see a **Maven-based setup without a Test Runner**? 🚀

**Cucumber Test Runner Class**

A **Runner Class** in Cucumber is used to execute **feature files** and define configurations like:  
✅ Location of **feature files**  
✅ Location of **step definitions**  
✅ **Reports & logs**  
✅ Filtering tests using **tags**

**📌 Example: Basic Cucumber Test Runner**

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features", // Path to feature files

glue = "stepDefinitions", // Path to step definitions

plugin = {"pretty", "html:target/cucumber-reports.html"}, // Report generation

monochrome = true // Cleaner console output

)

public class TestRunner {

**📌 CucumberOptions Breakdown**

| **Option** | **Description** |
| --- | --- |
| features | Path to .feature files |
| glue | Package where step definitions are located |
| plugin | Enables reporting (HTML, JSON, XML) |
| monochrome | Improves console output readability |

**Tidy Gherkin Plugin in Cucumber**

**Tidy Gherkin** is a Chrome extension that helps:  
✅ **Write and format Gherkin syntax** properly  
✅ **Generate step definitions** for Java, JavaScript, and Ruby  
✅ **Auto-suggest step definitions**

**📌 How to Install Tidy Gherkin?**

**Open Google Chrome**  
Go to **Chrome Web Store**  
Search for **Tidy Gherkin**  
Click **Add to Chrome**

**📌 Example: Running All Feature Files**

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features", // Runs all feature files in this folder

glue = "stepDefinitions", // Path to step definitions

plugin = {"pretty", "html:target/cucumber-reports.html"}, // Report generation

monochrome = true // Cleaner console output )

public class TestRunner {

}

✅ This will run **all .feature files** inside src/test/resources/features/.  
✅ You don't need to list them one by one—Cucumber will **automatically detect and execute all feature files**.

**What is glue in Cucumber?**

The glue option in @CucumberOptions specifies the **package** where Cucumber should look for **step definitions**.

**📌 Example: Using glue in Test Runner**

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features", // Location of feature files

glue = "stepDefinitions", // Package for step definitions

plugin = {"pretty", "html:target/cucumber-reports.html"},

monochrome = true

)

public class TestRunner {

}

📌 **Here, glue = "stepDefinitions" means:**  
➡️ Cucumber will search for **step definitions** inside the stepDefinitions package.

**📌 What Happens If glue is Incorrect?**

If the **package name** in glue does not match your actual **step definition package**, you will get an error like:

**Undefined step: Given User is on the login page**

✅ **Solution:** Make sure the glue path is correct and matches the package name of your step definitions.

**📌 Can glue Have Multiple Packages?**

Yes! You can specify **multiple glue paths**

**Using Background in Gherkin (Cucumber)**

The Background keyword in Gherkin allows you to **define common preconditions** that apply to all scenarios in a feature file.

✅ **Removes repetition** in Given steps  
✅ **Executes before each scenario** in the feature file

**📌 Example: Without Background (Repetitive Steps)**

Feature: Login Functionality

Scenario: Successful Login

Given User is on the login page

When User enters valid credentials

Then User should see the homepage

Scenario: Unsuccessful Login

Given User is on the login page

When User enters incorrect credentials

Then User should see an error message

📌 **Problem:** "User is on the login page" is repeated in every scenario.

**📌 Example: With Background (Cleaner Code)**

Feature: Login Functionality

Background:

Given User is on the login page

Scenario: Successful Login

When User enters valid credentials

Then User should see the homepage

Scenario: Unsuccessful Login

When User enters incorrect credentials

Then User should see an error message

✅ "User is on the login page" runs **before every scenario**  
✅ Avoids **code duplication**

**Scenario Outline & Examples in Gherkin (Cucumber)**

✅ Scenario Outline is used for **data-driven testing**  
✅ Examples provide multiple sets of test data  
✅ Avoids writing **separate scenarios** for different inputs

**📌 Example: Without Scenario Outline (Repetitive Code)**

Feature: Login Functionality

Scenario: Login with valid user1

Given User is on the login page

When User enters "user1" and "password1"

Then User should see the homepage

Scenario: Login with valid user2

Given User is on the login page

When User enters "user2" and "password2"

Then User should see the homepage

📌 **Problem:** The same steps are repeated with different data.

**📌 Example: With Scenario Outline (Optimized Code)**

Feature: Login Functionality

Scenario Outline: Successful Login

Given User is on the login page

When User enters "<username>" and "<password>"

Then User should see the homepage

Examples:

| username | password |

| user1 | password1 |

| user2 | password2 |

✅ Runs **twice**, once for each row in Examples  
✅ **"<username>" and "<password>"** are placeholders

**Cucumber Expressions in Java (Alternative to Regex in Step Definitions)**

Cucumber Expressions provide a **simpler way** to define step definitions compared to **regular expressions**.

✅ **More readable and user-friendly** than Regex  
✅ Supports **parameter types** (e.g., {int}, {string})  
✅ Works with **optional words** and **alternatives**

**📌 Example: Using Cucumber Expressions Instead of Regex**

**✅ Using Cucumber Expressions (Simpler)**

@Given("User enters {int} in the input field")

public void enterNumber(int number) {

System.out.println("Entered number: " + number);

}

🟢 **Uses {int}** instead of regex → Easier to understand

**📌 Supported Parameter Types**

| **Cucumber Expression** | **Data Type** | **Example Match** |
| --- | --- | --- |
| {int} | Integer | 42, 1000 |
| {float} | Float/Double | 3.14, 99.99 |
| {string} | String (with quotes) | "Hello", "Cucumber" |
| {word} | Single Word | Login, SignUp |
| {boolean} | Boolean | true, false |

**Can We Use Regular Expressions and Cucumber Expressions Together in Cucumber?**

✅ **Yes**, Cucumber allows you to use **both Regular Expressions and Cucumber Expressions** in the same step definition file.  
You can mix and match them based on your needs.

**Can We Use Both Regular Expressions and Cucumber Expressions for the Same Step?**

✅ **No, you cannot define the same step using both Regular Expressions and Cucumber Expressions in a single step definition method.**  
However, you can **define two separate step definitions**—one using **Cucumber Expressions** and another using **Regular Expressions**—to match the same feature file step.

**Using pretty in Cucumber options of Runner class**

The pretty option in Cucumber **formats the console output** for better readability. It **prints feature files and step definitions in a structured way**, making debugging easier.

**📌 Example: Runner Class with pretty**

**✅ TestRunner.java**

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features", // Path to feature files

glue = "stepDefinitions", // Path to step definitions

plugin = {"pretty"} // Enables pretty formatting

)

public class TestRunner {

}

**📌 What Does pretty Do?**

It improves console output **by adding indentation and formatting**, making it **easier to read**.

**DuplicateStepDefinitionException in Cucumber**

The DuplicateStepDefinitionException occurs when **multiple step definitions match the same Gherkin step** in a feature file. This confuses Cucumber about which method to execute.

**📌 Final Verdict**

| **Solution** | **Works?** | **When to Use?** |
| --- | --- | --- |
| **Remove duplicate step definition** | ✅ Yes | When the duplicate step is unnecessary |
| **Use different step texts** | ✅ Yes | When multiple steps with similar meanings are needed |
| **Use Regular Expressions** | ✅ Yes | When more flexibility is needed |

**AmbiguousStepDefinitionsException**

The AmbiguousStepDefinitionsException occurs when **multiple step definitions match the same Gherkin step**, and Cucumber **cannot determine which one to execute**.

**📌 Final Verdict**

| **Solution** | **Works?** | **When to Use?** |
| --- | --- | --- |
| **Remove the duplicate step definition** | ✅ Yes | If only one step is needed |
| **Use different step texts** | ✅ Yes | When both step definitions are required |
| **Use different parameter types** | ✅ Yes | When handling similar but different inputs |

**Difference Between DuplicateStepDefinitionException and AmbiguousStepDefinitionsException in Cucumber**

| **Exception** | **Cause** | **Example** | **Solution** |
| --- | --- | --- | --- |
| **DuplicateStepDefinitionException** | **Same step definition is implemented twice** in the same project. | Two methods have **exactly the same** step definition. | Remove the duplicate or merge both into one. |
| **AmbiguousStepDefinitionsException** | **Two or more step definitions match the same step** in the feature file. | One uses {string}, and another uses (.\*), both matching the same Gherkin step. | Modify one step to be more specific or unique. |

**📌 Quick Summary**

| **Exception** | **Happens When?** | **Fix** |
| --- | --- | --- |
| DuplicateStepDefinitionException | **Two identical step definitions** exist in the code. | Remove the duplicate. |
| AmbiguousStepDefinitionsException | **Two different step definitions match the same feature step.** | Make one more specific. |

**Comments in a Cucumber Feature File**

Cucumber allows you to **add comments** in feature files to improve readability and document your test cases.

**📌 How to Write Comments in a Feature File?**

* Use # at the **beginning** of a line.
* Cucumber **ignores** commented lines when executing tests.

**Tags in Cucumber**

Tags in Cucumber help you **organize and filter** test scenarios for execution. They are prefixed with @ and placed **above a Feature, Scenario, or Scenario Outline**.

**📌 How to Use Tags?**

* Tags are **custom labels** that allow you to run selected scenarios.
* You can use **single or multiple tags** to categorize tests.

**✅ Running Multiple Tags**

| **Tag Expression** | **Runs Scenarios That...** |
| --- | --- |
| tags = "@Smoke" | Have @Smoke tag |
| tags = "@Regression and @Smoke" | Have **both** @Regression and @Smoke tags |
| tags = "@Regression or @Smoke" | Have **either** @Regression or @Smoke |
| tags = "not @WIP" | Exclude scenarios with @WIP |

**📌 Why Use Tags?**

✅ Run specific tests based on priority  
✅ Organize test cases (Smoke, Regression, etc.)  
✅ Exclude tests that are not ready

📌 Example: Using Multiple Tags

@CucumberOptions(

features = "src/test/resources/features",

glue = "stepDefinitions",

tags = "@Regression **and** @Login" // Runs scenarios with both @Regression and @Login tags

)

**Combining not, and, or**

| **Expression (Tags)** | **Effect** |
| --- | --- |
| @Smoke or @Regression | Runs scenarios with **either** @Smoke or @Regression. |
| @Smoke and @Regression | Runs scenarios with **both** @Smoke and @Regression. |
| not @WIP | **Excludes** scenarios with @WIP. |
| @Regression and not @WIP | Runs scenarios with @Regression **but excludes** @WIP. |
| (@Smoke or @Regression) and not @WIP | Runs scenarios that have **either** @Smoke or @Regression, but **excludes** @WIP. |

**Hooks in Cucumber**

Cucumber **Hooks** are special blocks of code that run **before or after** scenarios. They are mainly used for **setup and teardown** tasks.

**📌 Types of Hooks:**

| **Hook** | **Runs When?** | **Purpose** |
| --- | --- | --- |
| @Before | Before each scenario | Setup tasks (e.g., open browser, initialize test data) |
| @After | After each scenario | Cleanup tasks (e.g., close browser, clear cookies) |
| @BeforeStep | Before each step | Actions before every step (e.g., logging) |
| @AfterStep | After each step | Actions after every step (e.g., screenshot on failure) |

**@BeforeStep and @AfterStep Hooks**

import io.cucumber.java.BeforeStep;

import io.cucumber.java.AfterStep;

public class Hooks {

@BeforeStep

public void beforeEachStep() {

System.out.println("🔹 Before each step...");

}

@AfterStep

public void afterEachStep() {

System.out.println("🔸 After each step...");

}

**Running Hooks in a Specific Order:** You can **prioritize hooks** using @Order(value).

* **Lower value = Higher priority** (executes first).

import io.cucumber.java.Before;

import io.cucumber.java.Order;

public class Hooks {

@Before(order = 1)

public void highPrioritySetup() {

System.out.println("🔹 High-priority setup (Runs first)");

}

@Before(order = 2)

public void lowPrioritySetup() {

System.out.println("🔸 Low-priority setup (Runs second)");

}

**Tagged Hooks in Cucumber:** Tagged hooks in Cucumber allow you to **execute hooks only for specific scenarios** based on tags.

**📌 How to Use Tagged Hooks?**

* Use @Before("@tagName") to run **before** specific tagged scenarios.
* Use @After("@tagName") to run **after** specific tagged scenarios.
* **Multiple tags** can be used for conditional execution.

**✅ Example: Running Hooks for a Specific Tag:**

import io.cucumber.java.Before;

import io.cucumber.java.After;

public class Hooks {

@Before("@Smoke")

public void beforeSmokeTests() {

System.out.println("🔥 Setup for @Smoke tests...");

}

@After("@Smoke")

public void afterSmokeTests() {

System.out.println("🧹 Cleanup after @Smoke tests...");

}

In Cucumber, the **value attribute** is used in **tagged hooks** to specify which scenarios the hook should apply to.

**order with value (Priority Control)**

* Lower order value **runs first**.
* Higher order value **runs later**.

import io.cucumber.java.Order;

@Before(value = "@Smoke", order = 1)

public void firstSetup() {

System.out.println("🚀 High-priority setup for @Smoke...");

}

@Before(value = "@Smoke", order = 2)

public void secondSetup() {

System.out.println("⏳ Low-priority setup for @Smoke...");

}

**Data Tables in Cucumber**

Cucumber **Data Tables** allow you to pass structured test data into step definitions. They are useful when testing multiple inputs or complex scenarios.

**Using Data Table as List<Map<String, String>>**

For **key-value pairs**, use a List<Map<String, String>>.

**✅ Example: Feature File**

Scenario: User login details

Given the following login credentials:

| Username | Password |

| admin | admin123 |

| user1 | pass1 |

**✅ Step Definition (Java)**

import io.cucumber.datatable.DataTable;

import java.util.List;

import java.util.Map;

@Given("the following login credentials:")

public void theFollowingLoginCredentials(DataTable dataTable) {

List<Map<String, String>> userList = dataTable.asMaps(String.class, String.class);

for (Map<String, String> user : userList) {

System.out.println("Username: " + user.get("Username") + ", Password: " + user.get("Password"));

}

}

**📌 Running a Specific Feature File in Cucumber**

You can run a specific **feature file** using the @CucumberOptions annotation in your **Runner class**.

**1. Specify the Feature File Path in @CucumberOptions**

Modify the features attribute in @CucumberOptions to point to the specific file.

2. **Running a Specific Feature File Using Tags**

If you don’t want to modify the Runner class, **use tags** in your feature file.

**📌 Running Cucumber Feature Files Using Maven**

**Run All Feature Files:** mvn test

**Run a Specific Feature File:** mvn test -Dcucumber.features=src/test/java/features/login.feature

**Run Scenarios with a Specific Tag:** mvn test -Dcucumber.options="--tags @Login"

**📌 Overriding Tags in Runner Class Using Maven Command Line**

If you have **tags** defined in your **Cucumber Runner class**, but you want to override them from the **Maven command line**, you can use the -Dcucumber.options parameter.

You can **override the @Smoke tag** and run a different tag from the command line.

**📍 Example: Run Tests Tagged @Regression**

mvn test -Dcucumber.filter.tags="@Regression"

**Default tags in Cucumber**

Cucumber has some **default tags** that help control test execution. These include:

1. **@smoke** – Used to mark smoke tests, which are quick tests to check core functionality.
2. **@regression** – Used for regression tests, which check if recent changes have affected existing features.
3. **@wip (Work in Progress)** – Skips tests marked as WIP; useful for incomplete scenarios.
4. **@ignore** – Prevents a scenario from being executed.
5. **@only** – Runs only the tagged scenarios (depends on test execution framework).
6. **@fast / @slow** – Categorizes tests based on execution speed.

These tags can be used in a cucumber.yml file

**Retrieving Scenario names into hooks**

In Cucumber, you can retrieve the Scenario name inside Hooks (@Before or @After) using the Scenario object

**Example: Retrieving Scenario Name in Hooks**: Track which scenario is running, useful for debugging failures.

@Before

public void beforeScenario(Scenario scenario) {

System.out.println("Starting Scenario: " + scenario.getName());

**Generating Cucumber XML & JSON Reports in Maven Project**

Add Cucumber Report Plugins in Runner.java

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features",

glue = "stepDefinitions",

plugin = {

"pretty", // Console output

**"json:target/cucumber-reports/Cucumber.json", // JSON Report**

**"junit:target/cucumber-reports/Cucumber.xml" // XML Report**

},

monochrome = true

)

public class TestRunner {

}

**monochrome = true in Cucumber**

The monochrome option in **CucumberOptions** controls the formatting of the console output.

**✅ When monochrome = true (Recommended)**

* Removes unnecessary special characters and ANSI escape sequences.
* Provides a **cleaner** and **more readable** console output.

**ClassNotFoundException in Cucumber Projects – Causes & Solutions**

The ClassNotFoundException in **Cucumber** projects usually happens due to:

1. **Missing dependencies**
2. **Incorrect package structure**
3. **Runner class misconfiguration**
4. **Issues with step definitions or glue path**

**✅ dryRun in Cucumber:** In Cucumber, a dry run is a way to check that all steps in a feature file have corresponding step definitions. It's a part of the Test Runner Class file. The value of dry run can be either true or false.

XPath (XML Path Language) is used to navigate and locate elements in XML/HTML documents. The basic syntax structure of XPath is:

//tagname[@attribute='value']

**XPath Structure Breakdown:**

1. // → Selects elements from anywhere in the document.
2. tagname → The HTML/XML tag to locate (e.g., div, input, button). Use \* to select any tag.
3. @attribute='value' → Filters elements based on attributes (e.g., @id='login').

The contains() function in XPath is used to find elements where an attribute or text contains a specific substring.

//tagname[contains(@attribute, 'value')]

**Examples:**

1. **Using contains() on href:**

//a[contains(@href, 'mail.google.com')]

Yes! XPath provides the starts-with() function to find elements where an attribute value begins with a specific substring.

//tagname[starts-with(@attribute, 'value')]

**Examples:**

1. **Using starts-with() on href:**

//a[starts-with(@href, 'https://mail.google.com')]

**Page Object Model (POM) in Selenium**

**Page Object Model (POM)** is a design pattern used in Selenium automation to enhance code maintainability and reusability by separating UI elements from test logic. It creates a **separate Java class** for each web page and defines **locators** and **methods** for interacting with elements on that page.

**Why Use POM?**✅ **Enhances Maintainability** – Changes in UI affect only one place (the page class).  
✅ **Improves Readability** – Test scripts are cleaner and more structured.  
✅ **Reduces Code Duplication** – Reusable methods for UI interactions.

**Running Parallel Tests in TestNG**

TestNG allows running tests in parallel using the **testng.xml** configuration file. You can run multiple tests simultaneously to **reduce execution time**.

**1. Parallel Execution at Suite Level**

Runs multiple test classes in parallel.

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="ParallelSuite" parallel="classes" thread-count="2">

<test name="Test1">

<classes>

<class name="tests.LoginTest"/>

<class name="tests.HomeTest"/>

</classes>

</test>

</suite>

🔹 parallel="classes" → Runs multiple classes in parallel.  
🔹 thread-count="2" → Runs 2 threads at a time.