ITMD 536

SOFTWARE TESTING AND MAINTENANCE

RESEARCH PAPER TOPIC :
AUTOMATION TESTING USING SELENIUM

STUDENT NAME :
DEBAYAN MITRA

SUBMISSION DATE :
NOVEMBER 10, 2023

PROFESSOR :
DR NAZNEEN HASMI

ABSTRACT

This research paper presents an in-depth exploration of automation testing using **Selenium, Java programming language, Maven build tool, and TestNG testing framework, with a focus on the implementation of the Page Factory Model.** The paper addresses the persistent challenges in software quality assurance and advocates for the adoption of this integrated toolset to enhance efficiency and reliability in testing processes.

The problem statement delineates the **limitations of manual testing and highlights** the increasing complexity of software applications, necessitating a robust automation framework. The research methodology systematically outlines the implementation approach, **incorporating Java as the programming language, Selenium as the automation tool, Maven for project management, TestNG for test execution, and the Page Factory Model for efficient maintenance of test scripts.**

Results from the automation testing endeavor showcase a substantial reduction in testing time, increased test coverage, and improved accuracy. The automation **test case reports, generated through TestNG,** provide valuable insights into test execution outcomes, aiding in quick defect detection and resolution. Real-world case studies illustrate the practical application of this integrated approach, demonstrating its adaptability across diverse software development environments.

The conclusion emphasizes the pivotal role of the **Java-Selenium-Maven-TestNG combination** in transforming software testing practices. The **Page Factory Model** emerges as a key contributor to maintainability and scalability, enhancing the robustness of automated test scripts. The paper recommends widespread adoption of this integrated toolset in industry practices, asserting its positive impact on software quality and development speed.

In summary, this research contributes a comprehensive understanding of **automation testing using Selenium, Java, Maven, and TestNG, employing the Page Factory Model.** The presented results and insights position this integrated approach as a powerful solution for organizations seeking to optimize their software quality assurance processes.

INTRODUCTION

-Background and Context

In the dynamic landscape of software development, the quest for robust and efficient testing methodologies has become paramount. The shift towards agile and continuous integration demands testing processes that are not only reliable but also capable of keeping pace with rapid development cycles. **Automation testing,** particularly through the **utilization of Selenium,** has emerged as a cornerstone in addressing these challenges. **Selenium, an open-source framework**, is renowned for its versatility in automating web applications, making it a pivotal player in the realm of software quality assurance.

-Problem Statement

The **manual testing paradigm** is beset with inherent challenges that hinder the agility and effectiveness of the software testing process. Time-intensive manual execution, limited coverage, and the inherent risk of human error pose significant obstacles to achieving the desired levels of software quality. Recognizing the limitations of manual testing, this research aims to explore and articulate the transformative potential of automation testing, with a primary focus on Selenium.

-Purpose of the Study

The overarching purpose of this research is to comprehensively investigate the role of Selenium in automation testing, particularly within the context of **Java, Maven, TestNG, and the Page Factory Model**. The study seeks to provide insights into the practical implementation of automation testing methodologies, evaluating their impact on testing efficiency, test coverage, and overall software quality. By elucidating the strengths and challenges of Selenium, the research endeavors to contribute valuable knowledge that can guide organizations in adopting and optimizing their automation testing strategies.

-Research Questions and Objectives

To guide this exploration, the research is structured around the following key questions and objectives:

Research Questions

1. **How does Selenium contribute to overcoming the challenges inherent in manual testing?**
2. **What is the impact of the Java-Selenium-Maven-TestNG combination on testing efficiency and accuracy?**
3. **How does the Page Factory Model enhance maintainability and scalability in automation testing?**

Objectives

1. To assess the efficacy of Selenium in **automating diverse testing scenarios.**
2. To evaluate the practical implications of **integrating Java, Maven, TestNG, and the Page Factory Model** in Selenium-based automation.
3. To provide recommendations for optimizing automation testing strategies based on empirical findings.

LITERATURE REVIEW

-Manual Testing: Types, Pros, Cons, and Importance

In the realm of software development, **manual testing** is a foundational process where **human testers meticulously assess applications to uncover bugs, defects, and errors.** Unlike its automated counterparts, manual testing involves direct interaction with the product, focusing on evaluating its functionality and overall user experience.

Manual testing encompasses various types, each serving a distinct purpose in the testing landscape. **Black Box Testing** concentrates on functionality without delving into code, **while White Box Testing** involves scrutinizing the application's structure and code. **Gray Box Testing,** a blend of black and white box testing, identifies bugs and structural issues. **Acceptance Testing** ensures software meets specified requirements, **Integration Testing** checks the collaboration of different application modules, and System Testing evaluates the entire system's compliance against specified requirements.

Several advantages underscore the significance of manual testing. It is accessible without the need for programming knowledge, making it particularly advantageous for black box testing. Manual testing proves ideal for dynamic GUI designs, effectively adapting to changing graphical user interface layouts. Its ease of learning allows new testers to quickly grasp testing principles, and the approach of real user interaction enables a comprehensive assessment from an end user's perspective. The primary goal is achieving a 100% bug-free software.

While manual testing holds distinct advantages, it also presents challenges. It demands a substantial number of human resources, making it resource-intensive. The initial investment can be significant, leading to short-term cost implications to avoid more substantial issues post-release. Manual testing is inherently time-consuming due to its meticulous nature. Subjective test case development can be a limitation, as testers rely on their skills, potentially overlooking certain functions. Additionally, test cases need separate development for each new application, limiting reusability.

Despite the prevalence of automated testing, manual testing remains indispensable. It ensures thorough testing for bugs, offering a unique perspective from the end user. Manual testing acts as a necessary precursor to automated testing, addressing nuanced issues that automation might overlook. The human element embedded in the testing process is crucial for ensuring accuracy and a comprehensive evaluation of software.

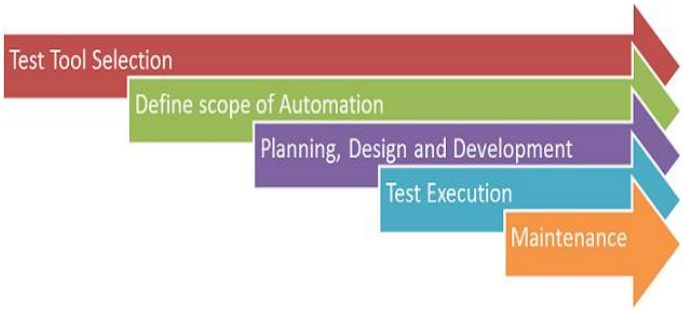
In conclusion, manual testing assumes a vital role in the software development life cycle, providing a meticulous examination that complements automated testing efforts. The combined approach results in the development of robust, user-friendly applications that meet the highest standards of quality.

- What is Automation Testing

Automation Testing is a software testing technique that performs using special automated testing software tools to execute a test case suite. The automation testing software can also enter test data into the System Under Test, compare expected and actual results and generate detailed test reports. Software Test Automation demands considerable investments of money and resources. Successive development cycles will require execution of same test suite repeatedly. Using a test automation tool, it's possible to record this test suite and re-play it as required. Once the test suite is automated, no human intervention is required.

- Automated Testing Process

Step 1) **Test Tool Selection**
Step 2) **Define scope of Automation**
Step 3) **Planning, Design and Development**
Step 4) **Test Execution**
Step 5) **Maintenance**



- Top Automation Testing Trends for 2023

In modern software testing, the focus has expanded beyond identifying errors to prioritize optimizing time-to-market, automating tasks, and enhancing team productivity. **Automation testing**, driven by technologies like **IoT and AI**, is

projected to reach USD 49.9 billion by 2026 with a CAGR of 19.2%. Anticipating 2023, trends include **Hyperautomation testing**, **cloud-based cross-browser testing**, and the rise of **non-functional testing** for efficiency and usability.

Automated regression testing ensures code changes don't break functionalities, while **API testing** becomes crucial for bug identification and seamless integration. **DevTestOps**, a notable trend, enhances deployment quality through continuous testing. **Codeless automation** is gaining popularity, with a 15.5% CAGR, and **mobile automation testing** remains vital, enhancing regression cases and reducing delivery time. The global mobile app testing services market is expected to reach USD 13,585.73 Million by 2026, growing at a CAGR of 20.3%.

In summary, automation testing is indispensable for organizations aiming to reduce delivery time, enhance software quality, and gain a competitive edge. Both large enterprises and startups can leverage these services to streamline operations, cut costs, and improve the overall customer experience. Staying informed about these trends is crucial for businesses adapting to the future of software testing.

-Evaluation of Selenium as an automation tool

The story of Selenium begins in 2004 at ThoughtWorks in Chicago, where **Jason Huggins** developed the Core mode, initially named "**JavaScriptTestRunner**," for testing an internal Time and Expenses application. ThoughtWorks, being committed to automated testing, embraced the tool's potential as a reusable testing framework for web applications. Colleagues such as **Paul Gross and Jie Tina Wang** contributed to its development, and discussions about open sourcing and a 'driven' mode ensued. Selenium gained traction within thought works and globally, with contributors like **Paul Hammant, Aslak Hellesoy, and Mike Melia** exploring different server ideas. The project evolved, with contributions from ThoughtWorkers worldwide, including **Mike Williams, Darrell Deboer, and Darren Cotterill.**

Selenium Remote Control emerged as a **standalone server**, thanks to contributions from **Dan Fabulich and Nelson Sprou**l at Bea. Meanwhile, **Pat Lightbody** pursued a commercial idea, "**Hosted QA**," and collaborated with **Dan and Nelson** to stabilize Selenium RC for large-scale deployment.

Shinya Kasatani in Japan integrated **Selenium's core code into a Firefox plugin**, creating an **IDE module for recording and playing back tests**.

Jennifer Bevan at Google implemented a **grid capability for Selenium RC**, and **Haw-bin Chai in Chicago joined the Selenium development team in 2007**. **Simon Stewart,** also at ThoughtWorks, developed a **different testing tool called WebDriver**, which showed promise with its client for each browser. **The merging of Selenium-RC and WebDriver became evident**, and **Simon**, now at Facebook, contributed to making that vision a reality. **Jason Huggins** later joined the Selenium support team at Google in 2007.

-Overview of Selenium and It's key features

It is a **software testing tool** used for **automation Testing.** It is an **open source testing tool** that provides playback and recording facility for Regression Testing. The **Selenium IDE** only supports **Mozilla Firefox web browser**.

- It provides the provision to export recorded script in other languages like **Java, Ruby, RSpec, Python, C#,** etc.
- It can be used with frameworks like **JUnit** and **TestNG**.
- It can execute **multiple tests** at a time
- Auto complete for Selenium commands that are common
- Walk through tests
- Identifies the element using **id, name, X-path**, etc.
- Store tests as **Ruby Script, HTML**, and any other format
- It provides an option to **assert** the title for every page
- It supports selenium **user-extensions.js** file
- It allows to insert comments in the middle of the script for better understanding and debugging

-Comparison with Other Automation Tools

Selenium, a widely used tool for automated software testing, faces comparisons with various alternative automation tools, each catering to distinct needs. Here's an overview of some key competitors:

- **Puppeteer:** A Node library maintained by the Chrome DevTools team, Puppeteer provides a high-level API for controlling headless Chrome or Chromium. Key features include **enhanced control over Chrome, web scraping capabilities, the ability to capture screenshots and PDFs for UI testing, and load time measurement** through the Chrome Performance Analysis tool.

- **Cypress:** Positioned as an **open-source test automation platform**, Cypress aligns with the latest development principles and offers tools like the **Cypress Test Runner for in-browser testing and the Cypress Dashboard for CI**

tools. Its **real-time testing capability, test snapshots from the Command Log, and automatic assertions** make it a preferred choice for developers.

- **WebdriverIO**: Founded by the OpenJS foundation, WebdriverIO is a **NodeJS-based end-to-end testing framework designed** for modern web and mobile applications. Notably, it supports **JavaScript/TypeScript** and is often used with WebdriverProtocol for features like scalable and stable test suites, built-in and community plugins for setup customization, and native mobile application testing.

- **Playwright:** Developed by Microsoft contributors, Playwright is **an open-source test automation library for Node.js**, automating browsers based on **Chromium, Firefox, and WebKit**. Key features encompass **easy setup and configuration, multi-browser support, multi-language compatibility, parallel browser testing capabilities, and support for multiple tabs/browsers.**

- **Cucumber**: As an automation tool for **behavior-driven development (BDD)**, Cucumber, originally in **Ruby**, now supports **Java and JavaScript**. It excels in engaging business stakeholders, focusing on end-user experience, enabling code reuse in tests through the Gherkin scripting language, and facilitating easy setup and execution.

- **NightwatchJS**: Developed and maintained by BrowserStack, NightwatchJS, **a Node.js-based framework,** utilizes the **Webdriver Protocol.** Its versatility allows seamless testing, including **End-to-End, component, visual regression, accessibility, API, unit, and integration testing.** Noteworthy features include **easy installation and setup, readable test scripts, support for multiple browser testing, and compatibility with native mobile app testing.**

In conclusion, while Selenium remains a popular choice, testers and developers can explore multiple alternatives, each offering a robust set of features. The selection should align with specific objectives, skillsets, testing scope, and other considerations within the DevOps team. Importantly, some alternatives can complement Selenium based on the testing requirements, showcasing the flexibility and diverse options available in the test automation landscape.

THEORITICAL FRAMEWORK

-Overview of Selenium Framework

The Selenium Framework stands as a versatile suite of tools crafted for the automation of web application testing, recognized for its portability and compatibility with all major browsers and operating systems. At its current version 3.142.6, with version 4.0 in beta, Selenium is not merely a singular framework but a customizable suite that can be configured into various frameworks tailored to diverse use cases. Offering an array of tools and customizations, it has established itself as a leading choice for automated testing needs. Its compatibility spans across major programming languages, browsers, and platforms, fostering extensive customization possibilities and seamless integration with other frameworks or dependencies. The framework's strength is amplified by a robust online community, providing ample support and assistance for learning.

The Selenium Framework is characterized by its noteworthy features, emphasizing compatibility with all major programming languages, browsers, and platforms. Its versatility is evident in the myriad of customization options available, empowering users to adapt the framework to their specific needs. Additionally, its capability to seamlessly interact with other frameworks or dependencies enhances its utility and flexibility.

The benefits of the Selenium Framework further underscore its value in the realm of automated testing. Engineered for consistent testing processes, it ensures repeatability and readability, significantly reducing person-hours invested in testing efforts. The framework's cross-platform testing capabilities add to its appeal, making it an invaluable tool for applications across diverse environments. As an open-source solution, the Selenium Framework not only minimizes costs but also enjoys strong community support, instilling confidence in users regarding continuous updates and improvements.

-Types of Selenium Frameworks: Enhancing Test Automation Efficiency

Selenium frameworks, designed based on a functional approach, can be categorized into three primary types, each offering unique advantages for efficient test automation:

- **Data-Driven Framework:**

The **data-driven framework** revolves around **utilizing various data sets stored in external files, often in Excel sheets**, which are then imported into the automation testing tool. This approach is particularly beneficial when dealing with a high volume of data sets. By keeping these data sets separate from the main script, testers can easily modify the script without disrupting or altering the data sets. This separation enhances robustness, allowing a single test to be applied to multiple data sets without the need for extensive code modifications.

- **Keyword-Driven Framework:**

The **keyword-driven framework** is centered around **keywords that encapsulate specific functionalities, taking parameters and producing relevant outputs.** This framework proves advantageous when dealing with extensive functionality that might require repetitive coding. To abstract the complexity of the code, keywords are created separately from the script. These keywords, often stored in an external Excel sheet, are then invoked in the code. This separation allows testers to manage each functionality independently by modifying individual keywords without affecting others.

- **Hybrid Driven Framework:**

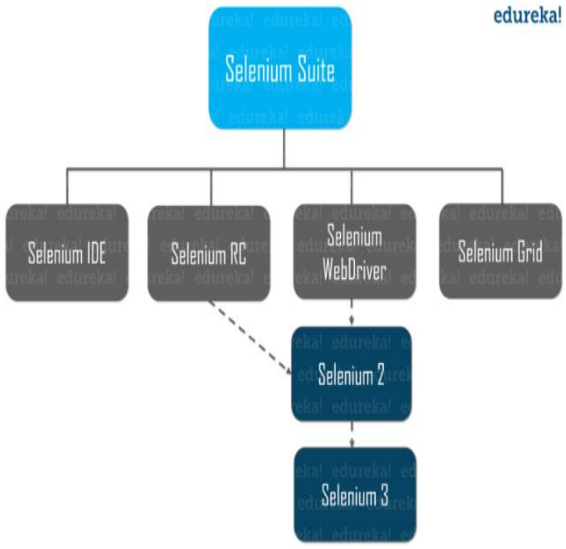
The **hybrid-driven framework** in Selenium **represents a fusion of both data-driven and keyword-driven approaches.** This framework employs various keywords and data sets, storing inputs and functions in separate files. Similar to the data-driven framework, it uses a consistent code for different data sets. This hybrid approach is particularly useful when dealing with a substantial number of data sets and functionalities, offering a balanced solution to avoid complexity.
The selection of a Selenium automation framework depends on the specific testing requirements. For scenarios where **numerous data sets need testing,** the **data-driven framework** proves beneficial by isolating them from the core code. In situations with a **multitude of functionalities to be tested**, the **keyword-driven framework** is recommended. Here, operations are stored as keywords in a separate table, facilitating easy modification. For projects with a **high number of data sets and functionalities**, the **hybrid-driven framework** strikes a balance, offering a comprehensive solution to streamline test automation efforts.

-Components of Selenium

Selenium, a powerful automation tool, consists of a suite of tools designed for different aspects of web application testing. The primary components include Selenium IDE, Selenium RC, Selenium WebDriver, and Selenium Grid.

- **Selenium IDE:**

Selenium IDE, an Integrated Development Environment, operates as a Firefox plugin and is the simplest framework in the Selenium Suite. It facilitates script recording and playback. However, for more advanced and robust test cases, one needs to complement it with Selenium RC or Selenium WebDriver. Selenium IDE has limitations, such as slow test case execution, restricted support for browsers beyond Firefox, and challenges in report generation.



- **Selenium RC:**

Formerly known as Selenium 1, Selenium RC relies on JavaScript for automation and supports various programming languages, including Ruby, PHP, Python, Perl, C#, Java, and Javascript. It operates on a client/server architecture, requiring manual initiation of the server for test execution. Selenium RC s upports parallel and remote test case execution using Selenium Grid. However, a drawback is the manual starting of the Selenium Standalone server before executing test cases.

- **Selenium WebDriver:**

Selenium WebDriver, a browser automation framework, communicates directly with the browser, accepting and executing commands. It is implemented through browser-specific drivers and supports multiple programming languages. WebDriver offers broad compatibility with various operating systems and browsers, making it a preferred choice for browser automation.

- **Selenium Grid:**

Selenium Grid, employed alongside Selenium RC, enables the parallel execution of tests on different machines against diverse browsers. It facilitates simultaneous testing against various machines with different operating systems and browsers. Selenium Grid is beneficial for scalable and efficient test execution.

Understanding these Selenium components is crucial for devising effective automation strategies, choosing the right tools for specific testing needs, and ensuring seamless web application testing.

-Selenium WebDriver Architecture

Selenium WebDriver, a pivotal component in browser automation, functions as a framework that interprets and dispatches commands to a web browser. Executed through a browser-specific driver, it directly communicates with the

browser, exercising control over its operations. Selenium WebDriver boasts compatibility with various programming languages, including Java, C#, PHP, Python, Perl, and Ruby.

In terms of operating system support, Selenium WebDriver is versatile, catering to Windows, Mac OS, Linux, and Solaris. Its browser support is extensive, encompassing Mozilla Firefox, Internet Explorer, Google Chrome (version 12.0.712.0 and above), Safari, Opera (version 11.5 and above), Android, iOS, and HtmlUnit 2.9.

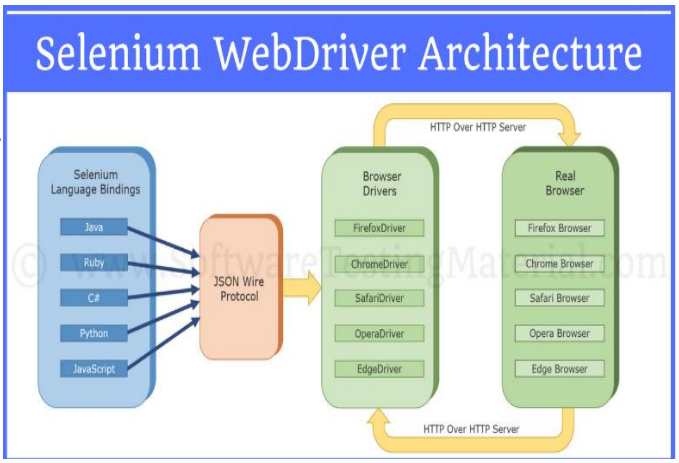
The Selenium Architecture comprises four integral components:

- **Selenium Client Library:**

Offering language bindings such as Java, Ruby, and Python, Selenium's client library facilitates multi-language support and flexibility.

- **JSON Wire Protocol over HTTP:**

Utilizing JavaScript Object Notation (JSON), this REST API transfers data between servers and clients on the web. The JSON Wire Protocol over HTTP acts as an intermediary, connecting the Selenium client library with browser drivers.



- **Browser Drivers:**

Individual browser drivers correspond to specific browsers and interact with them without exposing the internal functionalities. Each browser driver, like FirefoxDriver or ChromeDriver, has its HTTP server, ensuring seamless communication.

- **Browsers:**

Selenium's adaptability extends to various browsers, including Firefox, Chrome, Internet Explorer, Safari, and more.

Selenium's language bindings and the JSON Wire Protocol over HTTP collectively form the bridge between the Selenium client library and the browser drivers. The browser drivers, in turn, relay commands to their respective browsers, maintaining a cohesive and efficient automation framework. This architecture empowers Selenium to support a diverse range of browsers and operating systems, making it a preferred choice for web application testing.

- Understanding the Internal Mechanism of Selenium WebDriver Execution

Exploring the inner workings of Selenium WebDriver provides insights into how automation scripts are executed in real-time. By examining the process from script creation to browser launch, we gain a comprehensive understanding of the internal mechanisms at play.

- **Execution Workflow:**

In practical scenarios, developers utilize Selenium client libraries, such as Java, to write scripts in integrated development environments like Eclipse. Consider the following example:

```
WebDriver driver = new FirefoxDriver();
driver.get("https://www.softwaretestingmaterial.com");
```

Upon clicking "Run" to execute the script, the Firefox browser is launched, navigating to the specified website. Let's delve into the internal steps that unfold during this process.

- **Internal Process:**

- **Script Conversion:**

As you initiate the script execution, each statement is converted into a URL using the JSON Wire Protocol over HTTP. In the provided code, the Java client library transforms script statements into JSON format.

- **Communication with Browser Drivers:**

The generated URLs are then passed to the Browser Drivers, such as the FirefoxDriver in our case. The client library (Java) communicates with the chosen browser driver through HTTP, initiating the process.

- **HTTP Request Handling:**

Every Browser Driver operates a dedicated HTTP server to receive incoming HTTP requests. The URL, encapsulating the action to be performed (e.g., navigating to a URL), is transmitted to the Browser Driver via HTTP.

- **Browser Interaction:**

The Browser Driver forwards the HTTP request to the actual browser, facilitating the execution of commands on the browser. If the request is a POST request, an action is triggered on the browser. For GET requests, the browser generates a corresponding response.

- **Response Transmission:**

Upon executing the commands in the Selenium script, the response is sent back over HTTP to the Browser Driver. The JSON Wire Protocol is employed to facilitate seamless communication between the Browser Driver and the client library (e.g., Eclipse IDE).

In conclusion, understanding the intricate sequence of events—from script execution initiation to browser interaction—provides a foundational comprehension of how Selenium WebDriver operates internally. This knowledge is pivotal for practitioners seeking to optimize automation processes and troubleshoot potential issues in their testing workflows.

METHODOLOGY

- Research Design

This research adopts a mixed-methods research design, incorporating both **qualitative and quantitative approaches**. The **qualitative aspect** involves a comprehensive literature review to establish a theoretical framework, while the **quantitative aspect** involves practical experimentation and analysis of real-world case studies. This mixed-methods approach allows for a holistic understanding of the impact of Selenium in automation testing.

- Population and Sample

The population for this study comprises **software development projects that involve web application testing.** The sample is drawn from a diverse set of industries, ensuring a representative range of applications and testing scenarios. A stratified random sampling method is employed, categorizing projects based on industry, size, and testing requirements. The sample size is determined to ensure statistical significance and meaningful generalization of findings.

- Data Collection Methods

- **Literature Review:**

A comprehensive literature review is conducted to gather insights into the existing body of **knowledge related to Selenium, Java, Maven, TestNG, and the Page Factory Model** in automation testing. This forms the basis for establishing the theoretical framework.

- **Experimental Studies:**

Practical experimentation involves the **implementation of Selenium-based automation testing** in a controlled environment. Test cases are designed to cover various testing scenarios, and their execution is monitored to gather quantitative data on testing efficiency, accuracy, and other relevant metrics.

- **Case Studies:**

Real-world case studies are collected from industry projects that have **implemented Selenium for automation testing.** These cases provide practical insights into the challenges and successes of using Selenium in diverse software development settings.

- Tools and Technologies

- **Selenium WebDriver:**

The core automation tool for interacting with web elements and executing test scripts.

- **Java Programming Language:**

Utilized as the primary programming language for writing Selenium test scripts.

- **Maven:**

Employed as the build tool for managing project dependencies, compilation, and test execution.

- **TestNG:**

Selected as the testing framework for structuring and executing tests, enabling parallel test execution and providing detailed test reports.

➢ **Page Factory Model:**
Implemented for enhancing the maintainability of Selenium test scripts through the creation of a well-structured page object model.

- Variables and Measurements

➢ **Independent Variables:**
Selenium, Java, Maven, TestNG, Page Factory Model.

➢ **Dependent Variables:**
- Testing Efficiency (measured by execution time)
- Testing Accuracy (measured by the number of defects identified)
- Test Coverage (measured by the percentage of application functionality covered)
- Maintainability (measured by ease of script maintenance)
- Scalability (measured by the ability to handle an increasing number of test cases)

-Data Analysis Techniques

Quantitative data collected from experimental studies and case studies are analyzed using statistical methods. Descriptive statistics, such as mean and standard deviation, are employed to summarize key metrics. Inferential statistics, including t-tests and analysis of variance (ANOVA), are used to identify significant differences between various testing scenarios. Qualitative data from case studies are analyzed thematically to extract patterns and insights. The combined analysis provides a comprehensive understanding of the impact and effectiveness of Selenium in automation testing.

EXPLORING THE DIFFERENCES BETWEEN THE AUTOMATION TESTING GIANTS

- Selenium VS QTP

Let's compare QTP and Selenium based on several parameters:
QTP, a licensed tool incurring a high cost, mandates users to pay for all versions, while Selenium, an open-source tool, is freely accessible across its versions. While QTP has limitations in executing test cases across platforms, focusing primarily on Windows, Selenium provides high flexibility, supporting various platforms like Chrome and Firefox, albeit exclusively for web applications. QTP scripts are constrained to VBScript, developed by Microsoft, whereas Selenium scripts offer versatility in user-friendly languages like Java. QTP test cases are IDE-specific, unlike Selenium, which seamlessly operates across different IDEs. QTP relies on an inbuilt object repository, whereas Selenium utilizes web elements for testing, and while QTP supports diverse environments without additional plugins, Selenium accommodates plugins alongside its features.

- Selenium VS RPA

Selenium and Robotic Process Automation (RPA) serve distinct purposes in the realm of automation. Selenium is specifically designed for automating browser applications within web environments, focusing on user interface interactions. In contrast, RPA is dedicated to automating broader business processes, encompassing end-to-end tasks across various applications. Selenium, an open-source tool, utilizes Selenium Web Drivers to interact with browsers, while RPA employs RPA bots for tasks spanning different applications and systems. Selenium's automation occurs on the current browser page, emphasizing UI elements, whereas RPA executes tasks in the backend, manipulating data and systems without relying on a graphical interface. Selenium is more suited for browser-related processes, whereas RPA excels in handling low-value clerical processes and offers a simpler life cycle. Selenium is platform-dependent on browsers, while RPA is platform-independent, interfacing seamlessly with various systems. Selenium requires programming knowledge, whereas RPA operates on a more visual and intuitive level. Skills for effective Selenium use involve Selenium IDE, while RPA requires proficiency in SQL database management, analytical thinking, problem-solving, and knowledge of specific RPA tools.

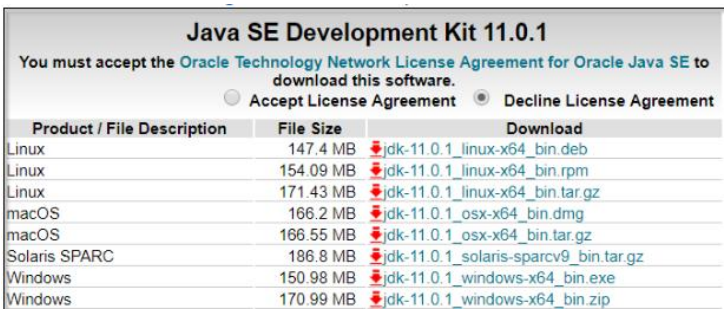
SELENIUM SET-UP AND CONFIGURATION

- Selenium Installation

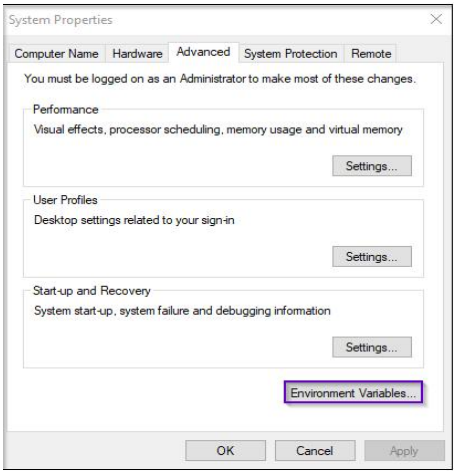
The process of installing Selenium involves three crucial steps: Java installation, Eclipse IDE configuration, and Selenium WebDriver installation.

➢ **Install Java:**
Java is a fundamental requirement for Selenium, and its installation involves several steps:

1. Visit the Java Downloads Page and select the Java Platform (JDK) option.

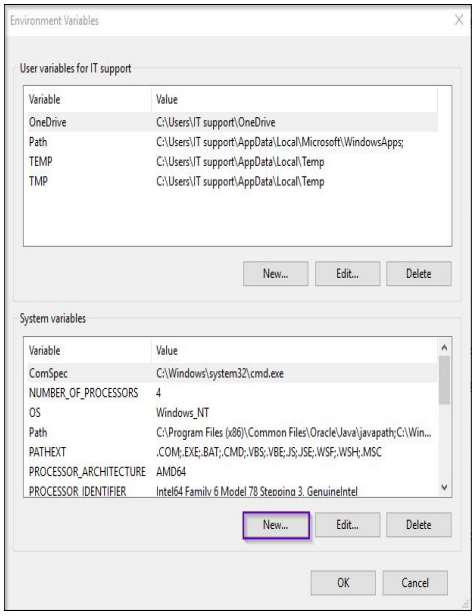


2. Accept the License Agreement and download the version suitable for your system configuration.
3. Run the installer, follow on-screen instructions, and configure the Java environment.



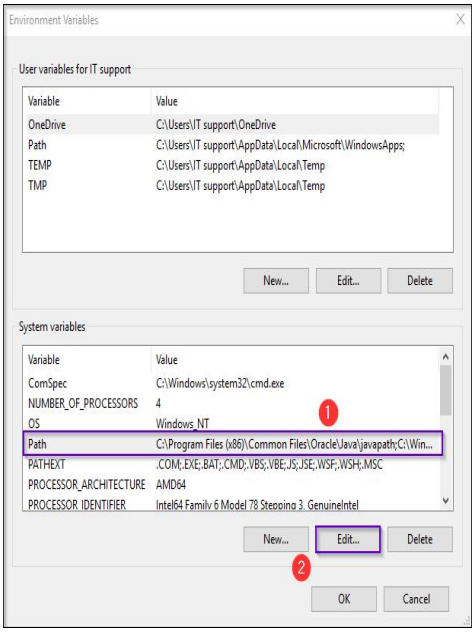
4. Set up the 'JAVA_HOME' system variable and configure the system variable path to include the Java installation directory.

STEP : 1



STEP : 2

STEP : 3



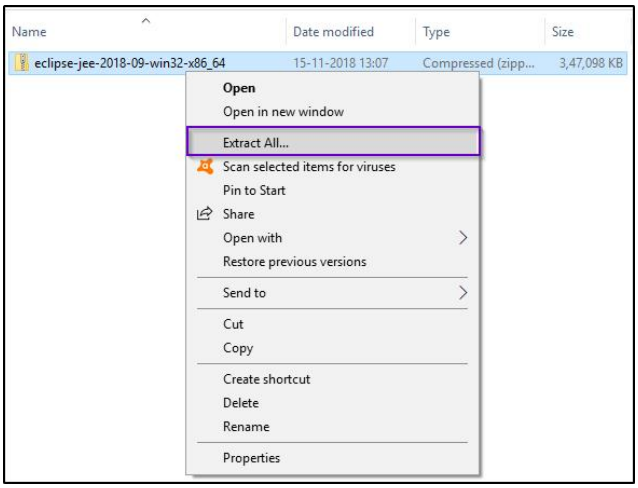
5. Cross-check the installation by running the command `java -version` in the command prompt.

➢ **Install Eclipse IDE:**

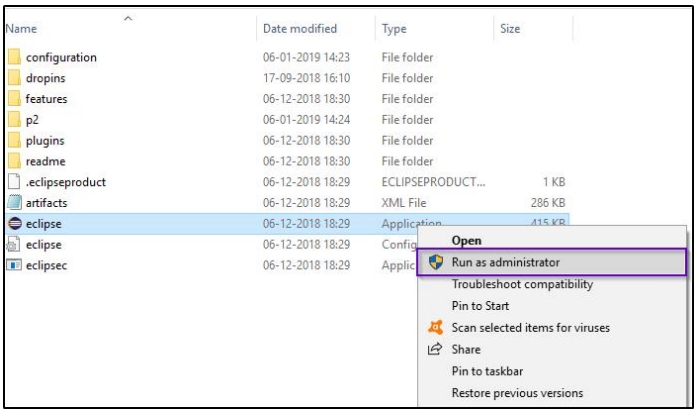
To configure Eclipse on your system, follow these steps:

1. Navigate to the Eclipse Downloads URL(https://www.eclipse.org/downloads/) and choose the appropriate version based on your system architecture.

2. Download and extract the zipped Eclipse file to any directory.



3. Launch Eclipse by running the `eclipse.exe` file.

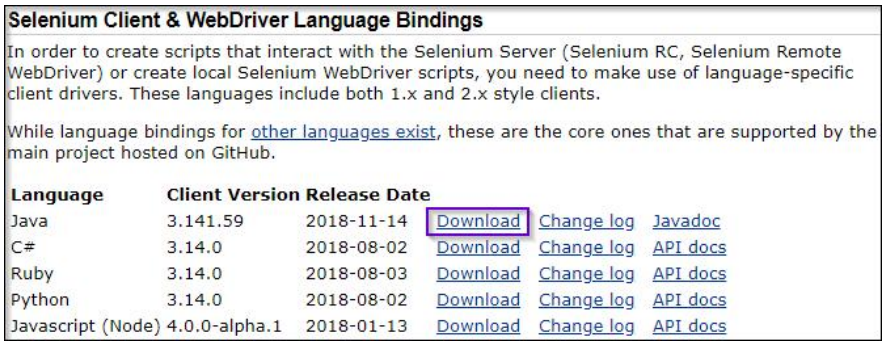


➢ **Install Selenium WebDriver:**

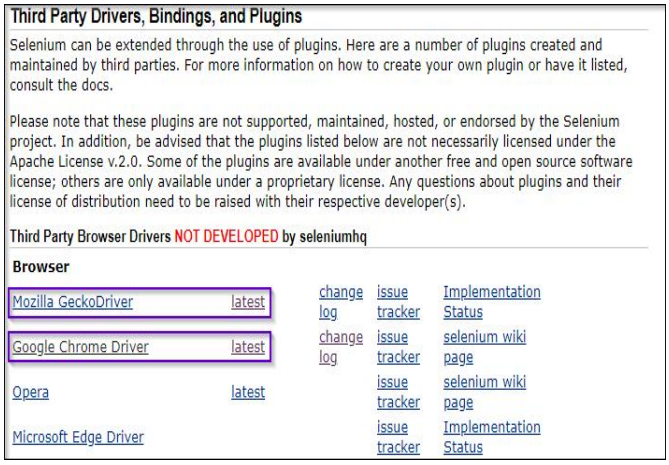
The final step involves installing Selenium WebDriver:

1. Visit the Selenium official website(**http://www.seleniumhq.org.**) and go to the Download section.
2. Download the Selenium Server standalone JAR and save it in a directory, e.g., "C:Selenium."

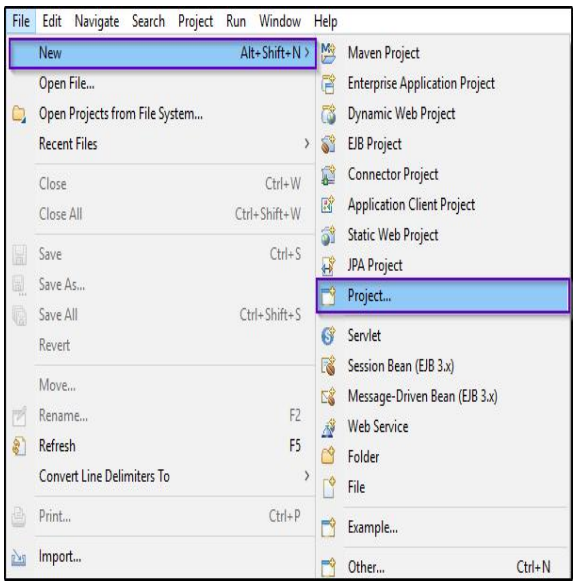
3. Download the Selenium Java Client by scrolling down to the Selenium Client and WebDriver Language section, and choose the Java download link.

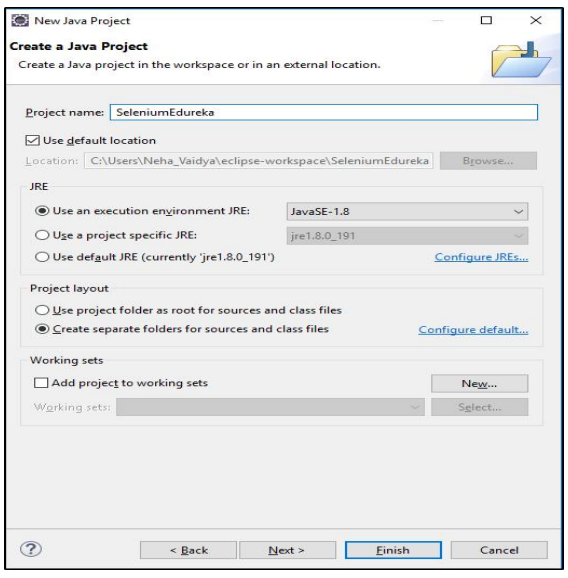


4. Extract the zip file and save it in the Selenium folder.
5. Download Chrome Driver and Gecko Driver if using Google Chrome and Mozilla Firefox, respectively.

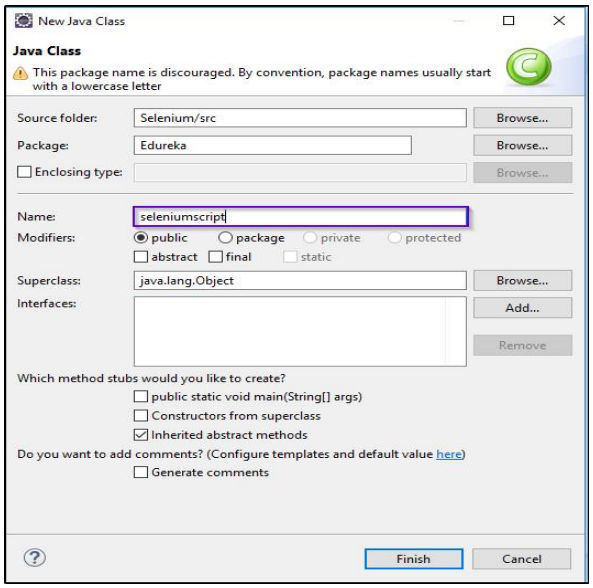


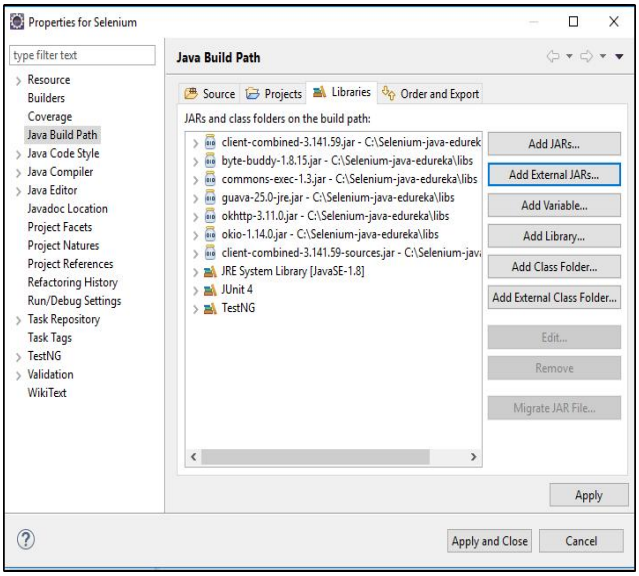
6.Create a new project in Eclipse, add external Selenium JAR files from the Selenium lib directory, and configure the build path.





7. Create a new class file, add referenced libraries, and you are set to execute Selenium programs.





This comprehensive installation process ensures a seamless setup for Selenium, enabling efficient browser automation.

- Execution of the first Selenium Program

```
package Edureka;
import java.util.concurrent.TimeUnit;
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.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class FirstSeleniumScript {
public static void main(String[] args) throws InterruptedException{
System.setProperty("webdriver.chrome.driver", "C:Selenium-java-edurekachromedriver_win32chromedriver.exe");
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();
driver.manage().deleteAllCookies();
driver.manage().timeouts().pageLoadTimeout(40, TimeUnit.SECONDS);
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
driver.get("<a href="https://login.yahoo.com/">https://login.yahoo.com/</a>");
driver.findElement(By.xpath("//input[@id='login-username']")).sendKeys("edureka@yahoo.com");
}
}
```

- Integration of Selenium with Maven and pom.xml

Integrating Selenium with Maven and configuring the `pom.xml` file is a common practice to manage dependencies, build projects, and streamline the Selenium setup. Below are the steps for integrating Selenium with Maven using the `pom.xml` file:

➢ **Create a Maven Project**

Create a new Maven project in Eclipse IDE or using the command line. Ensure Maven is installed on your system.

➢ **Configure pom.xml**

Open the `pom.xml` file in the root directory of your Maven project and add the necessary dependencies for Selenium and WebDriver.

```
```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```
```

```
  <groupId>your.group.id</groupId>
  <artifactId>your-artifact-id</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <!-- Selenium WebDriver -->
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.141.59</version> <!-- Use the latest version -->
    </dependency>

    <!-- Add other dependencies as needed -->
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version> <!-- Use the latest version -->
        <configuration>
          <source>1.8</source> <!-- or your Java version -->
          <target>1.8</target> <!-- or your Java version -->
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

In the above example, the `selenium-java` dependency is included for Selenium WebDriver. Make sure to use the latest version available.

- **Update Maven Project**

After modifying the `pom.xml` file, update the Maven project to download the dependencies.

- **Write Selenium Code**

Now, you can write your Selenium test scripts, and Maven will automatically manage the dependencies during the build process.

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

public class SeleniumExample {
  public static void main(String[] args) {
    System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
    WebDriver driver = new ChromeDriver();
    driver.get("https://www.example.com");
    // Your Selenium code here
    driver.quit();
  }
}
```

Ensure that the ChromeDriver executable path is correctly set in the `System.setProperty`.
This integration simplifies the management of Selenium dependencies and provides a standardized project structure with Maven.

ELEMENT IDENTIFICATION AND INTERACTION IN SELENIUM

- Finding Elements in Selenium WebDriver

In Selenium, the process of locating and interacting with web elements is crucial for automated testing. WebDriver provides a range of methods to find elements, such as **findElement()** and **findElements().** These methods utilize locators to identify elements on a web page, including options like ID, Name, Class Name, Tag Name, Link Text, and Partial Link Text. Developers need to choose appropriate locators to ensure robust and reliable test scripts.

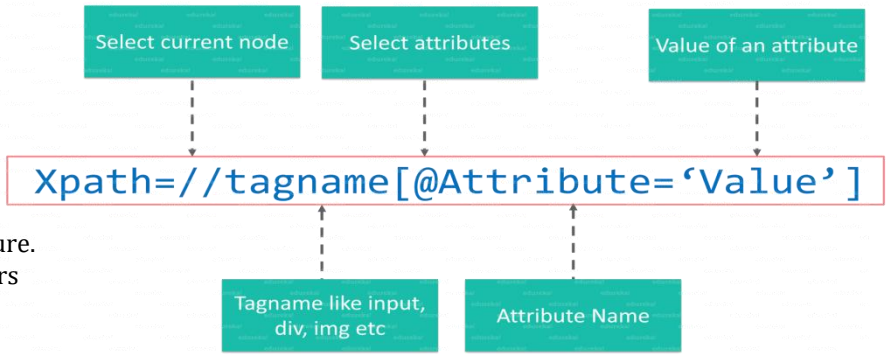
Syntax:
driver.findElement(By.tagName(<locator_value>)) ;//single web element
or
driver.findElements(By.tagName(<locator_value>)) ;//for list of elements

-Locators in Selenium - How to Locate Elements on Web-page

Locators act as addressing mechanisms to pinpoint HTML elements on a web page. Selenium supports a variety of locators, including ID, Name, Class Name, Tag Name, Link Text, Partial Link Text, CSS Selector, and XPath. The choice of locator depends on the specific attributes of the HTML elements. Understanding the structure of the web page and the uniqueness of elements aids in effective locator selection, which is fundamental for successful automation.

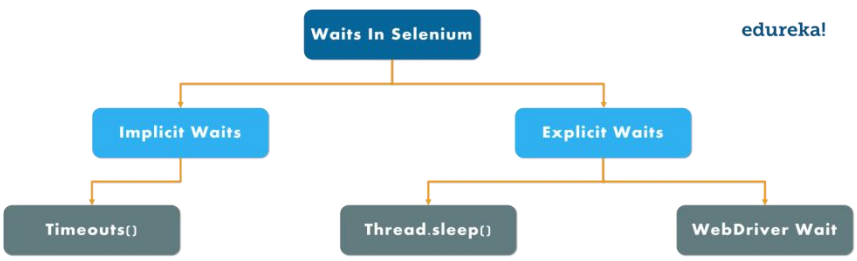
- XPath in Selenium

XPath is a powerful and flexible language for navigating XML documents, and it is widely used in Selenium for locating elements on a web page. It provides a hierarchical path to the elements, enabling precise identification. Absolute and relative XPath expressions are employed, with the latter being more recommended as it adapts better to changes in the HTML structure. XPath is especially useful when other locators struggle to uniquely identify elements.



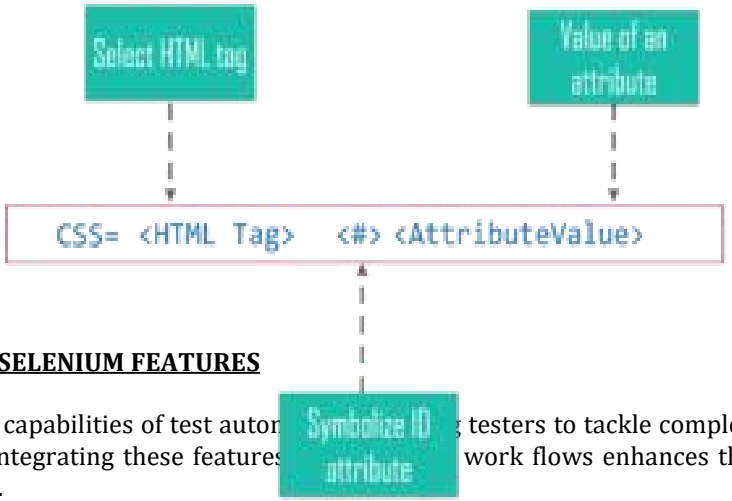
-Waits in Selenium

Waits in Selenium are essential for handling synchronization issues between the test script and the web page. Selenium offers **Implicit Waits, Explicit Waits, and Fluent Waits** to deal with different synchronization scenarios. Implicit Waits allow the WebDriver to wait for a certain amount of time before throwing an exception, while Explicit Waits wait for a specific condition to be met before proceeding. Fluent Waits combine flexibility and precision, making them suitable for dynamic web applications.



- Locate Web Elements Using CSS Selector in Selenium

CSS Selectors offer another way to locate elements in Selenium. They leverage the styling attributes of HTML elements to pinpoint them on a web page. CSS Selectors are concise and often more readable than XPath expressions. Choosing between XPath and CSS Selectors often depends on the structure and attributes of the web page. Both are powerful tools in the Selenium toolkit for efficient element location.



ADVANCED SELENIUM FEATURES

Exploration of advanced Selenium features expands the capabilities of test autor testers to tackle complex scenarios and ensure robust and reliable test scripts. Integrating these features work flows enhances the versatility and effectiveness of automated testing efforts.

-ChromeDriver and GeckoDriver in Selenium

ChromeDriver and **GeckoDriver** are browser-specific drivers used in Selenium for automating the **Chrome** and **Firefox** browsers, respectively. These drivers facilitate the **communication between Selenium WebDriver and the browsers,** enabling the execution of test scripts. As browsers regularly update, using the appropriate driver version is crucial for compatibility, making ChromeDriver and GeckoDriver integral components in Selenium test automation.

-Robot Class in Selenium WebDriver

The Robot class in Selenium is a **Java AWT (Abstract Window Toolkit) class** that provides a way to generate **native system input events** for controlling applications. It is often used to handle scenarios where Selenium WebDriver interactions are insufficient or impractical. The Robot class can simulate keyboard and mouse events, making it a powerful tool for automating complex user interactions and scenarios. Below are the methods of the Robot Class and their implementation :

- **KeyPress():** Simulates the pressing of a keyboard key.
- **KeyRelease():** Simulates the releasing of a pressed keyboard key.
- **MouseMove()**: Moves the mouse pointer to a specified location.
- **MousePress():** Simulates the pressing of a mouse button.
- **MouseRelease():** Simulates the releasing of a pressed mouse button.

-Actions Class in Selenium WebDriver

The Actions class in Selenium WebDriver is designed for **advanced user interactions such as mouse and keyboard events.** It provides a way to perform complex actions like **drag-and-drop, hovering, and multi-step operations.** The Actions class is particularly useful when dealing with web applications that involve intricate user interface components and behaviors, enhancing the precision and sophistication of test scripts. There are basically two methods which help in working with the actions in Selenium, namely:

- **Keyboard interface**
- **sendKeys(keysToSend):** Sends a series of keystrokes onto the element in Selenium WebDriver.
- **keyDown(theKey):** Sends a key press without releasing it, allowing subsequent actions to assume it as pressed, e.g., Keys.ALT, Keys.SHIFT, or Keys.CONTROL.
- **keyUp(theKey):** Performs a key release in Selenium WebDriver.

- Mouse interface
- **click():** Clicks on the element in Selenium WebDriver.
- **doubleClick():** Double clicks on the element.
- **contextClick():** Performs a context-click (right-click) on the element.
- **clickAndHold():** Clicks at the present mouse location without releasing.
- **dragAndDrop(source, target):** Clicks at the source location, moves to the target element, and releases the mouse.
- **dragAndDropBy(source, xOffset, yOffset):** Performs click-and-hold at the source location, shifts by a given offset, then frees the mouse.
- **moveByOffset(x-offset, y-offset):** Shifts the mouse by the given offset horizontally and vertically.
- **moveToElement(toElement):** Shifts the mouse to the center of the element.

- **release():** Releases the pressed left mouse button at the existing mouse location in Selenium WebDriver.

Code Snippet
```
Actions action = new Actions(driver);
action.moveToElement(element).click().perform();
```

-Handle Alerts and Pop-ups in Selenium

Handling alerts and pop-ups is a critical aspect of Selenium automation, especially **in scenarios where web applications generate prompts. Selenium WebDriver provides methods to interact with JavaScript alerts, confirms, and prompts.** Test scripts can **accept, dismiss, or input values** into these pop-ups, ensuring the smooth execution of test scenarios that involve user interactions. Below are the Alert Interface methods to handle alert messages.

- **void dismiss():** Dismisses the alert.

Code snippet
```
driver.switchTo().alert().dismiss();
```

- **void accept():** Accepts the alert.

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

- **String getText():** Retrieves the text from the alert.

Code snippet
```
driver.switchTo().alert().getText();
```

- **void sendKeys(String keysToSend):** Sends specified keys to the alert.

Code snippet
```
driver.switchTo().alert().sendKeys("Text");
```

-Select a Value from a Drop-down in Selenium WebDriver

Interacting with drop-downs is a common task in web automation. Selenium WebDriver offers the **Select class** to handle drop-downs efficiently. The Select class provides methods to choose options based on their **index, value, or visible text.** This functionality is essential for testing scenarios involving selection from lists or dropdown menus on web applications.

Code Snippet :
```
Select oSelect = new Select(driver.findElement(By.id(Element_ID)));
oSelect.selectByIndex(index)
oSelect.selectByIndex(index)
// Or can be used as
oSelect.selectByVisibleText(text)
oSelect.selectByVisibleText(text)
// Or can be used as
oSelect.selectByValue(value)
oSelect.selectByValue(value)
```

-Screenshot in Selenium WebDriver

Capturing screenshots during test execution is a **valuable practice for debugging and reporting purposes.** Selenium WebDriver provides the **TakesScreenshot interface,** which allows testers to capture the entire screen or specific web elements. Integrating screenshots into test automation workflows enhances the visibility into test execution and aids in identifying issues.

Code Snippet
```
WebDriver driver = new ChromeDriver();
driver.get("<a href="http://www.edureka.co/">http://www.edureka.co/</a>");
TakesScreenshot ts = (TakesScreenshot)driver;
File source = ts.getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(source, new File("./Screenshots/Screen.png"));
System.out.println("the Screenshot is taken");
```

- Exceptions in Selenium – Know How to Handle Exceptions

Exception handling is a critical aspect of Selenium test automation to manage unexpected events that may occur during test execution. Selenium WebDriver throws exceptions for various conditions such as **NoSuchElementException** or **TimeoutException.** Understanding exception types and employing proper handling mechanisms ensures that test scripts gracefully handle errors and provide meaningful feedback in reports. Types of Exceptions:

- **WebDriverException**

WebDriver Exception comes when we try to perform any action on the non-existing driver.

Code Snippet

```
WebDriver driver = new InternetExplorerDriver();
driver.get("<a href="http://google.com">http://google.com</a>");
driver.close();
driver.quit();
```

- **NoAlertPresentException**

When we try to perform an action i.e., either accept() or dismiss() which is not required at a required place; gives us this exception.

Code Snippet

```
try{
driver.switchTo().alert().accept();
}
catch (NoAlertPresentException E){
E.printStackTrace();
}
```

- **NoSuchWindowException**

When we try to switch to a window which is not present gives us this exception:

Code Snippet

```
WebDriver driver = new InternetExplorerDriver();
driver.get("<a href="http://google.com">http://google.com</a>");
driver.switchTo().window("Yup_Fail");
driver.close();
```

In the above snippet, line 3 throws us an exception, as we are trying to switch to a window that is not present.

- **NoSuchFrameException**

Similar to Window exception, Frame exception mainly comes during switching between the frames.

Code Snippet

```
WebDriver driver = new InternetExplorerDriver();
driver.get("<a href="http://google.com">http://google.com</a>");
driver.switchTo().frame("F_fail");
driver.close();
```

In the above snippet, line 3 throws us an exception, as we are trying to switch to a frame that is not present.

- **NoSuchElementException**

This exception is thrown when we WebDriver doesn't find the web-element in the DOM.

Code Snippet

```
WebDriver driver = new InternetExplorerDriver();
driver.get("<a href="http://google.com">http://google.com</a>");
driver.findElement(By.name("fake")).click();
```

USING JAVA AS PREFERRED LANGUAGE FOR SELENIUM AUTOMATION

In the realm of Selenium automation, Java stands out as the go-to programming language for a multitude of compelling reasons. Its popularity is well-founded, with distinct advantages that resonate with testers and developers alike. This section explores the key rationales for selecting Java as the language of choice for Selenium automation, highlighting its extensive community support, knowledge-sharing capabilities, rich ecosystem, platform independence, and the robust tooling that comes with strong type checking and IDE support. Furthermore, it provides a comprehensive roadmap for mastering Java essentials when embarking on a Selenium automation journey.

- Why Use Java for Selenium Automation?

Java has emerged as the most preferred programming language for Selenium automation, and there are compelling reasons for its widespread adoption. This choice offers a range of advantages that make it a popular selection among testers and developers alike. Here are some key reasons to use Java for Selenium:

- **Extensive Community Support:**

Java boasts a massive and active user community in the IT industry. This translates to abundant support and readily available resources. There's an extensive repository of references, tutorials, forums, and experts willing to assist with Java-related Selenium challenges.

- **Knowledge Sharing and Collaboration:**

Java's dominance within the Selenium testing community means that the knowledge base is vast, and knowledge sharing is both easy and rapid. With approximately 77% of Selenium testers using Java, collaborating with peers and finding solutions to common problems becomes straightforward.

- **Rich Ecosystem of Frameworks and Libraries:**

Java's longevity in the software development landscape has given rise to a rich ecosystem of frameworks, plugins, APIs, and libraries that support Java for test automation. This wealth of tools and resources simplifies test script development, making Java a practical choice.

- **Platform Independence:**

The utilization of the Java Virtual Machine (JVM) in Java renders it a platform-independent language. Selenium test scripts written in Java can be executed on any operating environment equipped with an installed JVM. This cross-platform compatibility presents a noteworthy advantage for teams operating in diverse environments.

- **Strong Type Checking and IDE Support:**

Java is a statically typed language, which means that Java Integrated Development Environments (IDEs) provide robust feedback on errors during coding. This feature ensures that code issues are identified early in the development process, reducing the likelihood of runtime errors.

-What to Learn in Java for Selenium

To effectively use Java for Selenium automation, it's essential to have a foundational understanding of Java. Here's a roadmap to get started:

- **Java Basics:**

Begin by learning Java fundamentals, including data types, keywords, operators, access modifiers, and methods.

- **Control Structures:**

Familiarize yourself with control structures such as iterative statements (for, while, do-while, for-each) and selection statements (if, if-else, switch).

- **Data Handling:**

Learn how to work with arrays, strings, and regular expressions for data manipulation.

- **Object-Oriented Programming (OOP) Concepts:**

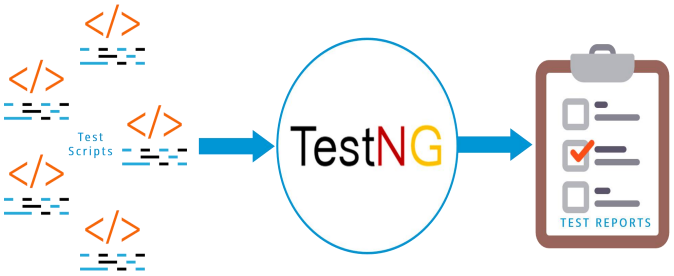
Explore object-oriented programming principles, including objects and classes, constructors, and concepts like inheritance, abstraction, encapsulation, and polymorphism.

- **Exception Handling:**

Understand exception handling in Java, which is crucial for writing robust Selenium scripts.

TESTNG IN SELENIUM FOR TEST CASE MANAGEMENT, REPORT GENERATION, AND DATAPROVIDER PARAMETERIZATION

-What is TestNG:



TestNG, short for "Test Next Generation," is a robust open-source test automation framework that draws inspiration from JUnit and NUnit. It serves as an upgrade to these frameworks, offering a suite of additional functionalities that enhance test case management. TestNG introduces features like **test annotations, grouping, prioritization, parameterization, and sequencing,** which were not readily available in its predecessors.

-Test Annotations: Controlling Test Execution:

TestNG leverages annotations to control the order in which methods are executed. Annotations are defined before each method in the test code, guiding the testing framework. For example, annotating a method with '**@Test**' designates it as the main test method to be executed. Other annotated methods, such as '**@BeforeMethod**' and '**@AfterMethod**', run before and after the primary test method, respectively. The execution order is determined by the type of annotation used. '**@BeforeClass**' and

'**@AfterClass**' run before and after the entire class, while '**@BeforeTest**' and '**@AfterTest**' manage the setup and teardown of test methods within a class. Below is a Sample code and it's output:

SAMPLE CODE

```
public class TestAnnotations {
 @Test
 public void myTestMethod() {
 System.out.println("Inside method:- myTestMethod");
 WebDriver driver = new FirefoxDriver();
 driver.get("<a
href="http://www.seleniumframework.com/Practiceform/">
http://www.seleniumframework.com/Practiceform/</a>");
 String title = driver.getTitle();
 System.out.println(title);
 driver.quit();
 }

 @BeforeMethod
 public void beforeMethod() {
 System.out.println("This piece of code is executed before
method:- myTestMethod");
 System.setProperty("webdriver.gecko.driver",
"C:UsersVardhanworkspaceSelenium
Projectfilesgeckodriver.exe");
 }

 @AfterMethod
 public void afterMethod() {
 System.out.println("This piece of code is executed after
method:- myTestMethod");
 }

 @BeforeClass
 public void beforeClass() {
 System.out.println("This piece of code is executed before the
class is executed");
 }

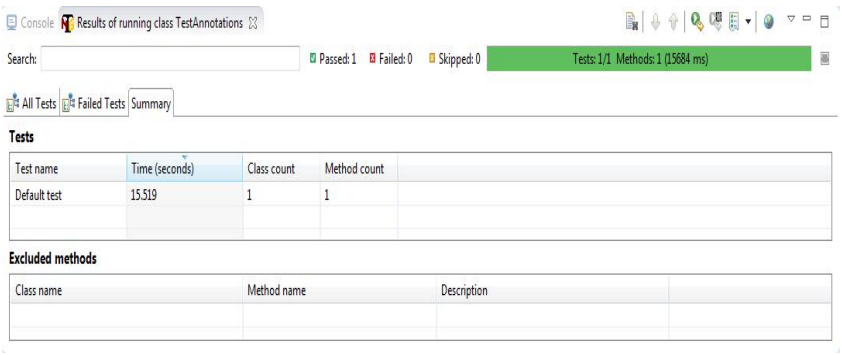
 @AfterClass
 public void afterClass() {
 System.out.println("This piece of code is executed after the
class is executed");
 }
}
```

OUTPUT

```
This piece of code is executed before the class is
executed
This piece of code is executed before method:-
myTestMethod
Inside method:- myTestMethod
1493192682118 geckodriver INFO Listening on
127.0.0.1:13676
1493192682713 mozprofile::profile INFO Using profile
path C:UsersVardhanAppDataLocalTemp
ust_mozprofile.wGkcwvwXkl2y
1493192682729 geckodriver::marionette INFO Starting
browser C:Program Files (x86)Mozilla
Firefoxirefox.exe
1493192682729 geckodriver::marionette INFO Connecting
to Marionette on localhost:59792
[GPU 6152] WARNING: pipe error: 109: file
c:/builds/moz2_slave/m-rel-w32-
0000000000000000000/build/src/ipc/chromium/src/chrom
e/common/ipc_channel_win.cc, line 346
1493192688316 Marionette INFO Listening on port 59792
Apr 26, 2017 1:14:49 PM
org.openqa.selenium.remote.ProtocolHandshake
createSession
INFO: Detected dialect: W3C
JavaScript error:
http://t.dtscout.com/i/?l=http%3A%2F%2Fwww.seleniumfr
amework.com%2FPracticeform%2F&j=, line 1: TypeError:
document.getElementsByTagName(...)[0] is undefined
Selenium Framework | Practiceform
1493192695134 Marionette INFO New connections will no
longer be accepted
Apr 26, 2017 1:14:57 PM
org.openqa.selenium.os.UnixProcess destroy
SEVERE: Unable to kill process with PID 6724
This piece of code is executed after
method:- myTestMethod
This piece of code is executed after the
class is executed
PASSED: myTestMethod
```

- Annotations Hierarchy:

@BeforeSuite: Runs before all tests in the suite.
@AfterSuite: Runs after all tests in the suite.
@BeforeTest: Executes before any test methods within a class.
@AfterTest: Executes after all test methods in a class.
@BeforeGroup: Runs before each group of test methods.
@AfterGroup: Runs after each group of test methods.
@BeforeClass: Executes once before the first test method in the class.
@AfterClass: Executes once after all test methods in the class.
@BeforeMethod: Runs before every test method within a class.
@AfterMethod: Runs after every test method within a class.
@Test: Marks the primary test method for execution.



SAMPLE TESTNG REPORT

- Test Prioritization and Disabling Test Cases:

- TestNG enables the **prioritization of test methods** by assigning numerical priorities using the **'Priority'** parameter. Smaller numbers indicate higher priority. When priorities are not explicitly set, tests run in alphabetical order.
- Additionally, TestNG allows to **disable specific test methods** without altering the code by using the **'enabled'** attribute. Test methods with **'enabled = false'** are skipped during execution.

SAMPLE CODE OF TEST PRIORITIZATION

```
@Test(Priority=2)
public static void FirstTest()
{
system.out.println("This is the Test Case number
Two because of Priority #2");
}

@Test(Priority=1)
public static void SecondTest()
{
system.out.println("This is the Test Case number
One because of Priority #1");
}

@Test
public static void FinalTest()
{
system.out.println("This is the Final Test Case
because there is no Priority");
}
```

SAMPLE CODE OF DISABILITNG TEST CASE

```
@Test(Priority=2, enabled = True)
public static void FirstTest()
{
system.out.println("This is the Test Case number
Two because of Priority #2");
}

@Test(Priority=1, enabled = True)
public static void SecondTest()
{
system.out.println("This is the Test Case number
One because of Priority #1");
}

@Test(enabled = false)
public static void SkippedTest()
{
system.out.println("This is the Skipped Test Case
because this has been disabled");
}

@Test(enabled = True)
public static void FinalTest()
{
system.out.println("This is the Final Test Case,
which is enabled and has no Priority");
}
```

- Method Dependency and Grouping:

- Method dependencies can be defined using the **'dependsOnMethods'** attribute, allowing one method to execute based on the successful completion of another method. When **'alwaysRun = true,'** a method is executed irrespective of the pass or fail status of the dependent method.
- Another useful feature is grouping, accomplished using the **'groups'** attribute, to categorize test methods. This functionality proves advantageous when aiming to execute specific groups of test methods rather than the entire suite.

SAMPLE CODE FOR METHOD DEPENDENCY

```
@Test
public static void FirstTest()
{
system.out.println("This is the first Test Case to be
executed");
}

@Test(dependsOnMethods = { "FirstTest" })
public static void SecondTest()
{
system.out.println("This is the second Test Case to be
executed; This is a Dependent method");
}

@Test(dependsOnMethods = { "SecondTest" })
public static void FinalTest()
{
system.out.println("This is the Final Test Case; It will be
executed anyway.");
}
```

SAMPLE CODE FOR METHOD GROUPING

```
@Test(groups = { "MyGroup" })
public static void FirstTest()
{
system.out.println("This is a part of the Group:
MyGroup");
}

@Test(groups = { "MyGroup" })
public static void SecondTest()
{
system.out.println("This is also a part of the Group:
MyGroup");
}

@Test
public static void ThirdTest()
{
system.out.println("But, this is not a part of the Group:
MyGroup");
}
```

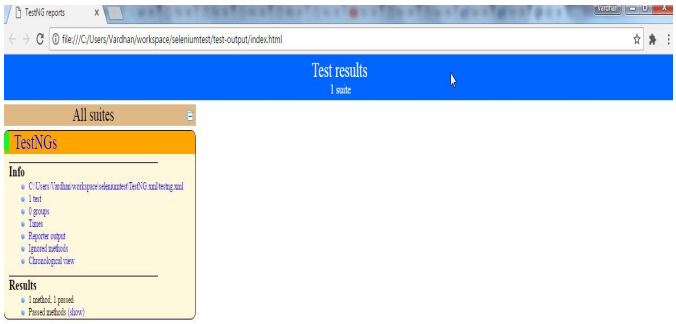
SAMPLE CODE

```
public class Assertions {
 @BeforeMethod
 public void beforeMethod() {
 System.setProperty("webdriver.gecko.driver",
 "C:UsersVardhanworkspaceSeleniumProjectfilesgeckodriver.exe");
 }
 public boolean isEqual(int a, int b) {
 if (a == b) {
 return true;
 } else {
 return false;
 }
 }
 @Test
 public void testEquality1() {
 Assert.assertEquals(true, isEqual(10, 10));
 System.out.println("This is a pass condition");
 }
 @Test
 public void testEquality2() {
 Assert.assertEquals(true, isEqual(10, 11));
 System.out.println("This is a fail condition");
 }
  @Test
 public void getTitle() {
 WebDriver driver = new FirefoxDriver();
 driver.get("<a href="https://www.gmail.com">https://www.gmail.com</a>");
 String title = driver.getTitle();
 Assert.assertEquals(title, "Gmail");
 System.out.println("This is again a pass condition");
 }
}
  </suite>
```

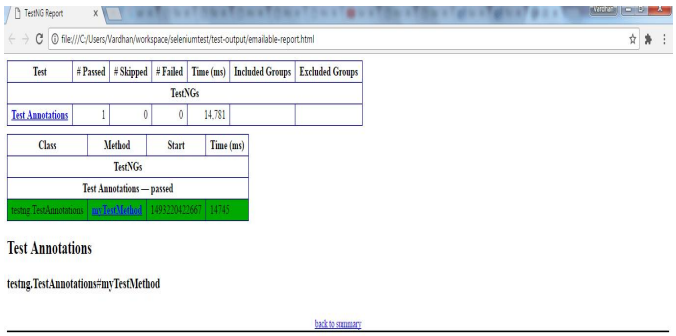
OUTPUT

```
1493277977348 geckodriver INFO Listening on 127.0.0.1:47035
1493277977993 mozprofile::profile INFO Using profile path C:UsersVardhanAppDataLocalT
ust_mozprofile.Z7X9uFdKODvi
1493277977994 geckodriver::marionette INFO Starting browser C:Program Files (x86)Mozi
1493277977998 geckodriver::marionette INFO Connecting to Marionette on localhost:5075
[GPU 6920] WARNING: pipe error: 109: file c:/builds/moz2_slave/m-rel-w32-000000000000
1493277981742 Marionette INFO Listening on port 50758
Apr 27, 2017 12:56:22 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
This is again a pass condition
This is a pass condition
PASSED: getTitle
PASSED: testEquality1
FAILED: testEquality2
java.lang.AssertionError: expected [false] but found [true]
 at org.testng.Assert.fail(Assert.java:93)
 at org.testng.Assert.failNotEquals(Assert.java:512)
 at org.testng.Assert.assertEqualsImpl(Assert.java:134)
 at org.testng.Assert.assertEquals(Assert.java:115)
 at org.testng.Assert.assertEquals(Assert.java:304)
 at org.testng.Assert.assertEquals(Assert.java:314)
 at testng.Assertions.testEquality2(Assertions.java:38)
```

- To access HTML reports, explore the test-output directory in your project's workspace. Within this folder, you'll find index reports and emailable reports that display test results and outcomes.



INDEX REPORT



EMAILABLE REPORT

In conclusion, TestNG empowers efficient and organized test case management, ensuring a comprehensive and reliable testing framework for Selenium WebDriver.

- Creating a sample test case using TESTNG annotations:

To create test cases using TestNG annotations, follow these steps:

Begin by setting up a project and including the TestNG library. Create a class file and write the test program.
Finally, craft an XML file to configure your test cases and execute them using TestNG Suite.
Now, let's delve deeper into how TestNG annotations enable test case grouping and program configuration.
In this particular test case, we'll utilize three different annotations to code the program.

```
package co.edureka.pages;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class AnnotationExample {
public String baseUrl = "<a href="https://www.edureka.co/">https://www.edureka.co/</a>";
String driverPath = "C://Users//Neha_Vaidya//Desktop//chromedriver_win32//chromedriver.exe";
public WebDriver driver ;

@BeforeTest
public void launchBrowser() {
System.out.println("launching Chrome browser");
System.setProperty("webdriver.chrome.driver", driverPath);
driver = new ChromeDriver();
driver.get(baseUrl);
}
@Test
public void verifyHomepageTitle() {
String expectedTitle = "Instructor-Led Online Training with 24X7 Lifetime Support | Edureka";
String actualTitle = driver.getTitle();
Assert.assertEquals(actualTitle, expectedTitle);
}
@AfterTest
public void terminateBrowser(){
driver.close();
}
}
```

OUTPUT

```
Console  Problems  Debug Shell
<terminated> AnnotationExample [TestNG] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (22-May-2019, 6:04:14 PM)
[RemoteTestNG] detected TestNG version 6.14.2
launching Chrome browser
Starting ChromeDriver 74.0.3729.6 (255758eccf3d244491b8a1317aa76e1ce10d57e9-refs/branch-heads/3729@{
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent access by malicious
May 22, 2019 6:04:20 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
PASSED: verifyHomepageTitle

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

- DataProviders in TestNG: An In-Depth Exploration:

To unlock the capabilities of DataProviders in TestNG, define a method returning a two-dimensional object array (Object[][]). The first array contains datasets, while the second stores parameter values. DataProvider methods can exist in the same test class or its superclasses. Alternatively, place DataProviders in a separate class, provided the method is static.Once established this method, applying the @DataProvider annotation to indicate its role as a DataProvider to TestNG. Optionally, assigning a name using the name attribute within the DataProvider annotation. If the name attribute is omitted, TestNG will use the method's name as a reference.

-Parameterization Using DataProvider

When confronted with scenarios demanding the management of extensive data sets for completing multiple web forms in your testing framework, DataProviders are the solution, facilitated through the @DataProvider annotation in TestNG. This annotation features a single attribute, 'name'. If the name attribute is not specified, the DataProvider will automatically adopt the name of the corresponding method by default. DataProviders provide an efficient method for handling diverse data sets during testing. Now, explore the practical implementation of DataProviders in TestNG through a tangible example.

```
package co.edureka.pages;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class DataProviderExample{

//This test method declares that its data should be supplied by the DataProvider
// "getdata" is the function name which is passing the data
// Number of columns should match the number of input parameters
@Test(dataProvider="getData")
public void setData(String username, String password)
{
System.out.println("your username is::"+username);
System.out.println("your password is::"+password);
}
```

```
@DataProvider
public Object[][] getData()
{
//Rows - Number of times your test has to be repeated.
//Columns - Number of parameters in test data.
Object[][] data = new Object[3][2];

// 1st row
data[0][0] ="user1";
data[0][1] = "abcdef";

// 2nd row
data[1][0] ="user2";
data[1][1] = "xyz";

// 3rd row
data[2][0] ="user3";
data[2][1] = "123456";

return data;
}
}
```

OUTPUT :

```
[RemoteTestNG] detected TestNG version 6.14.2
your username is::user1
your password is::abcdef
your username is::user2
your password is::xyz
your username is::user3
your password is::12345
PASSED: setData("user1", "abcdef")
PASSED: setData("user2", "xyz")
PASSED: setData("user3", "12345")

===============================================
    Default test
    Tests run: 3, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 3, Failures: 0, Skips: 0
===============================================
```

MASTERING DATABASE TESTING WITH SELENIUM: A COMPREHENSIVE GUIDE

In the evolving landscape of big data, databases play a pivotal role in managing records and preserving their integrity. Ensuring the absence of defects during data processing is a critical aspect of software testing. One tool that aids in this endeavor, particularly in the realm of automation testing, is Selenium.

-Java Database Connectivity (JDBC) – Bridging the Gap

Java Database Connectivity, or JDBC, stands as a standard Java API that facilitates database-independent connectivity between the Java programming language and a wide spectrum of databases. This API empowers you to encode access request statements in Structured Query Language (SQL), which are subsequently passed to the database management system. JDBC primarily involves tasks such as establishing a connection, creating an SQL database, executing SQL queries, and processing the output. With JDBC, you can harness the power to access tabular data stored in any relational database. This API allows you to perform operations like saving, updating, deleting, and fetching data from databases, drawing parallels to Microsoft's Open Database Connectivity (ODBC).

-Key Components of JDBC

The JDBC API offers essential interfaces and classes, including:

- **DriverManager:** Responsible for managing a list of database drivers, each recognizing a specific subprotocol under JDBC to establish a database connection.
- **Driver:** An interface that manages communications with the database server.
- **Connection:** An interface housing all the necessary methods for connecting to a database. The connection object serves as the communication context through which all interactions with the database occur.

-Steps to Develop a JDBC Application

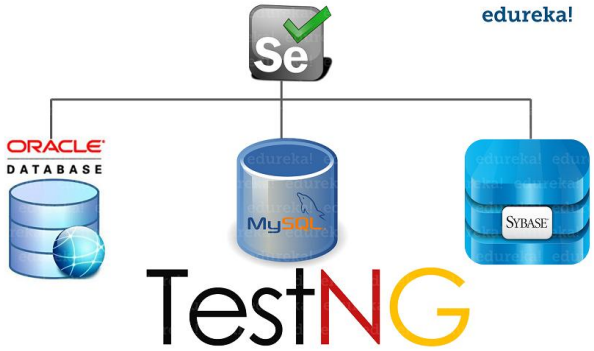
Creating a JDBC application entails a series of steps:

- **Import the packages:** Begin by importing the packages that contain the JDBC classes essential for database programming.
- **Register the JDBC driver:** Initialize a driver to open a communication channel with the database. Registration can be achieved using the command:

  Class.forName("com.mysql.jdbc.Driver");

- **Open a connection:** After driver registration, utilize the getConnection() method to create a Connection object, establishing a physical connection with the database.
- **Execute a query:** Use an object of type 'Statement' to build and submit an SQL statement to the database.
- **Extract data from the result set:** Retrieve data from the result set using the appropriate getXXX() method.
- **Clean up the environment:** Explicitly close all database resources, relying on JVM garbage collection.

-Database Testing Using Selenium



Although Selenium is not inherently equipped for database testing, it is possible to perform this task partially by leveraging JDBC and ODBC. In this discussion, we explore the connection between a Java program and a database to retrieve and verify data using TestNG.

-Step-by-Step Procedure for Database Testing

Step 1: Create a database.
Step 2: Once tables are created and values are inserted, establish a connection to the database.
Step 3: After establishing the connection, execute queries and process the records within your database.

-Integrating TestNG with JDBC for Database Testing: A Program Example

SAMPLE CODE

```
package co.edureka.pages;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class DatabaseTesingDemo {
// Connection object
static Connection con = null;
// Statement object
private static Statement stmt;
// Constant for Database URL
public static String DB_URL = "jdbc:mysql://localhost/emp";
// Constant for Database Username
public static String DB_USER = "your_user";
// Constant for Database Password
public static String DB_PASSWORD = "your_password";

@BeforeTest
public void setUp() throws Exception {
try{
// Make the database connection
String dbClass = "com.mysql.cj.jdbc.Driver";
Class.forName(dbClass).newInstance();
// Get connection to DB
Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
// Statement object to send the SQL statement to the Database
stmt = con.createStatement();
}
catch (Exception e)
{
e.printStackTrace();
}
}
```

EXPLANATION

The provided code specifies the database URL, username, and password for database access. It utilizes TestNG annotations to structure the test process. Before the test execution, it establishes a connection to the MySQL database by registering the driver and creating a statement object. The actual test, involving query execution, result processing, and record handling, is enclosed within the Test annotation.

Upon test completion, it closes the database connection using the AfterTest annotation. This organization separates the tasks effectively. When executed as a TestNG test, the code prints database details and runs the test cases.

```
@Test
public void test() {
try{
String query = "select * from employees";
// Get the contents of userinfo table from DB
ResultSet res = stmt.executeQuery(query);
// Print the result untill all the records are printed
// res.next() returns true if there is any next record else returns false
```

```
while (res.next())
{
System.out.print(res.getString(1));
System.out.print("  " + res.getString(2));
System.out.print("  " + res.getString(3));
System.out.println("    " + res.getString(4));
}
}
catch(Exception e)
{
e.printStackTrace();
}
}

@AfterTest
public void tearDown() throws Exception {
// Close DB connection
if (con != null) {
con.close();
}
}
}
```

OUTPUT

```
[RemoteTestNG] detected TestNG version 6.14.2
100 18 Zara Ali
101 25 Mahnaz Fatma
102 30 Zaid Khan
103 28 Sumit Mittal
PASSED: test

===============================================
Default test
Tests run: 1, Failures: 0, Skips: 0
===============================================

===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

THE IMPORTANCE AND PROCESS OF CROSS-BROWSER TESTING USING SELENIUM

-What is Cross-Browser Testing

Cross-browser testing is a vital aspect of ensuring the compatibility and performance of websites across various browsers and operating systems. It involves running the same set of test cases on different browsers like IE, Chrome, Firefox, and more. This process is particularly important because each browser uses different rendering engines, which can lead to variations in how a website is displayed. To guarantee a website functions seamlessly across different browsers and OS, cross-browser testing is necessary.
For instance, consider having a set of 20 manual test cases to complete, a task that might take a day or two. However, when the same 20 test cases need to be executed across five different browsers, it could potentially consume a week of effort. Nevertheless, by automating these 20 test cases and running them, the process can be completed in just an hour or two, depending on the complexity of the individual test cases. This is where the significance of cross-browser testing becomes evident.

-Reasons for Cross-Browser Testing

- Ensuring browser compatibility with different operating systems.
- Addressing image orientation issues.
- Handling variations in JavaScript interpretation among browsers.
- Preventing font size discrepancies and rendering problems.

- Confirming compatibility with new web frameworks.

-Performing Cross-Browser Testing:

Cross-browser testing can be efficiently executed using automation tools like Selenium WebDriver. The process involves the following steps:

- Utilize Selenium WebDriver to automate test cases for browsers like Internet Explorer, Firefox, Chrome, and Safari.
- Integrate the TestNG framework with Selenium WebDriver to execute test cases on multiple browsers simultaneously.
- Write and run the test cases to ensure cross-browser compatibility.

-Implementing cross-browser testing for Edureka with Selenium on 3 browsers

SAMPLE CODE

```
import java.util.concurrent.TimeUnit;
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.edge.EdgeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class CrossBrowserScript {

WebDriver driver;

/**
* This function will execute before each Test
*tag in testng.xml
* @param browser
* @throws Exception
*/
@BeforeTest
@Parameters("browser")
public void setup(String browser) throws Exception{
//Check if parameter passed from TestNG is 'firefox'
if(browser.equalsIgnoreCase("firefox")){
//create firefox instance
System.setProperty("webdriver.gecko.driver", "C:geckodriver-v0.23.0-win64geckodriver.exe");
driver = new FirefoxDriver();
}

//Check if parameter passed as 'chrome'
else if(browser.equalsIgnoreCase("chrome")){
//set path to chromedriver.exe
System.setProperty("webdriver.chrome.driver", "C:Selenium-java-edurekaNew folderchromedriver.exe");
driver = new ChromeDriver();

}
else if(browser.equalsIgnoreCase("Edge")){
//set path to Edge.exe
System.setProperty("webdriver.edge.driver","C:Selenium-java-edurekaMicrosoftWebDriver.exe");;span style="font-family:
verdana, geneva, sans-serif; font-size: 14px;"&gt;//create Edge instance&lt;/span&gt;
driver = new EdgeDriver();
}
else{
//If no browser passed throw exception
throw new Exception("Browser is not correct");
}
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
}
```

EXPLANATION

The provided code involves interactions with the Edureka website, including tasks like logging in and searching for a Selenium course. However, the primary goal is to assess cross-browser compatibility across Google Chrome, Mozilla Firefox, and Microsoft Edge. To achieve this, the code establishes system properties for these three browsers and utilizes locators for website interactions. In order to execute this program, a TestNG XML file containing the dependencies for the class file is required.

```
@Test
public void testParameterWithXML() throws InterruptedException{
driver.get("<a href="https://www.edureka.co/">https://www.edureka.co/</a>");
WebElement Login = driver.findElement(By.linkText("Log In"));
//Hit login button
Login.click();
Thread.sleep(4000);
WebElement userName = driver.findElement(By.id("si_popup_email"));
//Fill user name
userName.sendKeys("your email id");
Thread.sleep(4000);
//Find password'WebElement password = driver.findElement(By.id("si_popup_passwd"));
//Fill password
password.sendKeys("your password");
Thread.sleep(6000);

WebElement Next = driver.findElement(By.xpath("//button[@class='clik_btn_log btn-block']"));
//Hit search button
Next.click();
Thread.sleep(4000);
WebElement search = driver.findElement(By.cssSelector("#search-inp"));
//Fill search box
search.sendKeys("Selenium");
Thread.sleep(4000);
//Hit search button

WebElement searchbtn = driver.findElement(By.xpath("//span[@class='typeahead__button']"));
searchbtn.click();
}
}
```

Below code depicts the TestNG file. Within the XML file below, I define distinct classes for the drivers, enabling the instantiation of browsers for executing test cases on the website. This configuration facilitates the testing process.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "<a href="http://testng.org/testng-1.0.dtd">http://testng.org/testng-1.0.dtd</a>">
<suite name="TestSuite" thread-count="2" parallel="tests" >
<test name="ChromeTest">
<parameter name="browser" value="Chrome"/>
<classes>
<class name="co.edureka.pages.CrossBrowserScript">
</class>
</classes>
</test>
<test name="FirefoxTest">
<parameter name="browser" value="Firefox" />
<classes>
<class name="co.edureka.pages.CrossBrowserScript">
</class>
</classes>
</test>
<test name="EdgeTest">
<parameter name="browser" value="Edge" />
<classes>
<class name="co.edureka.pages.CrossBrowserScript">
</class>
</classes>
</test>
</suite>
```

UNDERSTANDING PAGE FACTORY AND PAGE OBJECT MODEL IN SELENIUM

-Introduction to Page Factory in Selenium

Page Factory is a Selenium design pattern that simplifies the creation of Page Objects for web automation testing. It reduces code duplication and enhances code readability.

-Advantages of Page Factory in Selenium

- **Reduced code duplication:** Page Factory minimizes the need for boilerplate code, improving maintainability.
- **Improved code readability**: It separates web element initialization from test code for better readability.
- **Better test maintenance:** Updating code for web page changes is easier, reducing errors and enhancing efficiency.
- **Enhanced test performance:** Page Objects are initialized once per test, optimizing test performance.

-Initializing Web Components in Page Factory

Page Factory in Selenium allows web elements to be initialized using the **@FindBy** annotation, streamlining the process of locating and initializing them on a web page. To achieve this, the **initElements** method is employed for initialization.

➢ **@FindBy annotation:**

The @FindBy annotation, a key feature of Selenium's Page Factory, serves to locate web elements via various types of locators such as ID, name, class, tag name, link text, partial link text, and CSS selector. This annotation maps a web element on a web page to a field in a Page Object, simplifying access and manipulation during test execution.

There are two methods for using the @FindBy annotation:

Method 1:

```
@FindBy(how = How.ID, using = "username")
private WebElement elementName;
```

In this case, the 'How' attribute is assigned a value from the How Enum, representing different locator types. For instance, How.ID is used to locate an element by its ID attribute, and the 'using' attribute specifies the value of the locator, which in this case is the web element's ID.

Method 2:

```
@FindBy(id = "search-box")
private WebElement searchBox;
```

In this example, the @FindBy annotation locates a web element with the ID attribute "search-box." The WebElement object "searchBox" is initialized with a reference to the web element found by this locator.
The @FindBy(id = "search-box") annotation simplifies the process of locating and initializing web elements within Selenium's Page Factory. It eliminates the need to write additional code for finding the web element using the By.id() method, making the process more efficient.

➢ **initElements():**

'initElements()' is a method provided by the Page Factory class for initializing elements defined in a Page Object. This method accepts two parameters: a WebDriver instance and a Page Object instance. It scans the Page Object for fields annotated with the @FindBy annotation and initializes them with references to the corresponding web elements on the page. 'initElements()' is an overloaded function, offering flexibility for modification and various uses.

-First Selenium Program using Page Factory Model

➢ **TestCase Scenerio :**

- Launch the Gmail website.
- Locate the element like email, password, and next button.
- Click on the Next button.

➢ **Code Snippet and Explanation :**

- **GmailLoginPage.java:**

In this scenario, the GmailLoginPage class represents the Gmail login page, using @FindBy annotations to identify elements like email, password, and the Next button. These elements come to life through the initElements() method from PageFactory.

Within the constructor, we set up the driver and navigate to the Gmail login page with driver.get(). Then, PageFactory's initElements() method is employed, passing both the driver and class instance to correctly configure the identified elements with their locators. Following this, we create methods for interacting with these elements, streamlining actions and simplifying manipulation through @FindBy annotations.

To utilize this Page Object in our tests, we instantiate the GmailLoginPage class and utilize its methods for efficient page interaction. This modular approach enhances code maintenance and readability.

```
package demo;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
public class GmailLoginPage {
   private WebDriver driver;
   // Using @FindBy to locate elements on the page
   @FindBy(xpath = "//input[@type='email']")
   private WebElement emailField;

   @FindBy(xpath = "//input[@type='password']")
   private WebElement passwordField;

   @FindBy(xpath = "//span[contains(text(), 'Next')]")
   private WebElement nextButton;

   // Constructor to initialize the driver and instantiate elements using PageFactory
   public GmailLoginPage(WebDriver driver) {
      this.driver = driver;

      PageFactory.initElements(driver, this);
   }

   public void enterEmail(String email) {
      emailField.sendKeys(email);
   }

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

   public void clickNextButton() {
      nextButton.click();
   }
}
```

- **Demo.java:**

It's essential to initialize the driver before creating a GmailLoginPage class instance. Additionally, we invoke the clickNextButton() method twice – once after entering the email and then after entering the password. This double usage is necessary because the Gmail login process is divided into two distinct pages.

```
package demo;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class demo  {
   public static void main(String[] args) throws Exception {
      // Set the path of the Chrome driver
      System.setProperty("webdriver.chrome.driver", "
C:\\Users\\sgokhru\\Downloads\\chromedriver_win32
\\chromedriver.exe");

      WebDriver driver = new ChromeDriver();
      driver.get("https://mail.google.com/");
      driver.manage().window().maximize();
      GmailLoginPage loginPage = new GmailLoginPage(driver);
      loginPage.enterEmail("myemail@gmail.com");
      loginPage.clickNextButton();
      loginPage.enterPassword("mypassword");
      loginPage.clickNextButton();
   }
```

}

-Introduction to Page Object Model (POM) in Selenium

POM segregates web elements from test code, improving organization, maintainability, and reusability.

-Advantages of Page Object Model (POM)

- **Improved code reusability:** The same Page Object class can serve multiple test cases.
- **Easy maintenance:** Web page updates can be made in the Page Object class without affecting test code.
- **Enhanced collaboration:** POM fosters collaboration between development and testing teams.
- **Increased test coverage:** POM allows creating more tests with less effort.
- **Better code readability:** POM makes test code more readable and understandable.

-First Selenium Program using Page Object Model

When working with the Page Object Model in Selenium Java testing, it's crucial to follow some guidelines for effective test case construction.

- First, avoid including assertions or related functionalities in a page object file. Such files should solely focus on storing web element variables and their interactions.
- Furthermore, it's essential to recognize that web page interactions often lead users to another page. In such cases, the page should return an object representing the next page, ensuring a smooth transition and organization within the test automation framework.

➢ **Scenario:**

Pages – GmailHomePage.java and GmailLoginPage.java
TestBase.java
GmailLoginTestCase.java

➢ **Code Snippet :**

- **GmailHomePage.java:**

```
public class GmailHomePage {

    WebDriver driver;

    @FindBy(link = “Sign In”)
    WebElement signInBtn;

    public GmailHomePage (WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    // This function is to click on signIn button
    public GmailLoginPage clickSignInButton() {
        signInBtn.click();
        return new GmailLoginPage(driver);
    }
```

- **GmailLoginPage.java:**

```
public class GmailLoginPage {

    private WebDriver driver;

    // Using @FindBy to locate elements on the page
    @FindBy(xpath = “//input[@type=’email’]”)
    private WebElement emailField;

    @FindBy(xpath = “//input[@type=’password’]”)
```

```
    private WebElement passwordField;

    @FindBy(xpath = "//span[contains(text(), 'Next')]")
    private WebElement nextButton;

    // Constructor to initialize the driver and instantiate elements using PageFactory
    public GmailLoginPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
    public void enterEmail(String email) {
        emailField.sendKeys(email);
    }

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

    public void clickNextButton() {
        nextButton.click();
    }
}
```

- **BaseClass.java :**

```
public class BaseClass{
    public static WebDriver driver;

    @BeforeSuite
    public void initializeWebDriver() throws IOException {
        System.setProperty("webdriver.chrome.driver",.\\chromedriver.exe");
        driver = new ChromeDriver();

        // To maximize browser
        driver.manage().window().maximize();

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

        // To open Gmail site
        driver.get("https:// www.gmail.com");
    }

    @AfterSuite
    public void quitDriver() {
        driver.quit();
    }
}
```

- **GmailLoginTest.java :**

```
public class GmailLoginTest extends BaseClass{

    @Test
    public void init() throws Exception {
        GmailHomePage gmailHomePage = new GmailHomePage(driver);
        GmailLoginPage gmailLoginpage =  gmailHomePage.clickSignInButton();
        gmailLoginpage.setEmail("abc@gmail.com");
        gmailLoginpage.clickNextButton();
        gmailLoginpage.setPassword("23456@qwe");
       gmailLoginpage.clickNextButton();

    }
```

}

-Comparing Page Factory and Page Object Model in Selenium: Features and Benefits

In comparing Page Factory to the Page Object Model (POM), it's essential to understand their distinctions.
Page Factory Model primarily serves the purpose of initializing WebElements within a page class, streamlining the process of identifying and interacting with these elements. It does so through an annotation-based approach, simplifying the code and making it more concise. However, this method may result in less code reusability due to the tighter coupling between elements and their interaction logic, which can be a drawback in scenarios requiring frequent updates for element changes. The initialization process in Page Factory can lead to slightly slower performance since it occurs at runtime. Nonetheless, it does support switching to different browser drivers, making it suitable for smaller and less complex test automation projects.

On the other hand, the **Page Object Model (POM)** aims to represent a web page as a Java class. It provides a structured and organized way to encapsulate web elements, interactions, and the overall functionality of a page. Unlike Page Factory, it relies on a design pattern that involves defining locators in a separate class, enhancing maintainability and code readability. The loose coupling between locators and the test code fosters greater code reusability, especially in large and complex test automation projects. Additionally, the initialization process in POM occurs at compile-time, leading to slightly faster performance. Similar to Page Factory, POM also supports switching to different browser drivers, making it a robust choice for extensive and intricate automation projects

CHALLENGES AND SOLUTIONS IN SELENIUM AUTOMATION

Selenium is a widely used open-source automation testing tool with its own set of limitations. In this article, we'll explore these limitations and provide solutions for each.

- **Limitation 1: Cannot Test Windows Applications**
- **Solution:** Selenium primarily focuses on web applications and does not support testing of Windows-based applications. To address this limitation, consider using alternative automation tools designed for testing desktop applications, like WinAppDriver for Windows applications and Appium for mobile apps.

- **Limitation 2: Mobile Testing Is Not Supported**
- **Solution:** While Selenium is excellent for testing web applications on desktop browsers, it lacks native support for mobile testing. To handle mobile testing, integrate Appium with Selenium. Appium extends Selenium to automate iOS and Android native, mobile, and hybrid apps via the WebDriver protocol.

- **Limitation 3: Limited Reporting**
- **Solution:** Selenium provides basic reporting capabilities. To enhance reporting, integrate Selenium with TestNG, a testing framework that offers more robust reporting features. With TestNG, you can generate detailed Extent reports, making test results more informative and easier to analyze.

- **Limitation 4: Handling Dynamic Elements**
- **Solution:** Dynamic elements with ever-changing attributes pose a challenge in Selenium automation. To address this, utilize dynamic XPath or dynamic CSS selectors. Functions like starts-with, contains, and ends-with help manage dynamic objects effectively.

- **Limitation 5: Handling Pop-Up Windows**
- **Solution:** Selenium lacks the capability to handle Windows-based pop-up windows as they are part of the operating system. To manage such pop-ups, consider using external tools like AutoIT or Robot Framework's AutoItLibrary, which provide solutions for automating interactions with these windows.

- **Limitation 6: Handling CAPTCHA**
- **Solution:** Selenium has limitations in automating CAPTCHA tests effectively due to their security measures. For CAPTCHA automation, explore third-party services or tools that specialize in solving CAPTCHAs, such as anti-captcha.com or 2Captcha.

- **Limitation 7: Image Testing Is Not Possible**

Solution: Selenium is not designed for image-based testing. To conduct image testing, consider integrating Selenium with SikuliX, an open-source tool for automating visual elements and image recognition. SikuliX enables you to automate tasks involving visual components and images.

In summary, Selenium is a powerful tool for web automation testing, but it has its limitations. Addressing these limitations often involves using complementary tools or frameworks to extend Selenium's capabilities and cover a wider range of testing scenarios.

CASE STUDIES

-Real world application of selenium :

Selenium, a powerful automation testing tool, has found extensive real-world applications across various domains. This section explores case studies and success stories that showcase the practical use of Selenium and the valuable lessons learned from its implementation.

- **Case Study 1: E-commerce Website Testing**
- **Challenge:** A prominent e-commerce company needed a comprehensive testing solution to ensure the functionality of its complex website across diverse browsers, devices, and operating systems. They faced challenges with dynamic elements, frequent updates, and cross-browser compatibility.

- **Solution:** The company leveraged Selenium for its ability to perform cross-browser testing and handle dynamic web elements. Selenium WebDriver scripts were designed to automate testing across browsers like Chrome, Firefox, and Internet Explorer. TestNG was integrated for detailed reporting. The outcome was significant time savings and improved test coverage, ensuring a seamless online shopping experience for customers.

- **Case Study 2: Financial Services Application Testing**
- **Challenge:** A financial services provider required a robust testing tool to verify the functionality and security of its web-based trading platform. Testing involved ensuring real-time data updates, secure login processes, and responsive user interfaces.

- **Solution:** Selenium was chosen for its versatility and support for multiple programming languages. Test scripts were developed to simulate user interactions and validate data integrity. Selenium Grid was employed to run tests simultaneously on different browser and OS combinations, leading to time and resource efficiency. Selenium played a pivotal role in ensuring the platform's performance, maintaining the trust of clients in the financial services sector.

- **Case Study 3: Healthcare Software Testing**
- **Challenge:** A healthcare software development firm was tasked with verifying the accuracy and regulatory compliance of its electronic health record (EHR) system. Systematic and scalable testing was required to meet healthcare industry regulations.

- **Solution:** Selenium was selected as the primary automation tool to validate the functionality and data integrity of the EHR system. Selenium's support for data-driven testing facilitated extensive validation of patient records and medical data. By automating repetitive tests, including data entry forms and report generation, Selenium improved testing cycles and minimized manual errors. The result was more efficient testing, enhanced adherence to healthcare regulations, and a reliable EHR system.

-Success Stories

- **Success Story 1: Microsoft Edge Testing**
- **Scenario:** Selenium's WebDriver played a crucial role in automating the testing of Microsoft Edge, Microsoft's web browser. This success story highlights Selenium's robust capabilities and its adoption for testing the browser's functionality.

- **Achievement:** Selenium WebDriver enabled Microsoft to conduct comprehensive testing of Edge, covering various browser versions, operating systems, and devices. Selenium's cross-browser testing capabilities and strong community support ensured that Edge met the required quality standards.

- **Success Story 2: Mozilla Firefox Testing**
- **Scenario:** Mozilla Firefox, a widely-used web browser, relied on Selenium for its testing requirements. Selenium's flexibility and support for multiple programming languages allowed for efficient testing of Firefox browser releases.

- **Achievement:** Selenium WebDriver significantly contributed to the success of Firefox's testing efforts. By automating the testing process across multiple platforms, Selenium helped maintain the browser's high-quality standards. It also demonstrated the tool's adaptability and compatibility with diverse web applications.

-Lessons Learned from Implementation

- **Lesson 1: Versatility and Scalability**

Selenium's versatility and scalability make it a preferred choice for testing various applications. The case studies illustrate its adaptability across different industries, from e-commerce and finance to healthcare and web browsers.

- **Lesson 2: Cross-browser Testing**

Selenium's cross-browser testing capabilities are invaluable for ensuring the compatibility of web applications across different browsers and platforms.

- **Lesson 3: Automation Efficiency**

Automation with Selenium reduces manual testing efforts, enhances test coverage, and minimizes errors, resulting in efficient and reliable testing processes.

FUTURE TRENDS

-Future of Automation Testing: Trends and Technologies

Staying competitive in software development hinges on keeping up with automation testing trends. "**Shift-Left" testing**, integrating testing early, cuts costs and defects. **Agile and DevOps** now rely on automation for shorter delivery cycles.

Robust test automation frameworks are vital, including **Behavior-Driven Development (BDD) and hybrid frameworks**. **AI and ML** are enhancing automation with self-healing scripts and smarter execution.

Robotic Process Automation (RPA) streamlines repetitive tasks, freeing up resources. **Cross-browser and cross-platform testing** is essential for user satisfaction, supported by tools like Selenium Grid.

The **IoT and blockchain** need specialized testing, and automation frameworks, like **IoTify**, provide solutions. By embracing these trends, companies enhance software quality, speed up releases, and satisfy users.

-AI-Based Testing: Shaping the Future of Software Testing

The application of Artificial Intelligence (AI) is rapidly expanding into various industries, including software testing. AI-driven testing is gaining traction, with businesses focusing on its growth, as indicated by the World Quality Report 2020-2021. **The incorporation of AI aims to make software testing more efficient, effective, and budget-friendly.**

AI-based testing utilizes AI and Machine Learning (ML) algorithms to enhance the testing process. It **automates test execution,** making it smarter and devoid of human intervention, thereby improving logical reasoning and problem-solving in testing.

AI's evolution in software testing is notable. It can mimic human intelligence, while ML allows computers to learn autonomously. **AI and ML algorithms extract patterns from data** to make informed decisions. This approach leads to **faster and continuous testing, complete automation, and quicker return on investment (ROI).**

The benefits of integrating AI into software testing are **multifold. Visual validation, using AI's pattern recognition and image analysis, ensures all visual elements function as intended.** It significantly boosts accuracy by eliminating human-prone errors and enhances test coverage by checking multiple aspects of software. **AI-based testing also saves time, money, and effort by handling repetitive tasks efficiently,** contributing to faster time-to-market and reduced defects.

AI-driven testing employs **four key approaches:**
Differential testing, which classifies differences in application versions; **Visual testing,** focusing on the look and feel of applications; **Declarative testing,** aiming to specify test intent in a natural language; and **Self-healing automation,** which corrects element selection in tests when UI changes.

AI-driven testing tools can be categorized into four main types:
Differential tools, which identify issues like code-related problems and security vulnerabilities through code scanning and unit test automation. **Visual AI testing tools**, such as Applitools and Percy by BrowserStack, examine the UI layer with image-based learning.**Declarative tools,** like Tricentis and UiPath Test Suite, automate end-to-end testing using AI and RPA.
Self-healing tools, exemplified by Mabl and Testim, address flakiness and reliability issues with the help of AI and ML.

In conclusion, AI-based testing is changing the landscape of software testing, allowing for faster and more reliable product releases. It supports **DevOps practices**, reduces human errors, and enables a higher quality of software in less time. Embracing AI-based testing can lead to efficient and accurate testing, ultimately benefiting businesses and their customers.

-Predictions for the Future of Selenium

The future of Selenium, the open-source automation testing tool, presents a landscape teeming with exciting possibilities and emerging trends. One of the most anticipated developments is the **seamless integration of Artificial Intelligence (AI) into Selenium.** AI-driven testing is expected to play a pivotal role, offering features like intelligent test case generation, self-healing

capabilities, and optimized test execution. These advancements are set to make Selenium more adaptable and efficient in handling complex web applications.

Selenium's reputation for cross-browser testing is poised for further enhancement. With a diverse array of browsers and platforms continually surfacing, Selenium is anticipated to offer more sophisticated tools and techniques to facilitate comprehensive cross-browser testing. This evolution will ensure that web applications maintain their compatibility and functionality across the spectrum of browsers and devices.

Parallel testing is another area that holds promise. Selenium Grid has laid the foundation for parallel and distributed testing, and the future promises even more advanced solutions. This will enable organizations to save precious time and resources by executing tests concurrently, thereby optimizing their test suites and enhancing productivity.

As the Internet of Things (IoT) continues to expand, **Selenium may evolve to cater to the specialized testing requirements of IoT devices and applications.** IoT ecosystems are complex, involving intricate interactions between hardware, software, and network components. Selenium is likely to introduce tools and frameworks tailored to simplify IoT testing processes.

In addition, with the growing prominence of blockchain technology, **Selenium may introduce specific tools designed for blockchain testing.** Testing smart contracts, consensus algorithms, and transaction verification is vital, and Selenium's potential role in ensuring the reliability and security of blockchain applications should not be underestimated. In this evolving landscape, Selenium remains poised to uphold its reputation as a powerful, adaptable, and open-source tool for automation testing.

CONCLUSION

-Summary of Key Findings:
In concluding our exploration of automation testing leveraging Selenium, a thorough summary of key findings emerges. Our study delves into the nuanced effectiveness of Selenium in enhancing software testing processes, revealing insights into its impact on test efficiency, reliability, and overall software quality assurance.

-Limitations of the Study:
Transparency is crucial in scholarly pursuits, and as we wrap up our research, it is imperative to acknowledge the study's limitations. These limitations, whether related to the scope, methodologies, or external factors, are presented candidly to provide a nuanced understanding of the context within which our findings are situated.

-Recommendations for Future Research:
The journey of Selenium in automation testing is dynamic, and our study contributes to this evolving narrative. In light of our current insights, we extend recommendations for future research initiatives. These recommendations aim to guide researchers toward unexplored facets, potential enhancements to Selenium's application, and emerging trends that warrant deeper investigation.

REFERENCES

Chrome driver and Gecko driver in selenium
https://www.edureka.co/blog/selenium-chromedriver-and-geckodriver/

Data provider in TestNg
https://www.edureka.co/blog/dataprovider-in-testng/

Page factory and page object model in selenium
https://www.browserstack.com/guide/page-object-model-in-selenium

Selenium webdriver architecture
https://www.javatpoint.com/selenium-webdriver

Online courses for Selenium
https://www.edureka.co/selenium-certification-training

APPENDICES

https://drive.google.com/drive/u/0/folders/1RjymMIZz6YuyeMGkuVgR3q9Ajt8aZ_mz
Orange HRM Login Page, have the code in the link attached.