**Selenium Tasks**

**Task1:**Eclipse IDE Basic Operations

Topic: Navigating the Eclipse Interface

Description: Demonstrate the basic operations in Eclipse by running a pre-written Java program, observing the console output, and then stopping the execution.

Steps:

Open Eclipse IDE and select the workspace that contains the pre-written Java program. Locate the Java program in the Package Explorer and open it in the editor. Click on the 'Run' button to start the program and observe the output in the Console window. Use the 'Stop' button to terminate the program execution. Describe what the 'Pause' button does in the context of running programs.

**Step 1:** **Open Eclipse IDE and select the workspace:**

* Launch Eclipse IDE on your computer.
* We will be prompted to select a workspace.
* Choose the workspace that contains the pre-written Java program you want to run.
* Click "OK" to proceed.

**Step 2: Locate the Java program in the Package Explorer :**

* In the Eclipse interface, we'll see the Package Explorer view on the left side.
* This view displays the hierarchy of projects, packages, and files in our workspace.
* Locate the Java program we want to run and double-click on it to open it in the editor.
* The editor will display the Java code.(eg: program.java)

**Step 3: Run our program for execution:**

* To run the Java program, click on the "Run" button in the toolbar.
* Eclipse will compile and execute the program.
* The output of the program will be displayed in the Console window, which is usually located at the bottom of the Eclipse interface.

**Step 4: Use the 'Stop' button to terminate the program execution:**

* Once we've observed the output of the program, we can stop the execution by clicking on the "Stop" button in the toolbar.
* This will terminate the program execution.

**Step5: 'Pause' the Program:**

* In the context of running programs, the "Pause" button allows you to temporarily suspend the execution of a program.
* When we click the "Pause" button, the program will stop executing, but it will not terminate.
* We can then inspect the program's state, examine variables, and set breakpoints. To resume the program execution, click the "Resume" button.
* The "Pause" button is only enabled when a program is running and is not applicable to this specific example, as our program is a simple Java program that terminates after execution.

**Task2:** Debugging with Eclipse

Topic: Error Handling and Debugging

Description: Utilize Eclipse's debugging features to identify and fix a simple logical error in a Java program.

Steps:

Open the provided Java program with a known logical error in Eclipse. Start the debugger by clicking on the 'Debug' button. Use breakpoints to pause the execution at critical points in the program. Step through the code using 'Step into' and 'Step over' to observe variable values. Identify the logical error based on variable values and code flow. Correct the error and rerun the program to ensure it's fixed.

**Step 1: Open the provided Java program with logical error in Eclipse:**

* Open Eclipse IDE and select the workspace that contains the Java program with a known logical error.
* We can either create a new project or import an existing one.
* Once we've opened the project, navigate to the Package Explorer view and open the Java file that contains the error.

Before you start Debug you need to fix or choose the breakpoints for critical points for better understanding and to know the execution process.

**Step 2: Start the debugger by clicking on the 'Debug' button:**

* To start the debugger, click on the "Debug" button in the toolbar.
* Alternatively, you can right-click on the Java file in the Package Explorer view and select "Debug As" > "Java Application".
* Eclipse will compile the program and start the debugger.

**Step 3: Set breakpoints to critical points in the program:**

* Breakpoints are points in the program where you want the execution to pause.
* To set a breakpoint, double-click in the left margin of the editor next to the line of code where you want to pause the execution.
* When we enter the breakpoint it will show the blue dot will appear, indicating that a breakpoint has been set.
* We can choose the breakpoints before a loop or before a conditional statement.

**Step 4: Step through the code using variable values:**

* When the program reaches a breakpoint, you can step through the code using the "Step into" and "Step over" buttons.
* There is a button in Eclipse we need to find Step into and Step over we can use it.
* **"Step into"** executes the current line of code and steps into any method calls.
* **"Step over"** executes the current line of code and steps over any method calls.
* Use these buttons to step through the code and observe the values of variables.
* We can use the "Variables" view to inspect the values of variables.

**Step 5: Identify the logical error:**

* As we step through the code, observe the values of variables and the code flow.
* Look for any unexpected values or behavior that may indicate a logical error.
* let's say we've identified that the program is not producing the expected output due to a mistake in a conditional statement.

**Step 6: Correct the error and rerun the program:**

* Once we've identified the logical error, correct it by modifying the code.
* Make the necessary changes and save the file. Rerun the program by clicking on the "Debug" button again.
* Eclipse will recompile the program and start the debugger.
* We've successfully utilized Eclipse's debugging features to identify and fix a simple logical error in a Java program.

**Example Debugging Process:** Consider a simple program with a logical error in a method that calculates the sum of an array of integers. Here's a sample code:

**Java Code:**

package testNGDemo;

import org.testng.annotations.Test;

import org.testng.annotations.Parameters;

public class parameetersDemo {

int a=10,b=20,result,sum;

public void sum() {

result=a+b; System.out.println(result);

}

@Test(enabled=false)

public void addTC()

{

add(40,50);

}

public void add(int num1,int num2)

{

sum=num1+num2; System.out.println(sum);

}

@Test @Parameters({"val1","val2"})

public void subtract(int a, int b)

{

int subtra=a-b; System.out.println(subtra);

}

}

**Task 3:** Writing and Running a Simple Selenium Test

Topic: First Test Case and WebDriver Basics

Description: Write a simple Selenium WebDriver test in Eclipse to open a web browser and navigate to a specified URL.

Steps:

Open Eclipse and create a new Java Project dedicated to Selenium tests.

Within the project, set up the Selenium WebDriver by adding the WebDriver JAR files to the build path.

Create a new Java class file for the test case.

Write a Java method using WebDriver to initiate a Firefox, Chrome, or Safari browser session.

Use WebDriver to navigate to 'http://example.com' or a similar simple web page.

Run the test to ensure the browser opens and navigates to the URL successfully.

**Step 1: Create a new Java Project dedicated to Selenium tests :**

* Launch Eclipse IDE and create a new Java project by going to "File" > "New" > "Java Project". Name the project, for example, "SeleniumTests".
* Click "Finish" to create the project.

**Step 2: Set up the Selenium WebDriver by adding the WebDriver JAR files to the build path:**

* To use Selenium WebDriver, you need to add the necessary JAR files to the project's build path.
* We can download the Selenium WebDriver JAR files from the official Selenium website.
* We have the JAR files, follow these steps to add them to the build path: Right-click on the project in the Package Explorer view and select "Properties".
* In the "Properties" window, select "Java Build Path" from the left menu.
* Click on the "Libraries" tab and then click on "Add External JARs".
* Select the Selenium WebDriver JAR files you downloaded and click "Open".
* Click "Apply and Close" to save the changes.

**Step 3: Create a new Java class file for the test case:**

* Create a new Java class file for the test case by right-clicking on the project in the Package Explorer view and selecting "New" > "Class".
* Name the class, for example, "SimpleSeleniumTest".
* Click "Finish" to create the class.

package Selinium;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.edge.EdgeDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class BrowserInvoke {

public static WebDriver invokeBrowser(String browserName)

{

WebDriver driver = null;

switch (browserName.toLowerCase())

{

case "chrome":

// Set the path to the ChromeDriver if necessary // System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

driver = new ChromeDriver();

break;

case "firefox":

// Set the path to the GeckoDriver if necessary // System.setProperty("webdriver.gecko.driver", "path/to/geckodriver");

driver = new FirefoxDriver();

break; case "edge":

// Set the path to the EdgeDriver if necessary // System.setProperty("webdriver.edge.driver", "path/to/msedgedriver");

driver = new EdgeDriver();

break;

default:

System.out.println("Unsupported browser: " + browserName);

break;

}

if (driver != null)

{

driver.manage().window().maximize();

}

return driver;

}

public static void main(String[] args) {

// Example usage

WebDriver driver = invokeBrowser("edge");

if (driver != null) {

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

System.out.println("Title: " + driver.getTitle()); driver.quit();

}

}

}