Selenium commands are also called ***Selenese***. Selenese is the set of commands which are used in Selenium IDE. These set of commands are used to test our web applications.

In selenese, one can test the existence of UI elements based on their HTML tags, test for specific content, test for broken links, input fields, selection list options, submitting forms, and table data among other things. In addition Selenium commands support testing of window size, mouse position, alerts, Ajax functionality, pop up windows, event handling, and many other web-application features.

Selenium IDE Commands (Selenese)

There are three types of Selenium IDE commands:

Actions

***Actions***are commands that generally manipulate the state of the application. They do things like ***type this box***, ***click this link*** or ***select option***. If an Action fails, or has an error, the execution of the current test is stopped.

Many Actions can be called with the ***AndWait*** suffix, e.g. ***clickAndWait***, ***typeAndWait***. This ***Suffix***tells Selenium that the action will cause the browser to make a call to the server, and that Selenium should wait for a new page to load.

Accessors

These commands examine the state of the application and store the result in variables, Like ***storeTitle***. They are also used to automatically generate Assertions.

Assertions

These commands are like ***accessors***, but they verify that the state of the application conforms to what is expected.  
Now the Selenium Assertions can also be categorized into three categories:

* ***Assert****: When an ‘Assert’ fails, the test is aborted.*
* ***Verify****: When a ‘Verify’ fails, the test will continue execution, logging the failure.*
* ***WaitFor****: Wait for some condition to become true. They will succeed immediately if the condition is already true. However, they will fail and halt the test if the condition does not become true within the current timeout setting.*

***Commonly used Selenium IDE commands:***

* ***type****: Sets the value of an input field, as though you typed it in.*
* ***open****: Opens a page using a URL.*
* ***click****: Clicks on a link, button, checkbox or radio button.*
* ***clickAndWait****: Clicks on a link, button, checkbox or radio button. If the click action causes a new page to load (like a link usually does), call waitForPageToLoad.*
* ***select****: Select an option from a drop-down using an option locator.*
* ***selectFrame****: Selects a frame within the current window.*
* ***verifyTitle/assertTitle****: Verifies an expected page title.*
* ***verifyTextPresent****: Verifies that the specified text pattern appears somewhere on the rendered page shown to the user.*
* ***verifyElementPresent****: Verifies that the specified element is somewhere on the page.*
* ***waitForPageToLoad****: Waits for a new page to load. You can use this command instead of the “AndWait” suffixes, “clickAndWait”, “selectAndWait”, “typeAndWait” etc.*
* ***highlight****: Briefly changes the backgroundColor of the specified element yellow. Useful for debugging.*
* ***pause****: Wait for the specified amount of time (in milliseconds)*
* ***store****: The name of a variable in which the result is to be stored. This command is a synonym for storeExpression.*
* ***echo****: Prints the specified message into the third table cell in your Selenese tables. Useful for debugging.*
* ***refresh****: Simulates the user clicking the “Refresh” button on their browser*

# Implicit Wait & Explicit Wait in Selenium

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

In this tutorial, you will learn-

1. [Why Do We Need Waits In Selenium?](https://www.guru99.com/implicit-explicit-waits-selenium.html#1)

2. [Implicit Wait](https://www.guru99.com/implicit-explicit-waits-selenium.html#2)

3. [Explicit Wait](https://www.guru99.com/implicit-explicit-waits-selenium.html#3)

4. [Fluent Wait](https://www.guru99.com/implicit-explicit-waits-selenium.html#4)

## 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.

Let's consider a scenario where we have to use both implicit and explicit waits in our test. Assume that implicit wait time is set to 20 seconds and explicit wait time is set to 10 seconds.

Suppose we are trying to find an element which has some **"ExpectedConditions** "(Explicit Wait), If the element is not located within the time frame defined by the Explicit wait(10 Seconds), It will use the time frame defined by implicit wait(20 seconds) before throwing an "**ElementNotVisibleException**".

**Selenium Web Driver Waits**

1. Implicit Wait
2. Explicit Wait

## Implicit Wait

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

The implicit wait will tell to 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);

package guru.test99;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.Test;

public class AppTest {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

driver = new ChromeDriver();

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

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

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

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.equals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

//close browser

driver.close();

}

}

**Explanation of Code**

In the above example,

**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 "**ExpectedCondtions**" 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

In the below example, we are creating reference wait for "**WebDriverWait**" class and instantiating using "**WebDriver**" reference, and we are giving a maximum time frame of 20 seconds.

**Syntax:**

WebDriverWait wait = new WebDriverWait(WebDriverRefrence,TimeOut);

package guru.test99;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

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

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

import org.testng.annotations.Test;

public class AppTest2 {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

driver = new ChromeDriver();

WebDriverWait wait=new WebDriverWait(driver, 20);

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

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

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.contentEquals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

WebElement guru99seleniumlink;

guru99seleniumlink= wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath( "/html/body/div[1]/section/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();

}

}

**Explanation of Code**

**Consider Following Code:**

WebElement guru99seleniumlink;

guru99seleniumlink= wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath( "/html/body/div[1]/section/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](https://www.guru99.com/java-tutorial.html)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

1. alertIsPresent()
2. elementSelectionStateToBe()
3. elementToBeClickable()
4. elementToBeSelected()
5. frameToBeAvaliableAndSwitchToIt()
6. invisibilityOfTheElementLocated()
7. invisibilityOfElementWithText()
8. presenceOfAllElementsLocatedBy()
9. presenceOfElementLocated()
10. textToBePresentInElement()
11. textToBePresentInElementLocated()
12. textToBePresentInElementValue()
13. titleIs()
14. titleContains()
15. visibilityOf()
16. visibilityOfAllElements()
17. visibilityOfAllElementsLocatedBy()
18. visibilityOfElementLocated()

## 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

Let's consider a scenario where an element is loaded at different intervals of time. The element might load within 10 seconds, 20 seconds or even more then that if we declare an explicit wait of 20 seconds. It will wait till the specified time before throwing an exception. In such scenarios, the fluent wait is the ideal wait to use as this will try to find the element at different frequency until it finds it or the final timer runs out.

**Syntax:**

Wait wait = new FluentWait(WebDriver reference)

.withTimeout(timeout, SECONDS)

.pollingEvery(timeout, SECONDS)

.ignoring(Exception.class);

package guru.test99;

import org.testng.annotations.Test;

import java.util.NoSuchElementException;

import java.util.concurrent.TimeUnit;

import java.util.function.Function;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

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

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

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

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

import org.testng.annotations.Test;

public class AppTest3 {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

driver = new ChromeDriver();

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

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

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.contentEquals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(30, TimeUnit.SECONDS)

.pollingEvery(5, TimeUnit.SECONDS)

.ignoring(NoSuchElementException.class);

WebElement clickseleniumlink = wait.until(new Function<Webdriver, WebElement>(){

public WebElement apply(WebDriver driver ) {

return driver.findElement(By.xpath("/html/body/div[1]/section/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"));

}

});

//click on the selenium link

clickseleniumlink.click();

//close~ browser

driver.close() ;

}

}

**Explanation of Code**

**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**"

**Consider Following Code:**

public WebElement apply(WebDriver driver ) {

return driver.findElement(By.xpath("/html/body/div[1]/section/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"));

}

We have created a new function to identify the Web Element on the page. (Ex: Here Web Element is nothing but the selenium link on the webpage).

Frequency is set to 5 seconds and the maximum time is set to 30 seconds. Thus this means that it will check for the element on the web page at every 5 seconds for the maximum time of 30 seconds. If the element is located within this time frame it will perform the operations else it will throw an"**ElementNotVisibleException**"

## Difference between Implicit Wait Vs Explicit Wait

|  |  |
| --- | --- |
| **Implicit Wait** | **Explicit Wait** |
| * Implicit Wait time is applied to all the elements in the script | * Explicit Wait time is applied only to those elements which are intended by us |
| * In Implicit Wait, we need **not** specify "ExpectedConditions" on the element to be located | * In Explicit Wait, we need to specify "ExpectedConditions" on the element to be located |
| * It is recommended to use when the elements are located with the time frame specified in implicit wait | * It is recommended to use when the elements are taking long time to load and also for verifying the property of the element like(visibilityOfElementLocated, elementToBeClickable,elementToBeSelected) |

**Conclusion:**

**Implicit, Explicit** and **Fluent Wait** are the different waits used in selenium. Usage of these waits are totally based on the elements which are loaded at different intervals of time. It is always **not recommended** to use **Thread.Sleep()**while[Testing](https://www.guru99.com/software-testing.html)our application or building our framework.

Selenium uses what is called locators to find and match the elements of your page that it needs to interact with. There are 8 locators strategies included in Selenium:

* Identifier
* Id
* Name
* Link
* DOM
* XPath
* CSS
* UI-element

6.1. Identifier

works with the **id** and **name** attributes of your html tags. Let’s consider the following example:

<html>

<body>

<form id="login">

<input name="username" type="text"/>

<input name="password" type="password"/>

<input name="submit" type="submit" value="Continue!"/>

</form>

</body>

</html>

Valid locators for this snippet are :

* **identifier=login**
* **identifier=username**
* **submit**

**PROS**:

* This strategy doesn’t rely on the structure of the page and will work even if it changes.

**CONS**:

* Easily matches several elements: try to name your **username** field as **login**

|  |  |
| --- | --- |
|  | **Note** |
| Usually, you don’t need to specify the locator prefix, Selenium will be able to infer the locator type by itself |

6.2. Id

The Id strategy looks for an element in the page having an **id** attribute corresponding to the specified pattern. **<label id="my\_id" />** will be matched by a locator like **id=my\_id** or just **my\_id**

**PROS**:

* Each id is supposed to be unique so no chance of matching several elements

**CONS**:

* Works well only on elements with fixed ids and not generated ones

6.3. Name

Like the Id strategy, but on the **name** attribute. You can also specify a filter to refine your locator. Currently, there are two filter types :

* **Value** : matches elements with a **name** attribute and where the **value** follows a pattern. The following example illustrates the interest of filters :
* <html>
* <body>
* <div id="pancakes">
* <button type="button" name="pancake" value="Blueberry">Blueberry</button>
* <button type="button" name="pancake" value="Banana">Banana</button>
* <button type="button" name="pancake" value="Strawberry">Strawberry</button>
* </div>
* </body>

</html>

Scenario:

we just added a strawberry pancake in our application and we want to test that the button that adds it into the cart works. With a locator like **name=pancake**, Selenium will find 3 elements and return the first one : the test will never fail even if the strawberry button is not here! Use a value filter like **name=pancake value=Strawberry** and the locator successfully identifies the Strawberry button.

* **Index** : same as name but works with an index. Using the previous example, the locator **name=pancake index=2** will select the Strawberry button.

|  |  |
| --- | --- |
|  | **Tip** |
| the index starts at 0 |

**PROS**:

* Works well with fixed list of similar elements

**CONS**:

* Difficult to use with data-bound lists

6.4. Link

This strategy is intended to select links only and selects the anchor element containing the specified text: **link=The text of the link**

**PROS**:

* Will only select anchor elements
* Useful when testing navigation

**CONS**:

* You have to know the text of the link before

6.5. DOM

The DOM strategy works by locating elements that matches the javascript expression refering to an element in the DOM of the page.

* **dom=document.div['pancakes'].button[0]**
* **document.div[0].button[2]**
* **dom=function foo() { return document.getElementById("pancakes"); }; foo();**

**PROS**:

* Javascript allows you to build dynamic locators

**CONS**:

* Relies on the structure of the page

6.6. XPath

While DOM is the recognized standard for navigation through an HTML element tree, XPath is the standard navigation tool for XML; and an HTML document is also an XML document (xHTML). XPath is used everywhere where there is XML. Valid XPath locators can be:

* **xpath=//button[@value="Blueberry"]**: matches the Blueberry button
* **//div[@id="pancakes"]/button[0]**: same thing

**PROS**:

* Allows very precise locators

**CONS**:

* Slower than CSS
* Relies on browser’s XPath implementation which is not always complete (especially on IE) and as such is not recommended for cross-browser testing

6.7. CSS

The CSS locator strategy uses CSS selectors to find the elements in the page. Selenium supports CSS 1 through 3 selectors syntax excepted CSS3 namespaces and the following:

| **pseudo-classes** | **pseudo-elements** |
| --- | --- |
| **:nth-of-type** | **::first-line** |
| **:nth-last-of-type** | **::first-letter** |
| **:first-of-type** | **::selection** |
| **:last-of-type** | **::before** |
| **:only-of-type** | **::after** |
| **:visited** |  |
| **:hover** |  |
| **:active** |  |
| **:focus** |  |
| **:indeterminate** |  |

* **css=div[id="pancakes"] > button[value="Blueberry"]** selects the button with its value property set at *Blueberry* if children of the *pancakes* div

**PROS**:

* Much faster than XPath
* Widely used
* Provides a good balance between structure and attributes
* Allows for selection of elements by their surrounding context

**CONS**:

* They tend to be more complex and require a steeper learning curve

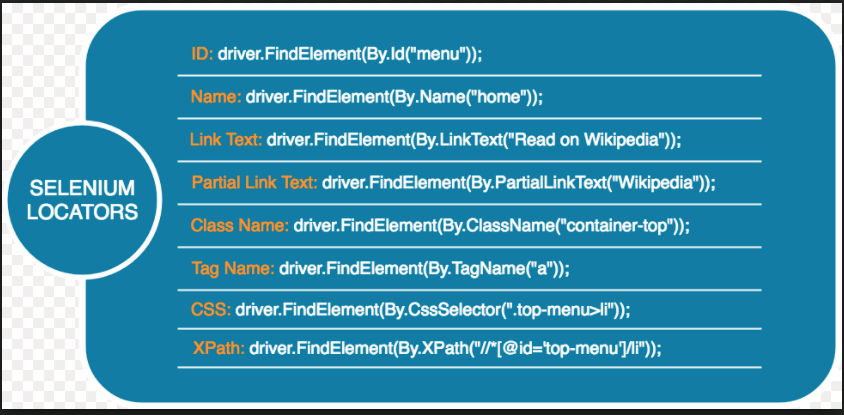
6.8. UI-Elements

* UI-element is a rather new locator
* It was at first a Selenium IDE extension
* It is now fully integrated into Selenium
* See the [Section 9.4, “UI-Elements:”](https://www.protechtraining.com/bookshelf/selenium_tutorial/testing_strategies#UI-Elements)

|  |  |
| --- | --- |
|  | **Caution** |
| As a general rule, keep in mind that if a locator matches several elements, only the first one will be effectively used by Selenium |

6.9. Structure-Dependent Or Not?

* Locators can be classified into two categories:
  + Structure-based locators: locators that rely on the structure of the page to find elements.
    - XPath
    - DOM
    - CSS
  + Attributes-based locators: locators that relies on the attributes of the elements to locate them
    - Identifier
    - Id
    - Name
    - Link
    - CSS
* You should consider this before choosing a locator strategy
* Most people choose CSS because it is the most flexible and gives a good balance between using structure and attributes to find the elements



<https://www.guru99.com/locators-in-selenium-ide.html>

<https://www.softwaretestingmaterial.com/javascriptexecutor-selenium-webdriver/>

<https://www.softwaretestingmaterial.com/selenium-interview-questions/>

<https://www.softwaretestingmaterial.com/api-testing-interview-questions/>

<https://www.softwaretestingmaterial.com/soap-interview-questions/>

APIs work using ‘requests’ and ‘responses.’ When an API requests information from a web application or web server, it will receive a response. The place that APIs send requests and where the resource lives, is called an endpoint.

What is HTTP?

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.

HTTP works as a request-response protocol between a client and server.

A web browser may be the client, and an application on a computer that hosts a web site may be the server.

Example: A client (browser) submits an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.

HTTP Methods

* **GET**
* **POST**
* **PUT**
* **HEAD**
* **DELETE**
* **PATCH**
* **OPTIONS**

The GET Method

**GET is used to request data from a specified resource.**

**GET is one of the most common HTTP methods.**

Note that the query string (name/value pairs) is sent in the URL of a GET request:

/test/demo\_form.php?name1=value1&name2=value2

**Some other notes on GET requests:**

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests is only used to request data (not modify)

The POST Method

**POST is used to send data to a server to create/update a resource.**

The data sent to the server with POST is stored in the request body of the HTTP request:

POST /test/demo\_form.php HTTP/1.1  
Host: w3schools.com  
name1=value1&name2=value2

**POST is one of the most common HTTP methods.**

**Some other notes on POST requests:**

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

The PUT Method

**PUT is used to send data to a server to create/update a resource.**

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

The HEAD Method

**HEAD is almost identical to GET, but without the response body.**

In other words, if GET /users returns a list of users, then HEAD /users will make the same request but will not return the list of users.

HEAD requests are useful for checking what a GET request will return before actually making a GET request - like before downloading a large file or response body.

The DELETE Method

**The DELETE method deletes the specified resource.**

The OPTIONS Method

**The OPTIONS method describes the communication options for the target resource.**

**Maven** is a build tool, in short a successor of ant. It helps in build and version control. However **Jenkins** is continuous integration system, where in **maven** is used for build. **Jenkins** can be used to automate the deployment process.

**What is Jenkins?**

Jenkins is the continuous integration tool. Jenkin's chief usage is to monitor any job which can be SVN checkout, cron or any application states. It fires pre-configured actions when a particular step occurs in jobs.

## Why Jenkins and Selenium?

* Running Selenium tests in Jenkins allows us to run our tests every time the software changes and deploy the software to a new environment when the tests pass.
* Jenkins can schedule the tests to run at specific time.
* we can save the execution history and Test Reports.
* Jenkins supports Maven for building and[Testing](https://www.guru99.com/software-testing.html)a project in continuous integration.

**Why Maven & Jenkins**

Selenium WebDriver is great for browser automation. Integrating Maven with Selenium provides

Maven provides support for managing the full lifecycle of a test project.

* Maven is used to define project structure, dependencies, build, and test management.
* Using pom.xml(Maven) we can configure dependencies needed for building testing and running code.
* Maven automatically downloads the necessary files from the repository while building the project.