**Java Notes**

Basic Program 🡪

**class** MyClass{

**public** **static** **void** main (String[]args) {

System.***out***.println("Hello VJ, you are doing good");

}

}

Here, public – anyone can access it.

static – method can be run without creating instance of a class containing the

main method.

void - method does not return any value.

main - name of the method.

Java Comments 🡪

Single line comments start with two forward // slashes.

Multiline comments start with forward slash and asterisk /\* and ends with an

asterisk and forward slash \*/

Data Types 🡪

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

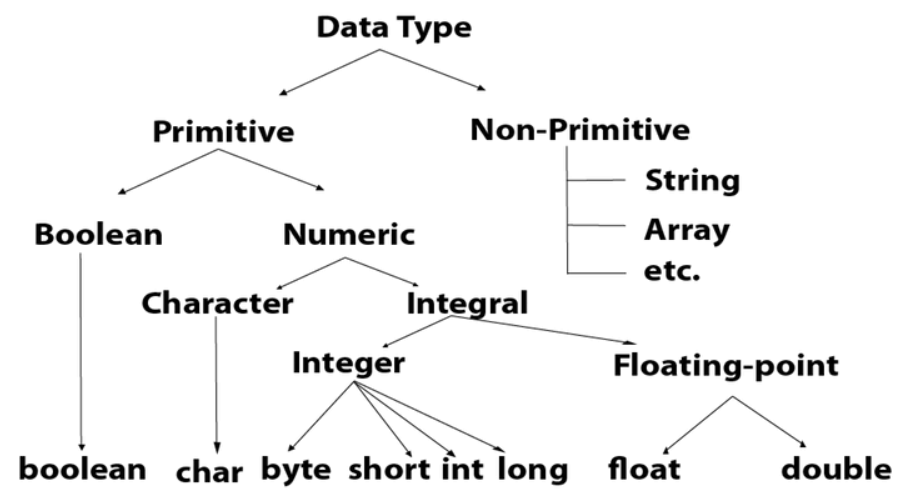
Example – String name = “Vijay”; // String must be written in double quotes

int age = 29;

double sz = 45.62;

char ch = ‘z’; // char must be written in single quote

boolean online = true;



Accepting Value/Input from user 🡪

**import** java.util.Scanner;

**class** MyClass{

**public** **static** **void** main (String[]args) {

Scanner sc = **new** Scanner(System.***in***);

String i = sc.nextLine();

System.***out***.println(i);

}

}

In order to accept integer from user, nextLine should be replaced by nextInt.

Reading a file 🡪

File file = **new** File("input.txt");

Scanner sc = **new** Scanner(file);

**while** (sc.hasNextLine()) {

String i = sc.nextLine();

System.***out***.println(i);

}

Conditional Statements 🡪

**If Else Statement**

**import** java.util.Scanner;

**class** MyClass{

**public** **static** **void** main (String[]args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter your age ");

**int** age = sc.nextInt();

**if** (age < 16) {

System.***out***.println("You are too young ");

} **else** {

System.***out***.println("Welcome to the world ");

}

}

}

**Switch Statement**

**import** java.util.Scanner;

**class** MyClass{

**public** **static** **void** main(String[]args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter your day umber in week ");

**int** day = sc.nextInt();

**switch**(day) {

**case** 6:

System.***out***.println("Toady is Saturday");

**break**;

**case** 7:

System.***out***.println("Today is Sunday");

**break**;

**default**:

System.***out***.println("Today is working day");

}

}

}

**While Loop**

**class** MyClass{

**public** **static** **void** main(String[]args) {

**int** num = 5;

**while** (num > 1) {

System.***out***.println(num);

num--;

}

}

}

**For Loop**

**class** MyClass{

**public** **static** **void** main(String[]args) {

**int** x = 8;

**for** (**int** i=0; i<x; i++) {

System.***out***.println(i);

}

}

}

**do….while Loop**

**class** MyClass{

**public** **static** **void** main(String[]args) {

**int** x=0;

**do** {

System.***out***.println(x);

x++;

}**while**(x<5);

}

}

**Arrays**

Arrays can be declared as,

int[] MyArr = new int[5];

Arrays can be defined as,

String[] MyNames = {“A”, “B”, “C”, “D”};

**Finding sum of array variables**

**class** MyClass{

**public** **static** **void** main (String[] args) {

**int**[] MyArr = {5,15,86,3652,4};

**int** sum = 0;

**for** (**int** i=0; i<MyArr.length; i++) {

sum = sum + MyArr[i];

}

System.***out***.println(sum);

}

}

**Enhanced for loop for arrays**

The enhanced for loop (also known as ‘for each’ loop) is used to transvers elements in an array.

**class** MyClass1{

**public** **static** **void** main (String[] args) {

String[] MyArr = {"I", "will", "do", "it"};

**for** (String t: MyArr) {

System.***out***.println(t);

}

}

}

**import** java.util.Scanner;

**public** **class** MyClass {

**public** **static** **void** main(String[] args) {

Scanner st = **new** Scanner (System.***in***);

System.***out***.println("Enter mail ");

String str = st.nextLine();

**int** len = str.length();

**char** [] li = **new** **char**[len];

li = str.toCharArray();

**for** (**int** i=0; i<len; i++) {

**if** (li[i]=='@') {

**for** (**int** j=i; j<len; j++) {

System.***out***.print(li[j]);

**if** (li[j]=='.') {

**break**;

}

}

}

}

}

}

**Object Orientated Programming**

Class defines what the object will be. Classes can be described as blueprints, descriptions or definition of an object.

Characteristics of an object are called attributes. Attributedescribes object’s current state.

Method defines behavior of an object. A method is a collection of statements that are grouped together to perform an operation.

**Creating new method in a class**

**class** MyClass{

**static** **int** twosum (**int** a, **int** b) {

**int** sum = a + b;

**return** sum;

}

**public** **static** **void** main (String[] args) {

**int** output = *twosum* (10,15);

System.***out***.println(output);

}

}

**Creating new classes and instantiating its objects**

**class** Vehicle{

**int** wheels = 4;

String colour = "White";

**void** horn() {

System.***out***.print("Beep !");

}

}

**class** MyClass{

**public** **static** **void** main (String[] args) {

Vehicle BMW = **new** Vehicle();

BMW.horn();

System.***out***.println(BMW.colour);

}

}

Constructor 🡪

Constructor is a member method with the same name as class name.

Object has two things, it knows something and it does something. Object knows something with the help of variables and does something with the help of methods.

To work with any non static method, we require object of that associated class.

‘new’ keyword is used to provide a memory.

Providing memory to objects is done by constructor.

Constructor overloading is possible.

Constructor never return anything.

                                                Vehicle obj1 = new Vehicle();

                                obj1 🡪 reference of Vehicle class is created as obj1

                                new 🡪 keyword used to provide the memory

                                Vehicle () 🡪 It allocates the memory equal to class Vehicle()

Constructor is,

-          Member method

-          Same name as classname

-          It will never return anything, so never mention ‘void’ while defining constructor

-          It is used to allocate memory

Polymorphism, Method Overriding 🡪

-          Polymorphism is of three types Method Overloading, Method Overriding and Constructor Overloading (same as method overloading but for constructors).

-          If you have two methods with same name and same parameters in two different classes (in which one is inherited from other), then is ‘Method Overriding’.

-          If you have two methods with same name and different parameters, then it is ‘Method Overloading’.

Encapsulation 🡪

-          Encapsulation is used to safe guard the data. By using the logs (println message), we can came to know that variable is changed and someone is trying to access the variable.

-          The concept of binding data with method is called ‘encapsulation’.

-          In case of object and class, in order to change variable value, you should use method. To set the value of variable and to get the value of variable you should use method.

-          First step of encapsulation is to define class variables as private, so that those will not be accessible outside the class.

-          Getter (get method) and Setter (set method) are used for this purpose. Get and Set methods should be equal to the number of variables and these methods should be public.

Access Modifiers à

-          In Java access modifiers are ‘final’, ‘abstract’, ‘private’, ‘public’, ‘protected’.

-          Only final, abstract and public modifiers can be used with class.

-          If you do not mention ‘public’ keyword to class, then you can’t access that class outside the package. Same case applies for methods also.

-          We can make one method private and can call this private method from other public method like we do in case of variables for encapsulation.

-          If you make any variable or method as,

o   Private – can be accessed in specific class

o   Default (no access modifier mentioned, also known as ‘package private’) – can be accessed in specific package

o   Public – can be accessed outside the package

o   Protected – can be accessed in subsiding class

Private keyword,

**class** Abc

{

**public** **void** method5()

       {

              method2();

       }

**private** **void** method2()

       {

              System.***out***.println("Hello, did it");

       }

}

**public** **class** MyClass4

{

**public** **static** **void** main(String[] args) {

              Abc obj1 = **new** Abc();

              obj1.method5();

       }

}

Protected keyword,

**class** A

{

**protected** **static** **int** *i* = 5;   // static keyword is used so that it can be accessed in static main method

}

**public** **class** MyClass3 **extends** A

{

**public** **static** **void** main(String[] args)

       {

              System.***out***.println(*i*);

       }

}

**A final class is a class that can't be extended.** Also methods could be declared as final to indicate that cannot be overridden by subclasses. Preventing the class from being subclassed could be particularly useful if you write APIs or libraries and want to avoid being extended to alter base behavior.

Constructor can not be associated with ‘final’ keyword but it can associate with ‘private’ keyword.

**Interface**

-          Interface is special kind of class which contains the method declaration only and not the method body.

- It contains variables which can’t be changed.

-          In technical words, ‘It has static constants and abstract methods only’.

Declaration / Signature of Interface Class

Interface declaration à

**package** Interface;

**public** **interface** Base {

**int** ***x*** = 90;

**public** **void** sum ();

}

Accessing interface à

**package** Interface;

**public** **class** Child **implements** Base

       {

**public** **static** **void** main(String[] args) {

              Child obj1 = **new** Child();

              obj1.sum();

       }

**public** **void** sum()

       {

              System.***out***.print("I am defined in Child Class");

       }

}

Webdriver driver = new ChromeDriver();  à Here we are creating new object of ChromeDriver class by referring to Webdriver interface.

**Abstract**

-          Abstraction is a process of showing the essential features and hiding the background details.

-          Any class with abstract keyword will be known as Abstract Class.

-          In a class even if single method is abstract method then class will become Abstract class.

-          An abstract class can contains abstract method and non-abstract method as well.

-          We cannot create object of Abstract class because Java says, “If your class is not fully implemented then we will not allow you to create object”.

-          Interface class is 100% abstract class.

- Abstract class can have a constructor.

Interface class à

**package** AbstactLearn;

**public** **interface** Emp {

**public** **void** salary();

**public** **void** goodies();

**public** **void** work();

}

Abstract class à

**package** AbstactLearn;

**public** **abstract** **class** ITDept **implements** Emp{

**public** **abstract** **void** salary();

**public** **void** goodies() {

              System.***out***.print("Laptop ");

       }

**public** **void** work() {

              System.***out***.print("Automation ");

       }

}

Main method à

**package** AbstactLearn;

**public** **class** HRDept **extends** ITDept {

**public** **static** **void** main(String[] args) {

              // 1st Scenario

              HRDept hr = **new** HRDept();

              hr.goodies();

              hr.work();

              hr.salary();

              hr.bonus();

              // 2nd Scenario

              ITDept it = **new** HRDept();

              it.goodies();

              it.work();

              it.salary();

//            it.bonus();       can not call bonus as bonus method belongs to HRDept but 'it' object has reference of ITDept

       }

**public** **void** salary() {

              System.***out***.println("CTC is good");

       }

**public** **void** bonus() {

              System.***out***.println("Bonus is best ");

       }

}

**Difference Between Interface and Abstract Class**

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| If we don’t know anything about implementation and we just have requirement specification, then we go for interface. | If we are talking about implementation but not completely (partial implementation) then we go for Abstract class. |
| All methods of Interface are always Public and abstract. So interface is 100% Abstract class. | In Abstract Class every method need not to be Public and Abstract, we can have concrete methods also. |
| All variables present in Interface are always public, static and final | NO such restriction on Abstract class variables. |
| For interface variables, we need to perform initialization at the time of declaration otherwise we get compile time error. | For abstract class variables, it is not required to perform initialization at the time of declaration. |
| Constructor concept is not applicable in interface class. | For abstract class, we can declare Constructor. |

Keyword ‘this’  à

- It is used to distinguish between local class \ method variables and arguments that method is accepting.

-          In order to use instance variable, we need to use ‘this’ keyword.

-          ‘this’ is not just an object but it represents current instance.

- It is used to indicate difference global and local variable.

**class** Abc{

**private** **int** x;

**public** **void** setX(**int** x) {

**this**.x = x;

              System.***out***.println(x);

       }

}

-          Variables can be grouped in three types as instance, local and class variables.

-          Instance variables are variables that are inside a class but outside a method.

-          Local variables are inside a method.

-          Class variables are defined inside a class but are of static type.

‘static’ Keyword à

- In java static keyword is used with variables (known as class variables), methods (known as class methods) and block.

-          Static variables are those variables which are same for all the objects like name of CEO is same for all the employees of the organization.

-          To call a static variable or method, we do not require object. We can call it directly using class name.

-          In same class, we cannot use non static variable in static method.

-          ‘static block’ is a special block in java class. It doesn’t matter how many times you create object of this class, it will execute only once.

-          To initialize non static variables, we use constructor. But to initialize static variable we use static block.

-          Constructor will get execute when you create an object. But static block will execute when you load a class.

-          Static block will get executed first and then constructor because class loads first then only you create the object.

**class** Emp

{

**int** EmpID;

**static** String *ceo*;

**static**    // static block, it executes when you loads a class

       {

*ceo* = "Brother";

       }

}

Wrapper Class à

-          For every primitive data type, we have a respective class. Like,

o    int  -  Integer

o    float  -  Float

o    double  -  Double

o    char  -  Character

-          Wrapper classes are widely used while working with collection, framework and advanced frameworks.

-          Converting a variable into object is wrapping, taking out the value from object is unwrapping.

We can define variable as, int i = 5;      // using primitive data type

It can also be defined as, Integer i = new Integer(5);         // using wrapper class

                           Integer obj1 = new Integer(i)    // this is wrapping

                           int j = obj1.intValue();         // this is unwrapping

Super Method  à

-          When you create a object of subclass, it is calling superclass constructor and subclass constructor.

-          Though user has created two constructors (one without any argument and other parameterized constructor) both in subclass and superclass, after creating instance of subclass by passing an argument by default it will call the parameterized constructor of subclass and non-parameterized constructor of superclass.

-          In subclass constructor, by default super() method is present in it. If user wants to call parameterized constructor of superclass also then he should send argument with super method.

- Means to call the parameterized constructor of superclass, Super method is used.

‘geckodriver.exe’ is Firefox browser specific driver executable, ‘chromedriver.exe’ is chrome browser and ‘IEDriverServer.exe’ is Internet Explorer specific driver executable.

**Apache POI 🡪**

* It is open source library to read excel file.
* We simply download this Apache POI, import and use them.

XSSFWorkbook class from Apache POI is used to work with .xlsx file

HSSFWorkbook class from Apache POI is used to work with .xls file

* **Importing Apache POI,**
* While creating new JAVA project in eclipse, import all the extracted .jar file from C:\Users\VJ\Downloads\poi-bin-4.1.1-20191023\poi-4.1.1 using ‘Add External Jars’ option.
* New -> Java Project -> (Provide Name) -> Next -> Libraries -> Add External JARs.

File src = **new** File("D:\\TestData.xlsx");

FileInputStream fis = **new** FileInputStream(src);

XSSFWorkbook wb = **new** XSSFWorkbook(fis); // Class from Apache POI

XSSFSheet sheet1 = wb.getSheetAt(0);

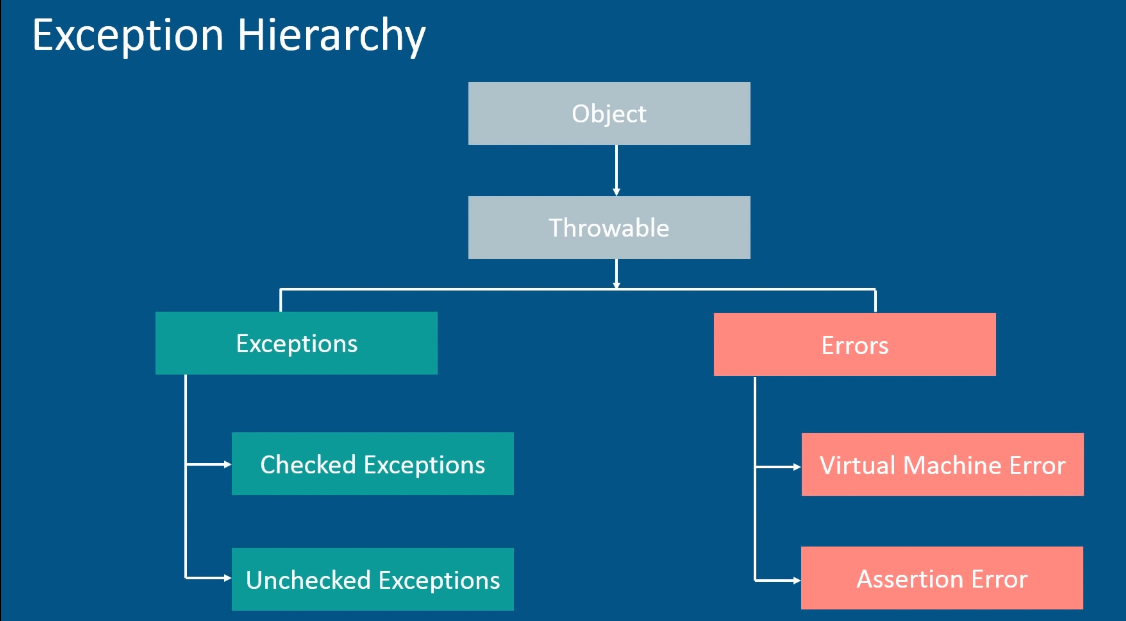
String data0 = sheet1.getRow(0).getCell(1).getStringCellValue();

System.***out***.println("Data from excel is " + data0);

wb.close();

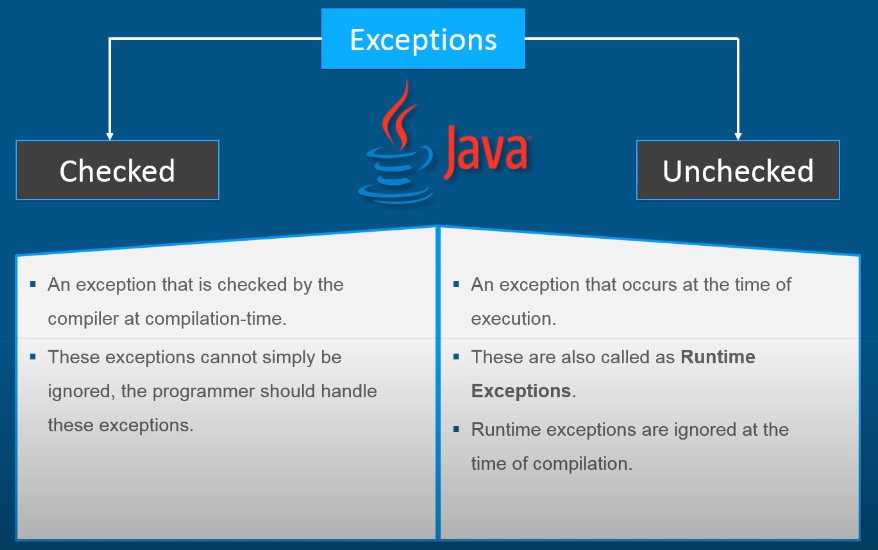
**Exceptions 🡪**

* Exception means abnormal condition.
* Exception will terminate the complete program.
* In technical words, Exception is a class.
* Main reason of exception handling is to maintain the normal flow of application.
* Hierarchy of Java Execution classes is as below,



|  |  |
| --- | --- |
| **Error** | **Exception** |
| It is impossible to recover from error. | It is possible to recover from Exceptions by handling them. |
| Errors are of unchecked type. | Exceptions can be Checked or Unchecked. |
| Error occurs at Runtime. | Exception occur at Runtime or Compile time. |

* Types of Exceptions,
  + Checked
  + Unchecked



* Checked exception -> until you don’t handle them, you wont be able to compile your code like IOException.
* Unchecked exception -> It is runtime exception like nullPointerException, ArrayIndexOutofBoundException

Handling Exception 🡪

Try Catch -> In try block we write the code which might throw exception and in Catch we write which type of exception it will throw. If we don’t know that, we can use finally. It doesn’t matter, whether you have catch block or not ‘finally’ code of block will always run.

**try**

{

**int** i = 10/0;

}

**catch**(ArithmeticException e)

{

System.***out***.println(e.getMessage());

**}**

**finally**

{

System.***out***.println("Its End of program ");

}

* User can write multiple catch blocks for same try block. **catch** (Exception e)

has to be written at end as it is parent class and if we write it as first catch block then specific catch blocks under it will never get executed.

* Dead code in programming is the code that will never get executed.

**Throws –** It is used to suppress the exceptions. It does not throw any exception but specifies that there may occur an exception.

It is compile type exception handling. It gives info to programmer that there may occur an exception so it is better to handle exception handling code. It is mainly used to handle Checked exception. It needs to be declared with method signature.

**Throw** – To forcefully throw an error, we use Throw keyword. Means for customizing (creating) the exception as per our need, we use ‘throw’ exception.

Example – If you want some output within some specific range then we can use Throw keyword if output moves out of it.

**public** **class** ThrowLearn {

**public** **static** **void** main (String[] args) {

**int** i = 5;

**try** {

**if** (i<10) {

**throw** **new** ArithmeticException();

}

}

**catch**(ArithmeticException e) {

System.***out***.println("Catched it");

}

}

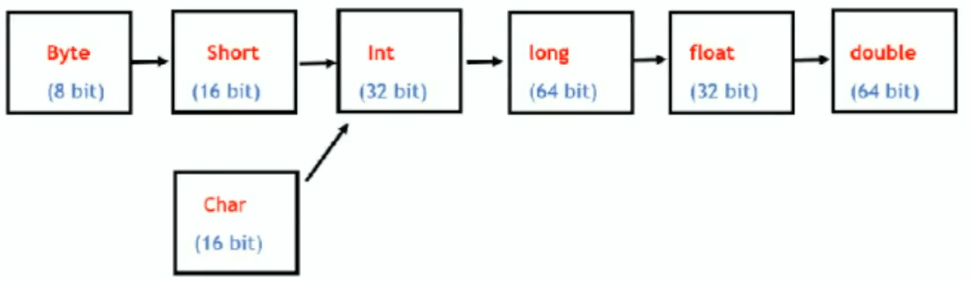
}

**Commonly used exceptions ->**

* NullPointerException: application try to use an object reference which has a null value.
* IOException: This exception happens when there is a failure during reading a file.
* NoSuchElementException: This exception is due to accessing an element which is not available on the page.
* NoSuchAttributeException: While trying to get attribute value but the attribute is not available in DOM.
* ElementNotVisibleException: If selenium tries to find an element but the element is not visible within the page.
* NoAlertPresentException: If a user tries to handle an alert box but the alert is not present.
* NoSuchWindowException
* NoSuchFrameException
* TimeoutException: This exception occurs when a command completion takes more than the wait time.
* ArithmeticException
* ArrayIndexOutOfBound Exception
* InputMismatchException

**Typecasting in Java**

* Typecasting is required where we assign value of variable type to other variable type.
* Typecasting is not possible for Boolean data type.
* It is of two types, implicit and explicit.
* Implicit casting means “smaller range to larger range”. It is done by compiler automatically.



* Explicit means “larger range to smaller range”.
* While assigning larger range data type to smaller range data type, we need to write smaller data type in round brackets.
* Information loss in Explicit data type.
  + Truncation
  + Out of range

**Lambda Functions (new enhancement in JAVA 8)**

Lambda Expression  à

Uptill now to implement the methods of Interface, we had two ways.

1)      Create a separate class to implement them

2)      While creating reference of Interface class, define that method

But using lambda in JAVA 8, we can define the method on the fly without using normal method defining syntax.

**interface** A

{

**public** **void** show();

}

**class** Xyz **implements** A     // use of this class is just to implement method of A interface and nothing else

{

**public** **void** show()

       {

              System.***out***.println("Hello Dude ");

       }

}

**public** **class** MyClass2 {

**public** **static** **void** main (String[] args)

       {

              A obj1 = **new** Xyz();

              obj1.show();

       }

}

So instead of having separate class to implement method of interface A, we can implement method while creating interface object itself, as seen below

**public** **class** MyClass2 {

**public** **static** **void** main (String[] args)

       {

              A obj1 = **new** A()

              {

**public** **void** show()

                     {

                           System.***out***.println("Hello ");

                     }

              };

              obj1.show();

       }

}

Instead of writing complete code while passing, we can simply use lambda expressions as,

A obj1 = () -> System.***out***.println("Hello ");

**Interfaces (and their methods) of Selenium 🡪**

1. SearchContext
   1. findElement()
   2. findElements()
2. WebDriver
   1. close()
   2. get()
   3. getTitle()
   4. getCurrentUrl()
   5. getWindowHandle()
   6. getWindowHandles()
   7. manage()
   8. navigate()
   9. quite()
   10. switchTo()
3. TakesScreenshot
   1. getScreenshotAs()
4. JavascriptExecutor
   1. executeScript()
   2. executeAsyncScript()
5. Navigation
   1. to()
   2. back()
   3. forward()
   4. refresh()
6. WebElement
   1. clear()
   2. sendKeys()
   3. click()
   4. getAttribute()
   5. getTagName()
   6. getText()
   7. getCssValue()
   8. getLocation()
   9. getRect()
   10. getSize()
   11. isDisplayed()
   12. isEnabled()
   13. isSelected()
   14. submit()
7. Alert
   1. getText()
   2. accept()
   3. dismiss()
8. Action
   1. contextClick()
   2. dragAndDrop()
   3. doubleClick()
   4. moveToElement()
   5. build()
9. ExpectedConditions
   1. alertIsPresent()
   2. elementToBeClickable()
   3. elementToBeLocated()
   4. titles()
   5. visibilityOf()
10. Options
11. TimeOuts
12. OutputType
13. TargetLocation

**Locators**

 Locator is a technique to find any element on web page.

-          Different locators in selenium,

o   ID

o   Name

o   ClassName

o   XPath

o   CSS

o   LinkText

o   PartialLinkText

o   Tagname (e.g. ‘a’, ‘iframe’)

-          Direct locators are always faster (id, name, classname)

-          For dynamic elements use XPath, CSS

XPath Syntax –

//tagname[@attribute\_name=’value’]

Here attributes are id, name, class, title, role, style, src

Commonly used methods with XPath –

o   contains()

o   starts-with()

o   text()

Example –

//input[@id=’user login’]

//input[@class=’input’][@name=’log’]

//input[@class=’input’ and @name=’log’]

//input[@class=’input’ OR @name=’log’]

//a[contains(@href,’login’)]

//a[text()=’Posts’]

//\*[starts-with(@id, ‘edit’)]

//\*[@id=’menu-dashboard’]//following::a        //will locate all following anchor tags

//\*[text()=’usename’]//following::input[1]

//\*[@id=’wp-submit’]//preceding::input

//\*[@id=’welcome-panel’]/div[1]//ancestor::div[1]      // moved to parent tag

XPath axes methods:

These XPath axes methods are used to find the complex or dynamic elements.

Following 🡪

//input[@type='text']//following::input[1]

Ancestor 🡪

The ancestor axis selects all ancestors element (grandparent, parent, etc.) of the current node

Xpath=//\*[text()='Enterprise Testing']//ancestor::div

There are 13 "div" nodes matching by using "ancestor" axis. If you want to focus on any particular element then you can use the below XPath, where you change the number 1, 2 as per your requirement:

Xpath=//\*[text()='Enterprise Testing']//ancestor::div[1]

Child 🡪

Selects all children elements of the current node

Xpath=//\*[@id='java\_technologies']//child::li[1]

Preceding 🡪

Select all nodes that come before the current node

Xpath=//\*[@type='submit']//preceding::input

Following-sibling 🡪

Select the following siblings of the context node. Siblings are at the same level of the current node

xpath=//\*[@type='submit']//following-sibling::input

 Parent 🡪

Selects the parent of the current node

Xpath=//\*[@id='rt-feature']//parent::div

|  |  |
| --- | --- |
| **XPATH Locator** | **CSS Selector** |
| XPATH stands for XML Path. | CSS (Cascading Style Sheet) selector is used to find element in HTML DOM using style sheet language. |
| With xpath we can traverse in forward or backward director in DOM hierarchy means here we can locate the parent element using child element. | With CSS selector, we can traverse only in forward direction in DOM hierarchy. |
| It can locate the element using the text it contains. | CSS can’t select the element using the text it contains. |
| IE do not have native xpath engine. Selenium injects its own xpath engine, so xpath is show in IE. | CSS is fast in IE. |
| XPath general syntax is  //tagname[@attribute\_name=’value’]  //input[@name = “Email”] | To find an element by an,  ID 🡪 #  Class name 🡪 .  Input[name=”Email”] |

**Handling Webtable**

Following tags are used to access different data in webtable.

<th> - Table heading

<tr> - Table row

<td> - Table data

Finding number of rows and columns in a web table à

List<WebElement> rows = driver.findElements(By.*xpath*("//table[@class='dataTable']/tbody/tr"));

List<WebElement> columns = driver.findElements(By.*xpath*("//table[@class='dataTable']/tbody/tr[1]/td"));

System.***out***.println(rows.size());

System.***out***.println(columns.size());

Dynamic webtable data access à

String rowdata = driver.findElement(By.xpath("//table[@class='dataTable']/tbody/tr[1]/td[1]")).getText();

System.out.println(rowdata);

String dynadata = driver.findElement(By.xpath("//a[contains(text(),'Star Cement')]/parent::\*/following-sibling::td[3]")).getText();

System.out.println(dynadata);

**Handling Windows**

Finding window handle of current browser à

String parent = driver.getWindowHandle();

Finding window handle of all open browsers à

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

Finding number of all open windows  à

int count = allWindows.size();

Converting all window handles from set to list  à

ArrayList<String> tabs = **new** ArrayList<>(allWindows);

Getting window handle of specific tab  à

tabs.get(0);

Switching to specific tab based on window index à

driver.switchTo().window(tabs.get(1));

Switching to specific tab based on window handle à

driver.switchTo().window(child);

**Q.) In Selenium, where we use concept of Set ?**

Ans 🡪 When we use getWindowHandles method, it returns the set of Strings. We convert this Set in ArrayList to use indexing as Set do not support indexing.

**Scroll Commands**

// Scroll to specific element

JavascriptExecutor jse = (JavascriptExecutor)driver;

WebElement dis = driver.findElement(By.*xpath*("//span[contains(text(), 'Up to 50% off home essentials | Amazon Brands & more')]"));

jse.executeScript("arguments[0].scrollIntoView(true)", dis);

**executeScript method accepts two Parameters:**

**script** The JavaScript to execute

**args** The arguments to the script. May be empty

// Scroll to specific location pixel

jse.executeScript("scroll(0,400)"); // 0 is horizontal and 400 pixel is vertical scroll

//  Scroll till bottom of page

jse.executeScript("window.scrollTo(0,document.body.scrollHeight)");

//  Finding scroll bar moving and it horizontally from left to right

WebElement slideBar = driver.findElement(By.*xpath*("//input[@type='range']"));

slideBar.sendKeys(Keys.***END***);

//  Scroll dynamically for loading page

**long** initialHeight = (**long**)(jse.executeScript("return document.body.scrollHeight"));

**while** (**true**) {

jse.executeScript("window.scrollTo(0,document.body.scrollHeight)");

       Thread.*sleep*(2000);

**long** currentHeight = (**long**)(jse.executeScript("return document.body.scrollHeight"));

**if** (initialHeight == currentHeight) {

**break**;

       }

initialHeight = currentHeight;

}

Complete video is available at <https://www.youtube.com/watch?v=BMaIfW8ikmU>

**Action class related commands**

Hover on element à

Actions action = **new** Actions(driver);

action.moveToElement(driver.findElement(By.*className*("menulink"))).build().perform();

Drag and Drop element à

WebElement source = driver.findElement(By.*id*("draggable"));

WebElement target = driver.findElement(By.*id*("droppable"));

action.dragAndDrop(source, target).build().perform();

OR     action.clickAndHold(source).moveToElement(target).release().build().perform();

Right Click à

WebElement rightClickElement = driver.findElement(By.*cssSelector*(".context-menu-one"));

action.contextClick(rightClickElement).build().perform();

**Handling Alerts in Selenium**

driver.findElement(By.*name*("proceed")).click();

Alert alert = driver.switchTo().alert();

System.***out***.println(alert.getText());

alert.accept();

In case of windows pop up (like Browse option to upload any file from system), do not use .click, we need to use .sendkeys and send system path of file to be uploaded).

**Handling iFrames**

Switching to particular iFrame 🡪

WebElement frame = driver.findElement(By.xpath(“//[@id’Single’]/iframe”));

driver.switchTo().frame(frame);

To come out of frame to home page 🡪

driver.switchTo().defaultContent();

Finding total number of frames in a webpage 🡪

int total\_frames = driver.findElements(By.tagName(“iframe”)).size();

System.out.print(total\_frames);

String operations for Selenium à

Converting string to character array

char[] ch1 = str1.toCharArray()

Converting character array to string

String str2 = new String (ch2) // ‘ch2’ is character array

Converting a single character to string

String str3 = Character.toString(ch3)

              String s1 = "Selenium Webdriver";

**boolean** status = s1.startsWith("Selenium");

              System.***out***.println("Status is "+status);

**boolean** status2 = s1.endsWith("driver");

**boolean** status3 = s1.equals(s2);

**boolean** status3 = s1.equalsIgnoreCase(s2);

**boolean** status5 = s1.contains("world");

String []Splited = fullname.split("-");

System.***out***.println(name2.trim());      // removes white space before and after the string

System.***out***.println(name.charAt(0));    // returns character at that index

       int i = s1.length();             // finds the total number of characters

       String s3 = s1.concat(s2);       // appends s1 with s2

       String s3 = s1 + s2;             // same as above

       s1 = s1.toUpperCase();           // converts string in upper case

-          String class objects are immutable means user can not modify string on same location.

-          If user tries to modify string, it creates new instance on heap memory means memory is wasted.

-          So StringBuffer class is introduced. It’s objects are mutable.

-          ‘compareTo’ function compares two strings. It returns 0 if both the strings are same (without ignoring upper / lower case), otherwise it returns +ve or –ve value by comparing unicode of mismatch.

              String str = **new** String("Hello");

// length()

**int** len = str.length();       // returns length of string

              System.***out***.println(len);

              // indexOf()

**int** index = str.indexOf("o");    // returns index value of provided character

              System.***out***.println(index);

              // charAt()

**char** a = str.charAt(2);        // returns character at provided index value

              System.***out***.println(a);

              // replace()

              String str1 = str.replace("ll", "ee");

              System.***out***.println(str1);         // returns a string that replaces old char with new char

              //  toLowerCase()

              String str2 = str.toLowerCase();     // returns string by converting to lower case

              System.***out***.println(str2);

              //  toUpperCase

              String str3 = str.toUpperCase();     // returns string by converting to upper case

              System.***out***.println(str3);

              //  compareTo()

**int** com = str.compareTo("hello");

              System.***out***.println(com);

              //  concat()

              String str4 = str.concat(" Boy");    // returns a string appended by provided string

              System.***out***.println(str4);

              //  trim()

              String str5 = str.trim();            // returns a string by trimming provided string with starting and ending white spaces

              System.***out***.println(str5);

StringBuffer  à

              StringBuffer sb = **new** StringBuffer("Welcome");

              sb.append(" to home");

              System.***out***.println(sb);       // appends provided string at the end

              sb.insert(2, "new");          // inserts provided string at specified index

              System.***out***.println(sb);

              sb.delete(2, 5);              // deletes string between provided index

              System.***out***.println(sb);

              sb.reverse();

              System.***out***.println(sb);       // reverses the string

              sb.reverse();

              sb.replace(0, 7, "Come");     // replaces the provided string within specified index range

              System.***out***.println(sb);

Q.) What is diff bet String and StringBuffer ?

- String objects are immutable while StringBuffer objects are mutable.

- Once we create an object of String class, we can’t perform any changes in the existing object. If we try to perform any changes, a new object will be created. This non changeable nature is called immutability.

- Once we create an object of StringBuffer class, we can perform any type of changes in the existing object. This changeable behavior is mutability.

StringBuffer and StringBuilder both are exactly same (including Methods and Constructors), except following points.

In String Buffer, every method is synchronized. So at a time, only one thread is allowed to operate on it means we can use thread safety here. But until first thread completes, second thread need to wait. So performance wise not recommended to use.

In String Builder, no method is synchronized. Any number of threads can operate simultaneously so thread safety is not achieved. Performance wise String builder is recommended.

Screen Shot  à

We take screen shot using getScreenshotAs() method of TakesScreenshot interface. We typecast our driver object to TakesScreenshot interface and using this object reference ‘ts’ we call getScreenshotAs() method.

**import** java.io.File;

**import** org.openqa.selenium.TakesScreenshot;

TakesScreenshot ts=(TakesScreenshot)driver;

File source = ts.getScreenshotAs(OutputType.FILE);

FileHandler.copy(source, new File("C:\\Users\\vtt1\\eclipse\_Dec23\\MyPro\\src\\facebook1.jpg"));

Selecting Dropdown Commands à

While working with drop down, we need to create instance of select class.

              WebElement month\_dropdown = driver.findElement(By.*id*("month"));

              Select month\_dd = **new** Select(month\_dropdown);

              month\_dd.selectByIndex(3);      // value is an attribute. It will select March as 0th index was Month title

                                month\_dd.selectByValue("10");   // It will select Octo

                                month\_dd.selectByVisibleText("Aug");   // This is most recommented

                                WebElement selected\_value = month\_dd.getFirstSelectedOption(); // get webelement of current selected option

              System.***out***.println(selected\_value.getText());                  // prints the selected value

              List<WebElement>month\_list = month\_dd.getOptions();

**int** total\_month = month\_list.size();

              System.***out***.println("Total month count is "+total\_month);   // prints total number of dropdown option

**for** (WebElement ele:month\_list)                           // prints all options from dropdown

              {

                     System.***out***.println(ele.getText());

              }

**Waits in Selenium**

There are three types of waits in Selenium. All waits are dynamic means if driver finds the element before reaching its maximum mentioned time limit, it will ignore the remaining time and move on.

Implicitly Wait 🡪

Implicit wait is applied globally means it will be applied to all the web elements with whom driver is interacting.

driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);

While interacting with web elements (like .click, .sendKeys) it will wait for max 20 seconds if element is not loaded. It is recommended for small applications.

Default value of implicit time is 0.

Explicitly Wait 🡪

Explicite wait will wait dynamically for specific conditions. Benefit of explicit wait is that we can change the time to specific locator according to our need. Generally for large applications explicitly wait will is preferred.

WebDriverWait wait = new WebDriverWait(driver, 30);

wait.until(ExpectedConditions.visibilityOf(element));

|  |  |
| --- | --- |
| **Implicit Wait** | **Explicit Wait** |
| It is applied globally means to all elements with whom driver is interacting. | It is applied to only those elements which are intended by us. |
| No need to specify any expected condition on elements to be located. | Need to specify a expected condition on elements to be located. |
| It is recommended when elements are loaded within a time that is mentioned in implicit wait. | It is recommended to use when some elements take longer time to load and also to verity property of element like visibilityOf, elementToBeClickble, elementToBeSelected. |

Fluent Time 🡪

It is used to tell webdriver to wait till the specific condition as well as the frequency with which we want to check condition before throwing an exception.

In explicit wait default polling time is 500ms, here we can change that as per our requirement.

**TestNG Framework**

-          TestNG (Testing Next Generation) is unit testing framework. Also known as TDD (Test Driven Development) framework.

-          Its purpose is design test cases in a proper systematic way.

o   It generates HTML report.

o   It has different annotations.

o   Many features like priority / sequence, dependency, grouping, data provider

-          TestNG is open source, free and available in the form of JAR files.

-          It can be used only with JAVA.

-          We never write main() method inside TestNG. TestNG automatically executes the class, so we don’t need main method executor.

-          Each and every annotation must be associated with some method.

 To install TestNG plugin à

-          In eclipse help -> Install new software, enter <https://beust.com/eclipse> and search TestNG will appear.

-          Now right click on project folder -> Properties -> Java Build Path -> Libraries -> Add Library -> Next -> Finish.

 -          Total number of test cases in a test suite are equal to the total @Test annotations in test suite.

o   @BeforeSuite

o   @ BeforeTest

o   @BeforeClass

o   @ BeforeMethod

o   @Test

o   @AfterMethod

o   @AfterClass

o   @AfterTest

o   @AfterSuite

-          Practically only @ BeforeMethod, @Test and @AfterMethod are used.

-          Before executing @Test everytime @BeforeMethod and @AfterMethod will be executed.

-          @Test cases are always independent.

- If you are not using any priority in your test method then TestNG assign by default

priority=0 to the Test Method. So if priority is assigned to only few test cases then test cases

with no priority will get executed first and then test cases with mentioned priority.

- If there is same priority assign to test methods then execution order will be alphabetically.

-          To see the HTML report, refresh the project folder in JAVA. ‘test-output’ folder will be created. Inside that folder, ‘index.html’ file will be generated.

Q.) What is difference between @BeforeTest and @BeforeMethod ?

* @BeforeTest will execute before starting of test (which is mentioned in testng.xml). If only one test (that may include many classes) is mentioned in testng.xml then it will execute only once throughout testcase. But @BeforeMethod will execute before executing each method during the testing.

-          Attributes of  @Testà

o   @Test(priority=1)

o    @Test(groups=”Logo”)  (in index.html we can see group wise report)

o   @Test(dependsOnMethod=”loginTest”)                    //If login passes then only this test case will be executed

o   @Test(invocationCount=10)                                        // It will run test case 10 times e.g. dumping data on server

o   @Test(invocationTimeOut=2)                                     // If this test case didn’t complete in 2 sec, then it will terminate this test case

o   @Test(expectedExceptions=NumberFormatException.call)        // If Number Format Exception comes, then also test case will pass, it will not fail. It can be used instead of try catch

-          To run some set of test cases again and again, we create XML Template file.

-          In src folder, right click -> other -> file->next-> enter ‘testng.xml’



Note 🡪

* In testng.xml always write class name as “package\_name.class\_name”, for default classes no need to mention package name.
* In testing.xml, do not provide .java extension for class names.

Passing parameter values to @Test using testing.xml

<parameter name=”url” value=”<https://login.yahoo.com/>”/>

<parameter name=”emailId” value=”[test@yahoo.com](mailto:test@yahoo.com)”>



 -          Using above parameters in script

@Test

@Parameters({“url”, “emailId”})

Public void yahooLoginTest(String url, String emailId)         // we can use url and emailId parameters in the script

 -          Data driven testing in TestNG framework

                @Test(dataprovider=”getData”) // here ‘getData’ is name attribute of dataprovider method

                Before defining method (containing data providing logic), we need to provide @dataprovider annotation

**import** org.testng.annotations.Test;

**public** **class** MyClass {

@Test(priority=1,description="About to start journey")

**public** **void** LoginPage() {

System.***out***.print("Hi Bro, you have logged in ");

}

* @Listener is used to generated reports / logs.

Parameterization in TestNG can be achieved by,

1)      Using @DataProvider annotation à

It is mandatory for a @DataProvider method to return the data in the form of array of Object class (**Object []**).

**import** [org.openqa.selenium.By](http://org.openqa.selenium.by/);

**import** org.openqa.selenium.WebDriver;

**import** org.openqa.selenium.WebElement;

**import** org.openqa.selenium.chrome.ChromeDriver;

**import** org.testng.annotations.DataProvider;

**import** org.testng.annotations.Test;

**public** **class** MyClass {

       @Test(dataProvider="worldpress")

**public** **void** MyTest(String user) **throws** InterruptedException {

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

              WebDriver driver = **new** ChromeDriver();

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

              driver.get("<https://wordpress.com/log-in>");

              WebElement ele = driver.findElement(By.*xpath*("//input[@id='usernameOrEmail']"));

              ele.sendKeys(user);

              Thread.*sleep*(5000);

              driver.quit();

       }

       @DataProvider(name = "worldpress")

**public** Object[] UserInput() {

              Object[] data = **new** Object[3];

              data[0]="VJ\_User\_One";

              data[1]="DS\_User\_two";

              data[2]="Ran\_Three";

**return** data;

       }

}

2)  Parameter Passing from ‘testng.xml’

**import** org.testng.annotations.AfterMethod;

**import** org.testng.annotations.AfterTest;

**import** org.testng.annotations.BeforeMethod;

**import** org.testng.annotations.BeforeTest;

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** File\_Testing {

       @Test

       @Parameters("emailid")

**public** **void** m1(String str1) {

              System.***out***.println("Test 1" + str1);

       }

       @Test

       @Parameters("password")

**public** **void** m2(String str2) {

              System.***out***.println("Test 2" + str2);

       }

       @BeforeMethod

**public** **void** beforeMethod() {

              System.***out***.println("Before method");

       }

       @AfterMethod

**public** **void** afterMethod() {

              System.***out***.println("After method");

       }

       @BeforeTest

**public** **void** beforeTest() {

              System.***out***.println("Before Test");

       }

       @AfterTest

**public** **void** afterTest() {

              System.***out***.println("After Test");

       }

}

testng.xml File  à

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "<http://testng.org/testng-1.0.dtd>">

<suite name=*"Practice Suite"*>

  <test name=*"Test Basics 1"*>

      <parameter name=*"emailid"* value=*"*[*tester456@gmail.com*](mailto:tester456@gmail.com)*"*/>

      <parameter name=*"password"* value=*"test@123"*/>

      <classes>

      <class name=*"File\_Testing"*/>

      </classes>

  </test>

</suite>

**Assertions in Selenium with TestNG framework**

Assert is a class in TestNG which helps to validate certain outcome. If assert fails then it throws AssertionError and not exception.

**import** org.testng.Assert;

**public** **class** MyClass {

@Test

**public** **void** test1() {

Assert.*assertEquals*(12, 13, “Message if assert fails”);

Assert.assertTrue(**False**);

}

Types of Assert 🡪

Hard Assert – means even if single assert fails, it will terminate the further test case execution and fails the test case.

Soft Assert – If validation fails then also it will continue with test execution and after executing entire script it will fail the test.

Whatever assert we have in hard we have same in softassert. Only the difference is we call these asserts using object of SoftAssert class. After using softAssert, we need to call assertAll() method.

SoftAssert assertion = **new** SoftAssert();

assertion.assertEquals(12, 13);

System.***out***.println("This will still execute");

assertion.assertAll();

Types ->

* + assertEquals
  + assertNotEquals
  + assertTrue
  + assertFalse
  + assertNull
  + assertNotNull

**ITestngListener 🡪**

ItestngListener has many methods like

* onTestStart
* onTestSuccess
* onTestFailure
* onTestSkipped

To use this listener, we need to create separate class that implements ITestListener.

we define above methods in that class (like we call Screenshot method in onTestFailure method).

**public** **class** CustomListener **extends** Base **implements** ITestListener{

**public** **void** onTestFailure(ITestResult result) {

failed(result.getMethod().getMethodName());

}

In order to know to our test case class that it should listen to above created listener, we need to use @listener annotation with listener class name at the starting of class declaration

@Listeners(CustomListener.**class**).

For more details, please visit C:\Users\VJ\ITestNGListener

Note – If your suite contains many classes and you want to use ITestngListener to all classes then instead of writing@Listeners(CustomListener.**class**) in every class, we can write it in testng.xml as

<suite name=*"Practice Suite"*>

<listeners>

<listener class-name = *"ListenerConcept.CustomListener"*/>

</listeners>

**Reporting in TestNG 🡪**

* TestNG will generate three reports. In working directory, ‘test-output’ folder will get created. Inside that folder ‘emailable-report.html’, testwise.html report under suite name folder and ‘index.html’ reports will get generated.

**Extent Report –** <http://extentreports.com/docs/versions/3/java/#basic-example>

* Extend report configuration should be done in BaseClass under @BeforeSuite tag.
* Extend report consist of three class,
  + ExtentHtmlReporter
  + ExtentReports
  + ExtentTest

**public** **void** extendReportSetup() {

ExtentHtmlReporter reporter = **new** ExtentHtmlReporter("./Reports/my\_report.html");

ExtentReports extent = **new** ExtentReports();

extent.attachReporter(reporter);

ExtentTest logger = extent.createTest("Login Test");

logger.info("Landed to Home Page");

logger.pass("Login Success");

extent.flush();

}

**Dependency –**

<dependency>

<groupId>com.aventstack</groupId>

<artifactId>extentreports</artifactId>

<version>3.1.5</version>

</dependency>

**Q.) How to achieve Parameterized testing in TestNG ?**

- using ‘testng.xml’ file and using @parameters annotation in test class definition.

**Q.) How to run test cases in parallel using TestNG ?**

- By using ‘parallel’ attribute in testng.xml file.

- The parallel attribute of suite tag accepts four values.

- tests 🡪 all test cases inside <test> of testing.xml will be executed in parallel

- classes 🡪 all the test cases inside java class will run in parallel

- methods 🡪 all methods with @Test annotation will run in parallel

<suite name=*"Practice Suite"* parallel = *'methods'* thread-count = *"4"*>

<test name=*"Test Basics 1"*>

<classes>

<class name=*"ertetr.MyClass1"*/>

</classes>

</test>

</suite>

* We can also provide ‘thread-count’ attribute in suite tag means that much number of @Test methods will start executing simultaneously (means that much amount of browsers are going to launch simultaneously).

Note – While running testng.xml if you are getting error as mentioned class name not found, then mention class name as “package\_name.class\_name” as shown in above example. It will work.

**Q.) How reports are generated in TestNG ?**

- After execution, in the ‘test-output’, index.html file gets generated.

- Apart from that ‘ExtentIReports’ we can generate reports.

**Q.) What is the difference between isDisplayed, isEnabled, isSelected ?**

- isDisplayed method will return True if element is seen on screen and will return false only if element is loaded in DOM but not visible on webpage. If user has provided some invalid element locator to isDisplayed method, then it will give NoSuchElement exception.

- isEnable is used to test options like Submit which becomes enable only after filling mandatory things.

- isSelected is used to test Checkbox, Dropdown and radio buttons.

**Interview Questions**

Q. What is the difference between final, finaly and finalize ?

a) final 🡪

- i) final keyword is used to define constant values means those values can not be

modified.

e.g. - final int a = 10;

ii) final keyword is used to prevent inheritance.

iii) final keyword is used to prevent method overriding.

b) finally 🡪

- It is associated with try catch block. Whatever code written in finally, will always get executed.

c) finalize 🡪

- This method is called before garbage collector is called (“System.gc()”). If user wants to perform certain closing operations before garbage collector is called then finalize method is called.

Q. Can we use final keyword with constructor ?

Apply a final keyword to a method means you cannot override that method. but by default in java no constructor can be overridden. so there is no need to use final keyword with a constructor . A constructor can't inherited so can't be overridden , so there is no use of making constructor final. If you provide, ‘final’ keyword with constructor then compile time error will occur.

Q. How to run failed test cases in Selenium ?

- In TestNG framework after execution if we refresh the project, we get new folder named ‘test-output’. This folder will have one subfolder with name of test suite that we have provided to TestNG suite. This folder contains ‘testng-failed.xml’. To rerun it, simply right click on it, select RunAs -> TestNG Suite.

Complete Video - <https://www.youtube.com/watch?v=kaUJgNyLaKs&t=337s>

Q. What is xpath ?

- Xpath is a syntax that is used to locate any element in a web application.

- It is XML PATH which helps to automate what we want.

Q. What is diff bet absolute (/) and relative (//) xpath ?

- Absolute xpath always starts from start node.

- In relative xpath, user can start xpath from node of his choice.

Q. What is the difference between findElement and findElements ?

* findElement will always return the first element matching our xpath.
* findElements returns an list of matching elements.
* findElement will return NoSuchElement exception if it does not find a match.
* findElements will return empty list when it does not find any match.

Q. How will you find total number of links in a webpage ?

List<WebElement> links = driver.findElements(By.tagName(“a”));

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

To print all links,

For (int i; i<links.size(); i++)

{

System.out.println(links.get(i).getText());

}

Q) What is the difference between == and .equals() ?

- ‘==’ is for reference / address comparison (i.e. whether both objects point to the same memory location or not) while ‘.equals()’ is used for content comparison.

**public** **class** MyClass

{

**public** **static** **void** main(String[] args)

       {

              String S1 = **new** String ("VJ");

              String S2 = **new** String ("VJ");

              System.***out***.println("s1==s2 is " + S1==S2);      // returns false

              System.***out***.println("s1.equals(s2) is " + S1.equals(S2)); // returns true

s1 = s2;

System.***out***.println(s1==s2); // returns true

System.***out***.println(s1.equals(s2)); // // returns true

       }

}

Q) What is the difference between JDK, JRE and JVM ?

       - JDK is Java Developers Kit, it contains JRE + Development tool.

       - JRE is Java Runtime Execution, it contains JVM + Library classes. JRE needs to be installed on client machine.

       - JVM is Java Virtual Machine, it is used to run the program.

Q) What is Stale Element exception ?

- Stale means old, not fresh.

- Whenever Selenium interacts with any elements, it creates its unique ID internally.

- After this first interaction, if page gets refreshed or some AJAX component gets

refreshed automatically then DOM structure of page changes and ID also changes.

* Now if we interact with that element again, Selenium tries to find it with old ID and can’t find it so stale element exception occurs.
* To avoid this, we should find element at the time of its interaction and not earlier.

Q.) How do you navigate within the browser ?

- By using Navigation interface.

- Navigation interface has non static methods, to access these methods we need to create object of Navigation interface.

- But we can’t create object of Navigation interface, so there is method called navigate() from WebDriver interface which returns the instance of navigation interface.

E.g. driver.navigate().to("www.javatpoint.com")

driver.navigate().forward()

Q.) How many ways we can execute our script ?

- Our script (means Maven project) can be executed by many ways.

- testng.xml

- pom.xml

- cmd command prompt

- batch file

- Jenkins

Q.) How to execute a project using Maven pom.xml

- Two plugins are required to execute project through pom.xml file.

maven-compiler-plugin

maven-surefire-plugin

* To execute project, in pom.xml, right click -> RunAs -> Maven test

Q.) How we can execute Maven project using command prompt ?

- Download Maven and extract zip file.

- Set Maven path in environment variable.

- Open command prompt

- cd “Enter maven project path that is created in Eclipse”

- mvn clean install

@.) Challenges and difficulties faced on Selenium ?

1. After implementing getScreenshot method in ITestListener, if more than one test cases are getting failed from suite then earlier screenshot was getting overrite. To overcome this problem, we passed method name as an argument to failed screenshot method which is called in default onTestFailure method in ITestListner implemented class,

**public** **void** onTestFailure(ITestResult result) {

failed(result.getMethod().getMethodName());

}

Note - For complete working code, "C:\Users\VJ\ITestNGListener"

1. Running some test cases on one URL and some test cases on different URL in the same scenario?

For this we need to use @Parameters annotation from TestNG framework. In testing.xml, We will pass the parameter ‘url1’ and its value from one test (that test will include the classes and methods that should be run on url1). We will capture this url1 in one of the launch browser initialization method using @Parameters attribute.

For running second url2, we will define it in second test in testng.xml and same above process will be repeated.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name=*"Practice Suite"*>

<test name=*"Test Basics 1"*>

<parameter name=*"url1"* value=*"https://www.google.com"*/>

<classes>

<class name=*"Different\_URL\_Testing.TestScenario"*>

<methods>

<include name = *"m1"*/>

<include name = *"m2"*/>

</methods>

</class>

</classes>

</test>

<test name=*"Test Basics 2"*>

<parameter name=*"url2"* value=*"https://www.facebook.com"*/>

<classes>

<class name=*"Different\_URL\_Testing.TestScenario"*>

<methods>

<include name = *"m3"*/>

<include name = *"m4"*/>

</methods>

</class>

</classes>

</test>

</suite>

**OOPs concepts in Selenium Framework 🡪**

**Abstraction** – Abstract page class is created which is super parent of all the page classes. It will have prototype methods including getInstance method which will return the object of page class that we are passing to it.

**Inheritance** – BasePage class will extend the above abstract page class. Every other page class will extend the BasePage Class.

**Method Overloading** – LoginPage method with same name. Create a method which accepts two arguments (user and password) and other which do not accept any argument (say authentication is not required).

**Method Overriding** – In BasePage class methods declared in abstract class will be defined using @Override operator.

**Encapsulations** – Page objects (locators) will be written with private keyword and public methods needs to be defined to access them (Getters).

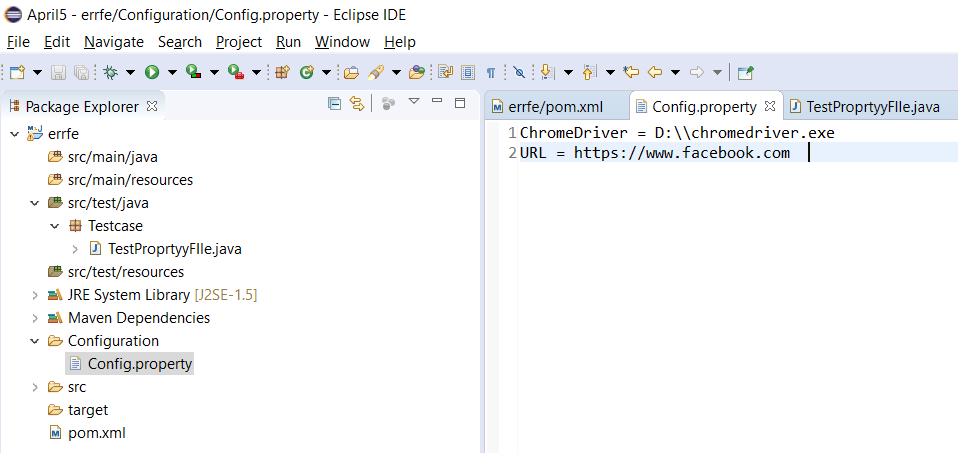
**Generics** –

**Automation Framework**

<https://www.youtube.com/watch?v=sPqf4ujbc0g&t=473s>

**config.property file**

* In practical application, some values like testing URL (might change as per release), chrome driver path will change as per time. So it is not advisable to change it in each and every time. Instead, we change it at one unique location (config.property file) and fetch it in test scripts.
* It is plain text file with key value pair.
* It contains data (hard coded) like chrome driver exe file path, URL.
* While creating this file, make sure that its extension is .property
* We generally create this config.property file in Configuration folder created under base Maven project.



@Test

**public** **void** testConfig() **throws** Exception {

File src = **new** File("./Configuration/Config.property");

FileInputStream fis = **new** FileInputStream(src);// for reading the file

Properties pro = **new** Properties(); // create object of property file

pro.load(fis);

String url = pro.getProperty("URL");

System.***out***.println("URL used for testing is " + url);

}

* In industry, this above code of reading values from config.proprty file is written in some other class under Utilities package.
* In test class, we create object of above utility class and just call that method to fetch the value from property file.

**Base Class**

* Base class consist of reusable methods like Browser setup (Initializing the WebDriver), loading configuration from property file (config.property) etc.
* Main purpose of base class is to avoid code duplication.
* Generally we write methods like @BeforeTest (for browser setup / launch), @AfterTest (quite browser).
* In order to use this, we just need to inherit above Base class in testing class (using ‘extend’ keyword).
* So consider a scenario where you have many @Test cases in one class file OR number of class files to be tested (using testng.xml) in that case this approach becomes very important.

**public** **class** Base {

**public** **static** WebDriver *driver*;

**public** **static** **void** initialization() {

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

*driver* = **new** ChromeDriver();

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

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

}

}

**public** **class** ScreenshotTest **extends** Base{

@BeforeMethod

**public** **void** setup() {

*initialization*();

}

@AfterMethod

**public** **void** teardown() {

*driver*.quit();

}

@Test

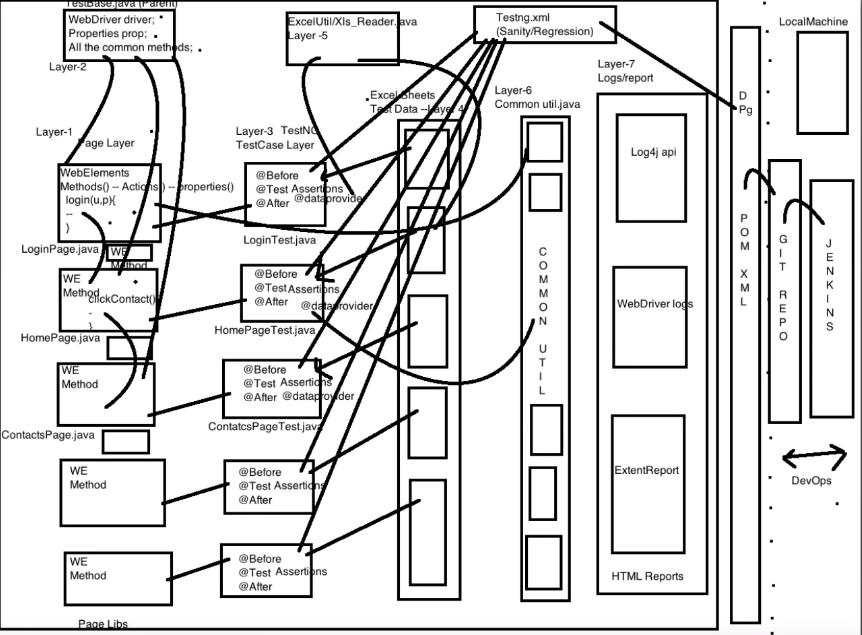
**public** **void** LoginTest() {

Assert.*assertEquals*(**true**, **false**);

}

}

Utilities Package 🡪 Taking screenshot, fetching data from excel sheet



Versions I use 🡪

* Java – 11 (Introduced in 2018)
* Eclipse – 4.13 (Introduced in 2019)
* Selenium – 3.142 (Introduced in 2019)

**Improving the Selenium script performance 🡪**

* Run the scripts in parallel
* Use direct selectors
* Use only explicit waits
* Create atomic tests
* Do not load any image in the webpage
* Use Headless browser

Running Script in Headless Browser 🡪

* Browser does not have Head means user can’t see (UI) but it runs internally
* To be used only with light weight applications and not having more complications.
* HTMLUnit is fastest and most light weight implementation of WebDriver.

WebDriver driver = new HtmlUnitDriver();

driver.get("http://www.facebook.com");

String fbTitle = driver.getTitle();

Assert.assertTrue(fbTitle.contains("log in"));

Note -: For HtmlUnitDriver, following maven dependency need to be added.

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-htmlunit-driver</artifactId>

<version>2.52.0</version>

</dependency>