Selenium

* **Selenium Tools:**

1. **Selenium IDE**

Selenium **I**ntegrated **D**evelopment **E**nvironment (IDE) is a Firefox plugin that lets testers to record their actions as they follow the workflow that they need to test.

1. **Selenium RC**

Selenium Remote Control (RC) was the flagship testing framework that allowed more than simple browser actions and linear execution. It makes use of the full power of programming languages such as Java, C#, PHP, Python, Ruby and PERL to create more complex tests.

1. **Selenium WebDriver**

Selenium WebDriver is the successor to Selenium RC which sends commands directly to the browser and retrieves results.

1. **Selenium Grid**

Selenium Grid is a tool used to run parallel tests across different machines and different browsers simultaneously which results in minimized execution time.

* **Advantages of Selenium:**

|  |  |
| --- | --- |
| **Selenium** | **QTP** |
| Selenium is an open-source tool. | QTP is a commercial tool and there is a cost involved in each one of the licenses. |
| Can be extended for various technologies that expose DOM. | Limited add-ons and needs add-ons for each one of the technologies. |
| Has capabilities to execute scripts across different browsers. | Can run tests in specific versions of Firefox , IE, and Chrome. |
| Can execute scripts on various operating systems. | Works only with Windows. |
| Supports mobile devices. | Supports mobile devices with the help of third-party tools. |
| Executes tests within the browser, so focus is NOT required while script execution is in progress. | Needs Focus during script execution, as the tool acts on the browser (mimics user actions). |
| Can execute tests in parallel with the use of Selenium Grids. | QTP cannot execute tests in parallel, however integrating QTP with QC allows testers to execute in parallel. QC is also a commercial tool. |

* **Disadvantages of Selenium:**

|  |  |
| --- | --- |
| **Selenium** | **QTP** |
| Supports only web based applications. | Can test both web and desktop applications. |
| No feature such as Object Repository/Recovery Scenario | QTP has built-in object repositories and recovery scenarios. |
| No IDE, so the script development won't be as fast as QTP. | More intuitive IDE; automation can be achieved faster. |
| Cannot access controls within the browser. | Can access controls within the browser such as favorites bar, backward, and forward buttons. |
| No default test report generation. | Default test result generation within the tool. |
| For parameterization, users has to rely on the programming language. | Parameterization is built-in and easy to implement. |

* **Sample Code:**

package selrcdemo;

import com.thoughtworks.selenium.DefaultSelenium;

import com.thoughtworks.selenium.Selenium;

public class rcdemo {

public static void main(String[] args) throws InterruptedException {

// Instatiate the RC Server

Selenium selenium = new DefaultSelenium("localhost", 4444 , "firefox", "http://www.calculator.net");

selenium.start(); // Start

selenium.open("/"); // Open the URL

selenium.windowMaximize();

// Click on Link Math Calculator

selenium.click("xpath = .//\*[@id = 'menu']/div[3]/a");

Thread.sleep(2500); // Wait for page load

// Click on Link Percent Calculator

selenium.click("xpath = .//\*[@id = 'menu']/div[4]/div[3]/a");

Thread.sleep(4000); // Wait for page load

// Focus on text Box

selenium.focus("name = cpar1");

// enter a value in Text box 1

selenium.type("css=input[name = \"cpar1\"]", "10");

// enter a value in Text box 2

selenium.focus("name = cpar2");

selenium.type("css = input[name = \"cpar2\"]", "50");

// Click Calculate button

selenium.click("xpath = .//\*[@id = 'content']/table/tbody/tr/td[2]/input");

// verify if the result is 5

String result = selenium.getText(".//\*[@id = 'content']/p[2]");

if (result == "5") {

System.out.println("Pass");

} else {

System.out.println("Fail");

}

}

}

* **Selenese Commands:**

Three types of commands:

1. Actions: Actions are commands that manipulate the state of the application.
2. Accessors: Accessors evaluate the state of the application and store the results in a variable which are used in assertions.

* assertErrorOnNext (message):

Pings Selenium to expect an error on the next command execution with an expected message.

* storeAllButtons (variableName):

Returns the IDs of all buttons on the page.

* storeAllFields (variableName):

Returns the IDs of all input fields on the page.

* storeAllLinks (variableName):

Returns the IDs of all links on the page.

* storeAllWindowIds (variableName):

Returns the IDs of all windows that the browser knows about in an array.

* storeAllWindowTitles (variableName):

Returns the names of all windows that the browser knows about in an array.

* storeAllWindowNames (variableName):

Returns the titles of all windows that the browser knows about in an array.

* storeAttribute (attributeLocator, variableName):

Gets the value of an element attribute. The value of the attribute may differ across browsers.

* storeBodyText (variableName):

Gets the entire text of the page.

* storeConfirmation (variableName):

Retrieves the message of a JavaScript confirmation dialog generated during the previous action.

* storeElementIndex (locator, variableName):

Get the relative index of an element to its parent (starting from 0).

* storeLocation (variableName):

Gets the absolute URL of the current page.

* storeSelectedIds (selectLocator,variableName):

Gets all element IDs for selected options in the specified select or multi-select element.

* storeSelectedIndex (selectLocator, variableName):

Gets index (option number, starting at 0) for selected option in the specified select element.

* storeSelectedLabel (selectLocator, variableName):

Gets label (visible text) for selected option in the specified select element.

* storeSelectedValue (selectLocator,variableName):

Gets value (value attribute) for selected option in the specified select element.

* storeSelectOptions (selectLocator,variableName):

Gets all labels in the specified select drop-down.

* storeTable (tableCellAddress, variableName):

Gets the text from a cell of a table. The cellAddress syntax: tableLocator.row.column, where row and column start at 0.

* storeText (locator, variableName):

Gets the text of an element. This works for any element that contains text.

* storeTitle (variableName):

Gets the title of the current page.

* storeValue (locator,variableName):

Gets the (whitespace-trimmed) value of an input field.

* storeChecked (locator, variableName):

Gets whether a toggle-button (checkbox/radio) is checked.

* storeElementPresent (locator, variableName):

Verifies that the specified element is somewhere on the page.

* storeTextPresent (pattern, variableName):

Verifies that the specified text pattern appears somewhere on the rendered page shown to the user.

* storeVisible (locator, variableName):

Determines if the specified element is visible.

1. **Assertions**:

Assertions enable us to verify the state of an application and compares against the expected.

* waitForErrorOnNext (message)

Waits for error; used with the accessor assertErrorOnNext.

* verifySelected(selectLocator, optionLocator)

Verifies that the selected option of a drop-down satisfies the optionSpecifier.

* waitForSelected (selectLocator, optionLocator)

Waits for getting the option selected; used with the accessor assertSelected.

* waitForNotSelected (selectLocator, optionLocator)

Waits for not getting the option selected; used with accessor the assertSelected.

* verifyAlert (pattern)

Verifies the alert text; used the with accessor storeAlert.

* waitForAlert (pattern)

Waits for the alert; used with the accessor storeAlert.

* verifyAllButtons (pattern)

Verifies the button; used with the accessor storeAllButtons.

* waitForAllButtons (pattern)

Waits for the button to load; used with the accessor storeAllButtons.

* verifyAllLinks (pattern)

Verifies all links; used with the accessor storeAllLinks.

* waitForAllLinks (pattern)

Waits for all links; used with the accessor storeAllLinks.

* verifyAllWindowIds (pattern)

Verifies the window id; used with the accessor storeAllWindowIds.

* waitForAllWindowIds (pattern )

Waits the window id; used with the accessor storeAllWindowIds.

* verifyAttribute(attributeLocator, pattern)

Verifies an attribute of an element; used with the accessor storeAttribute.

* waitForAttribute(attributeLocator, pattern)

Waits for an attribute of an element; used with accessor storeAttribute.

* verifyBodyText(pattern)

Verifies the body text; used with the accessor storeBodyText.

* waitForBodyText(pattern)

Waits for the body text; used with the accessor storeBodyText.

* waitForConfirmation(pattern)

Waits for confirmation; used with the accessor storeConfirmationPresent.

* **Locators:**

All these locators can be identified with the help of FirePath and FireBug plugin of Mozilla.

1. **identifier = id** Select the element with the specified "id" attribute and if there is no match, select the first element whose @name attribute is id.
2. **id = id** Select the element with the specified "id" attribute.
3. **name = name**
4. **dom = javascriptExpression**
5. **xpath = xpathExpression**
6. **link = textPattern**
7. **css = cssSelectorSyntax**

# Selenium – Webdriver:

WebDriver is a tool for automating testing web applications. It is popularly known as Selenium 2.0. WebDriver uses a different underlying framework, while Selenium RC uses JavaScript Selenium-Core embedded within the browser which has got some limitations. WebDriver interacts directly with the browser without any intermediary, unlike Selenium RC that depends on a server.

* Selenium RC Vs WebDriver

|  |  |
| --- | --- |
| **Selenium RC** | **Webdriver** |
| The architecture of Selenium RC is complicated, as the server needs to be up and running before starting a test. | WebDriver's architecture is simpler than Selenium RC, as it controls the browser from the OS level. |
| Selenium server acts as a middleman between the browser and Selenese commands. | WebDriver interacts directly with the browser and uses the browser's engine to control it. |
| Selenium RC script execution is slower, since it uses a Javascript to interact with RC. | WebDriver is faster, as it interacts directly with the browser. |
| Selenium RC cannot support headless execution as it needs a real browser to work with. | WebDriver can support the headless execution. |
| It's a simple and small API. | Complex and a bit large API as compared to RC. |
| Less object-oriented API. | Purely object oriented API. |
| Cannot test mobile Applications. | Can test iPhone/Android applications. |

* Most frequently used **commands** in WebDriver along with their syntax.

1. driver.get("URL") : To navigate to an application.
2. element.sendKeys("inputtext") : Enter some text into an input box.
3. element.clear() : Clear the contents from the input box.
4. select.deselectAll() : Deselect all OPTIONs from the first SELECT on the page.
5. select.selectByVisibleText("some text") : Select the OPTION with the input specified by the user.
6. driver.switchTo().window("windowName") : Move the focus from one window to another.
7. driver.switchTo().frame("frameName") : Swing from frame to frame.
8. driver.switchTo().alert() : Helps in handling alerts.
9. driver.navigate().to("URL") : Navigate to the URL.
10. driver.navigate().forward() : To navigate forward.
11. driver.navigate().back() : To navigate back.
12. driver.close() : Closes the current browser associated with the driver.
13. driver.quit() : Quits the driver and closes all the associated window of that driver.
14. driver.refresh() : Refreshes the current page.

* **Sample Code**

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

public class webdriverdemo {

public static void main(String[] args) {

WebDriver driver = new FirefoxDriver();

//Puts an Implicit wait, Will wait for 10 seconds before throwing exception

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

//Launch website

driver.navigate().to("http://www.calculator.net/");

//Maximize the browser

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

// Click on Math Calculators

driver.findElement(By.xpath(".//\*[@id = 'menu']/div[3]/a")).click();

// Click on Percent Calculators

driver.findElement(By.xpath(".//\*[@id ='menu']/div[4]/div[3]/a")).click();

// Enter value 10 in the first number of the percent Calculator

driver.findElement(By.id("cpar1")).sendKeys("10");

// Enter value 50 in the second number of the percent Calculator

driver.findElement(By.id("cpar2")).sendKeys("50");

// Click Calculate Button

driver.findElement(By.xpath(".//\*[@id = 'content']/table/tbody/tr[2]/td/input[2]")).click();

// Get the Result Text based on its xpath

String result = driver.findElement(By.xpath(".//\*[@id = 'content']/p[2]/font/b")).getText();

// Print a Log In message to the screen

System.out.println(" The Result is " + result);

//Close the Browser.

driver.close();

}

}

* **Selenium – Locators**
* findElement() returns a WebElement object based on a specified search criteria or ends up throwing an exception if it does not find any element matching the search criteria.
* findElements() returns a list of WebElements matching the search criteria. If no elements are found, it returns an empty list.
* **Interact with the application using some basic actions :**
* [Text Box Interaction](https://www.tutorialspoint.com/selenium/selenium_textbox.htm)

// Enter value 10 in the first number of the percent Calculator

driver.findElement(By.id("cpar1")).sendKeys("10");

Thread.sleep(5000);

// Get the text box from the application

String result = driver.findElement(By.id("cpar1")).getAttribute("value");

* [Radio Button Selection](https://www.tutorialspoint.com/selenium/selenium_radio_button.htm)

// Click on Radio Button

driver.findElement(By.id("cpayoff1")).click();

* [Check Box Selection](https://www.tutorialspoint.com/selenium/selenium_check_box.htm)

//Click on check Box

driver.findElement(By.id("caddoptional")).click();

* [Drop Down Item Selection](https://www.tutorialspoint.com/selenium/selenium_drop_down.htm)

//Selecting an item from Drop Down list Box

Select dropdown = new Select(driver.findElement(By.id("ccompound")));

dropdown.selectByVisibleText("continuously");

//you can also use dropdown.selectByIndex(1) to select second element as

//index starts with 0.

//You can also use dropdown.selectByValue("annually");

System.out.println("The Output of the IsSelected " +

driver.findElement(By.id("ccompound")).isSelected());

System.out.println("The Output of the IsEnabled " +

driver.findElement(By.id("ccompound")).isEnabled());

System.out.println("The Output of the IsDisplayed " +

driver.findElement(By.id("ccompound")).isDisplayed());

* [Synchronization](https://www.tutorialspoint.com/selenium/selenium_synchronization.htm)
* Thread.Sleep(1000); //Will wait for 1 second.

//Explicit wait : An 'explicit wait,' waits for a certain condition to occur before proceeding further. It is mainly used when we want to click or act on an object once it is visible.

WebDriver driver = new FirefoxDriver();

driver.get("Enter an URL"S);

WebElement DynamicElement =

(new WebDriverWait(driver, 10)).until(ExpectedConditions.presenceOfElementLocated(By.id("DynamicElement")));

//Implicit Wait : The WebDriver will wait for a specified implicit wait time and it will not try to find the element again during the specified time period.

Once the specified time limit is crossed, the webDriver will try to search the element once again for one last time. Upon success, it proceeds with the execution; upon failure, it throws exception.

WebDriver driver = new FirefoxDriver();

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

driver.get("Enter an URL");

WebElement DynamicElement = driver.findElement(By.id("DynamicElement"));

Fluent Wait

A FluentWait instance defines the maximum amount of time to wait for a condition to take place, as well as the frequency with which to check the existence of the object condition.

wait wait =

new FluentWait(driver).withTimeout(60, SECONDS).pollingEvery(10, SECONDS).ignoring(NoSuchElementException.class);

WebElement dynamicelement = wait.until(new Function<webdriver,webElement>() {

public WebElement apply(WebDriver driver) {

return driver.findElement(By.id("dynamicelement"));

}

});

* [Drag & Drop](https://www.tutorialspoint.com/selenium/selenium_drag_drop.htm)

WebElement From = driver.findElement(By.xpath(".//\*[@id='j3\_7']/a"));

WebElement To = driver.findElement(By.xpath(".//\*[@id='j3\_1']/a"));

Actions builder = new Actions(driver);

Action dragAndDrop = builder.clickAndHold(From).moveToElement(To).release(To).build();

dragAndDrop.perform();

driver.close();

* [Keyboard Actions](https://www.tutorialspoint.com/selenium/selenium_keyboard_actions.htm)

void sendKeys(java.lang.CharSequence keysToSend)

void pressKey(java.lang.CharSequence keyToPress)

void releaseKey(java.lang.CharSequence keyToRelease)

* [Mouse Actions](https://www.tutorialspoint.com/selenium/selenium_mouse_actions.htm)

void click(WebElement onElement)

void contextClick(WebElement onElement)

void doubleClick(WebElement onElement)

void mouseDown(WebElement onElement)

void mouseUp(WebElement onElement)

void mouseMove(WebElement toElement)

void mouseMove(WebElement toElement, long xOffset, long yOffset)

* [Multi Select](https://www.tutorialspoint.com/selenium/selenium_multi_select.htm)

// Perform Multiple Select

Actions builder = new Actions(driver);

WebElement select = driver.findElement(By.id("ContentHolder\_lbFeatures\_LBT"));

List<WebElement> options = select.findElements(By.tagName("td"));

System.out.println(options.size());

Action multipleSelect = builder.keyDown(Keys.CONTROL).click(options.get(2)).click(options.get(4)).click(options.get(6)).build();

multipleSelect.perform();

* -[Find All Links](https://www.tutorialspoint.com/selenium/selenium_find_all_links.htm)

WebDriver driver = new FirefoxDriver();

driver.navigate().to("http://www.calculator.net");

java.util.List<WebElement> links = driver.findElements(By.tagName("a"));

System.out.println("Number of Links in the Page is " + links.size());

for (int i = 1; i<=links.size(); i = i+1) {

System.out.println("Name of Link# " + i + links.get(i).getText());

}

# Selenium - Test design techniques

# Page Object Model

Advantages:

* POM is an implementation where test objects and functions are separated from each other, thereby keeping the code clean.
* The objects are kept independent of test scripts. An object can be accessed by one or more test scripts, hence POM helps us to create objects once and use them multiple times.
* Since objects are created once, it is easy to access as well as update a particular property of an object.

Step 1: Create a simple class (page\_objects\_perc\_calc.java) file within a package and create methods for each one of those object identifiers as shown below.

//Percentage Calc Link

public static webElement lnk\_percent\_calc(WebDriver driver) {

element = driver.findElement(By.xpath(".//\*[@id = 'menu']/div[4]/div[3]/a"));

return element;

}

Step 2: Create a class with main and import the package and create methods for each one of those object identifiers as shown below.

public class PercentCalculator {

private static WebDriver driver = null;

public static void main(String[] args) {

driver = new FirefoxDriver();

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

driver.get("http://www.calculator.net");

//Use page Object library now

page\_objects\_perc\_calc.lnk\_math\_calc(driver).click();

page\_objects\_perc\_calc.lnk\_percent\_calc(driver).click();

driver.close();

}

}

# Data Driven using Excel

Add required external JARs in libraries folder.

Sample code to access excel file:

import java.io.\*;

import org.apache.poi.xssf.usermodel.\*;

public class ExcelUtils {

private XSSFSheet ExcelWSheet;

private XSSFWorkbook ExcelWBook;

//Constructor to connect to the Excel with sheetname and Path

public Excelutils(String Path, String SheetName) throws Exception {

try {

// Open the Excel file

FileInputStream ExcelFile = new FileInputStream(Path);

// Access the required test data sheet

ExcelWBook = new XSSFWorkbook(ExcelFile);

ExcelWSheet = ExcelWBook.getSheet(SheetName);

} catch (Exception e) {

throw (e);

}

}

//This method is to set the rowcount of the excel.

public int excel\_get\_rows() throws Exception {

try {

return ExcelWSheet.getPhysicalNumberOfRows();

} catch (Exception e) {

throw (e);

}

}

//This method to get the data and get the value as strings.

public String getCellDataasstring(int RowNum, int ColNum) throws Exception {

try {

String CellData =

ExcelWSheet.getRow(RowNum).getCell(ColNum).getStringCellValue();

System.out.println("The value of CellData " + CellData);

return CellData;

} catch (Exception e) {

return "Errors in Getting Cell Data";

}

}

//This method to get the data and get the value as number.

public double getCellDataasnumber(int RowNum, int ColNum) throws Exception {

try {

double CellData =

ExcelWSheet.getRow(RowNum).getCell(ColNum).getNumericCellValue();

System.out.println("The value of CellData " + CellData);

return CellData;

} catch (Exception e) {

return 000.00;

}

}

}

# Log4j Logging

Log4j is an audit logging framework that gives information about what has happened during execution. It offers the following advantages −

* Enables us to understand the application run.
* Log output can be saved that can be analyzed later.
* Helps in debugging, in case of test automation failures.
* Can also be used for auditing purposes to look at the application's health.

Components :

1. Instance of Logger class.

2. Log level methods used for logging the messages as one of the following −

i. error ii. Warn iii. Info iv. Debug v. log

Use :

Add Log4j Jar files in library.

Sample Code :

package log4j\_demo;

import org.apache.log4j.LogManager;

import org.apache.log4j.Logger;

import org.apache.log4j.xml.DOMConfigurator;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

public class log4j\_demo {

static final Logger logger = LogManager.getLogger(log4j\_demo.class.getName());

public static void main(String[] args) {

DOMConfigurator.configure("log4j.xml");

logger.info("# # # # # # # # # # # # # # # # # # # # # # # # # # # ");

logger.info("TEST Has Started");

WebDriver driver = new FirefoxDriver();

//Puts a Implicit wait, Will wait for 10 seconds before throwing exception

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

//Launch website

driver.navigate().to("http://www.calculator.net/");

logger.info("Open Calc Application");

}

}

# Exception Handling

Syntax

The actual code should be placed in the try block and the action after exception should be placed in the catch block. Note that the 'finally' block executes regardless of whether the script had thrown an exception or NOT.

try {

//Perform Action

} catch(ExceptionType1 exp1) {

//Catch block 1

} catch(ExceptionType2 exp2) {

//Catch block 2

} catch(ExceptionType3 exp3) {

//Catch block 3

} finally {

//The finally block always executes.

}

* Example

If an element is not found (due to some reason), we should step out of the function smoothly. So we always need to have a try-catch block if we want to exit smoothly from a function.

public static WebElement lnk\_percent\_calc(WebDriver driver)throws Exception {

try {

element = driver.findElement(By.xpath(".//\*[@id='menu']/div[4]/div[3]/a"));

return element;

} catch (Exception e1) {

// Add a message to your Log File to capture the error

Logger.error("Link is not found.");

// Take a screenshot which will be helpful for analysis.

File screenshot = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);

FileUtils.copyFile(screenshot, new File("D:\\framework\\screenshots.jpg"));

throw(e1);

}

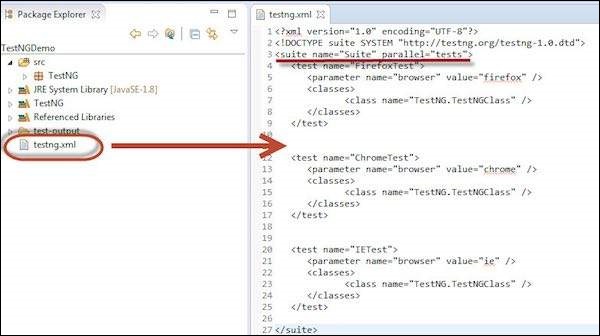
}

* Multi Browser Testing

Step1 : Prepare script.

e.g. package TestNG and class TestNGClass

Step2: Prepare xml to parameterize browser.



Step 3: Execute the script by performing right-click on the XML file and select 'Run As' >> 'TestNG' Suite

Output :

All the browser would be launched simultaneously and the result would be printed in the console.

# Capture Screenshots

Example :

import java.io.File;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

import org.apache.commons.io.FileUtils;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

public class WebdriverDemo {

public static void main(String[] args) throws IOException {

WebDriver driver = new FirefoxDriver();

// Puts an Implicit wait, Will wait for 10 seconds before throwing exception

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

// Launch website

driver.navigate().to("http://www.calculator.net/");

// Maximize the browser

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

// Click on Math Calculators

driver.findElement(By.xpath(".//\*[@id = 'menu']/div[3]/a")).click();

// Click on Percent Calculators

driver.findElement(By.xpath(".//\*[@id = 'menu']/div[4]/div[3]/a")).click();

// Enter value 10 in the first number of the percent Calculator

driver.findElement(By.id("cpar1")).sendKeys("10");

// Enter value 50 in the second number of the percent Calculator

driver.findElement(By.id("cpar2")).sendKeys("50");

// Click Calculate Button

driver.findElement(By.xpath(".//\*[@id = 'content']/table/tbody/tr/td[2]/input")).click();

// Get the Result Text based on its xpath

String result =

driver.findElement(By.xpath(".//\*[@id = 'content']/p[2]/span/font/b")).getText();

File screenshot = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);

FileUtils.copyFile(screenshot, new File("D:\\screenshots\\screenshots1.jpg"));

// Print a Log In message to the screen

System.out.println(" The Result is " + result);

//Close the Browser.

driver.close();

} }

# Capture videos

Add screen recorder JAR files into project library.

import java.io.File;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

import java.awt.\*;

import org.apache.commons.io.FileUtils;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.By;

import org.monte.media.math.Rational;

import org.monte.media.Format;

import org.monte.screenrecorder.ScreenRecorder;

import static org.monte.media.AudioFormatKeys.\*;

import static org.monte.media.VideoFormatKeys.\*;

public class webdriverdemo {

private static ScreenRecorder screenRecorder;

public static void main(String[] args) throws IOException, AWTException {

GraphicsConfiguration gconfig = GraphicsEnvironment

.getLocalGraphicsEnvironment()

.getDefaultScreenDevice()

.getDefaultConfiguration();

screenRecorder = new ScreenRecorder(gconfig,

new Format(MediaTypeKey, MediaType.FILE, MimeTypeKey, MIME\_AVI),

new Format(MediaTypeKey, MediaType.VIDEO, EncodingKey,

ENCODING\_AVI\_TECHSMITH\_SCREEN\_CAPTURE,

CompressorNameKey, ENCODING\_AVI\_TECHSMITH\_SCREEN\_CAPTURE,

DepthKey, (int)24, FrameRateKey, Rational.valueOf(15),

QualityKey, 1.0f,

KeyFrameIntervalKey, (int) (15 \* 60)),

new Format(MediaTypeKey, MediaType.VIDEO,

EncodingKey,"black", FrameRateKey, Rational.valueOf(30)), null);

WebDriver driver = new FirefoxDriver();

// Start Capturing the Video

screenRecorder.start();

// Puts an Implicit wait, Will wait for 10 seconds before throwing exception

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

// Launch website

driver.navigate().to("http://www.calculator.net/");

// Maximize the browser

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

// Click on Math Calculators

driver.findElement(By.xpath(".//\*[@id = 'menu']/div[3]/a")).click();

// Click on Percent Calculators

driver.findElement(By.xpath(".//\*[@id = 'menu']/div[4]/div[3]/a")).click();

// Enter value 10 in the first number of the percent Calculator

driver.findElement(By.id("cpar1")).sendKeys("10");

// Enter value 50 in the second number of the percent Calculator

driver.findElement(By.id("cpar2")).sendKeys("50");

// Click Calculate Button

driver.findElement(By.xpath(".//\*[@id = 'content']/table/tbody/tr/td[2]/input")).click();

// Get the Result Text based on its xpath

String result =

driver.findElement(By.xpath(".//\*[@id = 'content']/p[2]/span/font/b")).getText();

File screenshot = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);

FileUtils.copyFile(screenshot, new File("D:\\screenshots\\screenshots1.jpg"));

// Print a Log In message to the screen

System.out.println(" The Result is " + result);

// Close the Browser.

driver.close();

// Stop the ScreenRecorder

screenRecorder.stop();

}

}

* TestNG

TestNG is a powerful testing framework, an enhanced version of JUnit which was in use for a long time before TestNG came into existence. NG stands for 'Next Generation'.

TestNG framework provides the following features −

* Annotations help us organize the tests easily.
* Flexible test configuration.
* Test cases can be grouped more easily.
* Parallelization of tests can be achieved using TestNG.
* Support for data-driven testing.
* Inbuilt reporting.

Annotation:

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

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

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

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

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

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

@BeforeGroups : The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked.

@AfterGroups : The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked.

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

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

@DataProvider : Marks a method as supplying data for a test method. The annotated method must return an Object[ ][ ] where each Object[ ] can be assigned the parameter list of the test method. The @Test method that wants to receive data from this DataProvider needs to use a dataProvider name equals to the name of this annotation.

@Factory : Marks a method as a factory that returns objects that will be used by TestNG as Test classes. The method must return Object[ ].

@Listeners :Defines listeners on a test class.

@Parameters: Describes how to pass parameters to a @Test method.

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

In the following test, you will notice that there is NO main method, as testNG will drive the program execution flow. After initializing the driver, it will execute the '@BeforeTest' method followed by '@Test' and then '@AfterTest'. Please note that there can be any number of '@Test' annotation in a class but '@BeforeTest' and '@AfterTest' can appear only once.

package TestNG;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.annotations.AfterTest;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

public class TestNGClass {

WebDriver driver = new FirefoxDriver();

@BeforeTest

public void launchapp() {

// Puts an Implicit wait, Will wait for 10 seconds before throwing exception

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

// Launch website

driver.navigate().to("http://www.calculator.net");

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

}

@Test

public void calculatepercent() {

// Click on Math Calculators

driver.findElement(By.xpath(".//\*[@id='menu']/div[3]/a")).click();

// Click on Percent Calculators

driver.findElement(By.xpath(".//\*[@id='menu']/div[4]/div[3]/a")).click();

// Enter value 10 in the first number of the percent Calculator

driver.findElement(By.id("cpar1")).sendKeys("10");

// Enter value 50 in the second number of the percent Calculator

driver.findElement(By.id("cpar2")).sendKeys("50");

// Click Calculate Button

driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr/td[2]/input")).click();

// Get the Result Text based on its xpath

String result =

driver.findElement(By.xpath(".//\*[@id='content']/p[2]/span/font/b")).getText();

// Print a Log In message to the screen

System.out.println(" The Result is " + result);

if(result.equals("5")) {

System.out.println(" The Result is Pass");

} else {

System.out.println(" The Result is Fail");

}

}

@AfterTest

public void terminatetest() {

driver.close();

}

}

* Selenium Grid

Selenium Grid is a tool that distributes the tests across multiple physical or virtual machines so that we can execute scripts in parallel (simultaneously). It dramatically accelerates the testing process across browsers and across platforms by giving us quick and accurate feedback.

Selenium Grid has a Hub and a Node.

* **Hub** − The hub can also be understood as a server which acts as the central point where the tests would be triggered. A Selenium Grid has only one Hub and it is launched on a single machine once.
* **Node** − Nodes are the Selenium instances that are attached to the Hub which execute the tests. There can be one or more nodes in a grid which can be of any OS and can contain any of the Selenium supported browsers.

Working with Grid:

In order to work with the Grid, we need to follow certain protocols. Listen below are the major steps involved in this process −

* Configuring the Hub
* Configuring the Nodes
* Develop the Script and Prepare the XML File
* Test Execution
* Result Analysis