Common test automation interview questions & topics one can expect for 5+ years experience

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Java Basic

# 1. Types of polymorphism

## Overview

Polymorphism is one of the main aspects of Object-Oriented Programming(OOP). The word polymorphism can be broken down into Poly and morphs, as “Poly” means many and “Morphs” means forms. In simple words, we can say that ability of a message to be represented in many forms.

## Introduction

Let us understand the definition of polymorphism by an example; a lady can have different characteristics simultaneously. She can be a mother, a daughter, or a wife, so the same lady possesses different behavior in different situations.

Another example of polymorphism can be seen in carbon, as carbon can exist in many forms, i.e., diamond, graphite, coal, etc. We can say that both woman and carbon show different characteristics at the same time according to the situation. This is called polymorphism.

## How can polymorphism be achieved in Java?

Polymorphism in Java can be achieved in two ways i.e., method overloading and method overriding.

Polymorphism in Java is mainly divided into two types.

* Compile-time polymorphism
* Runtime polymorphism

Compile-time polymorphism can be achieved by method overloading, and Runtime polymorphism can be achieved by method overriding. In the further article, we will be discussing all the topics related to polymorphism in Java in more detail.

## Polymorphism in Java Example

The below code is the basic example of polymorphism in Java.

*// Parent class to illustrate run-time polymorphism*

classParent{

*// creating print method*

void print() {

System.out.println("Hi I am parent");

}

}

*// Child class extends Parent class*

classChildextendsParent{

*// overriding print method*

void print() {

System.out.println("Hi I am children");

}

}

*// Overload class to illustrate compile-time polymorphism*

classOverload{

*// Creating a statement method*

void statement(String name) {

System.out.println("Hi myself " + name);

}

*// overloading statement method*

void statement(String fname, String lname) {

System.out.println("Hi myself " + fname + " " + lname);

}

}

publicclassMain{

publicstaticvoid main(String[] args) {

*// creating instance of parent*

Parent obj1;

obj1 = new Parent();

obj1.print();

obj1 = new Child();

obj1.print();

*// creating instance of overload*

Overload obj2 = new Overload();

obj2.statement("Soham.");

obj2.statement("Soham", "Medewar.");

}

}

**Output**

Hi I am a parent

Hi I am children

Hi myself Soham.

Hi myself Soham Medewar.

**Explanation**

In the above example, the Child class extends the Parent class, and the print method is overridden, this represents run-time polymorphism in Java.

In the overload class, the statement function is overloaded, this represents compile-time

## Example of polymorphism in real-life

We can relate polymorphism in real life by the following example. Consider a parent class as living thing.

Living things exist on the planet in the forms of human beings, animals, plants, bacteria, etc. These are the child classes inherited from the parent class, i.e., living things.

The code below will show you the practical implementation of the real-life example.

*// parent class Livingthings*

classLivingthings{

*// creating live method*

publicvoid live() {

System.out.print("live on ");

}

}

*// Animals class extends Livingthings*

classAnimalsextendsLivingthings{

*// overriding live method*

publicvoid live() {

System.out.println("water, air, and land.");

}

}

*// Humanbeing class extends Livingthings*

classHumanbeingextendsLivingthings{

*// overriding live method*

publicvoid live() {

System.out.println("land.");

}

}

*// Plants class extends Livingthings*

classPlantsextendsLivingthings{

*// overriding live method*

publicvoid live() {

System.out.println("water and land.");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating new object of Livingthings class*

Livingthings LT = new Livingthings();

Livingthings LT1;

LT1 = new Animals();

System.out.print("Animals ");

LT.live();

LT1.live();

LT1 = new Humanbeing();

System.out.print("Human beings ");

LT.live();

LT1.live();

LT1 = new Plants();

System.out.print("Plants ");

LT.live();

LT1.live();

}

}

**Output**

Animals live on the water, air, and land.

Human beings live on land.

Plants live on water and land.

**Explanation**

In the above example, the parent class Livingthings has the method live(). Subclasses of Livingthings i.e., Animals, Humanbeing, and Plants have their way of living. By the principle of inheritance and polymorphism, each subclass can define its way of living with the help of the live() method.

## Types of Polymorphism in Java

There are two main types of polymorphism in Java.

1. Compile-time polymorphism

This type of polymorphism in Java is also called static polymorphism or static method dispatch. It can be achieved by method overloading. In this process, an overloaded method is resolved at compile time rather than resolving at runtime.

### Method overloading

Consider a class where multiple methods have the same name. It will be difficult for the compiler to distinguish between every method.

To overcome this problem, we pass a different number of arguments to the method or different types of arguments to the method. In this way, we achieve method overloading.

In other words, a class can have multiple methods of the same name, and each method can be differentiated either by bypassing different types of parameters or bypassing a different number of parameters.

**Example 1**

Passing different numbers of arguments to the function.

classCompileTime{

*// perimeter method with a single argument*

staticint perimeter(int a) {

return4 \* a;

}

*// perimeter method with two arguments (overloading)*

staticint perimeter(int l, int b) {

return2 \* (l + b);

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// calling perimeter method by passing a single argument*

System.out.println("Side of square : 4\nPerimeter of square will be : " + Compiletime.perimeter(4) + "\n");

*// calling perimeter method by passing two arguments*

System.out.println("Sides of rectangle are : 10, 13\nPerimeter of rectangle will be : " + Compiletime.perimeter(10, 13));

}

}

**Output**

Side of square : 4

Perimeter of square will be : 16

Sides of rectangle are : 10, 13

Perimeter of rectangle will be : 46

**Explanation**

In the above example, the CompileTime class has two functions, both having the same name, but the first function has a single argument to pass, and another one has two arguments to pass.

Here, the perimeter() function can calculate the perimeter of square and rectangle. In this way, two functions having the same name are distinguished, and compile-time polymorphism is achieved.

For the first time, when we call the perimeter method, we pass a single integer to the method, so the first method is evoked, and for the second time, we pass two integers to the method, and this time the second method is evoked.

**Example 2**

Passing different types of argument to the function.

classCompiletime{

*// contact method, which takes two arguments String and long*

staticvoid contact(String fname, long number) {

System.out.println("Name : "+fname+"\nNumber : "+number);

}

*// contact method, which takes two arguments and both are Strings (overloading)*

staticvoid contact(String fname, String mailid) {

System.out.println("Name : "+fname+"\nEmail : "+mailid);

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// calling first contact method*

Compiletime.contact("Soham", 1234567890);

System.out.print("\n");

*// calling second contact method*

Compiletime.contact("Soham", "soham@mail.com");

}

}

**Output**

Name : Soham

Number : 1234567890

Name : Soham

Email : soham@mail.com

**Explanation**

In the above example, the CompileTime class has two functions, both having the same name, but in the first function, we pass a string and long as an argument, and in the second function, we pass two strings as an argument.

It shows that we can save the person's contact by mobile number or email. In this way, compile-time polymorphism is achieved.

For the first time, when we call the contact() method, we pass a name and mobile number (string and long) as arguments, so the first contact method is evoked.

The second time while calling the contact method, we pass a name and email (string and string) as arguments, so the second contact method is evoked.

### 2. Runtime Polymorphism

Runtime polymorphism is also called Dynamic method dispatch. Instead of resolving the overridden method at compile-time, it is resolved at runtime.

Here, an overridden child class method is called through its parent's reference. Then the method is evoked according to the type of object. In runtime, JVM figures out the object type and the method belonging to that object.

Runtime polymorphism in Java occurs when we have two or more classes, and all are interrelated through inheritance. To achieve runtime polymorphism, we must build an "IS-A" relationship between classes and override a method.

### Method overriding

If a child class has a method as its parent class, it is called method overriding.

If the derived class has a specific implementation of the method that has been declared in its parent class is known as method overriding.

#### Rules for overriding a method in Java

1. There must be inheritance between classes.
2. The method between the classes must be the same(name of the class, number, and type of arguments must be the same).

**Example 1: Animal class**

*//parent class Animal*

classAnimal{

*// creating place method*

void place(){

System.out.println("Animals live on earth.");

}

}

*// Dog class extends Animal class*

classDogextendsAnimal{

*// overriding place method*

void place(){

System.out.println("Dog lives in kennel.");

}

}

*// Horse class extends Animal class*

classHorseextendsAnimal{

*// overriding place method*

void place(){

System.out.println("Horse lives in stable.");

}

}

*// Rabbit class extends Animal class*

classRabbitextendsAnimal{

*// overriding place method*

void place(){

System.out.println("Rabbit lives in burrow.");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating object of Animal class*

Animal A = new Animal();

A.place();

A = new Dog();

A.place();

A = new Horse();

A.place();

A = new Rabbit();

A.place();

}

}

**Output**

Animals live on earth.

Dog lives in kennel.

Horse lives in stable.

Rabbit lives in burrow.

**Explanation**

In the above example, the Animal class is parent class, and Dog, Horse, Rabbit are its derived class where the place() method is overridden.

We have created an instance of the Animal class. When an object of every child class is created, the method inside the child class is called. As, the parent class method is overridden by child class.

**Example 2: Shape Class**

*// parent class Shape*

classShape{

*// creating area method*

void area(){

System.out.println("Formula for areas.");

}

}

*// Square class extends Shape class*

classSquareextendsShape{

*// overriding area method*

void area(){

System.out.println("Area of square : a \* a");

}

}

*// Rectangle class extends Shape class*

classRectangleextendsShape{

*// overriding area method*

void area(){

System.out.println("Area of rectangle : 2 \* (a + b)");

}

}

*// Circle class extends Shape class*

classCircleextendsShape{

*// overriding area method*

void area(){

System.out.println("Area of circle : pi \* r \* r");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating new object of Shape class*

Shape S = new Shape();

S.area();

S = new Square();

S.area();

S = new Rectangle();

S.area();

S = new Circle();

S.area();

}

}

**Output**

Formula for areas.

Area of square : a \* a

Area of rectangle : 2 \* (a + b)

Area of circle : pi \* r \* r

**Explanation**

In the above example, the Shape class is parent class and Square, Rectangle, and Circle are its derived class where the area() method is overridden.

**Example 3: Bank Class**

*// parent class Bank*

classBank{

*// creating rateOfInterest method*

float rateOfInterest(){

return0;

}

}

*// ICICI class extends Bank class*

classICICIextendsBank{

*// overriding rateOfInterest method*

float rateOfInterest(){

return5.5f;

}

}

*// SBI class extends Bank class*

classSBIextendsBank{

*// overriding rateOfInterest method*

float rateOfInterest(){

return10.6f;

}

}

*// HDFC class extends Bank class*

classHDFCextendsBank{

*// overriding rateOfInterest method*

float rateOfInterest(){

return9.4f;

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating variable of Bank class*

Bank B;

B = new ICICI();

System.out.println("Rate of interest of ICICI is: "+B.rateOfInterest());

B = new SBI();

System.out.println("Rate of interest of SBI is: "+B.rateOfInterest());

B = new HDFC();

System.out.println("Rate of interest of HDFC is: "+B.rateOfInterest());

}

}

**Output**

Rate of interest of ICICI is: 5.5

Rate of interest of SBI is: 10.6

Rate of interest of HDFC is: 9.4

**Explanation**

In the above example, the Bank class is parent class and ICICI, SBI, and HDFC are its derived class where the rate() method is overridden.

**Example 4: Using Data Members**

*// Parent class*

classParent{

int value = 50;

}

*// Child class extends Parent class*

classChildextendsParent{

int value = 100;

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating new object of Parent class*

Parent obj1 = new Child();

System.out.println("Child value : "+obj1.value);

}

}

**Output**

Child value : 50

**Explanation**

In the above example, we can see that value in the child class is not overrode, as the "value" remains 50 after overriding. The above program concludes that data members are not overridden, so data members can't achieve polymorphism.

**Example 5: Multilevel Inheritance**

*// parent class Animal*

classAnimal{

*// creating move method*

void move(){

System.out.println("Animals can move.");

}

}

*// Tiger class extends Animal class*

classTigerextendsAnimal{

*// overriding move method*

void move(){

System.out.println("Tiger can walk as well as run.");

}

}

*// Cub class extends Tiger class*

classCubextendsTiger{

*// overriding move method*

void move(){

System.out.println("Cub can walk.");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// creating new object of Animal class*

Animal A = new Animal();

A.move();

A = new Tiger();

A.move();

A = new Cub();

A.move();

}

}

**Output**

Animals can move.

Tiger can walk as well as run.

Cub can walk.

**Explanation**

In the above example, the Animal class extends the Tiger class and the Tiger class extends the Cub class. move() function is overridden in the Tiger class and the Cub class.

## Characteristics of Polymorphism

Besides method overloading and method overriding, polymorphism has other characteristics as follows.

1. Coercion
2. Internal Operator Overloading
3. Polymorphic Variables or Parameters
4. Subtype polymorphism

### 1. Coercion

Let us understand this with an example, consider a variable whose data type is an integer and another variable whose data type is double. If we add these two numbers, we will get a type error.

Java has inbuilt functionality called coercion to avoid such errors, where a smaller data type is automatically typecasted into a more significant data type according to need. In this case, the integer value will be typecasted to double value, and then addition takes place. Hence type error is avoided.

The implicit conversion of one data type into another without changing its context is known as coercion. This type of conversion occurs to prevent type errors.

In other words, if the datum is present in one data type, but its context requires a different data type, then Coercion occurs.

**Example**

classPolymorphism{

publicstaticvoid main(String[] args) {

int num = 165;

String str = "Hello";

*// concatenating str and num*

String ans = str+num;

System.out.println(ans);

}

}

**Output**

Hello165

**Explanation**

In the above program, we can see that the int value “num” is added to the String “and”, then the compiler implicitly converts the int value into a string value to avoid type error.

2. Internal Operator Overloading

As explained in the article that Java does not support operator overloading. Still, there is a concept called internal operator overloading in Java, where an operator is used in more than one way. The characteristic of static polymorphism is observed here.

In Java, the ‘+’ symbol is used to add two numbers or used to concatenate two strings.

**Example**

classPolymorphism{

publicstaticvoid main(String[] args) {

*// adding numbers*

int num1 = 741, num2 = 852;

String str1 = "Hello", str2 = "World!";

*// concatenating two strings*

int sum = num1+num2;

String final\_str = str1 + str2;

System.out.println("Sum = "+sum);

System.out.println("Final String = "+final\_str);

}

}

**Output**

Sum = 1593

Final String = HelloWorld!

**Explanation**

In the above example, the ‘+’ operator is performing both addition and concatenation tasks.

3. Polymorphic Variables or Parameters

As I explained, the example of a lady is in the definition of polymorphism. If a lady has a child, she is called a mother; if she has a sibling, then she is called a sister; if she has a husband, she is called a wife. So here, a lady has different values under different situations; hence lady can be called a polymorphic variable.

Variables having different values under different circumstances is called polymorphic variable. It can be said that every object or instance variable in Java is a polymorphic variable because every object or instance variable has an IS-A relationship with its own classes and sub-classes.

Variables holding different values at the execution time are known as polymorphic variables.

A field name can be associated with several types, and a method name can be associated with different parameters and return types according to parametric polymorphism.

**Example: Polymorphic variable**

*// parent class India*

classIndia{

*// creating info method*

void info(){

System.out.println("I am India.");

}

}

*// Maharashtra class extends India class*

classMaharashtraextendsIndia{

*// overriding info method*

void info(){

System.out.println("I am Maharashtra, state of India.");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// declaring a variable of India class*

India ob;

ob = new India();

ob.info();

ob = new Maharashtra();

ob.info();

}

}

**Output**

I am India.

I am Maharashtra, state of India.

**Explanation**

Here, the ob object is a polymorphic variable because the same object refers to parent class (India) and child class (Maharashtra).

**Example: Polymorphic parameter**

classParent{

String display(int num){

String temp = ""+num;

return temp;

}

}

classChildextendsParent{

String data = "Hello World ";

String display(int num){

int data = num;

this.data = this.data+data;

returnthis.data;

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// declaring a variable of Parent class*

Parent obj;

obj = new Child();

System.out.println(obj.display(404));

}

}

**Output**

Hello World 404

**Explanation**

The above code is an example of parametric polymorphism. In the “Child” class, we define data as a string and later as an integer.

### 4. Subtype polymorphism

Let us consider an example, suppose a zoo has four distinct tigers, three distinct lions, and two distinct elephants. We are supposed to store the data of all the animals in a list. We can store the data by using the property of subtype polymorphism. Let us assume tiger, lion, and elephant are derived classes of parent class animal.

Traditionally we create an object of tiger class and store the information. Similarly, we do this for all the animals. But in subtype polymorphism, we make an array of animal class, and then we upcast objects of every tiger, lion, and elephant to animal class. Store the upcasted objects in an array of animal class.

The ability to use the subclass instead of the superclass is called subtype polymorphism. In other words, subtype polymorphism is about upcasting and late binding. Upcasting can be termed as typecasting a child object to a parent object, and late binding is simply dynamic binding or overriding.

**Read the following examplke for better under standing.**

**Example**

*// parent class Shape*

classShape{

*// creating area method*

void area(){

System.out.println("Area of various shapes are calculated here.");

}

}

*// Square class extends Shape class*

classSquareextendsShape{

int a;

*// parametric constructor*

Square(int side){

this.a = side;

}

void area(){

System.out.println("Side of square is : "+a);

System.out.println("Area of square is : "+(a\*a)+"\n");

}

}

*// Circle class extends Shape class*

classCircleextendsShape{

int r;

*// parametric constructor*

Circle(int a){

this.r = a;

}

void area(){

System.out.println("Radius of circle is : "+r);

System.out.println("Area of circle is : "+(3.14\*r\*r)+"\n");

}

}

*// Rectangle class extends Shape class*

classRectangleextendsShape{

int l, b;

*// parametric constructor*

Rectangle(int w, int h){

this.l = w;

this.b = h;

}

void area(){

System.out.println("Sides of rectangle are : "+l+", "+b);

System.out.println("Area of rectangle is : "+(2\*(l + b))+"\n");

}

}

classPolymorphism{

publicstaticvoid main(String[] args) {

*// array of Shape class*

Shape [] arr = {

new Square(10), new Circle(15), new Rectangle(10, 15)

};

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

arr[i].area();

}

}

}

**Output**

Side of square is : 10

Area of square is : 100

Radius of circle is : 15

Area of circle is : 706.5

Sides of rectangle are : 10, 15

Area of rectangle is : 50

**Explanation**

In the above example, the declaration of the Shape array illustrates upcasting. The Square, Circle, and Rectangle references are stored in Shape[0], Shape[1], and Shape[2] are upcast to type Shape. Each of Shape[0], Shape[1], and Shape[2] is regarded as a Shape.

Late binding is explained by the Shape[i].area(); method. When i equals 0, the compiler-generated instruction causes Square's area() method to be called. When i equals 1, Circle's area() method is called, and for i=2, Rectangle's area() method is called. This is a nature of subtype polymorphism.

## Advantages of Polymorphism

* Code reusability is the main advantage of polymorphism; once a class is defined, it can be used multiple times to create an object.
* In compile-time polymorphism, the readability of code increases, as nearly similar functions can have the same name, so it becomes easy to understand the functions.
* The same method can be created in the child class as in the parent class in runtime polymorphism.
* Easy to debug the code. You might have intermediate results stored in arbitrary memory locations while executing code, which might get misused by other parts of the program. Polymorphism adds necessary structure and regularity to computation, so it is easier to debug.

## Problems with Polymorphism

* Implementing code is complex because understanding the hierarchy of classes and its overridden method is quite difficult.
* Problems during downcasting because implicitly downcasting is not possible. Casting to a child type or casting a common type to an individual type is known as downcasting.
* Sometimes, when the parent class design is not built correctly, subclasses of a superclass use superclass in unexpected ways. This leads to broken code.
* Runtime polymorphism can lead to the real-time performance issue (during the process), it basically degrades the performances as decisions are taken at run time because the machine needs to decide which method or variable to invoke.

## Conclusion

* Polymorphism is one of the four parts of object-oriented programming.
* With proper implementation of polymorphism, one can achieve flexible and extensible class designs.
* There are two main types of polymorphism i.e. runtime polymorphism and compile-time polymorphism.
* Runtime polymorphism is achieved through method overriding, and compile-time polymorphism is achieved through method overloading.
* Java doesn't support operator overloading but does internal operator overloading.
* Other than the runtime and compile-time, java supports coercion polymorphism and parametric polymorphism.
* Using the concept of method overriding and inheritance enables us to reuse code without re-compilation.
* Closely related functions can be accessed through the common name using the method overloading technique.
* Internal operator overloading is achieved using coercion polymorphism.
* Method overriding provides a feature where a subclass can use all the general definitions provided by the superclass and add specialized definitions through overridden methods.

## FAQs

**Q.** What are 2 Types of Polymorphism?

**A.** Following are the two types of polymorphism

1. Compile-time polymorphism
2. Runtime polymorphism

**Q.** What are Polymorphism variables?

**A.** During the execution of a program, a polymorphic variable is defined as a variable that can hold values of several types. In other words, variables having different values under different conditions or variables holding different values at the execution time are called Polymorphism variables.

**Q.** What is polymorphism in programming?

**A.** Polymorphism is an ability of an object, variable, and function to take on multiple forms. The ability to apply a single method to derived classes and get a suitable result is called polymorphism.

**Q.** Why use Polymorphism in Java?

**A.** As we know java supports inheritance where properties of one class are derived into another class. Polymorphism uses the properties of inheritance to perform many tasks that's why we should use polymorphism in Java.

**Q.** What is Static Polymorphism?

**A.** Processes in static polymorphism are exhibited at compile time. The compiler knows which method has been called. Method overloading is an example of static polymorphism. Static polymorphism is also known as compile-time polymorphism. It can be achieved through method overloading.

**Q.** What is Dynamic Polymorphism?

**A.** In dynamic polymorphism instead of resolving the overridden method at compile-time, it is resolved at runtime. The compiler doesn't know which method has been called at compile-time. JVM decides which method is called at runtime. Dynamic polymorphism is also called as runtime polymorphism. It can be achieved through method overriding.

**Q.** What is the difference between Static & Dynamic Polymorphism?

**A.**

| **Static Polymorphism** | **Dynamic Polymorphism** |
| --- | --- |
| It is achieved through method overloading. | It is achieved through method overriding. |
| It is called compile-time polymorphism. | It is called runtime polymorphism. |
| Known as static binding or early binding. | Known as dynamic binding or late binding. |
| It has a high execution speed. | It has a low execution speed. |

# Difference between override and overload

# Methods that cannot be overloaded

# Static block and instance block

# Difference in Static and non static

# Encapsulation and common use cases

# Abstract vs interface

# Inheritance in Java

# Methods cannot be overloaded

# Type casting in Java

## Overview

Type casting in Java is the process of converting one data type to another. It can be done automatically or manually. Automatic type casting occurs when a value of one data type is assigned to another compatible data type.

There are two types of casting: widening type casting and narrowing type casting.

## Introduction to Type Casting in Java

First of all, let’s understand the literal meaning of **Type Casting** with context to programming. **Type** means the data type of a variable or entity and **Casting** means converting the data type of an entity to another data type.

### Example

Let us see a quick example where we will convert a string to an integer using Integer.parseInt() method in Java.

**Code:**

publicclassStringtoInt{

publicstaticvoid main(String args[]) {

*// Declaring String variable*

String s = "1000";

*// Convert to int using Integer.parseInt()*

int i = Integer.parseInt(s);

*// Printing value of i*

System.out.println(i);

}

}

**Output:**

1000

**Explanation:**

* We have created a String variable s in the code above.
* Next, we have converted the string variable to an integer using Integer.parseInt() method.

## What is Type Casting/Conversion in Java

Type casting, also known as type conversion, is the process of changing the data type of a variable in Java. It allows you to convert a variable from one type to another, either widening or narrowing the range of possible values.

Type casting is useful when you need to perform operations on variables of different types or when you want to assign a value of one type to a variable of another type. It can be seen in the above example, when we assigned the value of a String variable to an Integer.

## Primitive Data types

There are eight types of primitive data types in Java as given below:

* byte
* short
* int
* long
* float
* double
* char
* boolean

**Primitive data types** in Java are like the different kinds of Lego blocks you can use to build something. Type casting is like when you need to change one Lego block into another to make it fit or work together with a different block.

## Types of Casting in Java

Type Casting in Java is mainly of two types.

1. Widening Type Casting
2. Narrow Type Casting

### Widening Type Casting in Java

Widening type casting refers to the conversion of a lower data type into a higher one. It is also known as implicit type casting or casting down. It happens on its own. It is secure since there is no possibility of data loss.

Widening Type Casting happens on the following scenarios or conditions:

* **Both the data types must be compatible with each other.** For example, converting a string to an integer is not possible as the string may contain alphabets that cannot be converted to digits.
* The target variable holding the type casted value must be larger than the value being type casted.

In the above image, double is larger data type than float, long, int, etc. Similarly, int is larger data type than short and byte. When converting from a lower data type to a larger data type, no loss of data occurs as the range of supported values is wider in the larger data type.

### Example

publicclassWideningTypeCastingExample{

publicstaticvoid main(String[] args) {

int i = 10;

*// Automatic Casting from int to long*

long l = i;

*// Automatic Casting from int to double*

double d = i;

System.out.println("int i = " + i);

System.out.println("long l = " + l);

System.out.println("double d = " + d);

}

}

### Output:

int i = 10

long l = 10

double d = 10.0

## Narrow Type Casting in Java (on default data types)

Narrowing Type Casting in Java refers to the conversion of a larger data type into a lower one. It is also known as explicit type casting or casting up. It does not happen on its own.

We must do it explicitly otherwise compile-time error is thrown.

**Narrowing Type Casting in Java is not secure** as loss of data can occur due to shorter range of supported values in lower data type.

Explicit Casting is done with the help of cast operator.

### Syntax

*// var is of lowerDataType*

var = (lowerDataType) expr;

In the above image, int is a lower data type as compared to double, float, long, etc. When we convert a long to an int, it is possible that the value of the long variable is out of range of int variable.

### Example 1:

publicclassNarrowingTypeCastingExample{

publicstaticvoid main(String[] args) {

double a = 100.245;

*// Narrowing Type Casting*

short b = (short) a;

int c = (int) a;

System.out.println("Before Casting Original Value " + a);

System.out.println("After Casting to short " + b);

System.out.println("After casting to int " + c);

}

}

### Output:

Before Casting Original Value 100.245

After Casting to short 100

After casting to int 100

**Explanation:**

* As it is visible in the code, there is loss of decimal digits after conversion from double to either short or int.
* We perform explicit type casting using the cast () operator and mentioning the name of target data type inside the brackets.

### Lossy NarrowingType Casting in Java

**Example of data loss:**

long l = 2147483648L;

*//*

*// T*

*// And we are trying to store a value in i*

*// greater than its upper limit*

int i = (int) l;

publicclassLossyConversion{

publicstaticvoid main(String[] args) {

long l = 2147483648L;

int i = (int) l;

System.out.println(i);

}

}

**Output:**

-2147483648

**Explanation:**

* As you can see the output, i contains **-2147483648**. This is because the range of int is -2147483648 to 2147483647.
* And we are trying to store a value in i greater than its upper limit resulting in overflow situation.

## Type Conversion in Java

Type conversion in Java refers to the process of converting one data type to another. It allows you to change the representation or format of a value to make it compatible with another data type.

There are two types of type conversion in Java:

### Explicit Type Conversion

Explicit type conversion in Java, also known as type casting, is used when you want to convert a value of one data type to another data type that cannot be automatically done by the compiler.

**Example 1: Converting an integer to a double**

int num = 5;

double convertedNum = (double) num;

**Example 2: Converting a double to an integer**

double num = 3.14;

int convertedNum = (int) num;

### Implicit Type Conversion

Implicit type conversion in Java, also known as automatic type promotion, occurs when the compiler automatically converts a value of one data type to another data type without requiring explicit instructions from the programmer.

**Example 1: Converting an integer to a double**

int num = 5;

double convertedNum = num;

**Example 2: Combining an integer with a long**

int num1 = 10;

long num2 = 20;

long sum = num1 + num2;

## When does Automatic type conversion in Java?

Automatic type conversion occurs in Java when there is a need to convert a value from a smaller or narrower data type to a larger or wider data type.

### Scenarios:

* **Operations between different data types:** Java automatically promotes smaller types to larger types for compatibility.
* **Assignment to a larger type:** Smaller values can be assigned to larger types without explicit conversion.
* **Method parameter matching:** Java automatically converts compatible argument types to match expected parameter types.

## Advantages and Disadvantage of Explicit Type casting in Java

### Advantages

* **Precision Control:** Explicit type casting allows precise control over data precision and range, ensuring accurate calculations or data representation.
* **Data Compatibility:** Explicit type casting enables seamless integration between different types, facilitating operations and assignments that require matching data types.

### Disadvantages

* **Data Loss:** Explicit type casting can result in data loss or truncation when narrowing values from a larger type to a smaller type, potentially affecting accuracy and introducing inaccuracies.
* **Potential Errors:** Incorrect usage of explicit type casting can lead to runtime errors. Incompatible casting or exceeding the target type's range may cause unexpected behavior or exceptions, such as ClassCastException, resulting in program instability or crashes.

## Difference Between Type Casting and Type Conversion in Java

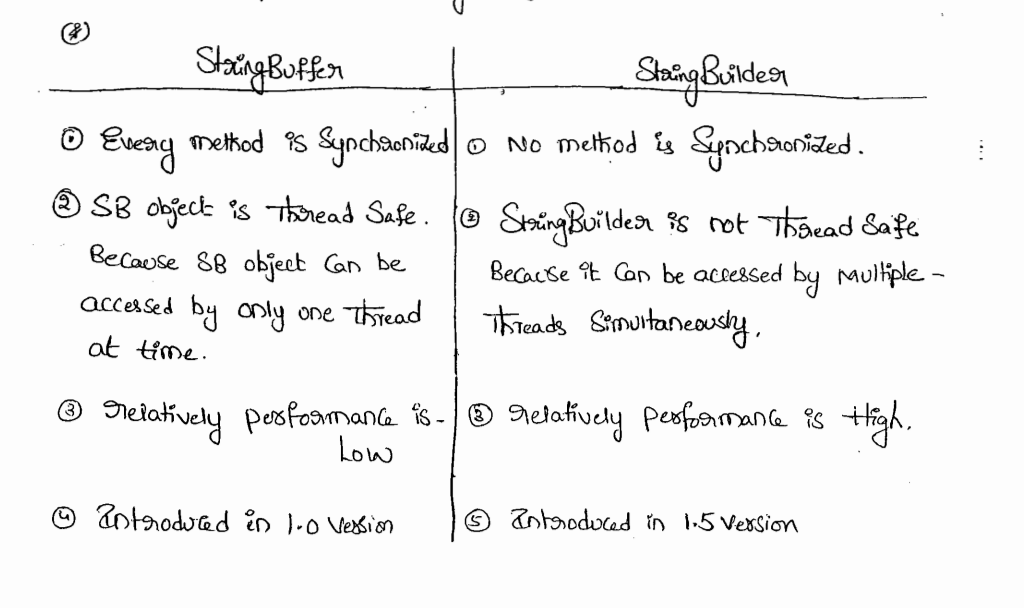
| **Type Casting** | **Type Conversion** |
| --- | --- |
| Involves changing the data type of a variable. | Involves changing the representation or format of a value. |
| Performs conversion between compatible data types. | Can involve conversion between incompatible data types as well. |
| Requires explicit instructions using casting syntax. | Can be done implicitly by the Java compiler. |
| Can result in loss of data or precision if narrowing conversion is performed. | Can involve manipulation or transformation of data without necessarily changing the data type. |
| Example: (double) num converts num from an integer to a double explicitly. | Example: Integer.toString(num) converts num to a string without changing its data type. |

## Conclusion

* Type Casting in Java refers to the process of converting one data type to another data type.
* The process of conversion of higher data type to lower data type is known as narrowing typecasting. It is also known as Explicit TypeCasting as it must be done explicitly.
* The process of conversion of lower data type to higher data type is known as widening typecasting. It is also called Implicit TypeCasting as it can be done implicitly by the compiler.

# String buffer and String builder

|  |  |
| --- | --- |
| **String Buffer** | **String Builder** |
| Every method is synchronized | No method is synchronized |
| SB object is tread safe, because SB object can be accessed by only one thread at one time. | SB object is not tread safe, because SB object can be accessed by multiple thread simultaneously. |
| Relatively performance is low | Relatively performance is high |
| Introduced in 1.0 version | Introduced in 1.5 version |



# Why String is immutable in java

* In the Case of string several reference can be pointing to the same object. By using one reference, if we are performing any change in the existing object the ramming references will be impacted to resolve this problem SUN people declared as string objects are immutable. According to this once we created a string object we can’t perform any changes in the existing object.
* If we are trying to perform any changes, with those a new object is created i.e. SCP is the only reason why the string objects are immutable.

# How to handle exceptions?

Ans. Handling Java exceptions happens with the try, catch, and finally blocks. They can be used to define how to handle exceptions when they occur.  
 Try blocks should include code that might throw an exception. If your code throws more than one exception, you can choose whether to:  
 Use a separate try block for every statement that may throw an exception or  
 Use one try block for several statements that could throw multiple exceptions.  
 Each Catch block should handle exceptions thrown by the try blocks. The finally block gets executed whether or not an exception is thrown after the successful execution of the try block or after one of the catch blocks.  
 There is no limit to add catch blocks, as you can include as many Catch blocks as you want. However, you can add only one finally block to each try block.  
 After the try and catch blocks have been executed, programmers usually use finally blocks to do cleanup.

# Can I write try catch without the catch block?

Ans. Yes, It is possible to have a try block without a catch block by using a final block.

As we know, a final block will always execute even there is an exception occurred in a try block, except System.exit() it will execute always.

**Example 1**

public class TryBlockWithoutCatch {

   public static void main(String[] args) {

      try {

         System.out.println("Try Block");

      } finally {

         System.out.println("Finally Block");

      }

   }

}

**Output**

Try Block

Finally Block

# Difference between throws and throw?

| **S. No.** | **Key Difference** | **throw** | **throws** |
| --- | --- | --- | --- |
| 1. | Point of Usage | The throw keyword is used inside a function. It is used when it is required to throw an Exception logically. | The throws keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions. |
| 2. | Exceptions Thrown | The throw keyword is used to throw an exception explicitly. It can throw only one exception at a time. | The throws keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then. |
| 3. | Syntax | Syntax of throw keyword includes the instance of the Exception to be thrown. Syntax wise throw keyword is followed by the instance variable. | Syntax of throws keyword includes the class names of the Exceptions to be thrown. Syntax wise throws keyword is followed by exception class names. |
| 4. | Propagation of Exceptions | throw keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword. | throws keyword is used to propagate the checked Exceptions only. |

# Use of iterator in java

# Difference in Final, finally the catch block?

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **final** | **finally** | **finalize** |
| Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| Functionality | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| Execution | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the obj  ect is destroyed. |

# Boxing and un boxing in java

# Increment and decrement operation

# Variable Args

# This and super keyword in java

# Issues during Switch case without break

# Up casting and down casting

# Base class of all class in java

# Base class of error and exceptions

# Access Specifies

# Continue and break statement

# Can main method return any value

# Can we overload main method? What happens when overloaded

# How to execute and statement before main method

# Difference between == and equals()

# Can user declare constructor as final

# Can we cast any other type to Boolean data with type casting.

# Dose java compile if user use ‘static public void’ instead of ‘public static void ’

# Can we use this() and super() in a constructor

# Can we create object of abstract class

# Can we create reference for abstract class

# Can we declare a class as static

# What is instance of keyword?

# What’s the load factor of HashMap

# How to prevent a class from being sub classes

# Final variable, final method and final class

# Ways to create a string variable

# What is gc() – garbage collector

# Subclass and innerclass

# Infinite loop in java

# How to make copy of an object

# Checked and unchecked exceptions?

|  |  |
| --- | --- |
| checked Exception | Unchecked Exception |
| Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| The compiler checks a checked exception. | The compiler does not check these types of exceptions. |
| These types of exceptions can be handled at the time of compilation. | These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program. |
| They are the sub-class of the exception class. | They are runtime exceptions and hence are not a part of the Exception class. |
| Here, the JVM needs the exception to catch and handle. | Here, the JVM does not require the exception to catch and handle. |
| Examples of Checked exceptions:   * File Not Found Exception * No Such Field Exception * Interrupted Exception * No Such Method Exception * Class Not Found Exception | Examples of Unchecked Exceptions:   * No Such Element Exception * Undeclared Throwable Exception * Empty Stack Exception * Arithmetic Exception * Null Pointer Exception * Array Index Out of Bounds Exception * Security Exception |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Java Collection

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Classes inside List interface, Set interface, Map interface
2. ArrayList vs LinkedList
3. ArrayList vs Array
4. ArrayList vs vector or stack
5. Which class of List Interface to be used if user have more insertions and deletions
6. Which class of List interface to be used if user have more retrieval
7. Set Interface: HashSet,TreeSet, Sortedset
8. Map –HashMap, HashTable, LinkedHashMap
9. Stack and Queue
10. How to maintain insertion order in Set, List and Map
11. How to sort elements in ascending order in Set and Map

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Java Program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. String reverse

static string Reverse(string str)

{

string reverse\_str = "";

for(int i = str.Length - 1; i >= 0; i--)

{

reverse\_str = reverse\_str + str[i];

}

return reverse\_str;

}

1. String Palindrome

static string Reverse(string str)

{

string reverse\_str = "";

for(int i = str.Length - 1; i >= 0; i--)

{

reverse\_str = reverse\_str + str[i];

}

return reverse\_str;

}

static string CheckPalindromString(string str1, string str2)

{

if (str1.Equals(str2))

{

return str1 + " : is a Palindrome String";

}

else

return str1 + " : is not a Palindrome String";

}

1. String Anagram

public static bool areAnagram(string str1, string str2)

{

if (str1.Length == str2.Length)

{

char[] charArr1 = str1.ToCharArray();

char[] charArr2 = str2.ToCharArray();

System.Array.Sort(charArr1);

System.Array.Sort(charArr2);

for (int i = 0; i < charArr1.Length; i++)

{

if (charArr1[i] != charArr2[i])

{

Console.WriteLine(str1 + " : is not anagram");

return false;

}

}

return true;

}

else

return false;

}

1. Find occurrences of characters in a String

string s1 = "Java is awesome";

char c = 'a';

int frequency = 0;

for (int i=0;i<s1.Length;i++)

{

if (s1[i]==c)

frequency++;

}

Console.WriteLine(frequency);

1. Find the count of Capital and small in a string

string str1 = "Welcome";

int smallCount = 0;

int capitalCount = 0;

for(int i=0; i<str1.Length; i++) {

if (str1[i] >= 'A' && str1[i] <= 'Z')

{

capitalCount++;

}

else if (str1[i] >= 'a' && str1[i] <= 'z')

smallCount++;

}

Console.WriteLine(capitalCount);

Console.WriteLine(smallCount);

1. Remove duplicate characters from string
2. Swap two number, without temporary variable
3. Reverse number

int num = 123;

int rem,sum=0;

while(num > 0)

{

rem = num%10;

sum = (sum \* 10) + rem;

num /= 10;

}

Console.WriteLine(sum);

1. Factorial
2. Fibonacci
3. Count number, alphabet and special characters
4. How to swap two numbers without 3rd variable

X = 10; Y = 20;

x = y - x; //10

y = y - x; //10

x = x + y; //20

1. How to multiple two numbers without any operator

int x = Convert.ToInt32 (Console.ReadLine());

int y = Convert.ToInt32 (Console.ReadLine());

int sum = 0;

for(int i = 1; i <= y; i++)

{

sum+= x;

}

Console.WriteLine(sum);

1. Write script to get prime numbers

**static** **boolean** isPrime(**int** n) {

**if**(n==0||n==1)

**return** **false**;

**for**(**int** i=2;i<n;i++) {

**if**(n%i==0)

**return** **false**;

}

**return** **true**;

}

1. Find number of words in given string

string s1 = " Find number of words in given string ";

s1 = s1.Trim();

int count = 0;

for(int i = 0; i < s1.Length; i++)

{

if (s1[i]==' ')

count++;

}

Console.WriteLine(++count);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Selenium

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Difference in driver.get() and driver.navigate().to()**

Below are the difference between driver.get() and driver.navigate().to() method:

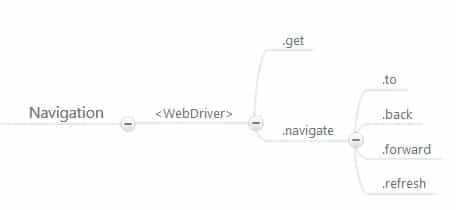
* **Browser History:**driver.get(url) does not maintain the browser history and cookies where as driver.navigate.to(url) maintains browser history and cookies.
* **Page Load:**driver.get(url) wait till the page fully loaded whereas driver.navigate.to(url) does not wait till the page fully loaded.
* **Forward/Backward:**We can not move forward/backward between the pages with driver.get() but we can navigate between the pages back, forward and refresh with driver.navigate().
* **Speed:** navigate() is faster than get() because navigate() does not wait for the page to load fully or completely.
* **Execution:**get() method execute next line of code once the page is completely loaded but  navigate() method does not wait for loading of page, it executes once the url hit into search-box.

1. **Navigation method in selenium**

Selenium **Navigation** is the first part of **Webdriver API** and we will cover all navigation methods such as **.get(<url>)**, **.navigate.to**, **.navigate.back(**), **.navigate.forward()**, etc…

**Selenium Navigation Methods**

Webdriver Navigation methods are shown below figure.

[](https://www.swtestacademy.com/wp-content/uploads/2015/12/navigation-1.jpg)

**.get (URL)**

We can go to any URL with **driver.get(“http://www.yahoo.com”)** command.

**.navigate.to(url)**

We can go to any URL with **driver.navigate().to(“http://www.yahoo.com”)** command.

**.navigate.back()**

We can go back to previous page with**driver.navigate().back()** command.

**.navigate.forward()**

We can go forward from the current page to the last opened page with**driver.navigate().next()**command.

**.navigate.refresh()**

We can refresh the webpage with**driver.navigate().refresh()**command.

**Webdriver Navigation Example**

@TestInstance**(**TestInstance.Lifecycle.PER\_CLASS**)**

@TestMethodOrder**(**MethodOrderer.MethodName.class**)**

**publicclass** NavigationTests **{**

**private** WebDriver driver;

**private** WebDriverWait wait;

**finalprivate**String URL1 = "http://www.yahoo.com";

**finalprivate**String URL2 = "http://www.amazon.com";

@BeforeEach

**public**voidsetupTest**(){**

WebDriverManager.chromedriver**()**.setup**()**;

driver = newChromeDriver**()**;

wait = newWebDriverWait**(**driver, Duration.ofSeconds**(**10**))**;

**}**

@AfterEach

**public**voidquitDriver**(){**

driver.quit**()**;

**}**

//.get Example

@Test

**public**voidT01\_getURLExample**(){**

//Go to www.yahoo.com

driver.get**(**URL1**)**;

//Check title is correct

Assertions.assertEquals**(**driver.getTitle**()**, "Yahoo"**)**;

**}**

//.Navigate().to example

@Test

**public**voidT02\_navigateToExample**(){**

//Go to www.amazon.com

driver.navigate**()**.to**(**URL2**)**;

//Check title is correct

Assertions.assertEquals**(**"Amazon.com. Spend less. Smile more.", driver.getTitle**())**;

**}**

@Test

//Back - Forward - Refresh Example

**public**voidT03\_backForwardRefreshExample**(){**

//Go to www.yahoo.com

driver.navigate**()**.to**(**URL1**)**;

wait.until**(**driver -**>** driver.getTitle**()**.contentEquals**(**"Yahoo"**))**;

//Check title is correct

Assertions.assertEquals**(**driver.getTitle**()**, "Yahoo"**)**;

//Go to www.amazon.com

driver.navigate**()**.to**(**URL2**)**;

wait.until**(**driver -**>** driver.getTitle**()**.contentEquals**(**"Amazon.com. Spend less. Smile more."**))**;

//Check title is correct

Assertions.assertEquals**(**"Amazon.com. Spend less. Smile more.", driver.getTitle**())**;

//\*\*\*Navigate Back\*\*\*

driver.navigate**()**.back**()**;

wait.until**(**driver -**>** driver.getTitle**()**.contentEquals**(**"Yahoo"**))**;

//Check title is correct

Assertions.assertEquals**(**driver.getTitle**()**, "Yahoo"**)**;

//\*\*\*Navigate Forward\*\*\*

driver.navigate**()**.forward**()**;

wait.until**(**driver -**>** driver.getTitle**()**.contentEquals**(**"Amazon.com. Spend less. Smile more."**))**;

//Check title is correct

Assertions.assertEquals**(**"Amazon.com. Spend less. Smile more.", driver.getTitle**())**;

//\*\*\*Refresh The Page\*\*\*

driver.navigate**()**.refresh**()**;

wait.until**(**driver -**>** driver.getTitle**()**.contentEquals**(**"Amazon.com. Spend less. Smile more."**))**;

Assertions.assertEquals**(**"Amazon.com. Spend less. Smile more.", driver.getTitle**())**;

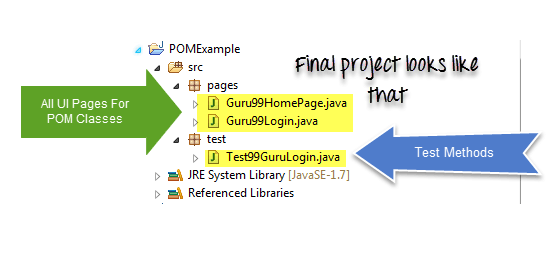
**}**

**}**

1. **What is POM**

**Page Object Model (POM)** is a design pattern, popularly used in test automation that creates Object Repository for web UI elements. The advantage of the model is that it reduces code duplication and improves test maintenance.

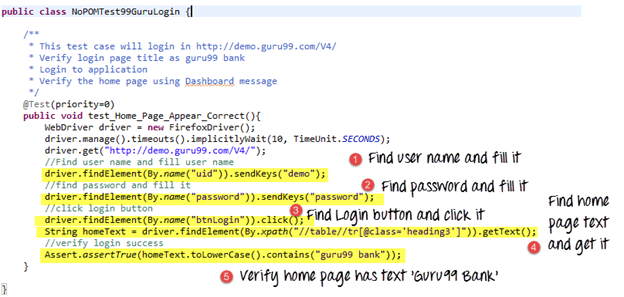
Under this model, for each web page in the application, there should be a corresponding Page Class. This Page class will identify the WebElements of that web page and also contains Page methods which perform operations on those WebElements. Name of these methods should be given as per the task they are performing, i.e., if a loader is waiting for the payment gateway to appear, POM method name can be waitForPaymentScreenDisplay().



Why Page Object Model?

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

Consider this simple script to login into a website



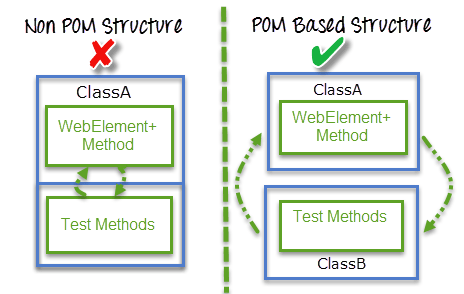
As you can observe, all we are doing is finding elements and filling values for those elements.

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

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

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

This approach is called Page Object Model in Selenium. It helps make the code more readable, maintainable, and reusable.



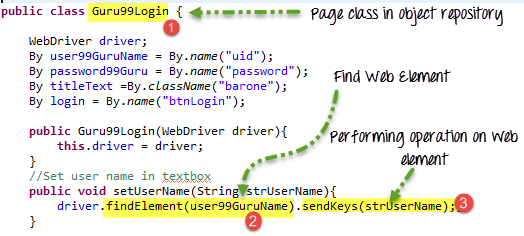
**Advantages of POM**

1. Page Object Design Pattern says operations and flows in the UI should be separated from verification. This concept makes our code cleaner and easy to understand.
2. The Second benefit is the object repository is independent of test cases, so we can use the same object repository for a different purpose with different tools. For example, we can integrate Page Object Model in Selenium with TestNG/JUnit for functional[Testing](https://www.guru99.com/software-testing.html)and at the same time with JBehave/Cucumber for acceptance testing.
3. Code becomes less and optimized because of the reusable page methods in the POM classes.
4. Methods get more realistic names which can be easily mapped with the operation happening in UI. i.e. if after clicking on the button we land on the home page, the method name will be like ‘gotoHomePage()’.

**How to implement POM?**

Simple POM:

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



1. Implicit wait
2. Explicit wait – use of explicit wait, how to ignore exceptions using explicit wait
3. Fluent wait
4. Examples for expected conditions in explicit wait
5. Default timeout of pageLoadTimeout
6. Use of desired capabilities
7. Use of ChromeOptions, firefoxOptions, ieOptions, etc
8. How to disable notification in selenium
9. Difference between getText() and getAttribute()
10. How to extract css property value
11. Difference between driver.close() and driver.quite()
12. How to handle dropdown elements with select tag and without tag
13. How to handles frames and windows
14. How to switch back to parent window
15. How to perform page scroll in selenium
16. Use of Robot Keys
17. Use of AutoIt
18. Use of Sikuli
19. How to perform file upload
20. How to click or send value to an element using element.click() and element.sendKeys()
21. Relative and absolute xpath
22. Explain few Xpath functions
23. How to login to site shoeing authentications popup in selenium
24. Mention few Selenium exceptions
25. Difference between single and double slash
26. What are the scenarios that cannot be automated using selenium
27. How to perform right click on selenium
28. Actions class and it’s methods
29. Use of HTMLUnit diver
30. Can we create an object of the WebDriver() in selenium
31. What is stale element in exception and how to handle it
32. Xpath and CSS selector

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# TestNG

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Annotations in testing
2. Difference between @BeforTestand @BeforeMethod
3. Soft assert vs Hard assert
4. How to pass parameters to test methods
5. Use of data provider
6. Use of invocationCount, enabled, priority, depedsOnMethod, depedsOnGroup, alwaysRun, dataProvider, groups, timeout, expectedExceptions, singleThteded, etc
7. TestNG xml structure
8. How to group testcases
9. Use of IRetryAnalyzer class
10. How to run failed testcases
11. How to perform parallel testing
12. What level user can perform parallel testing
13. Use of ISuiteListeners Methods
14. Use of TestListenersAdaptor methods
15. How to ignore packages and classes
16. Can user provide negative value for settings priority

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# REST API

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Client/server Error Codes
2. Success codes
3. Use of GET, PUT, POST, PATCH, DELETE
4. Difference between Put and Patch
5. Difference between Put and Post
6. How to handle authentication
7. Difference between webservers and API
8. Json/xml structure format
9. How to validate json response
10. Use of JsonPath
11. How to construct json body for Put, Post, Patch, request
12. Use of JsonObject
13. What’s is URI
14. What’s a resource?
15. What are the common headers used?
16. Common content types?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# GIT

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Use of common commands used: add, status, commit, push, pull, restore, checkout, clone
2. How to handle merge conflicts?
3. How merge issues occur?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Cucumber

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Keywords used inside CucumberOptions
2. Difference betweenFryRun and Strict in cucumberOptions
3. And/OR operations in Tags
4. Annotations in Cucumber
5. Use of order function inside cucumber annotations
6. Use of background keyword
7. Keywords used in cucumber
8. How to perform singleline and multiline comment in cucumber
9. How data is passed in cucumber
10. Use of ScenarioOutline and example
11. What is dataTable
12. Can user provide multiple examples for one scenario outline

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*