

Test Automation Induction Program

Date	Phase name	Author	Version
23.11.2018	Document establishment	D. Plaku F. Bejko	1.0
28.01.2019	Document editing	S. Zaja F. Doda	1.0
20.06.2019	Review and quality check	S. Zaja F. Doda	1.0
06.04.2020	Updating	S. Bano E. Llanaj L. Çili F. Çeka	2.0

Preface

This documentation is aimed to provide an overview of the topics and subjects related to the Automation Testing training program.

It will be explained how the induction will be performed, as well as how students will be monitored and evaluated.

In this document you will find also several projects that will be assigned to the students under training, during the induction period.

Contents

Test Automation Induction Program	1
Preface	2
Introduction.....	5
Monitoring the progress	5
1. Introduction to Testing Fundamentals	6
1.1 What is testing?	6
1.2 Why should you test?.....	7
1.3 Automation Testing Vs. Manual Testing: What's the Difference?.....	9
1.4 Unit Tests, Integration Tests & End-to-End Tests	14
1.5 Tools for Testing.....	17
References	26
2. Element Selectors	27
2.1 Locating by CSS Selector	27
2.2 Other selectors.....	34
References	35
3. Introduction to Selenium.....	36
3.1 Selenium WebDriver, Installation and Configuration	36
3.2 First Script	40
3.3 Introduction to WebElement, findElement(), findElements()	44
3.4 Keyboard and Mouse Events using Action Class in Selenium WebDriver	73
3.5 How to handle Alert and Frame in Selenium WebDriver	75
3.6 Implicit Wait, Explicit Wait & Fluent Wait in Selenium	77
3.7 Common exceptions when working with Selenium.....	75
3.8 Scroll in Selenium.....	79
3.9 Assert Vs Verify Commands in Selenium.....	81
3.10 Difference between driver.close() and driver.quit().....	82
4. Introduction to TestNg & Junit	87
4.1 TestNG Annotations.....	88
4.2 JUnit Annotations.....	92

5. Selenium with Cucumber	93
5.1 What is Cucumber testing?	93
5.2 Cucumber features, scenarios and steps	93
5.3 Why use Cucumber with Selenium?	95
6. Page Object Model (POM) & Page Factory: Selenium WebDriver Tutorial	96
6.1 What is Page Object Model?	96
6.2 How to implement POM?	97
6.3 What is Page Factory?	98
7. Most Common Challenges in Selenium Automation	99
8. Overview	101

Introduction

New interns are supposed to work with Test Automation technologies as software testers will undertake two weeks of induction/training for these technologies. The induction process will be divided accordingly to concepts learning and practical examples for each explained topic. Information about the theoretic topic covered and the projects to be delivered, can be found in the following chapters.

This program will be covered and monitored from Daniel Plaku and Flavio Bejko. They are responsible for lection organization, choosing the topics to be explained and study materials, track learning process for each student and evaluate their achievements and progress.

Lectures will be held 2 times/week for a total time period of eight weeks. This period could be extended with one extra session. Each lesson will last from one to one-and-half hour, depending on the required time needed to fulfill the training goals.

Monitoring the progress

The mentor will assure that the student will be in time with theoretical topics and the project tasks as planned.

During the week, the mentor will write down notes about the progress, topics that are completely covered, or those that seem more difficult to the students.

The last day of the week a test will be prepared that covers the topics that have been studied. Alternatively, a face to face exercise session may be held. Depending on the result of the test / exercise session, the induction program will continue as predicted or may be modified (add more time for those topics with difficulties and cut down time for those completely covered topics.) The mentor must keep in mind that the overall objectives of the induction program won't fail to comply.

1. Introduction to Testing Fundamentals

This chapter covers the basic theoretical knowledge a tester requires. It is aimed at anyone involved at software testing field, including people in roles such as testers, test analysts, test engineers, test consultants, test managers, user acceptance testers and software developers.

Expectations:

At the end of this session, student should be able to answer the questions related to testing notions. They should know the reason why we test, what are testing benefits and how is the workflow process.

1.1 What is testing?

The student learns the basic terminology related to testing, the reasons why testing is required, what objectives are and the principles of successful testing.

The student understands the test process, the major activities and artifacts.

The tester learns the test process itself and how it interacts with software development and maintenance life cycles.

Terms

Debugging, requirement, review, test case, testing, test objective

Background

A common perception of testing is that it only consists of running tests, i.e., executing the software. This is part of testing, but not all of the testing activities.

Test activities exist before and after test execution. These activities include planning and control, choosing test conditions, designing and executing test cases, checking results, evaluating exit criteria, reporting on the testing process and system under test, and finalizing or completing closure activities after a test phase has been completed. Testing also includes reviewing documents (including source code) and conducting static analysis.

Both dynamic testing and static testing can be used as a means for achieving similar objectives, and will provide information that can be used to improve both the system being tested and the development and testing processes.

Testing can have the following objectives:

- Finding defects
- Gaining confidence about the level of quality
- Providing information for decision-making ☐ Preventing defects

The thought process and activities involved in designing tests early in the life cycle (verifying the test basis via test design) can help to prevent defects from being introduced into code. Reviews of documents (e.g., requirements) and the identification and resolution of issues also help to prevent defects appearing in the code.

Different viewpoints in testing take different objectives into account. For example, in development testing (e.g., component, integration and system testing), the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. In acceptance testing, the main objective may be to confirm that the system works as expected, to gain confidence that it has met the requirements. In some cases the main objective of testing may be to assess the quality of the software (with no intention of fixing defects), to give information to stakeholders of the risk of releasing the system at a given time. Maintenance testing often includes testing that no new defects have been introduced during development of the changes. During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Debugging and testing are different. Dynamic testing can show failures that are caused by defects. Debugging is the development activity that finds, analyzes and removes the cause of the failure.

Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure. The responsibility for these activities is usually testers test and developers debug.

1.2 Why should you test?

Describe with examples, the way in which a defect in software can cause harm to a person, to the environment, or to a company.

Distinguish between the root cause of a defect and its effects.

Give reasons why testing is necessary by giving examples.

Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality.

Terms

Bug, defect, error, failure, fault, mistake, quality, risk

➤ Software Systems Context

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time or business reputation, and could even cause injury or death.

➤ Causes of Software Defects

A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.

Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions.

Failures can be caused by environmental conditions as well. For example, radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing the hardware conditions.

➤ Role of Testing in Software Development, Maintenance and Operations

Rigorous testing of systems and documentation can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if the defects found are corrected before the system is released for operational use.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

➤ Testing and Quality

With the help of testing, it is possible to measure the quality of software in terms of defects found, for both functional and non-functional software requirements and characteristics (e.g., reliability, usability, efficiency, maintainability and portability). For more information on software characteristics see 'Software Engineering – Software Product Quality' (ISO 9126).

Testing can give confidence in the quality of the software if it finds few or no defects. A properly designed test that passes reduces the overall level of risk in a system. When testing does find defects, the quality of the software system increases when those defects are fixed.

Lessons should be learned from previous projects. By understanding the root causes of defects found in other projects, processes can be improved, which in turn should prevent those defects from reoccurring and, as a consequence, improve the quality of future systems. This is an aspect of quality assurance.

Testing should be integrated as one of the quality assurance activities (i.e., alongside development standards, training and defect analysis).

➤ How Much Testing is enough?

Deciding how much testing is enough should take account of the level of risk, including technical, safety, and business risks, and project constraints such as time and budget.

Testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system being tested, for the next development step or handover to customers.

1.3 Automation Testing Vs. Manual Testing: What's the Difference?

In this section, a tester learns the main differences between:

- Automation Testing
- Manual Testing.
- Difference Between Manual Testing and Automation Testing
- Manual Testing Pros and Cons
- Automated Testing Pros and Cons

➤ What is Manual Testing?

Manual testing is testing of the software where tests are executed manually by a QA Analysts. It is performed to discover bugs in software under development.

In Manual testing, the tester checks all the essential features of the given application or software. In this process, the software testers execute the test cases and generate the test reports without the help of any automation software testing tools.

It is a classical method of all testing types and helps find bugs in software systems. It is generally conducted by an experienced tester to accomplish the software testing process.

➤ What is Automation Testing?

In Automated Software Testing, testers write code/test scripts to automate test execution. Testers use appropriate automation tools to develop the test scripts and validate the software. The goal is to complete test execution in a less amount of time.

Automated testing entirely relies on the pre-scripted test which runs automatically to compare actual result with the expected results. This helps the tester to determine whether or not an application performs as expected.

Automated testing allows you to execute repetitive task and regression test without the intervention of manual tester. Even though all processes are performed automatically, automation requires some manual effort to create initial testing scripts.

Manual Testing



Automated Testing



VS

➤ Difference Between Manual Testing and Automation Testing

Parameter	Automation Testing	Manual Testing
Definition	Automation Testing uses automation tools to execute test cases.	In manual testing, test cases are executed by a human tester and software.
Processing time	Automated testing is significantly faster than a manual approach.	Manual testing is time-consuming and takes up human resources.
Exploratory Testing	Automation does not allow random testing	Exploratory testing is possible in Manual Testing
Initial investment	The initial investment in the automated testing is higher. Though the ROI is better in the long run.	The initial investment in the Manual testing is comparatively lower. ROI is lower compared to Automation testing in the long run.
Reliability	Automated testing is a reliable method, as it is performed by tools and scripts. There is no testing Fatigue.	Manual testing is not as accurate because of the possibility of the human errors.
UI Change	For even a trivial change in the UI of the AUT, Automated Test Scripts need to be modified to work as expected	Small changes like change in id, class, etc. of a button wouldn't thwart execution of a manual tester.
Investment	Investment is required for testing tools as well as automation engineers	Investment is needed for human resources.

Cost-effective	Not cost effective for low volume regression	Not cost effective for high volume regression.
Test Report Visibility	With automation testing, all stakeholders can login into the automation system and check test execution results	Manual Tests are usually recorded in an Excel or Word, and test results are not readily available.
Human observation	Automated testing does not involve human consideration. So it can never give assurance of user-friendliness and positive customer experience.	The manual testing method allows human observation, which may be useful to offer user-friendly system.
Performance Testing	Performance Tests like Load Testing, Stress Testing, Spike Testing, etc. have to be tested by an automation tool compulsorily.	Performance Testing is not feasible manually
Parallel Execution	This testing can be executed on different operating platforms in parallel and reduce test execution time.	Manual tests can be executed in parallel but would need to increase your human resource which is expensive
Batch testing	You can Batch multiple Test Scripts for nightly execution.	Manual tests can not be batched.
Programming knowledge	Programming knowledge is a must in automation testing.	No need for programming in Manual Testing.
Set up	Automation test requires less complex test execution set up.	Manual testing needs have a more straightforward test execution setup
Engagement	Done by tools. Its accurate and never gets bored!	Repetitive Manual Test Execution can get boring and error-prone.
Ideal approach	Automation testing is useful when frequently executing the same set of test cases	Manual testing proves useful when the test case only needs to run once or twice.
Build Verification Testing	Automation testing is useful for Build Verification Testing (BVT).	Executing the Build Verification Testing (BVT) is very difficult and time-consuming in manual testing.
Deadlines	Automated Tests have zero risks of missing out a pre-decided test.	Manual Testing has a higher risk of missing out the pre-decided test deadline.

Framework	Automation testing uses frameworks like Data Drive, Keyword, Hybrid to accelerate the automation process.	Manual Testing does not use frameworks but may use guidelines, checklists, stringent processes to draft certain test cases.
Documentation	Automated Tests acts as a document provides training value especially for automated unit test cases. A new developer can look into a unit test cases and understand the code base quickly.	Manual Test cases provide no training value
Test Design	Automated Unit Tests enforce/drive Test Driven Development Design.	Manual Unit Tests do not drive design into the coding process
Devops	Automated Tests help in Build Verification Testing and are an integral part of DevOps Cycle	Manual Testing defeats the automated build principle of DevOps
When to Use?	Automated Testing is suited for Regression Testing, Performance Testing, Load Testing or highly repeatable functional test cases.	Manual Testing is suitable for Exploratory, Usability and Adhoc Testing. It should also be used where the AUT changes frequently.

➤ Manual Testing Pros and Cons

➤ Pros of Manual Testing:

- Get fast and accurate visual feedback
- It is less expensive as you don't need to spend your budget for the automation tools and process
- Human judgment and intuition always benefit the manual element
- While testing a small change, an automation test would require coding which could be time-consuming. While you could test manually on the fly.

➤ Cons of Manual Testing:

- Less reliable testing method because it's conducted by a human. Therefore, it is always prone to mistakes & errors.
- The manual testing process can't be recorded, so it is not possible to reuse the manual test.
- In this testing method, certain tasks are difficult to perform manually which may require an additional time of the software testing phase.

➤ Automated Testing Pros and Cons

➤ Pros of automated testing:

- Automated testing helps you to find more bugs compare to a human tester
- As most of the part of the testing process is automated, you can have a speedy and efficient process
- Automation process can be recorded. This allows you to reuse and execute the same kind of testing operations
- Automated testing is conducted using software tools, so it works without tiring and fatigue unlike humans in manual testing
- It can easily increase productivity because it provides fast & accurate testing result
- Automated testing support various applications
- Testing coverage can be increased because of automation testing tool never forget to check even the smallest unit

➤ Cons of Automated Testing:

- Without human element, it's difficult to get insight into visual aspects of your UI like colors, font, sizes, contrast or button sizes.
- The tools to run automation testing can be expensive, which may increase the cost of the testing project.
- Automation testing tool is not yet foolproof. Every automation tool has their limitations which reduces the scope of automation.
- Debugging the test script is another major issue in the automated testing. Test maintenance is costly.

➤ KEY DIFFERENCE

- Manual Testing is done manually by QA analyst (Human) whereas Automation Testing is done with the use of script, code and automation tools (computer) by a tester.
- Manual Testing process is not accurate because of the possibilities of human errors whereas the Automation process is reliable because it is code and script based.
- Manual Testing is a time-consuming process whereas Automation Testing is very fast.
- Manual Testing is possible without programming knowledge whereas Automation Testing is not possible without programming knowledge.
- Manual Testing allows random Testing whereas Automation Testing doesn't allow random Testing.

1.4 Unit Tests, Integration Tests & End-to-End Tests

In this section, a tester learns about test levels, test types and impact analysis when working with maintenance testing.

➤ Unit Testing / Component Testing

Test basis:

- Component requirements
- Detailed design
- Code

Typical test objects:

- Components
- Programs
- Data conversion / migration programs
- Database modules

Component testing (also known as unit, module or program testing) searches for defects in, and verifies the functioning of, software modules, programs, objects, classes, etc., that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Stubs, drivers and simulators may be used.

Component testing may include testing of functionality and specific non-functional characteristics, such as resource-behavior (e.g., searching for memory leaks) or robustness testing, as well as structural testing (e.g., decision coverage). Test cases are derived from work products such as a specification of the component, the software design or the data model.

Typically, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. In practice, component testing usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects.

One approach to component testing is to prepare and automate test cases before coding. This is called a test-first approach or test-driven development. This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests correcting any issues and iterating until they pass.

➤ Integration Testing

Test basis:

- Software and system design
- Architecture
- Workflows
- Use cases

Typical test objects:

- Subsystems
- Database implementation
- Infrastructure
- Interfaces
- System configuration and configuration data

Integration testing tests interfaces between components, interactions with different parts of a system, such as the operating system, file system and hardware, and interfaces between systems.

There may be more than one level of integration testing and it may be carried out on test objects of varying size as follows:

1. Component integration testing tests the interactions between software components and is done after component testing
2. System integration testing tests the interactions between different systems or between hardware and software and may be done after system testing. In this case, the developing organization may control only one side of the interface. This might be considered as a risk. Business processes implemented as workflows may involve a series of systems. Cross-platform issues may be significant.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

Systematic integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang”.

Testing of specific non-functional characteristics (e.g., performance) may be included in integration testing as well as functional testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in testing the communication between the modules, not the functionality of the individual module as that was done during component testing. Both functional and structural approaches may be used.

Ideally, testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, those components can be built in the order required for most efficient testing.

➤ End-to-End Tests / System Testing

Test basis:

- System and software requirement specification
 - Use cases
 - Functional specification
 - Risk analysis reports
- Typical test objects:
- System, user and operation manuals
 - System configuration and configuration data

System testing is concerned with the behavior of a whole system/product. The testing scope shall be clearly addressed in the Master and/or Level Test Plan for that test level.

In system testing, the test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found in testing.

System testing may include tests based on risks and/or on requirements specifications, business processes, use cases, or other high level text descriptions or models of system behavior, interactions with the operating system, and system resources.

System testing should investigate functional and non-functional requirements of the system, and data quality characteristics. Testers also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based techniques (white-box) may then be used to assess the thoroughness of the testing with respect to a structural element, such as menu structure or web page navigation.

An independent test team often carries out system testing.

1.5 Tools for Testing

Classify different types of test tools according to their purpose and to the activities of the fundamental test process and the software life cycle.

Explain the term test tool and the purpose of tool support for testing.

Summarize the potential benefits and risks of test automation and tool support for testing

Terms

Configuration management tool, coverage tool, debugging tool, dynamic analysis tool, incident management tool, load testing tool, modeling tool, monitoring tool, performance testing tool, probe effect, requirements management tool, review tool, security tool, static analysis tool, stress testing tool, test comparator, test data preparation tool, test design tool, test harness, test execution tool, test management tool, unit test framework tool

➤ Types of Test Tools

- Tool Support for Testing

Test tools can be used for one or more activities that support testing. These include:

1. Tools that are directly used in testing such as test execution tools, test data generation tools and result comparison tools
2. Tools that help in managing the testing process such as those used to manage tests, test results, data, requirements, incidents, defects, etc., and for reporting and monitoring test execution
3. Tools that are used in reconnaissance, or, in simple terms: exploration (e.g., tools that monitor file activity for an application)
4. Any tool that aids in testing (a spreadsheet is also a test tool in this meaning)

Tool support for testing can have one or more of the following purposes depending on the context:

- Improve the efficiency of test activities by automating repetitive tasks or supporting manual test activities like test planning, test design, test reporting and monitoring
- Automate activities that require significant resources when done manually (e.g., static testing)
- Automate activities that cannot be executed manually (e.g., large scale performance testing of client-server applications)
- Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

The term “test frameworks” is also frequently used in the industry, in at least three meanings:

- Reusable and extensible testing libraries that can be used to build testing tools (called test harnesses as well)
- A type of design of test automation (e.g., data-driven, keyword-driven)
- Overall process of execution of testing

For the purpose of this syllabus, the term “test frameworks” is used in its first two meanings as described in Section 6.1.6.

- **Test Tool Classification**

There are a number of tools that support different aspects of testing. Tools can be classified based on several criteria such as purpose, commercial / free / open-source / shareware, technology used and so forth. Tools are classified in this syllabus according to the testing activities that they support.

Some tools clearly support one activity; others may support more than one activity, but are classified under the activity with which they are most closely associated. Tools from a single provider, especially those that have been designed to work together, may be bundled into one package.

Some types of test tools can be intrusive, which means that they can affect the actual outcome of the test. For example, the actual timing may be different due to the extra instructions that are executed by the tool, or you may get a different measure of code coverage. The consequence of intrusive tools is called the probe effect.

Some tools offer support more appropriate for developers (e.g., tools that are used during component and component integration testing). Such tools are marked with “(D)” in the list below.

- **Tool Support for Management of Testing and Tests**

Management tools apply to all test activities over the entire software life cycle.

Test Management Tools

These tools provide interfaces for executing tests, tracking defects and managing requirements, along with support for quantitative analysis and reporting of the test objects. They also support tracing the test objects to requirement specifications and might have an independent version control capability or an interface to an external one.

Requirements Management Tools

These tools store requirement statements, store the attributes for the requirements (including priority), provide unique identifiers and support tracing the requirements to individual tests. These tools may also help with identifying inconsistent or missing requirements.

Incident Management Tools (Defect Tracking Tools)

These tools store and manage incident reports, i.e., defects, failures, change requests or perceived problems and anomalies, and help in managing the life cycle of incidents, optionally with support for statistical analysis.

Configuration Management Tools

Although not strictly test tools, these are necessary for storage and version management of testware and related software especially when configuring more than one hardware/software environment in terms of operating system versions, compilers, browsers, etc.

- Tool Support for Static Testing

Static testing tools provide a cost effective way of finding more defects at an earlier stage in the development process.

Review Tools

These tools assist with review processes, checklists, review guidelines and are used to store and communicate review comments and report on defects and effort. They can be of further help by providing aid for online reviews for large or geographically dispersed teams.

Static Analysis Tools (D)

These tools help developers and testers find defects prior to dynamic testing by providing support for enforcing coding standards (including secure coding), analysis of structures and dependencies. They can also help in planning or risk analysis by providing metrics for the code (e.g., complexity).

Modeling Tools (D)

These tools are used to validate software models (e.g., physical data model (PDM) for a relational database), by enumerating inconsistencies and finding defects. These tools can often aid in generating some test cases based on the model.

- Tool Support for Test Specification

Test Design Tools

These tools are used to generate test inputs or executable tests and/or test oracles from requirements, graphical user interfaces, design models (state, data or object) or code.

Test Data Preparation Tools

Test data preparation tools manipulate databases, files or data transmissions to set up test data to be used during the execution of tests to ensure security through data anonymity.

- [Tool Support for Test Execution and Logging](#)

Test Execution Tools

These tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language and usually provide a test log for each test run. They can also be used to record tests, and usually support scripting languages or GUI-based configuration for parameterization of data and other customization in the tests.

Test Harness/Unit Test Framework Tools (D)

A unit test harness or framework facilitates the testing of components or parts of a system by simulating the environment in which that test object will run, through the provision of mock objects as stubs or drivers.

Test Comparators

Test comparators determine differences between files, databases or test results. Test execution tools typically include dynamic comparators, but post-execution comparison may be done by a separate comparison tool. A test comparator may use a test oracle, especially if it is automated.

Coverage Measurement Tools (D)

These tools, through intrusive or non-intrusive means, measure the percentage of specific types of code structures that have been exercised (e.g., statements, branches or decisions, and module or function calls) by a set of tests.

Security Testing Tools

These tools are used to evaluate the security characteristics of software. This includes evaluating the ability of the software to protect data confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Security tools are mostly focused on a particular technology, platform, and purpose.

- [Tool Support for Performance and Monitoring](#)

Dynamic Analysis Tools (D)

Dynamic analysis tools find defects that are evident only when software is executing, such as time dependencies or memory leaks. They are typically used in component and component integration testing, and when testing middleware.

Performance Testing/Load Testing/Stress Testing Tools

Performance testing tools monitor and report on how a system behaves under a variety of simulated usage conditions in terms of number of concurrent users, their ramp-up pattern, frequency and relative

percentage of transactions. The simulation of load is achieved by means of creating virtual users carrying out a selected set of transactions, spread across various test machines commonly known as load generators.

Monitoring Tools

Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems.

- Tool Support for Specific Testing Needs

Data Quality Assessment

Data is at the center of some projects such as data conversion/migration projects and applications like data warehouses and its attributes can vary in terms of criticality and volume. In such contexts, tools need to be employed for data quality assessment to review and verify the data conversion and migration rules to ensure that the processed data is correct, complete and complies with a pre- defined context-specific standard.

Other testing tools exist for usability testing.

➤ Effective Use of Tools: Potential Benefits and Risks

Terms

Data-driven testing, keyword-driven testing, scripting language

- Potential Benefits and Risks of Tool Support for Testing (for all tools)

Simply purchasing or leasing a tool does not guarantee success with that tool. Each type of tool may require additional effort to achieve real and lasting benefits. There are potential benefits and opportunities with the use of tools in testing, but there are also risks.

Potential benefits of using tools include:

- Repetitive work is reduced (e.g., running regression tests, re-entering the same test data, and checking against coding standards)
- Greater consistency and repeatability (e.g., tests executed by a tool in the same order with the same frequency, and tests derived from requirements)
- Objective assessment (e.g., static measures, coverage)

- Ease of access to information about tests or testing (e.g., statistics and graphs about test progress, incident rates and performance) Risks of using tools include:
- Unrealistic expectations for the tool (including functionality and ease of use)
- Underestimating the time, cost and effort for the initial introduction of a tool (including training and external expertise)
- Underestimating the time and effort needed to achieve significant and continuing benefits from the tool (including the need for changes in the testing process and continuous improvement of the way the tool is used)
- Underestimating the effort required to maintain the test assets generated by the tool
- Over-reliance on the tool (replacement for test design or use of automated testing where manual testing would be better)
- Neglecting version control of test assets within the tool
- Neglecting relationships and interoperability issues between critical tools, such as requirements management tools, version control tools, incident management tools, defect tracking tools and tools from multiple vendors
- Risk of tool vendor going out of business, retiring the tool, or selling the tool to a different vendor
- Poor response from vendor for support, upgrades, and defect fixes
- Risk of suspension of open-source / free tool project
- Unforeseen, such as the inability to support a new platform

- **Special Considerations for Some Types of Tools**

Test Execution Tools

Test execution tools execute test objects using automated test scripts. This type of tool often requires significant effort in order to achieve significant benefits.

Capturing tests by recording the actions of a manual tester seems attractive, but this approach does not scale to large numbers of automated test scripts. A captured script is a linear representation with specific data and actions as part of each script. This type of script may be unstable when unexpected events occur.

A data-driven testing approach separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic test script that can read the input data and execute the same test script with different data. Testers who are not familiar with the scripting language can then create the test data for these predefined scripts.

There are other techniques employed in data-driven techniques, where instead of hard-coded data combinations placed in a spreadsheet, data is generated using algorithms based on configurable parameters at run time and supplied to the application. For example, a tool may use an algorithm, which generates a random user ID, and for repeatability in pattern, a seed is employed for controlling randomness.

In a keyword-driven testing approach, the spreadsheet contains keywords describing the actions to be taken (also called action words), and test data. Testers (even if they are not familiar with the scripting language) can then define tests using the keywords, which can be tailored to the application being tested.

Technical expertise in the scripting language is needed for all approaches (either by testers or by specialists in test automation).

Regardless of the scripting technique used, the expected results for each test need to be stored for later comparison.

Static Analysis Tools

Static analysis tools applied to source code can enforce coding standards, but if applied to existing code may generate a large quantity of messages. Warning messages do not stop the code from being translated into an executable program, but ideally should be addressed so that maintenance of the code is easier in the future. A gradual implementation of the analysis tool with initial filters to exclude some messages is an effective approach.

Test Management Tools

Test management tools need to interface with other tools or spreadsheets in order to produce useful information in a format that fits the needs of the organization.

➤ Introducing a Tool into an Organization

Terms

No specific terms.

Background

The main considerations in selecting a tool for an organization include:

- Assessment of organizational maturity, strengths and weaknesses and identification of opportunities for an improved test process supported by tools
- Evaluation against clear requirements and objective criteria
- A proof-of-concept, by using a test tool during the evaluation phase to establish whether it performs effectively with the software under test and within the current infrastructure or to identify changes needed to that infrastructure to effectively use the tool
- Evaluation of the vendor (including training, support and commercial aspects) or service support suppliers in case of non-commercial tools
- Identification of internal requirements for coaching and mentoring in the use of the tool
- Evaluation of training needs considering the current test team's test automation skills
- Estimation of a cost-benefit ratio based on a concrete business case

Introducing the selected tool into an organization starts with a pilot project, which has the following objectives:

- Learn more detail about the tool
- Evaluate how the tool fits with existing processes and practices, and determine what would need to change
- Decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g., deciding on naming conventions for files and tests, creating libraries and defining the modularity of test suites)
- Assess whether the benefits will be achieved at reasonable cost

Success factors for the deployment of the tool within an organization include:

- Rolling out the tool to the rest of the organization incrementally
- Adapting and improving processes to fit with the use of the tool
- Providing training and coaching/mentoring for new users
- Defining usage guidelines
- Implementing a way to gather usage information from the actual use
- Monitoring tool use and benefits
- Providing support for the test team for a given tool
- Gathering lessons learned from all teams

Glossary

Debugging - The process of finding, analyzing and removing the causes of failures in software.
Requirement- A provision that contains criteria to be fulfilled.

Review- A type of static testing during which a work product or process is evaluated by one or more individuals to detect issues and to provide improvements.

Test case - A set of preconditions, inputs, actions (where applicable), expected results and post-conditions, developed based on test conditions.

Testing - The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

Test objective - A reason or purpose for designing and executing a test.

Defect - An imperfection or deficiency in a work product where it does not meet its requirements or specifications.

Failure- An event in which a component or system does not perform a required function within specified limits.

Quality - The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

Risk - A factor that could result in future negative consequences.

Alpha testing - Simulated or actual operational testing conducted in the developer's test environment, by roles outside the development organization.

Beta testing - Simulated or actual operational testing conducted at an external site, by roles outside the development organization.

Component Testing - The testing of individual hardware or software components. Synonyms: module testing, unit testing.

Functional requirement - A requirement that specifies a function that a component or system must be able to perform.

Integration testing - Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

Non-functional requirement - A requirement that describes how the component or system will do what it is intended to do.

System testing - Testing an integrated system to verify that it meets specified requirements.

Test environment - An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.

Test level - A specific instantiation of a test process.

Test-driven development - A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

User acceptance testing - Acceptance testing conducted in a real or simulated operational environment by intended users focusing their needs, requirements and business processes.

Configuration management tool - A tool that provides support for the identification and control of configuration items, their status over changes and versions, and the release of baselines consisting of configuration items.

Coverage tool - A tool that provides objective measures of what structural elements, e.g., statements, branches have been exercised by a test suite.

Debugging tool - A tool used by programmers to reproduce failures, investigate the state of programs and find the corresponding defect. Debuggers enable programmers to execute programs step by step, to halt a program at any program statement and to set and examine program variables.

Dynamic analysis tool - A tool that provides run-time information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic and to monitor the allocation, use and de-allocation of memory and to flag memory leaks.

Probe effect - The effect on the component or system by the measurement instrument when the component or system is being measured, e.g., by a performance testing tool or monitor. For example performance may be slightly worse when performance testing tools are being used.

Static analysis tool - The process of evaluating a component or system without executing it, based on its form, structure, content, or documentation.

Test design tool - A tool that supports the test design activity by generating test inputs from a specification that may be held in a CASE tool repository, e.g., requirements management tool, from specified test conditions held in the tool itself, or from code.

Test execution tool - A test tool that executes tests against a designated test item and evaluates the outcomes against expected results and post-conditions.

Test management tool - A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as test-ware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting.

Unit test framework tool - A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities.

References

- International Software Testing Qualifications Board (ISTQB), Certified Tester Foundation Level Syllabus", (Released Version 2011).

2. Element Selectors

One of the most important aspects of test automation is to be able to identify different elements on the page to interact with. Without the ability to find those elements, it's not possible to do any interaction with the page. Therefore, this chapter covers all the main methods used for locating elements on a web page.

2.1 Locating by CSS Selector

CSS Selectors are string patterns used to identify an element based on a combination of HTML tag, id, class, and attributes. Locating by CSS Selector is more complicated than the previous methods, but it is the most common locating strategy of advanced Selenium users because it can access even those elements that have no ID or name.

CSS Selectors have many formats, but we will only focus on the most common ones.

- Tag and ID
- Tag and class
- Tag and attribute
- Tag, class, and attribute
- Inner text

When using this strategy, we always prefix the Target box with "css=" as will be shown in the following examples.

1. Locating by CSS Selector - Tag and ID

Again, we will use Facebook's Email text box in this example. As you can remember, it has an ID of "email," and we have already accessed it in the "Locating by ID" section. This time, we will use a CSS Selector with ID in accessing that very same element.

Syntax	Description
<i>css=tag#id</i>	<ul style="list-style-type: none">□ tag = the HTML tag of the element being accessed□ # = the hash sign. This should always be present when using a CSS Selector with ID

- id = the ID of the element being accessed

Keep in mind that the ID is always preceded by a hash sign (#).

2. Locating by CSS Selector - Tag and Class

Locating by CSS Selector using an HTML tag and a class name is similar to using a tag and ID, but in this case, a dot (.) is used instead of a hash sign.

Syntax	Description
<i>css=tag.class</i>	<ul style="list-style-type: none"> • tag = the HTML tag of the element being accessed • . = the dot sign. This should always be present when using a CSS Selector with class
	<ul style="list-style-type: none"> □ class = the class of the element being accessed

3. Locating by CSS Selector - Tag and Attribute

This strategy uses the HTML tag and a specific attribute of the element to be accessed.

Syntax	Description
<i>css=tag[attribute=value]</i>	<ul style="list-style-type: none"> □ tag = the HTML tag of the element being accessed □ [and] = square brackets within which a specific attribute and its corresponding value will be placed □ attribute = the attribute to be used. It is advisable to use an attribute that is unique to the element such as a name or ID. □ value = the corresponding value of the chosen attribute.

4. Locating by CSS Selector - tag, class, and attribute

Syntax	Description
<i>css=tag.class[attribute=value]</i>	<ul style="list-style-type: none"> □ tag = the HTML tag of the element being accessed

- . = the dot sign. This should always be present when using a CSS Selector with class
- class = the class of the element being accessed
- [and] = square brackets within which a specific attribute and its corresponding value will be placed
- attribute = the attribute to be used. It is advisable to use an attribute that is unique to the element such as a name or ID.
- value = the corresponding value of the chosen attribute.

5. Locating by CSS Selector - inner text

As you may have noticed, HTML labels are seldom given id, name, or class attributes. So, how do we access them? The answer is through the use of their inner texts. **Inner texts are the actual string patterns that the HTML label shows on the page.**

Syntax	Description
css=tag:contains("inner text")	<ul style="list-style-type: none"> □ tag = the HTML tag of the element being accessed □ inner text = the inner text of the element

2.2.2 CSS Attribute Selectors

1. CSS [attribute] Selector

The **[attribute]** selector is used to select elements with a specified attribute.

The following example selects all <a> elements with a target attribute:

Example

```
a[target]
{
background-color: yellow;
}
```

3. CSS [attribute~="value"] Selector

The `[attribute~="value"]` selector is used to select elements with an attribute value containing a specified word.

The following example selects all elements with a title attribute that contains a space-separated list of words, one of which is "flower":

Example

`[title~="flower"]`

```
{  
  border: 5px solid yellow;  
}
```

4. CSS [attribute|="value"] Selector

The `[attribute|="value"]` selector is used to select elements with the specified attribute starting with the specified value.

The following example selects all elements with a class attribute value that begins with "top":

Note: The value has to be a whole word, either alone, like `class="top"`, or followed by a hyphen (-), like `class="top-text"`!

Example

`[class|="top"]`

```
{  
  background: yellow;  
}
```

5. CSS [attribute^="value"] Selector

The `[attribute^="value"]` selector is used to select elements whose attribute value begins with a specified value.

The following example selects all elements with a class attribute value that begins with "top":

Note: The value does not have to be a whole word!

Example

```
[class^="top"]
```

```
{  
  background: yellow;  
}
```

6. CSS [attribute\$="value"] Selector

The `[attribute$="value"]` selector is used to select elements whose attribute value ends with a specified value.

The following example selects all elements with a class attribute value that ends with "test":

Note: The value does not have to be a whole word!

Example

```
[class$="test"]
```

```
{  
  background: yellow; }
```

7. CSS [attribute*="value"] Selector

The `[attribute*="value"]` selector is used to select elements whose attribute value contains a specified value.

The following example selects all elements with a class attribute value that contains "te":

Note: The value does not have to be a whole word!

Example

```
[class*="te"]
```

```
{  
  background: yellow;  
}
```

2.2.3 Pseudo-classes and pseudo-elements

2.2.3.1 Pseudo-classes

A CSS *pseudo-class* is a keyword added to a selector that specifies a special state of the selected element(s).

For example, `:hover` can be used to change a button's color when the user's pointer hovers over it.

```
/* Any button over which the user's pointer is hovering */ button:hover
{
  color: blue;
}
```

Pseudo-classes let you apply a style to an element not only in relation to the content of the document tree, but also in relation to external factors like the history of the navigator (`:visited`, for example), the status of its content (like `:checked` on certain form elements), or the position of the mouse (like `:hover`, which lets you know if the mouse is over an element or not).

Syntax

```
selector:pseudo-class { property:
value;
}
```

Like regular classes, you can chain together as many pseudo-classes as you want in a selector.

2.2.3.2 Pseudo-elements

A CSS **pseudo-element** is a keyword added to a selector that lets you style a specific part of the selected element(s).

For example, `::first-line` can be used to change the font of the first line of a paragraph.

```
/* The first line of every <p> element. */
p::first-line {
  color: blue;
  text-transform: uppercase;
}
```

Note: In contrast to pseudo-elements, [pseudo-classes](#) can be used to style an element based on its *state*.

Syntax

```
selector::pseudo-element { property:
value;
}
```

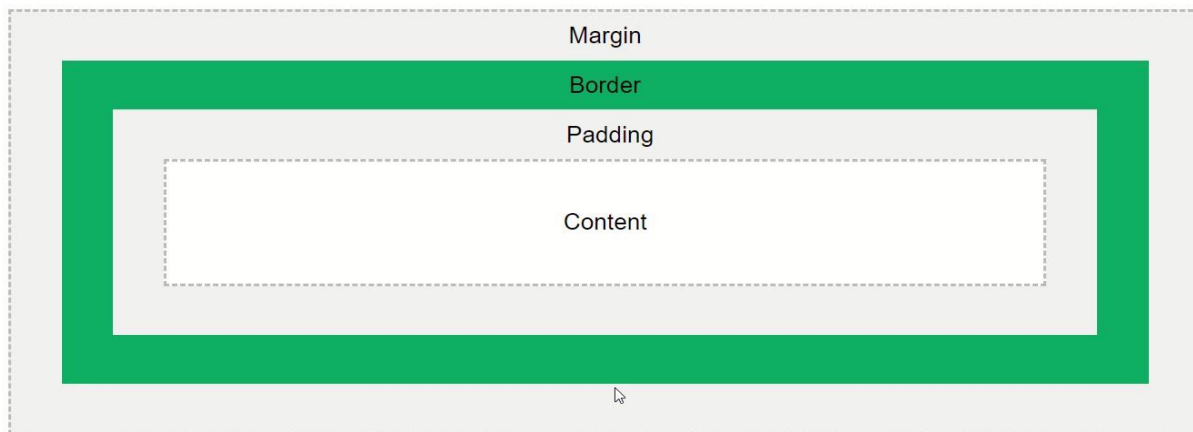

You can use only one pseudo-element in a selector. It must appear after the simple selectors in the statement.

Note: As a rule, double colons (::) should be used instead of a single colon (:). This distinguishes pseudoclasses from pseudo-elements. However, since this distinction was not present in older versions of the W3C spec, most browsers support both syntaxes for the original pseudo-elements.

1.2.4 The CSS box model

All HTML elements can be considered as boxes. In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content. The image below illustrates the box model:



Explanation of the different parts:

- **Content** - The content of the box, where text and images appear
- **Padding** - Clears an area around the content. The padding is transparent
- **Border** - A border that goes around the padding and content
- **Margin** - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.

Example

```
div { width: 300px;  
border: 25px solid green;  
padding: 25px;  
margin: 25px;  
}
```

2.2. Other selectors

➤ ID selector

This is the most common way of locating elements since ID's are supposed to be unique for each element. Use this when you know *id* attribute of an element. With this strategy, the first element with the *id* attribute value matching the location will be returned. If no element has a matching *id* attribute, a `NoSuchElementException` will be raised.

Target Format: `id=id of the element`

➤ Class Name selector

Locating elements by name are very similar to locating by ID, except that we use the "**name=**" prefix instead.

Target Format: `name=name of the element`

➤ Tag Name selector

Use this when you want to locate an element by tag name. With this strategy, the first element with the given tag name will be returned. If no element has a matching tag name, a `NoSuchElementException` will be raised.

➤ Link Text selector

Use this when you know link text used within an anchor tag. If there are multiple elements with the same link text then the first one will be selected. If no element has a matching link text attribute, a `NoSuchElementException` will be raised.

Target Format: `link=link_text`

➤ Partial Link Text selector

In some situations, we may need to find links by a portion of the text in a Link Text element. it contains. In such situations, we use Partial Link Text to locate elements.

Summary for locating elements

Variation	Description	Sample
By.className	finds elements based on the value of the "class" attribute	<code>findElement(By.className("someClassName"))</code>
By.cssSelector	finds elements based on the driver's underlying CSS Selector engine	<code>findElement(By.cssSelector("input#email"))</code>
By.id	locates elements by the value of their "id" attribute	<code>findElement(By.id("someId"))</code>
By.linkText	finds a link element by the exact text it displays	<code>findElement(By.linkText("REGISTRATION"))</code>
By.name	locates elements by the value of the "name" attribute	<code>findElement(By.name("someName"))</code>
By.partialLinkText	locates elements that contain the given link text	<code>findElement(By.partialLinkText("REG"))</code>
By.tagName	locates elements by their tag name	<code>findElement(By.tagName("div"))</code>
By.xpath	locates elements via XPath	<code>findElement(By.xpath("//html/body/div/table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[2]/td[3]/form/table/tbody/tr[5]"))</code>

References

- CSS Tutorial. Available: <https://www.w3schools.com/css/>
- Selenium Tutorial. Available: <https://www.guru99.com/selenium-tutorial.html>
- CSS: Cascading Style Sheets. Available: <https://developer.mozilla.org/en-US/docs/Web/CSS>
- <https://www.softwaretestingmaterial.com/>
- <https://selenium-python.readthedocs.io/>

3. Introduction to Selenium

3.1 Selenium WebDriver, Installation and Configuration

WebDriver is a web automation framework that allows you to **execute your tests against different browsers**.

WebDriver also enables you to **use a programming language** in creating your test scripts.

You can now use **conditional operations** like if-then-else or switch-case. You can also perform **looping** like do-while.

Following programming languages are supported by WebDriver:

- Java
- .Net
- PHP
- Python
- Perl
- Ruby

In this section, students will install Webdriver (Java only) and Configure Eclipse.

Step 1 - Install Java on your computer

Download and install the **Java Software Development Kit (JDK)**

This JDK version comes bundled with Java Runtime Environment (JRE), so you do not need to download and install the JRE separately.

Step 2 - Install Eclipse IDE


Download latest version of "**Eclipse IDE for Java Developers**". Be sure to choose correctly between Windows 32 Bit and 64 Bit versions.

You should be able to download an exe file named "eclipse-inst-win64" for Setup.

Step 3 - Download the Selenium Java Client Driver

You can download the **Selenium Java Client Driver** on Selenium official page. You will find client drivers for other languages there, but only choose the one for Java.

this is the download for Java client Driver



Language	Client Version	Release Date	Download	Change log	Javadoc
Java	3.0.1	2016-10-18	Download	Change log	Javadoc
C#	3.0.0	2016-10-13	Download	Change log	API docs
Ruby	3.0.0	2016-10-13	Download	Change log	API docs
Python	3.0.2	2016-11-29	Download	Change log	API docs
Javascript (Node)	3.0.0-beta-2	2016-08-07	Download	Change log	API docs

This download comes as a ZIP file named "selenium-2.25.0.zip". For simplicity, extract the contents of this ZIP file on your C drive so that you would have the directory "C:\selenium-2.25.0\". This directory contains all the JAR files that we would later import on Eclipse.

Step 4 - Configure Eclipse IDE with WebDriver

1. Launch the "eclipse.exe" file inside the "eclipse" folder that we extracted in step 2. If you followed step 2 correctly, the executable should be located on C:\eclipse\eclipse.exe.
2. When asked to select for a workspace, just accept the default location.
3. Create a new project through File > New > Java Project. Name the project as "newproject".

A new pop-up window will open enter details as follow

- Project Name
- Location to save project
- Select an execution JRE
- Select layout project option
- Click on Finish button

4. In this step,

- Right-click on the newly created project and
- Select New > Package, and name that package as "newpackage".

A pop-up window will open to name the package,

- Enter the name of the package
- Click on Finish button

5. Create a new Java class under newpackage by right-clicking on it and then selecting- New > Class, and then name it as "MyClass". Your Eclipse IDE should look like the image below.



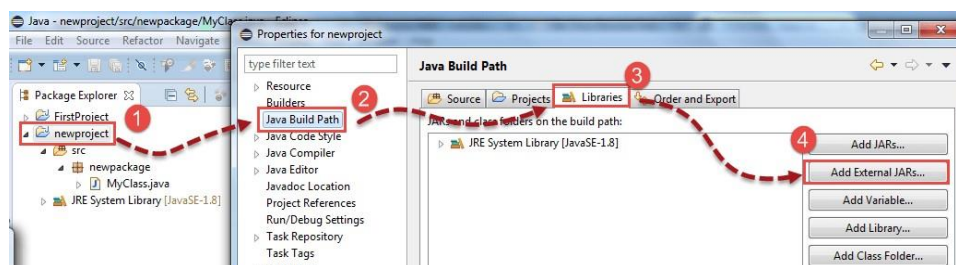
When you click on Class, a pop-up window will open, enter details as

- Name of the class
- Click on Finish button

Now selenium WebDriver's into Java Build Path

In this step,

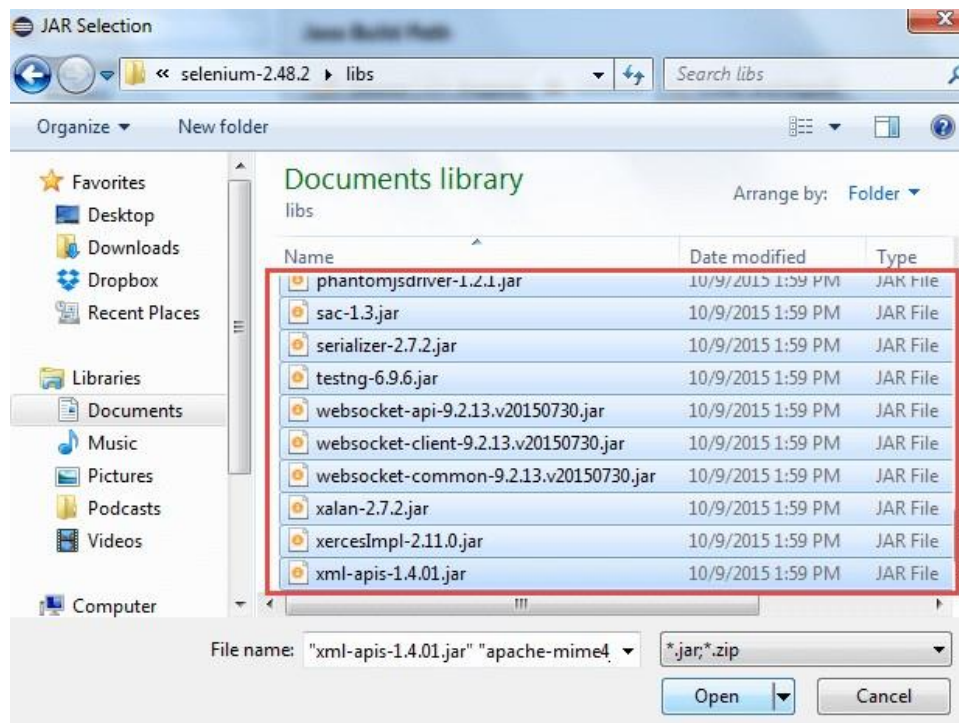
4. Right-click on "newproject" and select **Properties**.
5. On the Properties dialog, click on "Java Build Path".
6. Click on the **Libraries** tab, and then
7. Click on "Add External JARs.."



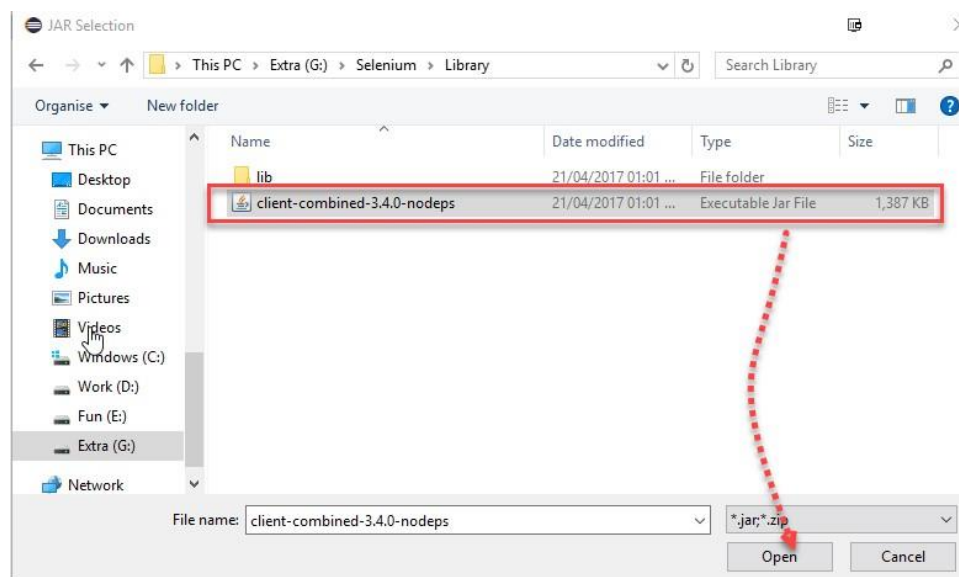
When you click on "Add External JARs.." It will open a pop-up window. Select the JAR files you want to add.

After selecting jar files, click on OK button.

Select all files inside the lib folder.

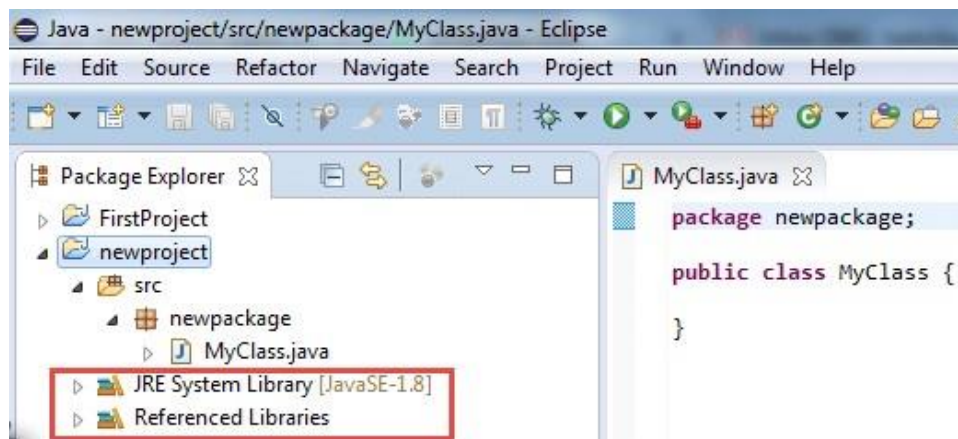


Select files outside lib folder



Once done, click "Apply and Close" button

6. Add all the JAR files inside and outside the "libs" folder. Your Properties dialog should now look similar to the image below.



7. Finally, click OK and we are done importing Selenium libraries into our project.

Summary

Aside from a browser, you will need the following to start using WebDriver

- **Java DevelopmentKit (JDK).**

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

- **Eclipse IDE** - <http://www.eclipse.org/downloads/>
- **Java Client Driver** - <http://seleniumhq.org/download/>

When starting a WebDriver project in Eclipse, do not forget to import the Java Client Driver files onto your project. These files will constitute your Selenium Library.

3.2. First Script

Using the configuration that we created in the previous section, students will create a WebDriver script that would:

- Fetch Mercury Tours' homepage
- Verify its title
- Print out the result of the comparison
- Close it before ending the entire program.

WebDriver Code

Below is the actual WebDriver code for the logic presented by the scenario above

```
package newproject;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

//comment the above line and uncomment below line to use Chrome
//import org.openqa.selenium.chrome.ChromeDriver;

public class PG1 {

    public static void main(String[] args) {

        // declaration and instantiation of objects/variables
        System.setProperty("webdriver.firefox.marionette", "C:\\\\geckodriver.exe");
        WebDriver driver = new FirefoxDriver();

        //comment the above 2 lines and uncomment below 2 lines to use Chrome
        //System.setProperty("webdriver.chrome.driver", "G:\\\\chromedriver.exe");
        //WebDriver driver = new ChromeDriver();

        String baseUrl = "http://demo.guru99.com/test/newtours/";
        String expectedTitle = "Welcome: Mercury Tours";

        String actualTitle = "";

        // launch Fire fox and direct it to the Base URL
        driver.get(baseUrl);

        // get the actual value of the title
        actualTitle = driver.getTitle();

        /*
        *      compare the actual title of the page with the expected one
        and print
        *      the result as "Passed" or "Failed"          */

        if (actualTitle.contentEquals(expectedTitle)){
            System.out.println("Test Passed!");
        } else {
            System.out.println("Test Failed");
        }
        //close Firefox
        driver.close();
    }
}
```

Explaining the code

Importing Packages

To get started, you need to import following two packages:

1. **org.openqa.selenium.***- contains the WebDriver class needed to instantiate a new browser loaded with a specific driver
2. **org.openqa.selenium.firefox.FirefoxDriver** - contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class

If your test needs more complicated actions such as accessing another class, taking browser screenshots, or manipulating external files, definitely you will need to import more packages.

Instantiating objects and variables

Normally, this is how a driver object is instantiated.

```
WebDriver driver = new FirefoxDriver();
```

A FirefoxDriver class with no parameters means that the default Firefox profile will be launched by our Java program. The default Firefox profile is similar to launching Firefox in safe mode (no extensions are loaded).

For convenience, we saved the Base URL and the expected title as variables.

Launching a Browser Session

WebDriver's **get()** method is used to launch a new browser session and directs it to the URL that you specify as its parameter.

```
driver.get(baseUrl);
```

Get the Actual Page Title

The WebDriver class has the **getTitle()** method that is always used to obtain the page title of the currently loaded page.

```
actualTitle = driver.getTitle();
```

Compare the Expected and Actual Values

This portion of the code simply uses a basic Java if-else structure to compare the actual title with the expected one.

```

if (actualTitle.equals(expectedTitle)) {
    System.out.println("Test Passed!");
} else {
    System.out.println("Test Failed!");
}

```

Terminating a Browser Session

The "**close()**" method is used to close the browser window.

```
driver.close();
```

Terminating the Entire Program

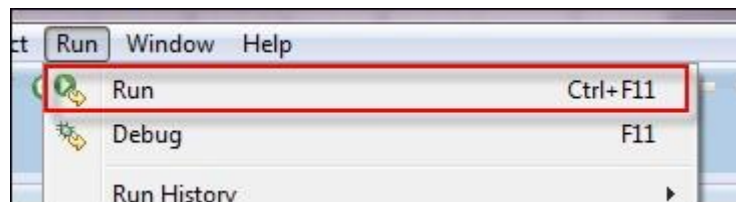
If you use this command without closing all browser windows first, your whole Java program will end while leaving the browser window open.

```
System.exit(0);
```

Running the Test

There are two ways to execute code in Eclipse IDE.

1. On Eclipse's menu bar, click **Run > Run**.
2. Press **Ctrl+F11** to run the entire code.



If you did everything correctly, Eclipse would output "Test Passed!"



3.3 Introduction to WebElement, findElement(), findElements()

Forms are the fundamental web elements to receive information from the website visitors. Web forms have different GUI elements like Text boxes, Password fields, Checkboxes, Radio buttons, dropdowns, file inputs, etc.

The students will learn how to access these different form elements using Selenium Web Driver with Java. **Selenium encapsulates every form element as an object of WebElement.** It provides API to find the elements and take action on them like entering text into text boxes, clicking the buttons, etc. The students will see the methods that are available to access each form element.

In this section, students will see how to identify the following form elements:

- Input Box
- Entering Values in Input Boxes
- Deleting Values in Input Boxes
- Buttons
- Submit Buttons
- Radio Button
- Check Box
- getText() and getAttribute()
- Switch to new tab in Selenium
- Select Option from Drop-Down
- Checking the WebElement state: isDisplayed(), isEnabled(), isSelected()
- Accessing links and web tables
- Upload & Download File

Selenium Web Driver encapsulates a simple form element as an object of **WebElement**.

There are various techniques by which the WebDriver identifies the form elements based on the different properties of the Web elements like ID, Name, Class, XPath, Tagname, CSS Selectors, link Text, etc.

Web Driver provides the following two methods to find the elements.

- **findElement()** – finds a single web element and returns as a WebElement object.
- **findElements()** – returns a list of WebElement objects matching the locator criteria.

Let's see the code snippets to get a single element – Text Field in a web page as an object of WebElement using findElement() method. We shall cover the findElements() method of finding multiple elements in subsequent tutorials.

Step 1: We need to import this package to create objects of Web Elements

```
import org.openqa.selenium.WebElement;
```

Step 2: We need to call the `findElement()` method available on the `WebDriver` class and get an object of `WebElement`.

Refer below to see how it is done.

- **Input Box**

Input boxes refer to either of these two types:

1. **Text Fields**- text boxes that accept typed values and show them as they are.
2. **Password Fields**- text boxes that accept typed values but mask them as a series of special characters (commonly dots and asterisks) to avoid sensitive values to be displayed.



Locators

The method `findElement()` takes one parameter which is a locator to the element. Different locators like `By.id()`, `By.name()`, `By.xpath()`, `By.CSSSelector()` etc. locate the elements in the page using their properties like `id`, `name` or `path`, etc.

You can use plugins like Fire path to get help with getting the `id`, `xpath`, etc. of the elements.

Using the example site <http://demo.guru99.com/test/login.html> given below is the code to locate the "Email address" text field using the `id` locator and the "Password" field using the `name` locator.

```
// Get the WebElement corresponding to the Email Address(TextField)
WebElement email = driver.findElement(By.id("email")); 1

// Retrieve the WebElement corresponding to the Password Field
WebElement password = driver.findElement(By.name("passwd")); 2
```

- 1) email text field is located by ID
- 2) Password field is located by name

- Entering Values in Input Boxes

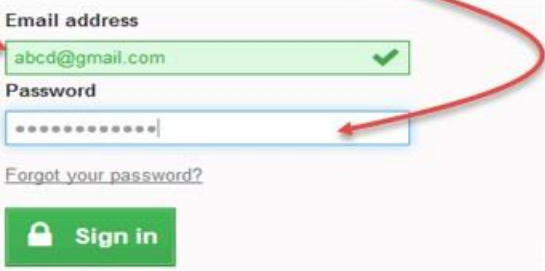
To enter text into the Text Fields and Password Fields, `sendKeys()` is the method available on the `WebElement`.

Using the same example of <http://demo.guru99.com/test/login.html>, here is how we find the Text field and Password fields and enter values into them.

```
// Get the WebElement corresponding to the Email Address(TextField)
WebElement email = driver.findElement(By.id("email")); 1

// Retrieve the WebElement corresponding to the Password Field
WebElement password = driver.findElement(By.name("passwd")); 2

email.sendKeys("abcd@gmail.com"); 3
password.sendKeys("abcdefghijkl"); 4
```



1) Find the "E-mail Address" Text Field using id locator

2) Find the "Password" Field using name locator

3) Enter text into the "Email Address"

4) Enter password into the "Password" using `sendKeys()`

- Deleting Values in Input Boxes

The `clear()` method is used to delete the text in an input box. **This method does not need a parameter.** The code snippet below will clear out the text from the Email or Password fields

Email address

abcd@gmail.com ✓

Password

.....

email.clear();

password.clear();

after clear()

Email address

Password

- Buttons

The buttons can be accessed using the click() method. In the example above

1. Find the button to Sign in
2. Click on the "Sign-in" Button in the login page of the site to login to the site.



- Submit Buttons

Submit buttons are used to submit the entire form to the server. We can either use the click () method on the web element like a normal button as we have done above or use the submit () method on any web element in the form or on the submit button itself.



When submit() is used, WebDriver will look up the DOM to know which form the element belongs to, and then trigger its submit function.

- Radio Button

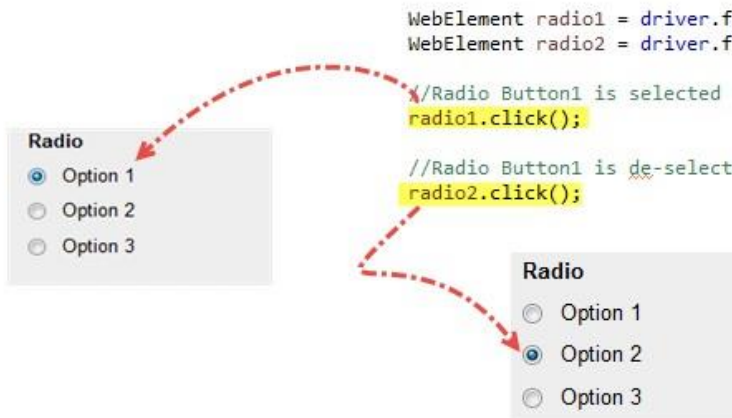
Radio Buttons too can be toggled on by using the click() method.

Using <http://demo.guru99.com/test/radio.html> for practice, see that radio1.click() toggles on the "Option1" radio button. radio2.click() toggles on the "Option2" radio button leaving the "Option1" unselected.

```
WebElement radio1 = driver.findElement(By.id("vfb-7-1"));
WebElement radio2 = driver.findElement(By.id("vfb-7-2"));

//Radio Button1 is selected
radio1.click();

//Radio Button1 is de-selected and Radio button2 is selected
radio2.click();
```



Radio

☒ Option 1

☐ Option 2

☐ Option 3

Radio

☐ Option 1

☒ Option 2

☐ Option 3

click() is the method to toggle on the radio buttons

isSelected() method is used to know whether the Checkbox is toggled on or off.

Here is another example: <http://demo.guru99.com/test/radio.html>

```
@Test
public void tryCheckbox(){

    WebElement option1 = driver.findElement(By.id("vfb-6-0"));

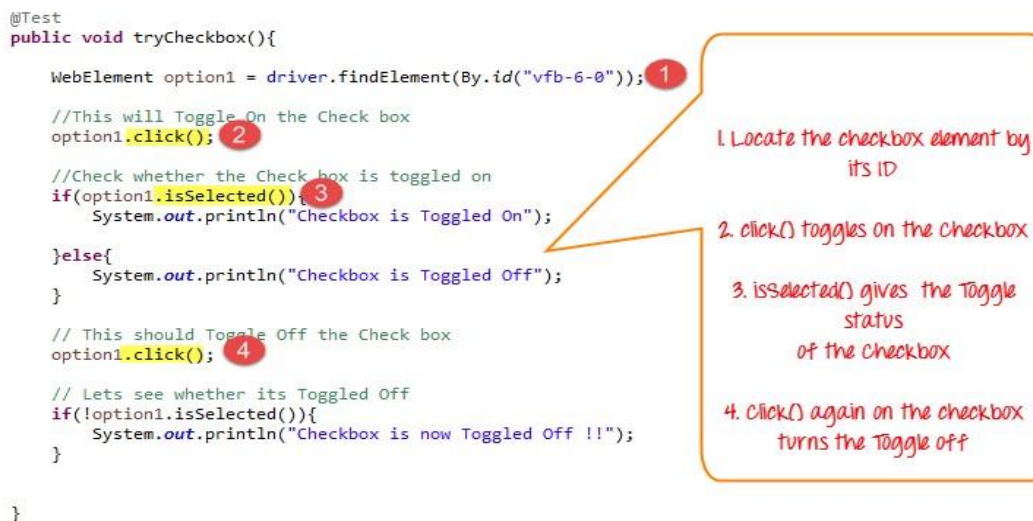
    //This will Toggle On the Check box
    option1.click();

    //Check whether the Check box is toggled on
    if(option1.isSelected()){
        System.out.println("Checkbox is Toggled On");
    }else{
        System.out.println("Checkbox is Toggled Off");
    }

    // This should Toggle Off the Check box
    option1.click();

    // Lets see whether its Toggled Off
    if(!option1.isSelected()){
        System.out.println("Checkbox is now Toggled Off !!");
    }

}
```



1. Locate the checkbox element by its ID
2. click() toggles on the checkbox
3. isSelected() gives the Toggle status of the checkbox
4. click() again on the checkbox turns the Toggle off

- `getText()` and `getAttribute()`

As most of times, you need to test if the expected results differ with actual results, `getText()` and `getAttribute()` methods help you on this.

Suppose there is HTML tag like

```
<input name="Name Locator" value="selenium">Hello</input>
```

getAttribute() : It fetches the text that containing one of any attributes in the HTML tag now `getAttribute()` fetch the data of the attribute of value which is "Selenium"

Returns: The attribute's current value or null if the value is not set.

```
driver.findElement(By.name("Name Locator")).getAttribute("value")
```

The field value is retrieved by the `getAttribute("value")` Selenium WebDriver predefined method and assigned to the String object.

selenium will be shown.

getText() : delivers the `innerText` of a `WebElement`. Get the visible (i.e. not hidden by CSS) `innerText` of this element, including sub-elements, without any leading or trailing whitespace.

Returns: The `innerText` of this element.

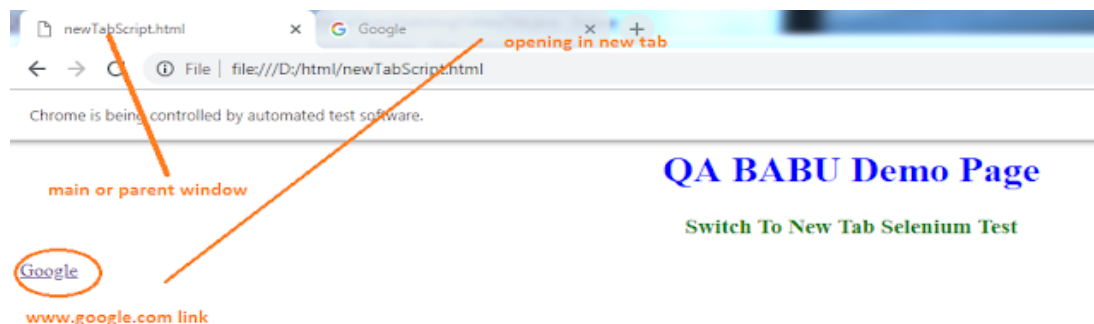
```
driver.findElement(By.name("Name Locator")).getText();
```

Hello will be shown.

- How to switch to new tab in Selenium automation?

You might have seen an application in which clicking on a link opens another web page/application in the new tab, and performing some activity on the application that opened in the new tab.

We will see automating above activity using Selenium WebDriver. Consider below application (QA BABU Demo Page) which as "Google" link clicking on this link will open Google search page in the new tab.



Step 1: Launch "file:///D:/html/newTabScript.html" (this is a local file) in chrome browser using Selenium Web Driver. You can download the file from [here](#)

//launching the application

```
driver.get("file:///D:/html/newTabScript.html");
```

Step 2: Get the parent window handle to String variable using getWindowHandle()

//getting parent window

```
String parentWindow = driver.getWindowHandle();
```

Step 3: Now click on the "Google" link, Google search page should be opened in the new tab

//click Google link

```
driver.findElement(By.linkText("Google")).click();
```

Step 4: Get all the window handles to a Set<String> using getWindowHandles(). Now get number of windows present Set<String>

//get all the windows

```
Set<String> allWindows = driver.getWindowHandles();
```

//No of windows=2

```
System.out.println("No of windows or tabs after clicking: " + allWindows.size());
```

Step 5: Remove parent window handle from the all the window handles, so that Set<> contains only new tab window only in it.

Then switch to the new tab window as shown below.

//removing parent window,

```
allWindows.remove(parentWindow);
```

```
Iterator<String> ite = allWindows.iterator();
```

//So now Set contains only new tab window only,so switch to it

```
driver.switchTo().window((String) ite.next());
```

Step 6: Enter some text to Google search text field and search, if the driver switched to new tab we will be able to search

```
driver.findElement(By.name("q")).sendKeys("QABABU");
```

```
driver.findElement(By.xpath("//div[@class='FPdLc VlcLAe']/input[@value='Google Search']")).click();
```

This is one of the way to switch to new tab in Selenium automation.

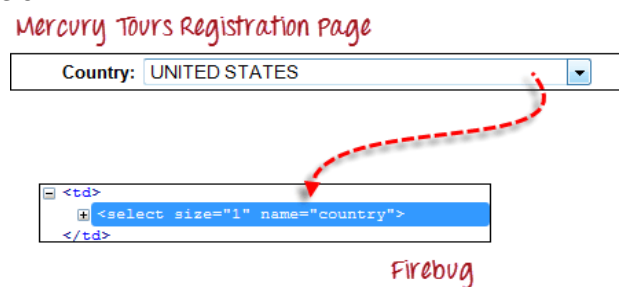
- [How to Select Option from Drop-Down using Selenium WebDriver](#)

In this section, the students will learn how to handle Drop Down and Multiple Select Operations.

Before we can control drop-down boxes, we must do following two things:

- Import the package `org.openqa.selenium.support.ui.Select`
- Instantiate the drop-down box as a "Select" object in WebDriver

Consider the example below:



Step 1

Import the "Select" package.

```
import org.openqa.selenium.support.ui.Select;
```

Step 2

Declare the drop-down element as an instance of the Select class. In the example below, we named this instance as "drpCountry".

```
Select drpCountry = new Select(driver.findElement(By.name("country")));
```

Step 3

We can now start controlling "drpCountry" by using any of the available Select methods. The sample code below will select the option "ANTARCTICA."

```
drpCountry.selectByVisibleText("ANTARCTICA");
```

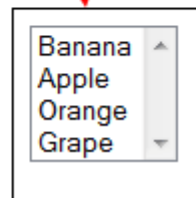
➤ Selectin Items in a Multiple SELECT elements

We can also use the **selectByVisibleText()** method in selecting multiple options in a multi SELECT element.

page source

```
<select id="fruits" multiple="">
  <option value="banana">Banana </option>
  <option value="apple">Apple </option>
  <option value="orange">Orange </option>
  <option value="grape">Grape </option>
</select>
```

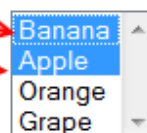
HTML page



The code below will select the first two options using the selectByVisibleText() method.

```
public static void main(String[] args) {
    WebDriver driver = new FirefoxDriver();

    driver.get("http://jsbin.com/osebed/2");
    Select fruits = new Select(driver.findElement(By.id("fruits")));
    fruits.selectByVisibleText("Banana");
    fruits.selectByIndex(1);
}
```



- Checking the WebElement state

- The `isDisplayed()` method

The `isDisplayed` action verifies whether an element is displayed on the web page and can be executed on all the WebElements. The API syntax for the `isDisplayed()` method is as follows:

```
boolean isDisplayed()
```

The preceding method returns a Boolean value specifying whether the target element is displayed on the web page. The following is the code to verify whether the Search box is displayed, which obviously should return true in this case:

```
@Test
public void elementStateExample() {
    WebElement searchBox = driver.findElement(By.name("q"));
    System.out.println("Search box is displayed: "
        + searchBox.isDisplayed());
}
```

The preceding code uses the `isDisplayed()` method to determine whether the element is displayed on a web page. The preceding code returns true for the Search box:

```
Search box is displayed: true
```

- The `isEnabled()` method

The `isEnabled` action verifies whether an element is enabled on the web page and can be executed on all the WebElements. The API syntax for the `isEnabled()` method is as follows:

```
boolean isEnabled()
```

The preceding method returns a Boolean value specifying whether the target element is enabled on the web page. The following is the code to verify whether the Search box is enabled, which obviously should return true in this case:

```
@Test
public void elementStateExample() {
    WebElement searchBox = driver.findElement(By.name("q"));
    System.out.println("Search box is enabled: ")
```

```
+ searchBox.isEnabled());  
}
```

The preceding code uses the `isEnabled()` method to determine whether the element is enabled on a web page. The preceding code returns true for the Search box:

Search box is enabled: true

➤ The `isSelected()` method

The `isSelected` method returns a boolean value if an element is selected on the web page and can be executed only on a radio button, options in select, and checkbox WebElements. When executed on other elements, it will return false. The API syntax for the `isSelected()` method is as follows:

```
boolean isSelected()
```

The preceding method returns a Boolean value specifying whether the target element is selected on the web page. The following is the code to verify whether the Search box is selected on a search page:

```
@Test  
public void elementStateExample() {  
    WebElement searchBox = driver.findElement(By.name("q"));  
    System.out.println("Search box is selected: "  
        + searchBox.isSelected());  
}
```

The preceding code uses the `isSelected()` method. It returns false for the Search box, because this is not a radio button, options in select, or a checkbox. The preceding code returns false for the Search box:

Search box is selected: false

To select a Checkbox or Radio button, we need to call the `WebElement.click()` method, which toggles the state of the element. We can use the `isSelected()` method to see whether it's selected.

- [Accessing links and web tables using selenium WebDriver](#)

In this section, students will learn the available methods to find and access the Links & Tables using Webdriver. Also, will be discussed some of the common problems faced while accessing Links and will further be discussed on how to resolve them.

Here is what they will learn:

Part 1) Accessing Links

- Exact Match
- Partial Match
- All Links
- Case-sensitivity
- Links Outside and Inside a Block
- Accessing Image Links

➤ Accessing links using Exact Text Match

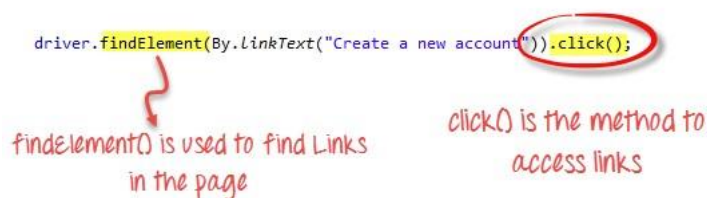
Accessing links using their exact link text is done through the `By.linkText()` method. However, if there are two links that have the very same link text, this method will only access the first one. Consider the HTML code below



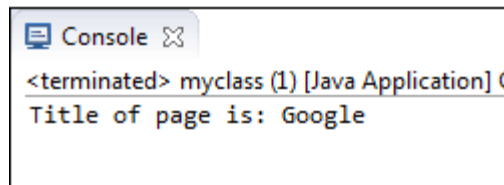
When you try to run the WebDriver code below, you will be accessing the first "click here" link

```
public static void main(String[] args) {  
    String baseUrl = "file:///D:/newhtml.html";  
    WebDriver driver = new FirefoxDriver();  
  
    driver.get(baseUrl);  
    driver.findElement(By.linkText("click here")).click();  
    System.out.println("Title of page is: " + driver.getTitle());  
    driver.quit();  
}
```

Here is how it works-



As a result, you will automatically be taken to Google.



➤ Accessing links using Partial Text Match

Accessing links using a portion of their link text is done using the `By.partialLinkText()` method. If you specify a partial link text that has multiple matches, only the first match will be accessed. Consider the HTML code below.

```
<html>
  <head>
    <title>Partial Match</title>
  </head>
  <body>
    <a href="http://www.google.com">go here</a>
    <br>
    <a href="http://www.fb.com">click here</a>
  </body>
</html>
```

When you execute the WebDriver code below, you will still be taken to Google.

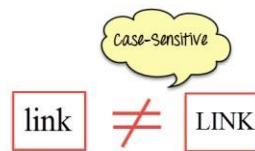
```
public static void main(String[] args) {
    String baseUrl = "file:///D:/partial_match.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    driver.findElement(By.partialLinkText("here")).click();
    System.out.println("Title of page is: " + driver.getTitle());
    driver.quit();
}
```


➤ Getting multiple links with the same Link text

In cases where there are multiple links with the same link text, and we want to access the links other than the first one, different locators `By.xpath()`, `By.cssSelector()` or `By.tagName()` are used. Most commonly used is `By.xpath()`. It is the most reliable one but it looks complex and non-readable too.

➤ Case- sensitivity for Link Text

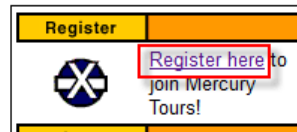


The parameters for `By.linkText()` and `By.partialLinkText()` are both case-sensitive, meaning that capitalization matters. For example, in Mercury Tours' homepage, there are two links that contain the text "egis" - one is the "REGISTER" link found at the top menu, and the other is the "Register here" link found at the lower right portion of the page.

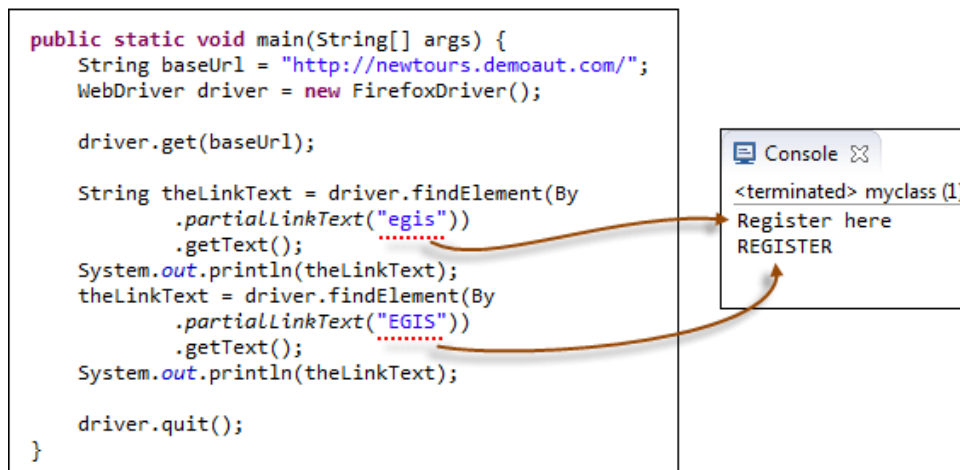
The link at the top menu



The link at the lower right portion of the page



Though both links contain the character sequence "egis," one is the "`By.partialLinkText()`" method will access these two links separately depending on the capitalization of the characters. See the sample code below.

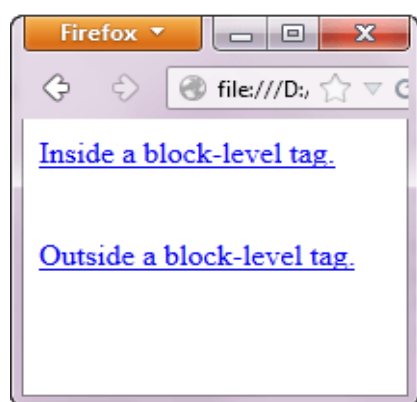


➤ Links Outside and Inside Block

The latest HTML5 standard allows the `<a>` tags to be placed inside and outside of block-level tags like `<div>`, `<p>`, or `<h3>`. The `"By.linkText()"` and `"By.partialLinkText()"` methods can access a link located outside and inside these block-level elements. Consider the HTML code below.

```
<body>
  <p>
    <a href="http://www.google.com">Inside a block-level tag.</a>
  </p>

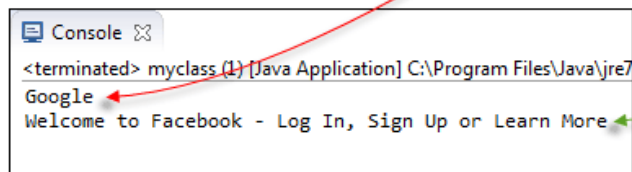
  <br>
  <a href="http://www.fb.com">
    <div>
      <span>Outside a block-level tag.</span>
    </div>
  </a>
</body>
```



The WebDriver code below accesses both of these links using By.partialLinkText() method.

```
public static void main(String[] args) {
    String baseUrl = "file:///D:/Links%20Outside%20and%20Inside%20a%20Block.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    driver.findElement(By.partialLinkText("Inside")).click();
    System.out.println(driver.getTitle());
    driver.navigate().back();
    driver.findElement(By.partialLinkText("Outside")).click();
    System.out.println(driver.getTitle());
    driver.quit();
}
```



Console

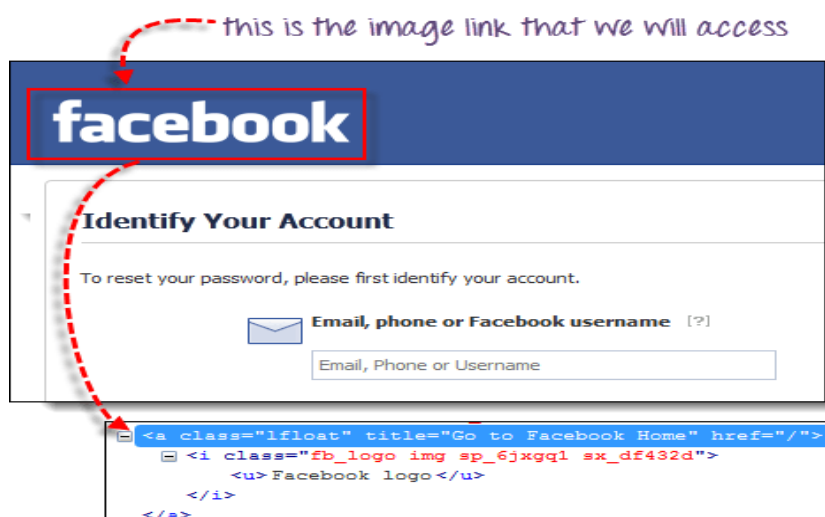
```
<terminated> myclass (1) [Java Application] C:\Program Files\Java\jre7\bin\java.exe
Google
Welcome to Facebook - Log In, Sign Up or Learn More
```

Links can be accessed by the By.linkText() and By.partialLinkText() whether they are inside or outside block-level elements.

➤ Accessing Image links

Image links are the links in web pages represented by an image which when clicked navigates to a different window or page. Since they are images, we cannot use the By.linkText() and By.partialLinkText() methods because image links basically have no link texts at all. In this case, we should resort to using either By.cssSelector or By.xpath. The first method is more preferred because of its simplicity.

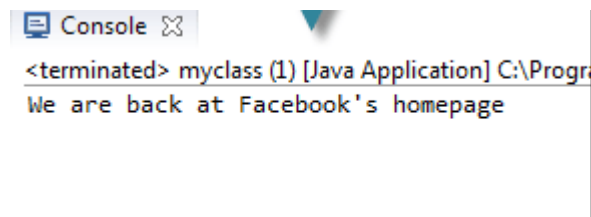
In the example below, we will access the "Facebook" logo on the upper left portion of Facebook's Password Recovery page.



We will use `By.cssSelector` and the element's "title" attribute to access the image link. And then we will verify if we are taken to Facebook's homepage.

```
driver.findElement(By.cssSelector("a[title=\"Go to Facebook home\"]")).click();
```

Result:



Part 2) Reading a Table

- XPath Syntax
- Accessing Nested Tables
- Using Attributes as Predicates
- Shortcut: Use Firebug for Accessing Tables in Selenium

- Reading a HTML Web Table

Consider the HTML code below.

```
<html>
  <head>
    <title>Sample</title>
  </head>
  <body>
    <table border="1">
      <tbody>
        <tr>
          <td>first cell</td>
          <td>second cell</td>
        </tr>
        <tr>
          <td>third cell</td>
          <td>fourth cell</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

We will use XPath to get the inner text of the cell containing the text "fourth cell."

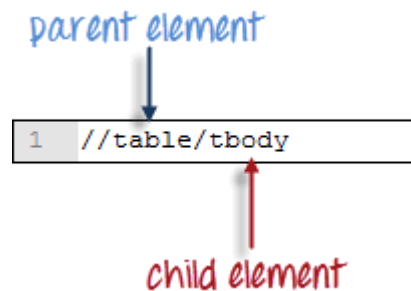


Step 1 - Set the Parent Element (table)

XPath locators in WebDriver always start with a double forward slash "/" and then followed by the parent element. Since we are dealing with tables, the parent element should always be the `<table>` tag. The first portion of our XPath locator should, therefore, start with `//table`.

Step 2 - Add the child elements

The element immediately under `<table>` is `<tbody>` so we can say that `<tbody>` is the "child" of `<table>`. And also, `<table>` is the "parent" of `<tbody>`. All child elements in XPath are placed to the right of their parent element, separated with one forward slash "/" like the code shown below.



Step 3 - Add Predicates

The `<tbody>` element contains two `<tr>` tags. We can now say that these two `<tr>` tags are "children" of `<tbody>`. Consequently, we can say that `<tbody>` is the parent of both the `<tr>` elements.

Another thing we can conclude is that the two `<tr>` elements are siblings. Siblings refer to child elements having the same parent.

To get to the `<td>` we wish to access (the one with the text "fourth cell"), we must first access the second `<tr>` and not the first. If we simply write `//table/tbody/tr`, then we will be accessing the first `<tr>` tag.

So, how do we access the second `<tr>` then? The answer to this is to use Predicates.

Predicates are numbers or HTML attributes enclosed in a pair of square brackets "[]" that distinguish a child element from its siblings. Since the `<tr>` we need to access is the second one, we shall use `"[2]"` as the predicate.

```
1 //table/tbody/tr[2]
```

The [2] predicate denotes that we are accessing the 2nd <tr> of the parent <tbody>

If we won't use any predicate, XPath will access the first sibling. Therefore, we can access the first <tr> using either of these XPath codes.

Step 4 - Add the Succeeding Child Elements Using the Appropriate Predicates

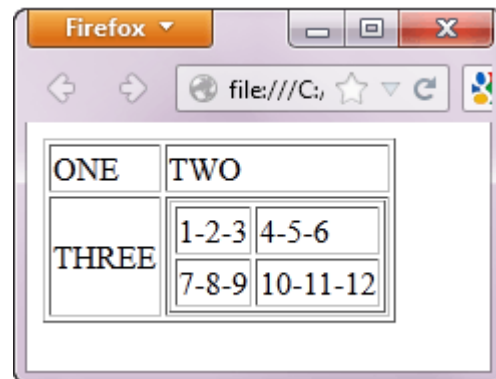
The next element we need to access is the second <td>. Applying the principles we have learned from steps 2 and 3, we will finalize our XPath code to be like the one shown below.

```
//table/tbody/tr[2]/td[2]
```

➤ Accessing Nested Tables

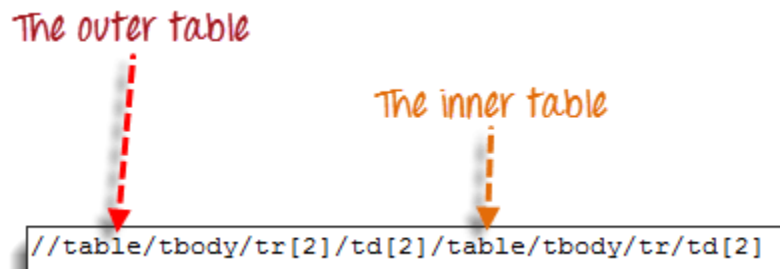
The same principles discussed above applies to nested tables. Nested tables are tables located within another table. An example is shown below.

```
<html>
<head>
  <title>Sample</title>
</head>
<body>
  <!--outer table-->
  <table border="1">
    <tbody>
      <tr>
        <td>ONE</td>
        <td>TWO</td>
      </tr>
      <tr>
        <td>THREE</td>
        <td>
          <!--inner table-->
          <table border="1">
            <tbody>
              <tr>
                <td>1-2-3</td>
                <td>4-5-6</td>
              </tr>
              <tr>
                <td>7-8-9</td>
                <td>10-11-12</td>
              </tr>
            </tbody>
          </table>
        </td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```



ONE	TWO				
THREE	<table border="1"><tr><td>1-2-3</td><td>4-5-6</td></tr><tr><td>7-8-9</td><td>10-11-12</td></tr></table>	1-2-3	4-5-6	7-8-9	10-11-12
1-2-3	4-5-6				
7-8-9	10-11-12				

To access the cell with the text "4-5-6" using the "//parent/child" and predicate concepts from the previous section, we should be able to come up with the XPath code below.



The WebDriver code below should be able to retrieve the inner text of the cell which we are accessing.

```
public static void main(String[] args) {
    String baseUrl = "file:///C:/newhtml.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(By
        .xpath("//table/tbody/tr[2]/td[2]/table/tbody/tr/td[2]"))
        .getText();
    System.out.println(innerText);
    driver.quit();
}
```

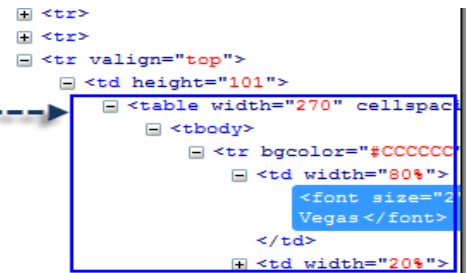
➤ Using Attributes as Predicates

If the element is written deep within the HTML code such that the number to use for the predicate is very difficult to determine, we can use that element's unique attribute instead.

In the example below, the "New York to Chicago" cell is located deep into Mercury Tours homepage's HTML code.

Specials	
Atlanta to Las Vegas	\$398
Boston to San Francisco	\$513
Los Angeles to Chicago	\$168
New York to Chicago	\$198
Phoenix to San Francisco	\$213

this is the `<table>` that holds the New York to Chicago cell. Notice that it is buried deep into the HTML code, and the number to use as Predicate is difficult to determine.



```
<tr>
<tr>
<tr valign="top">
  <td height="101">
    <table width="270" cellpadding="10">
      <tbody>
        <tr bgcolor="#CCCCFF">
          <td width="80%">
            <font size="24">
              Vegas</font>
            </td>
            <td width="20%">
```

In this case, we can use the table's unique attribute (`width="270"`) as the predicate. Attributes are used as predicates by prefixing them with the `@` symbol. In the example above, the "New York to Chicago" cell is located in the first `<td>` of the fourth `<tr>`, and so our XPath should be as shown below. Remember that when we put the XPath code in Java, we should use the escape character backward slash `"\"` for the double quotation marks on both sides of `"270"` so that the string argument of `By.xpath()` will not be terminated prematurely.

```
By.xpath("//table[@width=\"270\"]/tbody/tr[4]/td")
```

use the escape characters here

We are now ready to access that cell using the code below.

```
public static void main(String[] args) {
    String baseUrl = "http://newtours.demoaut.com/";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(By
        .xpath("//table[@width=\"270\"]/tbody/tr[4]/td"))
        .getText();
    System.out.println(innerText);
    driver.quit();
}
```

➤ **Shortcut: Use Inspect Element for accessing Tables in Selenium**

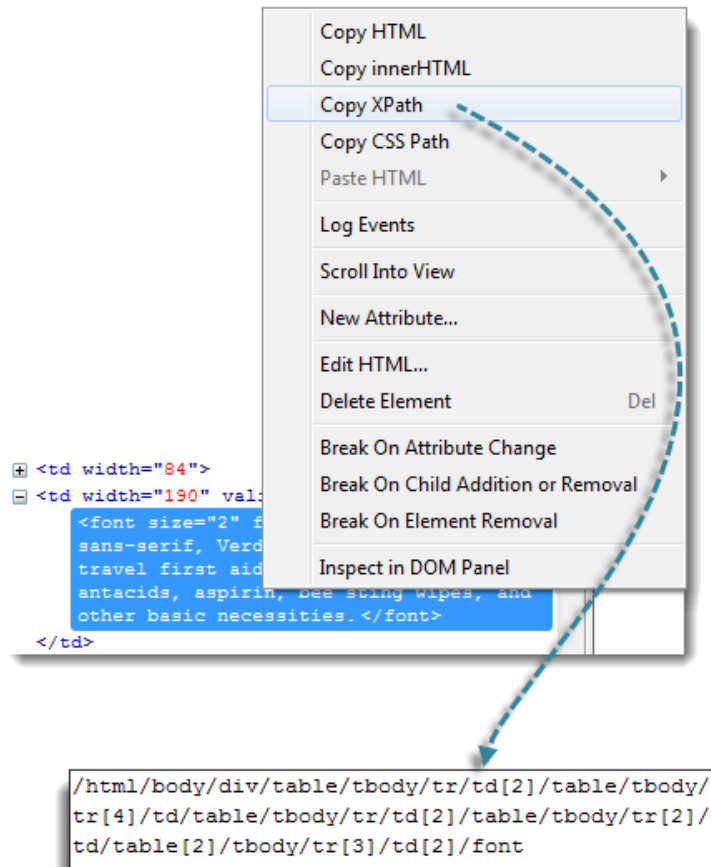
If the number or attribute of an element is extremely difficult or impossible to obtain, the quickest way to generate the XPath code is using Inspect Element. Consider the example below from Mercury Tours homepage.



this is the text that
we will access

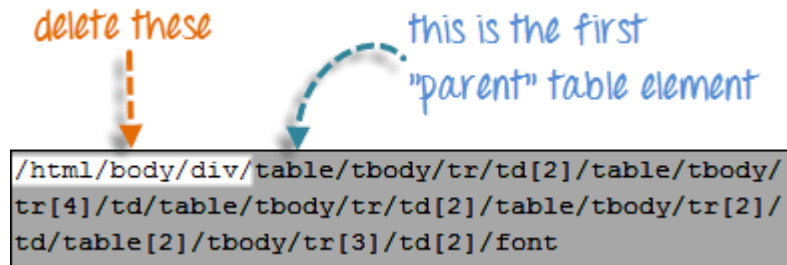
Step 1

Use Firebug to obtain the XPath code.



Step 2

Look for the first "table" parent element and delete everything to the left of it.



Step 3

Prefix the remaining portion of the code with double forward slash "/" and copy it over to your WebDriver code.

The remaining portion of the code, trimmed and prefixed with "/"

```
//table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[2]/td/table[2]/tbody/tr[3]/td[2]/font
```



```
By.xpath("//table/tbody/tr/td[2]"  
+ "/table/tbody/tr[4]/td"  
+ "/table/tbody/tr/td[2]"  
+ "/table/tbody/tr[2]/td"  
+ "/table[2]/tbody/tr[3]/td[2]/font")
```

When pasted onto the `By.xpath()` method

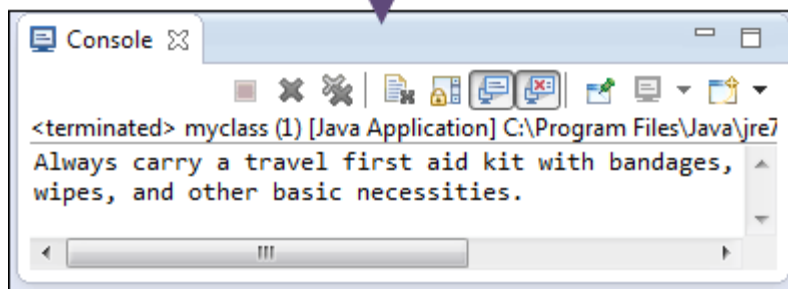
The WebDriver code below will be able to successfully retrieve the inner text of the element we are accessing.

```

public static void main(String[] args) {
    String baseUrl = "http://newtours.demoaut.com/";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(
        By.xpath("//table/tbody/tr/td[2]"
            + "/table/tbody/tr[4]/td"
            + "/table/tbody/tr/td[2]"
            + "/table/tbody/tr[2]/td"
            + "/table[2]/tbody/tr[3]/td[2]/font"))
        .getText();
    System.out.println(innerText);
    driver.quit();
}

```



Part 3) XPath Selectors

- Types of XPath
- What are XPath axes?
- Advanced XPath methods

What does XPath do? How does it work?

XPath is a syntax language for finding any element on the web page using XML path expression. XPath is designed to allow the navigation of XML documents, with the purpose of selecting individual elements, attributes, or some other part of an XML document for specific processing.

Why should I use XPath?

XPath is used to find the location of any element on a webpage using HTML DOM structure.

Standard syntax for creating XPath:



Sometimes, we may not identify the element using the locators such as id, class, name, etc. In those cases, we use XPath to find an element on the web page. XPath produces reliable locators but in performance wise it is slower, compared to CSS Selectors.

➤ Types of XPath

There are two types of XPath:

1. Absolute XPath
2. Relative XPath

Absolute XPath:

It is the direct way to find the element. It begins with the single forward slash (/), which means you can select the element from the root node. The disadvantage of the absolute XPath is that if there are any changes made in the path of the element then that XPath gets failed.

Relative XPath:

The path starts from the middle of the HTML DOM structure. It starts with double forward slash (//), which means it can search the element anywhere at the webpage. You can start from the middle of the HTML DOM structure and no need to write long XPath.

➤ What are XPath axes?

XPath axes search different nodes in XML document from current context node. XPath Axes are the methods used to find dynamic elements, which otherwise not possible by normal XPath method having no ID, Classname, Name, etc.

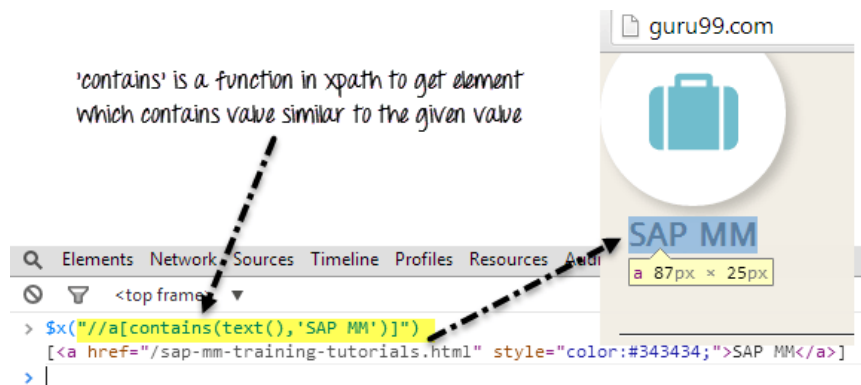
Axes methods are used to find those elements, which dynamically change on refresh or any other operations. There are few axes methods commonly used in Selenium Webdriver like child, parent, ancestor, sibling, preceding, self, etc.

➤ Advanced XPath methods

If a simple [XPath](#) is not able to find a complicated web element for our test script, we need to use the functions from XPath 1.0 library. With the combination of these functions, we can create more specific XPath.

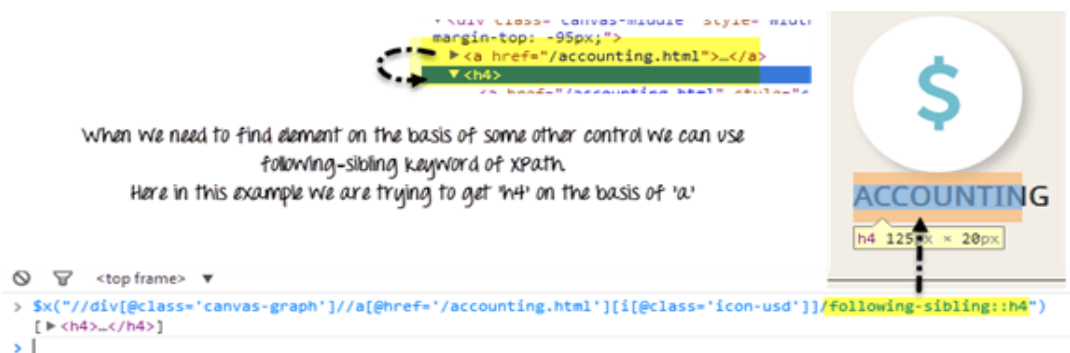
Contains

By using 'contains' function in XPath, we can extract all the elements which matches a particular text value.



Sibling

Using sibling keyword, we can fetch a web element on the which is related to some other element.



Ancestor: To find an element on the basis of the parent element we can use ancestor attribute of XPath.



Ancestor function

We can achieve the same functionality with the help of a function 'ancestor' as well.

Now suppose we need to Search All elements in 'Popular course' section with the help of ancestor of the anchor whose text is 'SELENIUM'

Here our xpath query will be like:

```
//div[./a[text()='SELENIUM']]/ancestor::div[@class='rt-grid-2 rt-omega']/following-sibling::div
```

Using AND and OR

By using AND and OR you can put 2 conditions in our XPath expression.

- In case of AND both 2 conditions should be true then only it finds the element.
- In case of OR any one of the 2 conditions should be true then only it finds the element.

Parent

By Using Parent, you can find the parent node of the current node in the web page.



Starts-with

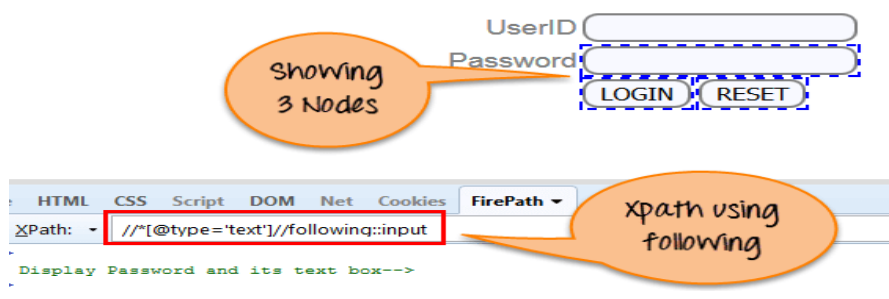
Using Starts-with function, you can find the element whose attribute dynamically changes on refresh or other operations like click, submit, etc.



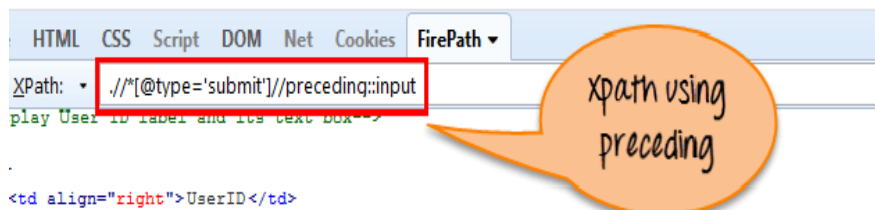
Xpath axes

By using XPath axes, you can find the dynamic and very complex elements on a web page. XPath axes contain several methods to find an element. Here, will discuss a few methods.

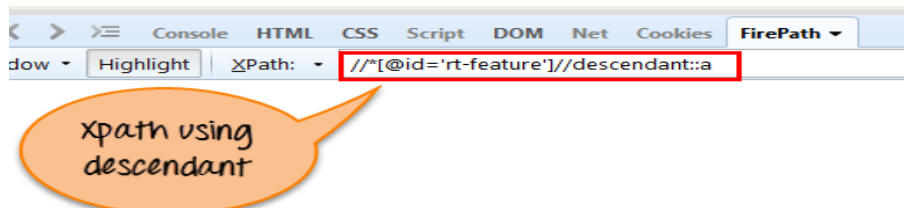
following: This function will return the immediate element of the particular component.



Preceding: This function will return the preceding element of the particular element.



d) **Descendant:** This function will return the descendant element of the particular element.

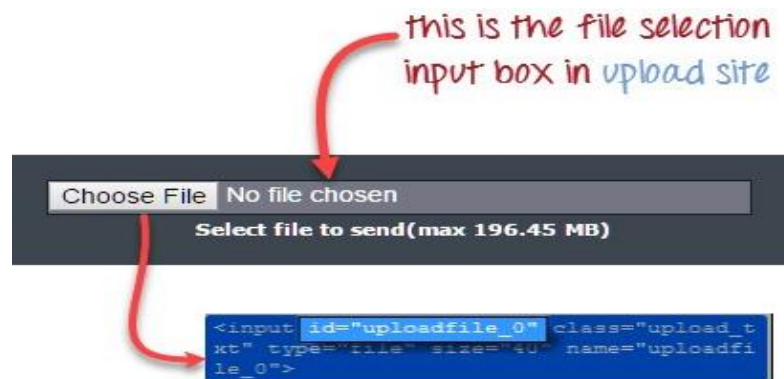


- **Upload & Download File**

For this section, students will use a demo website as our test application. This site easily allows any visitor to upload files without requiring them to sign up.

- **Uploading Files**

Uploading files in WebDriver is done by simply using the `sendKeys()` method on the file-select input field to enter the path to the file to be uploaded.



Handle File upload popup in Selenium Web Driver

Remember following two things when uploading files in WebDriver

- There is no need to simulate the clicking of the "Browse" button. WebDriver automatically enters the file path onto the file-selection text box of the `<input type="file">` element
- When setting the file path in your Java IDE, use the proper escape character for the back-slash.


```
// enter the file path onto the file-selection input field  
uploadElement.sendKeys("C:\\newhtml.html");
```

When used within a string, each back-slash in your file path is represented by a double back-slash

Sometimes when we try to upload a file, a dialog box is open, which is not part of our browser so it is not possible to find a selector for it. So under these circumstances, the Robot class helps us.

➤ Downloading Files

WebDriver has no capability to access the Download dialog boxes presented by browsers when you click on a download link or button. However, we can bypass these dialog boxes using a separate program called "wget".

What is Wget?

Wget is a small and easy-to-use command-line program used to automate downloads. Basically, we will access Wget from our WebDriver script to perform the download process.

3.4 Keyboard and Mouse Events using Action Class in Selenium WebDriver

In this section, students will learn handling Keyboard and Mouse Event in Selenium Webdriver. Handling special keyboard and mouse events are done using the Advanced User Interactions API. It contains the Actions and the Action classes that are needed when executing these events.

Handling special keyboard and mouse events are done using the Advanced User Interactions API. It contains the Actions and the Action classes that are needed when executing these events. The following are the most commonly used keyboard and mouse events provided by the Actions class. Frequently used Keyword and Mouse Events are doubleClick(), keyUp, dragAndDropBy, contextClick & sendKeys.

Method	Description
clickAndHold()	Clicks (without releasing) at the current mouse location.

contextClick()	Performs a context-click at the current mouse location. (Right Click Mouse Action)
doubleClick()	Performs a double-click at the current mouse location.
dragAndDrop(source, target)	<p>Performs click-and-hold at the location of the source element, moves to the location of the target element, then releases the mouse.</p> <p>Parameters:</p> <p>source- element to emulate button down at.</p> <p>target- element to move to and release the mouse at.</p>
dragAndDropBy(source, x-offset, y-offset)	<p>Performs click-and-hold at the location of the source element, moves by a given offset, then releases the mouse.</p> <p>Parameters:</p> <p>source- element to emulate button down at.</p> <p>xOffset- horizontal move offset.</p> <p>yOffset- vertical move offset.</p>
keyDown(modifier_key)	<p>Performs a modifier key press. Does not release the modifier key - subsequent interactions may assume it's kept pressed.</p> <p>Parameters:</p> <p>modifier_key - any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL)</p>
keyUp(modifier_key)	<p>Performs a key release.</p> <p>Parameters:</p> <p>modifier_key - any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL)</p>

moveByOffset(x-offset, y-offset)	<p>Moves the mouse from its current position (or 0,0) by the given offset.</p> <p>Parameters:</p> <p>x-offset- horizontal offset. A negative value means moving the mouse left.</p> <p>y-offset- vertical offset. A negative value means moving the mouse down.</p>
moveToElement(toElement)	<p>Moves the mouse to the middle of the element.</p> <p>Parameters:</p> <p>toElement- element to move to.</p>
release()	<p>Releases the depressed left mouse button at the current mouse location</p>
sendKeys(onElement, charsequence)	<p>Sends a series of keystrokes onto the element.</p> <p>Parameters:</p> <p>onElement - element that will receive the keystrokes, usually a text field</p> <p>charsequence - any string value representing the sequence of keystrokes to be sent</p>

Example:

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

Use perform() to execute the actions.

3.5 How to handle Alert and Frame in Selenium WebDriver

- **Handle Alert**

Alert interface provides the below few methods which are widely used in Selenium Webdriver.

Step 1: To click on the 'Cancel' button of the alert.

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

Step 2: To click on the 'OK' button of the alert.

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

Step 3: To capture the alert message.

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

Step 4: To send some data to alert box.

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

We can easily switch to alert from the main window by using Selenium's **.switchTo()** method.

- [Handle frame](#)

What is iFrame? An iFrame (Inline Frame) is an HTML document embedded inside the current HTML document on a website. iFrame HTML element is used to insert content from another source, such as an advertisement, into a Web page. A website can have multiple frames on a single page.

Using the SwitchTo().frame function

For a browser to work with several elements in iframes, it is crucial for the browser to identify all the iframes. For this purpose, we need to use the **SwitchTo().frame** method. This method enables the browser to switch between multiple frames. It can be implemented in the following ways:

- [Switching Frames in Selenium using Index](#)

1. **switchTo.frame(int *frame number*):** Defining the frame index number, the Driver will switch to that specific frame

- [Switching Frames using Name or ID](#)

2. **switchTo.frame(string *frameNameOrId*):** Defining the frame element or Id, the Driver will switch to that specific frame

To switch between an iframe using the **name** attribute, one can use the switch command as follows:

```
//Switch by frame name
driver.switchTo().frame("iframeResult"); //BY frame name
```

To switch between an iframe using the **Id** attribute, one can use the switch command as follows:

```
//Switch by frame name
driver.switchTo().frame("iframeResult");// Switch By ID
```

➤ Switching Frames using WebElement

3.switchTo.frame(WebElement frameElement): Defining the frame web element, the Driver will switch to that specific frame

```
//First finding the element using any of locator strategy
WebElement iframeElement = driver.findElement(By.id("iframeResult"));
//now using the switch command
driver.switchTo().frame(iframeElement);
```

3.6 Implicit Wait, Explicit Wait & Fluent Wait in Selenium

In selenium "Waits" play an important role in executing tests. In this section, students will learn various aspects of both "Implicit" and "Explicit" waits in Selenium.

In this section, you will learn:

- Why Do We Need Waits in Selenium?
- Implicit Wait
- Explicit Wait
- Fluent Wait

Most of the web applications are developed using Ajax and Javascript. When a page is loaded by the browser the elements which we want to interact with may load at different time intervals.

Not only it makes this difficult to identify the element but also if the element is not located it will throw an **"ElementNotVisibleException"** exception. Using Waits, we can resolve this problem.

➤ Why do we need waits in Selenium

Most of the web applications are developed using Ajax and Javascript. When a page is loaded by the browser the elements which we want to interact with may load at different time intervals.

Not only it makes this difficult to identify the element but also if the element is not located it will throw an **"ElementNotVisibleException"** exception. Using Waits, we can resolve this problem.

➤ Implicit Wait

Selenium Web Driver has borrowed the idea of implicit waits from Watir.

The implicit wait will tell the web driver to wait for certain amount of time before it throws a **"No Such Element Exception"**. The default setting is 0. Once we set the time, web driver will wait for that time before throwing an exception.

In the below example we have declared an implicit wait with the time frame of 10 seconds. It means that if the element is not located on the web page within that time frame, it will throw an exception.

To declare implicit wait: **Syntax:**

```
driver.manage().timeouts().implicitlyWait(Timeout, TimeUnit.SECONDS);
```

Consider Following Code:

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

Implicit wait will accept 2 parameters, the first parameter will accept the time as an integer value and the second parameter will accept the time measurement in terms of SECONDS, MINUTES, MILISECOND, MICROSECONDS, NANOSECONDS, DAYS, HOURS, etc.

➤ Explicit Wait

The explicit wait is used to tell the Web Driver to wait for certain conditions (Expected Conditions) or the maximum time exceeded before throwing an **"ElementNotVisibleException"** exception.

The explicit wait is an intelligent kind of wait, but it can be applied only for specified elements. Explicit wait gives better options than that of an implicit wait as it will wait for dynamically loaded Ajax elements. Once we declare explicit wait we have to use **"ExpectedConditions"** or we can configure how frequently we want to check the condition using Fluent Wait. These days while implementing we are using Thread.Sleep() generally it is not recommended to use.

Syntax:

```
WebDriverWait wait = new WebDriverWait(WebDriverReference,Timeout); Consider
```

Following Code:

```
WebElement guru99seleniumlink;guru99seleniumlink =  
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("/html/body/div[1]/sect  
ion/div[2]/div/div[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div  
/a/i")));guru99seleniumlink.click();
```

In the above example, wait for the amount of time defined in the **"WebDriverWait"** class or the **"ExpectedConditions"** to occur whichever occurs first.

The above [Java](#) code states that we are waiting for an element for the time frame of 20 seconds as defined in the **"WebDriverWait"** class on the webpage until the **"ExpectedConditions"** are met and the condition is **"visibilityOfElementLocated"**.

The following are the Expected Conditions that can be used in Explicit Wait

- `alertIsPresent()`
- `elementSelectionModeToBe()`
- `elementToBeClickable()`
- `elementToBeSelected()`
- `frameToBeAvaliableAndSwitchToIt()`
- `invisibilityOfTheElementLocated()`
- `visibilityOf()`
- `visibilityOfAllElements()`

➤ Fluent Wait

The fluent wait is used to tell the web driver to wait for a condition, as well as the frequency with which we want to check the condition before throwing an **"ElementNotVisibleException"** exception.

Frequency: Setting up a repeat cycle with the time frame to verify/check the condition at the regular interval of time

Syntax:

```
Wait wait = new FluentWait(WebDriver reference).withTimeout(timeout,
SECONDS).pollingEvery(timeout, SECONDS).ignoring(Exception.class);
```

Consider Following Code:

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

In the above example, we are declaring a fluent wait with the timeout of 30 seconds and the frequency is set to 5 seconds by ignoring **"NoSuchElementException"**

Project Overview:

This is the first important project that will be developed by students at the end of this chapter. Now, after they have been introduced with the core feature of Selenium library (main classes and methods, architecture etc.) should be able to create their first test cases.

Firstly, any web browser will be called and initialized to navigate to a given web-page. Then different actions like clicking a button, waiting for an element to be shown on a certain page, checking options of a drop-down element will be undertaken. At the end, browser will be closed and a report will be generated. Students should be able even to analyse the results and debug in case of test failure.

3.7 Common exception when working with Selenium

In this section, you will learn:

- NoSuchElementException
- ElementNotVisibleException
- StaleElementReferenceException
- InvalidElementStateException
- SessionNotFoundException

➤ NoSuchElementException

This is probably the most straightforward exception that you will come across. The element you are trying to find does not exist. There are three common causes for this exception:

- The locator you are using to find the element is incorrect
- Something has gone wrong and the element has not been rendered
- You tried to find the element before it was rendered

The first one is pretty easy to check. You can use the Google Chrome development tools to test your locator.

The second one is a little harder to diagnose, as you will need to walk through your code and see what happens to cause the failure.

Screenshots can be a big help in diagnosing NoSuchElementException issues, as they give a good view of the state of the application you are testing when the error failed. If the cause of the problem is that the element has not yet been rendered, it is possible that the element was not rendered when the error occurred but it was rendered when the screenshot was taken. In this case, the screenshot will appear to show the element was there, when in actual fact it was missing when Selenium tried to find it.

This brings us nicely to the third potential problem. Lots of modern websites use technologies such as jQuery or AngularJS, which use JavaScript to manipulate the DOM. Selenium is fast; in many cases it's ready to start interacting with your website before all of the JavaScript has finished doing its job. When this happens, things may seem to be missing when in reality they just haven't been created yet. There are some tricks that you can use to wait for JavaScript to finish rendering the page, but the real solution is to know the application that you are automating. You should know what is required for the page to be ready to use and write your code to be aware of these conditions. A good question to ask in many situations is: *What would I do if I were testing this manually?*

Normally, you would wait for the page to load before starting your testing; you have to write your code so that it can do the same thing. Explicit waits are usually your best friend in this scenario. You should never use `Thread.sleep()` to wait for the page to load.

➤ ElementNotVisibleException

This is a very useful exception that you will probably come across on a regular basis. It tells you that the WebElement that you are trying to interact with is not visible to the user. If the element is not visible to the user, they are not going to be able to interact with it.

If you see this exception, there is a problem that needs to be fixed with your code. Selenium is very fast and will often try and interact with an element before it has had a chance to render on the screen. Your code should be aware of what needs to happen for the element to be displayed to the end user. You will need to wait for the page rendering (or at least the part of the page you are interested in) to complete before trying to interact with the element.

When you see this exception, the best thing to do is walk through the code manually and check whether you can see things that load slowly. The usual fix is to then add an explicit wait to wait for the correct conditions to be met before trying to interact with the element in question.

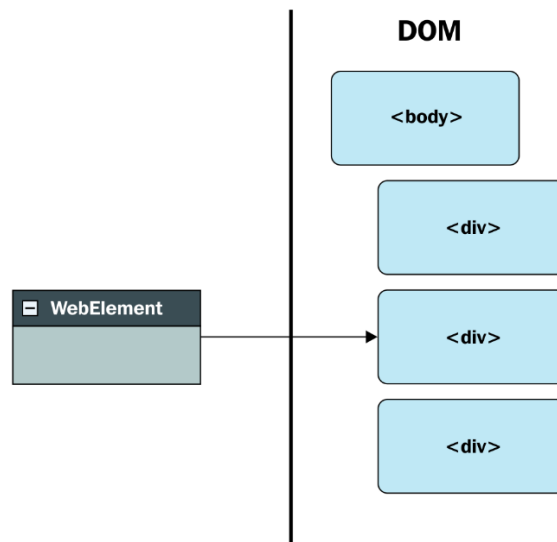
➤ StaleElementReferenceException

This is an exception that you will quite often see if you work with AJAX or JavaScript-heavy websites where the DOM is continuously being manipulated.

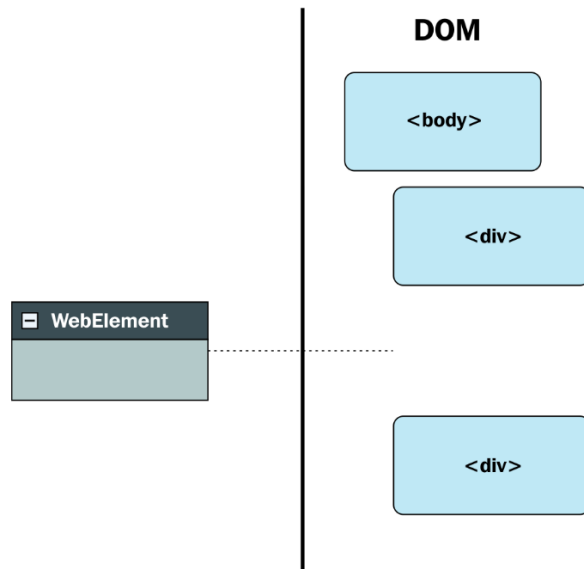
You are probably used to seeing code like this:

```
WebElement googleSearchBar = driver.findElement(By.name("q"));
```

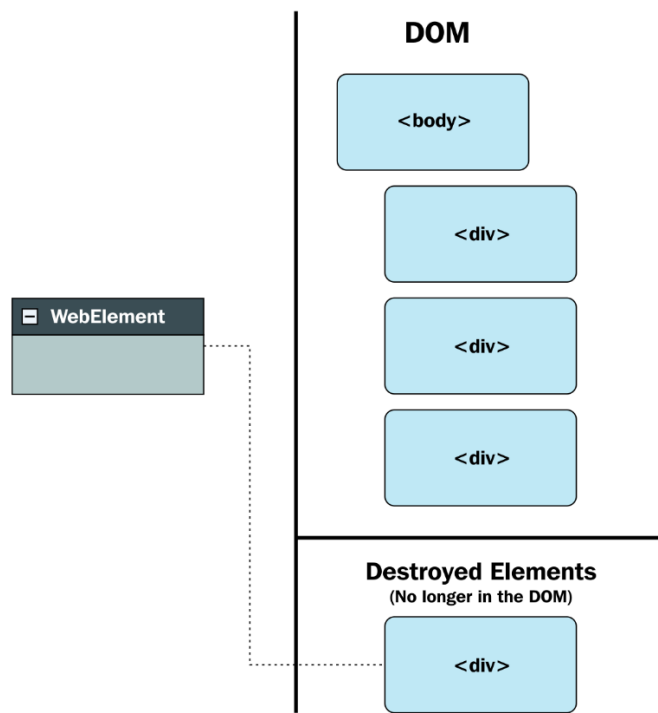
The WebElement object that you have created is actually a reference to a specific element in the DOM:



When the DOM is manipulated and the old element is destroyed, that reference no longer links to an element in the DOM and it becomes stale.



This can get very confusing when the element we have a reference to has been destroyed, and then another identical looking element has replaced it.



The Selenium solution is just as simple, you ask Selenium to find that element again:

```
googleSearchBar = driver.findElement(By.name("q"));
```

The reference is updated and you can carry on interacting with that WebElement.

Maybe we were expecting the element to be destroyed and recreated and we want to check for that. If so, we could use a conditional wait to wait for the element to become stale before continuing with our test.

```
WebDriverWait wait = new WebDriverWait(driver, 10);
```

```
wait.until(ExpectedConditions.stalenessOf(googleSearchBar));
```

It's actually pretty easy to do; Selenium's Java bindings have some predefined expected conditions that we can use without having to write our own explicit wait condition.

➤ InvalidElementStateException

This is an exception that you probably won't see that often, but when it does pop up it is not always instantly clear what it means. InvalidElementStateException is thrown when the WebElement that you are trying to interact with is not in a state that would allow you to perform the action that you would like to perform.

Think of a <select> element that gives you a list of countries to select when filling in an address form; this element will allow you to select the country associated with your address.

Now, what if the developers have added some validation that will not let you enter a postcode (or ZIP code) until you have selected a country, so that they can trigger the correct postcode validation routine? In this case, the <input> element where you enter your postcode may be disabled until you have selected your country.

If you try to enter a postcode into this disabled <input> element, you will get InvalidElementStateException.

The fix is to do whatever a user manually testing the site would do to enable the <input> element; in this case, select a country from the <select> element.

➤ SessionNotFoundException

Occasionally, when you are running your tests, things will go wrong and you will lose connection with the browser instance you are driving. When you lose connection to the browser instance, SessionNotFoundException will be thrown.

This is a similar error to UnreachableBrowserException but in this case, you have a much smaller list of things to check since you know you were successfully talking to the RemoteWebDriver instance for a while.

The following problems usually cause this error:

- You inadvertently quit the driver instance
- The browser crashed

3.8 Scroll in Selenium

To scroll using Selenium, you can use JavaScriptExecutor interface that helps to execute JavaScript methods through Selenium Webdriver

Syntax :

```
JavaScriptExecutor js = (JavaScriptExecutor) driver;
```

```
js.executeScript(Script,Arguments);
```

- Script – This is the JavaScript that needs to execute.
- Arguments – It is the arguments to the script. It's optional.

Selenium Script to scroll down the page

Let's, see the scroll down a web page using the selenium webdriver with following 4 scenarios :

- Scenario 1: To scroll down the web page by pixel.
- Scenario 2: To scroll down the web page by the visibility of the element.
- Scenario 3: To scroll down the web page at the bottom of the page.
- Scenario 4: Horizontal scroll on the web page.

Scenario 1: To scroll down the web page by pixel.

```
JavaScriptExecutor js = (JavaScriptExecutor) driver;
```

```
// This will scroll down the page by 1000 pixel vertical  
js.executeScript("window.scrollTo(0,1000)");
```

Scenario 2: To scroll down the web page by the visibility of the element.

```
JavascriptExecutor js = (JavascriptExecutor) driver;  
WebElement Element = driver.findElement(By.linkText("Home"));  
//This will scroll the page till the element is found  
js.executeScript("arguments[0].scrollIntoView();", Element);  
"arguments[0]" means first index of page starting at 0.
```

Scenario 3: To scroll down the web page at the bottom of the page.

```
JavascriptExecutor js = (JavascriptExecutor) driver;  
//This will scroll the web page till end.  
js.executeScript("window.scrollTo(0, document.body.scrollHeight);");  
Javascript method scrollTo() scroll the till the end of the page .  
"document.body.scrollHeight" returns the complete height of the body i.e web page.
```

Scenario 4: Horizontal scroll on the web page.

```
JavascriptExecutor js = (JavascriptExecutor) driver;  
WebElement Element = driver.findElement(By.linkText("VBScript"));  
//This will scroll the page Horizontally till the element is found  
js.executeScript("arguments[0].scrollIntoView();", Element);  
Javascript method scrollIntoView() scrolls the page until the mentioned element is in full view.
```

3.9 Assert Vs Verify Commands in Selenium

The validation checks made in Selenium usually come in two flavors: one using Assertions and the other inculcating “Verify” statements. Though both serve the same purpose there persists one major difference in their working functionality.

Verify checks get test results for multiple conditions even if one of them fails wherein Hard Assertions put a stringent restriction on the test script when it fails thereby terminating the program execution further. Although Soft Assertions work in a way different from Hard Assertion where normal flow of execution resumes although there is failure in the asserting script. Usage of either Assertions or verify statement purely falls on the users cup of tea.

Commonly used Assertions

Assert Equals & Assert Not Equals

Assert Equals works by comparing the Expected condition with that of the Actual condition based on which the test results are displayed.

“Assert Not Equals” serves the purpose of negation testing for Testers. If the Actual and Expected do not match, then the test Script passes else it fails.

```
Assert.assertEquals("Log In – Perficient Wiki ", driver.getTitle());
```

```
Actualtext = driver.findElement(By.xpath("//h3/span")).getText();
```

```
Assert.assertNotEquals(Actualtext, "Here comes the surprise gift for every order on Sunday, 01 January 2017", "Expected and Actual do not match as the test is performed on December 31st, 2016");
```

Assert True & Assert False

“Assert True” passes the Test Step only when the boolean value returned is “True” and “Assert False” passes the Test Step only when the boolean value returned is “False”.

```
defaultcheck = driver.findElement(By.name("os_cookie"));
```

```
System.out.print("\n" + defaultchk1.isSelected());
```

```
Assert.assertTrue(defaultcheck.isSelected());
```

```
System.out.print("Assertion Passed successfully ");
```

3.10 Difference between driver.close() and driver.quit()

Selenium webdriver provides two methods for closing a browser window driver.close() and driver.quit(). Some people incorrectly use them interchangeably but the two methods are different. In this post, we will study the difference between the two and also see where to use them effectively.

driver.close()

The driver.close() command is used to close the current browser window having focus. In case there is only one browser open then calling driver.close() quits the whole browser session.

Usability

It is best to use driver.close() when we are dealing with multiple browser tabs or windows e.g. when we

click on a link that opens another tab. In this case after performing required action in the new tab, if we want to close the tab we can call the `driver.close()` method.

driver.quit()

The `driver.quit()` is used to quit the whole browser session along with all the associated browser windows, tabs and pop-ups.

Usability

It is best to use `driver.quit()` when we no longer want to interact with the driver object along with any associated window, tab or pop-up. Generally, it is one of the last statements of the automation scripts. In case, we are working with Selenium with TestNG or JUnit, we call `driver.quit()` in the `@AfterSuite` method of our suite. Thus, closing it at the end of the whole suite.

4.Introduction to TestNg & Junit

4.1 TestNg Annotations

A Selenium test can be automated with TestNG by adding annotations to them. Annotations follow the syntax `@ + 'annotation'` and are inserted in the line before the method or class we want to mark it with:

```
@Test
public void testCase() {
    System.out.println("This is a Test Case example");
    // Input username
    // Input password
    // Hit the login button
}
```

The first annotation we should learn is `@Test`. This establishes a method or class that is part of the test. The following table presents the most important attributes of the `@Test` annotation:

- **alwaysRun**: When set to true, it indicates that the test will be always executed, even if it depends on a test that failed.
- **dataProvider**: The name of the data provider for the test, which is useful when running a test that needs different parameters.
- **enabled**: Specifies whether the test is enabled or disabled.
- **description**: A text description of the test.
- **expectedExceptions**: When a test is expected to throw one or more exceptions as a result, the list of exceptions can be declared in this annotation. If the test does not throw any of the expected exceptions, it will be marked as failed.

- **invocationCount:** A test can be invoked more than once with this annotation. For example, with `invocationCount=10`, the test will be executed 10 times.
- **threadPoolSize:** Establishes the thread pool size for the test, usually combined with `invocationCount`. If `invocationCount=10` and `threadPoolSize=1`, the test will be invoked 10 times sequentially. But if `threadPoolSize=10`, then the test will be invoked 10 times at the same time (in parallel).

Besides the classes and methods that contain the instructions of the actual test, we might have other pieces of codes that are required before, after, or at a specific point in our test. For example, before testing a login page, some setup might be required:

```
@BeforeMethod
public void beforeMethod() {
    System.out.println("This will execute before the Test Case
method");
    // Setup WebDriver
    // Open website
    // Set a wait
}
```

But we might also need to perform some tasks after our test:

- **@BeforeClass:** If this annotation is used in a method, it means that the annotated method will be executed before any of the declared tests in the same class. It is useful to execute setup activities that are needed for the tests in that class.
- **@AfterClass:** Similar to the previous annotation, but after all tests have been executed. It is commonly used to perform a clean-up after tests have finished.
- **@BeforeMethod:** When a method has this annotation, it will be executed before each one of the tests in the class where it exists. It is useful to do a small setup or clean up before each test; it should be used for light or simple tasks.
- **@AfterMethod:** Similar to the previous annotation, but after each test is executed. In most of the cases, either `BeforeMethod` or `AfterMethod` is used, not both. If you are using both, be aware of what each before or after action is exactly doing.
- **@BeforeSuite:** When a method is annotated with `BeforeSuite`, it will be executed before any of the tests contained in the test suite. It is normally used for heavy and complex setup tasks needed for the tests, like creating test users in a database, creating files needed for testing, or to check that the system under test is up and responsive (smoke tests).
- **@AfterSuite:** This is conceptually similar to the previous annotation, but it is executed after all tests have finished. In it generally used to perform clean up tasks, like deleting the created users for testing.

More detailed and up to date information about TestNG annotations can be found at <http://testng.org/doc/documentation-main.html#annotations>.

In the previous table, we included the `@BeforeGroups` and `@AfterGroups` annotations. In TestNG, groups can be used to execute batches of tests in any given order. Let's consider the following tasks to be tested on an online store application:

- Login.
- Search for products.
- Modify the cart (add products to the cart, delete products from the cart, and change the quantities of the products already in the cart).

Checkout process (verify the cart, the billing and shipping address, and make the payment). If we want to perform different tests on each of these tasks, we might need to execute them in the same order as a real client would when they use the website:

- All tests related to the login functionality.
- All tests related to the product's search functionality.
- All tests related to managing of the cart.
- All tests related to the checkout process.

With the `@BeforeGroups` and `@AfterGroups` annotations, our code would look similar to this:

```
public class OnlineStoreTest {
    @BeforeGroups("Login")
    public void setupLogin() {
        System.out.println("setupLogin()");
    }

    @Test (groups = { "Login" })
    public void LoginTest1() {
        System.out.println("Login Functionality - Test 1");
    }

    @Test (groups = { "Login" })
    public void LoginTest2() {
        System.out.println("Login Functionality - Test 2");
    }

    @AfterGroups("Login")
    public void cleanUpLogin() {
        System.out.println("cleanUpLogin()");
    }

    @BeforeGroups("Search")
    public void setupSearch() {
        System.out.println("setupSearch()");
    }
}
```

```

@Test (groups = { "Search" })
public void SearchTest1() {
    System.out.println("Search Functionality - Test 1");
}

@Test (groups = { "Search" })
public void SearchTest2() {
    System.out.println("Search Functionality - Test 2");
}

@AfterGroups("Search")
public void cleanUpSearch() {
    System.out.println("CleanUp Search()");
}

@BeforeGroups("Payment")
public void setupPayment() {
    System.out.println("setupPayment()");
}

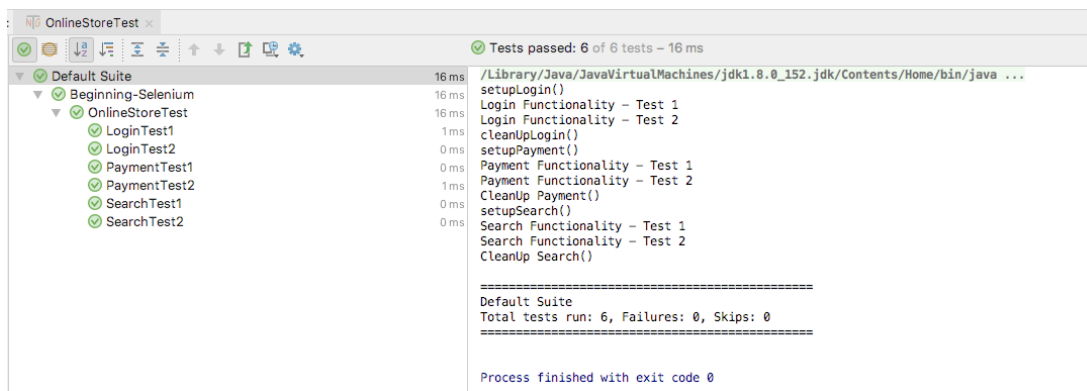
@Test (groups = { "Payment" })
public void PaymentTest1() {
    System.out.println("Payment Functionality - Test 1");
}

@Test (groups = { "Payment" })
public void PaymentTest2() {
    System.out.println("Payment Functionality - Test 2");
}

@AfterGroups("Payment")
public void cleanUpPayment() {
    System.out.println("CleanUp Payment()");
}
}

```

If we execute the previous code in our IDE, we should see an output similar to the following:



The following list contains the attributes for the `@BeforeGroups` and `@AfterGroups` annotations:

- `@BeforeGroups`: If a method is annotated with `BeforeGroups` (with a list of groups), then this method will be executed before any of the tests that belong to any of the listed groups is invoked. Similar to the other `Before*` annotations, this one is useful to perform setup tasks that are needed to run tests that belong to any given group(s).
- `@AfterGroups`: This is conceptually similar to the previous one, but the method will be executed after all tests that belong to the group list have already finished. This is commonly used for cleanup tasks.

These annotations can be extended with the following attributes:

- `dependsOnGroups`: A test can depend on groups of tests, which means that a given group of tests will be executed before the annotated test is executed. For example, when a "user area" test is executed after a group of "login" tests.
- `dependsOnMethods`: This is similar to the previous one, but there is an emphasis on test names. This creates a direct dependency because text "X" must be executed before test "Y".
- `enabled`: This specifies whether the test is enabled or disabled.
- `groups`: This sets the group where the test or the class containing the test belongs.

4.2 JUnit Annotations

What is Junit?

JUnit is a unit testing framework for Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.

Why you need JUnit testing

- It finds bugs early in the code, which makes our code more reliable.
- JUnit is useful for developers, who work in a test-driven environment.
- Unit testing forces a developer to read code more than writing.
- You develop more readable, reliable and bug-free code which builds confidence during development.

What is JUnit Annotations?

Annotation is a special form of syntactic meta-data that can be added to [Java](#) source code for better code readability and structure. Variables, parameters, packages, methods and classes can be annotated.

Some of the JUnit annotations which can be useful are:

S.No.	Annotations	Description
1.	@Test	This annotation is a replacement of <code>org.junit.TestCase</code> which indicates that public void method to which it is attached can be executed as a test Case.
2.	@Before	This annotation is used if you want to execute some statement such as preconditions before each test case.
3.	@BeforeClass	This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
4.	@After	This annotation can be used if you want to execute some statements after each Test Case for e.g. resetting variables, deleting temporary files ,variables, etc.
5.	@AfterClass	This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
6.	@Ignore	This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
7.	@Test(timeout=500)	This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.

5. Selenium with Cucumber

In this chapter, we will introduce students with Cucumber – a Behavior Driven Development (BDD) framework which is used with Selenium for performing acceptance testing.

In order to understand cucumber, they need to know the most important features of cucumber and its usage, which is the main goal of this chapter.

5.1 What is Cucumber testing?

Cucumber is a tool that supports Behaviour Driven Development (BDD). It offers a way to write tests that anybody can understand, regardless of their technical knowledge.

- What is Behaviours Driven Development(BDD)
- Advantages of Cucumber

➤ What is cucumber?

Cucumber is a tool that supports Behaviour Driven Development (BDD). It offers a way to write tests that anybody can understand, regardless of their technical knowledge. In BDD, users (business analysts, product owners) first write scenarios or acceptance tests that describes the behaviour of the system from the customer's perspective, for review and sign-off by the product owners before developers write their codes. Cucumber use Ruby programming language.

➤ Advantages of Cucumber

- It is helpful to involve business stakeholders who can't easily read code
- Cucumber Testing focuses on end-user experience
- Style of writing tests allow for easier reuse of code in the tests
- Quick and easy set up and execution
- Efficient tool for testing

5.2 Cucumber features, scenarios and steps

A Feature File is an entry point to the Cucumber tests. This is a file where you will describe your tests in Descriptive language (Like English). It is an essential part of Cucumber, as it serves as an automation test script as well as live documents.

In this chapter we will explain the following topics:

- What is "Feature File"
- What is "Step Definition"

➤ What is "Feature File"?

Features file contain high level description of the Test Scenario in simple language. It is known as Gherkin. Gherkin is a plain English text language which helps you to describe business behaviour without the need to go into detail of implementation.

Feature File consist of following components:

Feature: A feature would describe the current test script which has to be executed.

Scenario: Scenario describes the steps and expected outcome for a particular test case.

Scenario Outline: Same scenario can be executed for multiple sets of data using scenario outline. The data is provided by a tabular structure separated by (| |).

Given: It specifies the context of the text to be executed. By using datatables "Given", step can also be parameterized.

When: "When" specifies the test action that has to performed

Then: The expected outcome of the test can be represented by "Then"

Also in Cucumber you can pass parameters from feature files to your test scripts. This can be done inline when you pass a parameter between two single quotes or with cucumber data table. With cucumber data table you can pass parameters in tabular format and then you can use this data in step definition methods in the form of Lists and Maps.

Sample Feature File Example:

Feature: Visit **career guide** page in `career.guru99.com`

Scenario: Visit `career.guru99.com`

Given: I am on `career.guru99.com`

When: I click on career guide menu

Then: I should see career guide page

➤ What is “Step Definition”?

Step definition maps the Test Case Steps in the feature files(introduced by Given/When/Then) to code. It which executes the steps on Application Under Test and checks the outcomes against expected results. For a step definition to be executed, it must match the given component in a feature. Step definition is defined in ruby files under "features/step_definitions/*_steps.rb".

5.3 Why use Cucumber with Selenium?

What is Selenium? Selenium is a free (open source) automated testing suite for web applications across different browsers and platforms. Testing done using Selenium tool is usually referred as Selenium Testing. Selenium supports different language like java, ruby, python C#, etc.

Cucumber and Selenium are two popular technologies.

Most of the organizations use Selenium for functional testing. These organizations which are using Selenium want to integrate Cucumber with selenium as Cucumber makes it easy to read and to understand the application flow.

Cucumber tool is based on the Behavior Driven Development framework that acts as the bridge between the following people:

1. Software Engineer and Business Analyst.
2. Manual Tester and Automation Tester.
3. Manual Tester and Developers.

Project Overview:

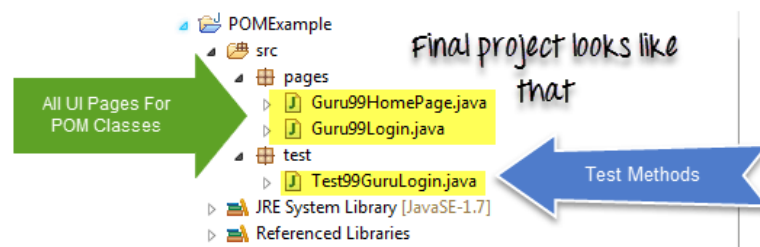
This is the final project and the most important one. Here will be integrated almost all Selenium features that are necessary to work in a test automation project. Also, scenarios will be organised on Cucumber features.

6. Page Object Model (POM) & Page Factory: Selenium WebDriver Tutorial

6.1 What is Page Object Model?

Page Object Model is a design pattern to create **Object Repository** for web UI elements. Under this model, for each web page in the application, there should be corresponding page class. This Page class will find the WebElements of that web page and also contains Page methods which perform operations on those WebElements.

Name of these methods should be given as per the task they are performing, i.e., if a loader is waiting for the payment gateway to appear, POM method name can be `waitForPaymentScreenDisplay()`.



➤ Why Page Object Model?

Starting an UI Automation in Selenium WebDriver is NOT a tough task. You just need to find elements, perform operations on it.

Consider a simple script to login into a website

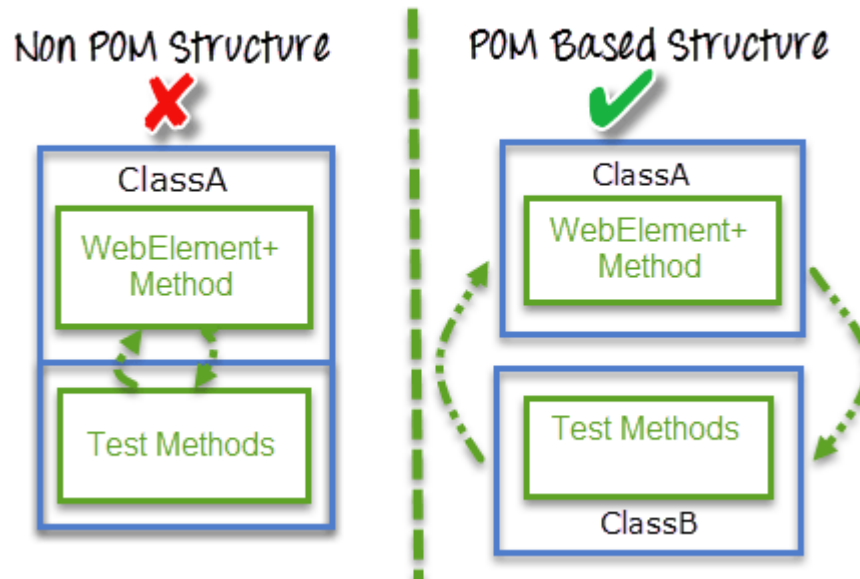
As you know now, all we are doing is finding elements and filling values for those elements.

This is a small script. Script maintenance looks easy. But with time test suite will grow. As you add more and more lines to your code, things become tough.

The chief problem with script maintenance is that if 10 different scripts are using the same page element, with any change in that element, you need to change all 10 scripts. This is time consuming and error prone.

A better approach to script maintenance is to create a separate class file which would find web elements, fill them or verify them. This class can be reused in all the scripts using that element. In future, if there is a change in the web element, we need to make the change in just 1 class file and not 10 different scripts.

This approach is called **Page Object Model(POM)**. It helps make the code **more readable, maintainable, and reusable**.



Advantages of POM

1. Page Object Pattern says operations and flows in the UI should be separated from verification. This concept makes our code cleaner and easy to understand.
2. The Second benefit is the **object repository is independent of test cases**, so we can use the same object repository for a different purpose with different tools. For example, we can integrate POM with TestNG/JUnit for functional [Testing](#) and at the same time with JBehave/Cucumber for acceptance testing.
3. Code becomes less and optimized because of the reusable page methods in the POM classes.
4. Methods get more realistic names which can be easily mapped with the operation happening in UI. i.e. if after clicking on the button we land on the home page, the method name will be like 'gotoHomePage()'.

6.2 How to implement POM?

Simple POM:

It's the basic structure of Page object model (POM) where all Web Elements of the **AUT** and the method that operate on these Web Elements are maintained inside a class file. A task like **verification** should be **separate** as part of Test methods.

```

public class Guru99Login {
    WebDriver driver;
    By user99GuruName = By.name("uid");
    By password99Guru = By.name("password");
    By titleText = By.className("barone");
    By login = By.name("btnLogin");

    public Guru99Login(WebDriver driver){
        this.driver = driver;
    }
    //Set user name in textbox
    public void setUsername(String strUserName){
        driver.findElement(user99GuruName).sendKeys(strUserName);
    }
}

```

6.3 What is Page Factory?

Page Factory is an inbuilt Page Object Model concept for Selenium WebDriver but it is very optimized.

Here as well, we follow the concept of separation of Page Object Repository and Test Methods. Additionally, with the help of PageFactory class, we use annotations **@FindBy** to find WebElement. We use `initElements` method to initialize web elements

```

@FindBy(xpath="//table//tr[@class='heading3']")
WebElement homePageUserName;

public Guru99HomePage(WebDriver driver){
    this.driver = driver;
    //This initElements method will create all WebElements
    PageFactory.initElements(driver, this);
}

```

Summary

1. Page Object Model is an Object Repository design pattern in Selenium WebDriver.
2. POM creates our testing code maintainable, reusable.
3. Page Factory is an optimized way to create object repository in POM concept.

7. 7 Most Common Challenges in Selenium Automation

Even though Selenium has made web testing far less complex many teams and enterprises across the globe, it has a considerable amount of issues because of its open-source nature. A large portion of the difficulties that testers encounter has moderately straightforward solutions, which is the reason we outline the commonest Selenium Webdriver challenges and how to settle them.

1. Can't Test Windows Apps

It doesn't support windows based apps, it only supports web-based applications.

2. Can't Test Portable Applications

We can test on [any OS and browser on the desktop](#) using it; however, we can't deal with mobile testing with Selenium alone. But there is a solution to this. You can make use of [Appium](#) to deal with Android and iOS local, portable, and crossbreed applications using the WebDriver protocol.

3. Limited Reports

Having limited reports is one of the core difficulties and challenges of using Selenium. As with selenium WebDriver, you may not be able to create a decent report, but there is a workaround. You can make reports using [TestNG](#) or Extent reports.

4. Pop-up Windows

When a simple or confirmation alert pops up, [it tends to be the hardest job to automate it to either accept or close](#). There are basically three distinct kinds of pop-up windows:

- Prompts Alert – It informs the user to input something
- Confirmation alert – It mainly requests the user for confirmation
- Simple alert – It is something that shows some message

Moreover, Windows-based alerts are beyond the abilities of the Selenium since they are part of the OS instead of the browser. However, since Selenium WebDriver can operate diverse windows, web-based alerts can, by and large, be taken care of with the switchTo method for controlling the pop-up while keeping the browser in the background.

5. Page Load

A handful of website pages are client-specific. These pages load different components that rely upon the specific user. Most of the time, components appear to rely on prior activity. If you select, for example, a nation from a list of nations dropdown, then cities related that country will load in the cities dropdown. In runtime, the script couldn't find the component. To overcome this, we require utilizing explicit waits in the script to give the components enough time to load and to detect the component.

6. Captcha

[Handling captcha is restricted in Selenium.](#) There are some outside party tools to automate it yet we can't accomplish 100% outcomes.

7. Scalability

While Selenium WebDriver enables you to test on essentially any browser or OS, it's constrained in the number of tests it can keep running without delay and how fast it can run them dependent on what number of node/hub designs the tester has. Without a Selenium Grid, you can only test sequentially. In any case, with a Selenium Grid and a third-party cloud tool for cross-browser testing, you can test in equivalent. It can diminish the time it takes to run automated testing and escalating the configurations you can test on.

8. Overview

	Duration (number of sessions)	Type of Evaluation	Resources
Chapter 1	1	Exam	ISTQB Foundation Level Syllabus.
Chapter 2	2	Exam	W3C web-element selectors documentation.
Chapter 3	4	Project	Selenium documentation. Web-page used for project.
Chapter 4	2 (+1 Optional)	Final Project	Cucumber documentation. Web-page used for project.