**OOPs concepts in Java**

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of “objects” that contain data and methods. The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs. Object oriented programming brings together data and its behaviour(methods) in a single location(object) makes it easier to understand how a program works.

# Polymorphism

Polymorphism is one of the [OOPs](https://beginnersbook.com/2013/04/oops-concepts/) feature that allows us to perform a single action in different ways. For example, let’s say we have a class Animal that has a method sound(). Since this is a generic class so we can’t give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

public class Animal{

...

public void sound(){

System.out.println("Animal is making a sound");

}

}

Now lets say we two subclasses of Animal class: Horse and Cat that extends (see [Inheritance](https://beginnersbook.com/2013/03/inheritance-in-java/)) Animal class. We can provide the implementation to the same method like this:

public class Horse extends Animal{

...

@Override

public void sound(){

System.out.println("Neigh");

}

}

and

public class Cat extends Animal{

...

@Override

public void sound(){

System.out.println("Meow");

}

}

## What is polymorphism in programming?

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.  
Which sound() method will be called is determined at runtime so the example we gave above is a **runtime polymorphism example**.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:  
1. [Method Overloading in Java](https://beginnersbook.com/2013/05/method-overloading/) – This is an example of compile time (or static polymorphism)  
2. [Method Overriding in Java](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) – This is an example of runtime time (or dynamic polymorphism)  
3. [Types of Polymorphism – Runtime and compile time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) – This is our next tutorial where we have covered the types of polymorphism in detail. I would recommend you to go though method overloading and overriding before going though this topic.

## Example 1: Polymorphism in Java

**Runtime Polymorphism example:**  
Animal.java

public class Animal{

public void sound(){

System.out.println("Animal is making a sound");

}

}

Horse.java

class Horse extends Animal{

@Override

public void sound(){

System.out.println("Neigh");

}

public static void main(String args[]){

Animal obj = new Horse();

obj.sound();

}

}

Output:

Neigh

Cat.java

public class Cat extends Animal{

@Override

public void sound(){

System.out.println("Meow");

}

public static void main(String args[]){

Animal obj = new Cat();

obj.sound();

}

}

Output:

Meow

## Example 2: Compile time Polymorphism

Method Overloading on the other hand is a compile time polymorphism example.

class Overload

{

void demo (int a)

{

System.out.println ("a: " + a);

}

void demo (int a, int b)

{

System.out.println ("a and b: " + a + "," + b);

}

double demo(double a) {

System.out.println("double a: " + a);

return a\*a;

}

}

class MethodOverloading

{

public static void main (String args [])

{

Overload Obj = new Overload();

double result;

Obj .demo(10);

Obj .demo(10, 20);

result = Obj .demo(5.5);

System.out.println("O/P : " + result);

}

}

Here the method demo() is overloaded 3 times: first method has 1 int parameter, second method has 2 int parameters and third one is having double parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at  compile time so this type of polymorphism is known as compile time polymorphism.

**Output:**

a: 10

a and b: 10,20

double a: 5.5

O/P : 30.25

1) **Static Polymorphism** also known as compile time polymorphism  
2) **Dynamic Polymorphism** also known as runtime polymorphism

## Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.  
**Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. We have already discussed Method overloading here: If you didn’t read that guide, refer: [Method Overloading in Java](https://beginnersbook.com/2013/05/method-overloading/)

### Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

class SimpleCalculator

{

int add(int a, int b)

{

return a+b;

}

int add(int a, int b, int c)

{

return a+b+c;

}

}

public class Demo

{

public static void main(String args[])

{

SimpleCalculator obj = new SimpleCalculator();

System.out.println(obj.add(10, 20));

System.out.println(obj.add(10, 20, 30));

}

}

**Output:**

30

60

## Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism. I have already discussed method overriding in detail in a separate tutorial, refer it: [Method Overriding in Java](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/).

**Example**  
In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of [inheritance in Java](https://beginnersbook.com/2013/03/inheritance-in-java/).

class ABC{

public void myMethod(){

System.out.println("Overridden Method");

}

}

public class XYZ extends ABC{

public void myMethod(){

System.out.println("Overriding Method");

}

public static void main(String args[]){

ABC obj = new XYZ();

obj.myMethod();

}

}

**Output:**

Overriding Method

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.  
Since both the classes, child class and parent class have the same method animalSound. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

**Few more overriding examples:**

ABC obj = new ABC();

obj.myMethod();

// This would call the myMethod() of parent class ABC

XYZ obj = new XYZ();

obj.myMethod();

// This would call the myMethod() of child class XYZ

ABC obj = new XYZ();

obj.myMethod();

// This would call the myMethod() of child class XYZ

In the third case the method of child class is to be executed because which method is to be executed is determined by the type of object and since the object belongs to the child class, the child class version of myMethod() is called.

# Method Overloading in Java with examples

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](https://beginnersbook.com/category/oops-concept/)

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to [constructor overloading](https://beginnersbook.com/2013/05/constructor-overloading/) in Java, that allows a class to have more than one constructor having different argument lists.

let’s get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

## Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:  
1. Number of parameters.  
For example: This is a valid case of overloading

add(int, int)

add(int, int, int)

2. Data type of parameters.  
For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.  
For example:

add(int, float)

add(float, int)

**Invalid case of method overloading:**  
When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

int add(int, int)

float add(int, int)

**Method overloading** is an example of [Static Polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/). We will discuss [polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/) and types of it in a separate tutorial.

**Points to Note:**  
1. Static Polymorphism is also known as compile time binding or early binding.  
2. [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

## Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

### Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

class DisplayOverloading

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(char c, int num)

{

System.out.println(c + " "+num);

}

}

class Sample

{

public static void main(String args[])

{

DisplayOverloading obj = new DisplayOverloading();

obj.disp('a');

obj.disp('a',10);

}

}

**Output:**

a

a 10

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

### Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

class DisplayOverloading2

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(int c)

{

System.out.println(c );

}

}

class Sample2

{

public static void main(String args[])

{

DisplayOverloading2 obj = new DisplayOverloading2();

obj.disp('a');

obj.disp(5);

}

}

Output:

a

5

### Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

class DisplayOverloading3

{

public void disp(char c, int num)

{

System.out.println("I’m the first definition of method disp");

}

public void disp(int num, char c)

{

System.out.println("I’m the second definition of method disp" );

}

}

class Sample3

{

public static void main(String args[])

{

DisplayOverloading3 obj = new DisplayOverloading3();

obj.disp('x', 51 );

obj.disp(52, 'y');

}

}

**Output:**

I’m the first definition of method disp

I’m the second definition of method disp

## Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

**What it has to do with method overloading?**  
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.  
Lets take an example to see what I am talking here:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* I am passing float value as a second argument but

\* it got promoted to the type double, because there

\* wasn't any method having arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn’t any method with argument list as (int, float)

But this type promotion doesn’t always happen, lets see another example:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

void disp(int a, float b){

System.out.println("Method C");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* This time promotion won't happen as there is

\* a method with arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method C

As you see that this time type promotion didn’t happen because there was a method with matching argument type.  
**Type Promotion table:**  
The data type on the left side can be promoted to the any of the data type present in the right side of it.

byte → short → int → long

short → int → long

int → long → float → double

float → double

long → float → double

## Lets see few Valid/invalid cases of method overloading

Case 1:

int mymethod(int a, int b, float c)

int mymethod(int var1, int var2, float var3)

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

int mymethod(int a, int b)

int mymethod(float var1, float var2)

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

int mymethod(int a, int b)

int mymethod(int num)

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

float mymethod(int a, float b)

float mymethod(float var1, int var2)

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

int mymethod(int a, int b)

float mymethod(int var1, int var2)

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.

Guess the answers before checking it at the end of programs:  
**Question 1 – return type, method name and argument list same.**

class Demo

{

public int myMethod(int num1, int num2)

{

System.out.println("First myMethod of class Demo");

return num1+num2;

}

public int myMethod(int var1, int var2)

{

System.out.println("Second myMethod of class Demo");

return var1-var2;

}

}

class Sample4

{

public static void main(String args[])

{

Demo obj1= new Demo();

obj1.myMethod(10,10);

obj1.myMethod(20,12);

}

}

**Answer:**  
It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

**Question 2 – return type is different. Method name & argument list same.**

class Demo2

{

public double myMethod(int num1, int num2)

{

System.out.println("First myMethod of class Demo");

return num1+num2;

}

public int myMethod(int var1, int var2)

{

System.out.println("Second myMethod of class Demo");

return var1-var2;

}

}

class Sample5

{

public static void main(String args[])

{

Demo2 obj2= new Demo2();

obj2.myMethod(10,10);

obj2.myMethod(20,12);

}

}

**Answer:**  
It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn’t matter in case of overloading.**

# Abstract Class

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body).

An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

## Why we need an abstract class?

Lets say we have a class Animal that has a method sound() and the subclasses(see [inheritance](https://beginnersbook.com/2013/03/inheritance-in-java/)) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class.

## Abstract class Example

//abstract parent class

abstract class Animal{

//abstract method

public abstract void sound();

}

//Dog class extends Animal class

public class Dog extends Animal{

public void sound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.sound();

}

}

## Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

//Declaration using abstract keyword

abstract class A{

//This is abstract method

abstract void myMethod();

//This is concrete method with body

void anotherMethod(){

//Does something

}

}

## Rules

**Note 1:** As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

### Why can’t we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen?There would be no actual implementation of the method to invoke.  
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

## What is an interface

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: [Java abstract method](https://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/)). Also, the variables declared in an interface are public, static & final by default.

## What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

## Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

interface MyInterface

{

/\* compiler will treat them as:

\* public abstract void method1();

\* public abstract void method2();

\*/

public void method1();

public void method2();

}

class Demo implements MyInterface

{

/\* This class must have to implement both the abstract methods

\* else you will get compilation error

\*/

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public static void main(String arg[])

{

MyInterface obj = new Demo();

obj.method1();

}

}

## Difference Between Abstract Class and Interface in Java

|  |  |  |
| --- | --- | --- |
|  | **Abstract Class** | **Interface** |
| 1 | An abstract class can extend only one class or one abstract class at a time | An interface can extend any number of interfaces at a time |
| 2 | An abstract class can extend another concrete (regular) class or abstract class | An interface can only extend another interface |
| 3 | An abstract class can have both abstract and concrete methods | An interface can have only abstract methods |
| 4 | In abstract class keyword “abstract” is mandatory to declare a method as an abstract | In an interface keyword “abstract” is optional to declare a method as an abstract |
| 5 | An abstract class can have protected and public abstract methods | An interface can have only have public abstract methods |
| 6 | An abstract class can have static, final or static final variable with any [access specifier](https://beginnersbook.com/2013/05/java-access-modifiers/) | interface can only have public static final (constant) variable |

### Difference No.1: Abstract class can extend only one class or one abstract class at a time

class Example1{

public void display1(){

System.out.println("display1 method");

}

}

abstract class Example2{

public void display2(){

System.out.println("display2 method");

}

}

abstract class Example3 extends Example1{

abstract void display3();

}

class Example4 extends Example3{

public void display3(){

System.out.println("display3 method");

}

}

class Demo{

public static void main(String args[]){

Example4 obj=new Example4();

obj.display3();

}

}

**Interface can extend any number of interfaces at a time**

//first interface

interface Example1{

public void display1();

}

//second interface

interface Example2 {

public void display2();

}

//This interface is extending both the above interfaces

interface Example3 extends Example1,Example2{

}

class Example4 implements Example3{

public void display1(){

System.out.println("display2 method");

}

public void display2(){

System.out.println("display3 method");

}

}

class Demo{

public static void main(String args[]){

Example4 obj=new Example4();

obj.display1();

}

}

### Difference No.2: Abstract class can be extended(inherited) by a class or an abstract class

class Example1{

public void display1(){

System.out.println("display1 method");

}

}

abstract class Example2{

public void display2(){

System.out.println("display2 method");

}

}

abstract class Example3 extends Example2{

abstract void display3();

}

class Example4 extends Example3{

public void display2(){

System.out.println("Example4-display2 method");

}

public void display3(){

System.out.println("display3 method");

}

}

class Demo{

public static void main(String args[]){

Example4 obj=new Example4();

obj.display2();

}

}

**Interfaces can be extended only by interfaces. Classes has to implement them instead of extend**

interface Example1{

public void display1();

}

interface Example2 extends Example1{

}

class Example3 implements Example2{

public void display1(){

System.out.println("display1 method");

}

}

class Demo{

public static void main(String args[]){

Example3 obj=new Example3();

obj.display1();

}

}

### Difference No.3: Abstract class can have both abstract and concrete methods

abstract class Example1 {

abstract void display1();

public void display2(){

System.out.println("display2 method");

}

}

class Example2 extends Example1{

public void display1(){

System.out.println("display1 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

**Interface can only have abstract methods, they cannot have concrete methods**

interface Example1{

public abstract void display1();

}

class Example2 implements Example1{

public void display1(){

System.out.println("display1 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

### Difference No.4: In abstract class, the keyword ‘abstract’ is mandatory to declare a method as an abstract

abstract class Example1{

public abstract void display1();

}

class Example2 extends Example1{

public void display1(){

System.out.println("display1 method");

}

public void display2(){

System.out.println("display2 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

**In interfaces, the keyword ‘abstract’ is optional to declare a method as an abstract because all the methods are abstract by default**

interface Example1{

public void display1();

}

class Example2 implements Example1{

public void display1(){

System.out.println("display1 method");

}

public void display2(){

System.out.println("display2 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

### Difference No.5: Abstract class can have protected and public abstract methods

abstract class Example1{

protected abstract void display1();

public abstract void display2();

public abstract void display3();

}

class Example2 extends Example1{

public void display1(){

System.out.println("display1 method");

}

public void display2(){

System.out.println("display2 method");

}

public void display3(){

System.out.println("display3 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

**Interface can have only public abstract methods**

interface Example1{

void display1();

}

class Example2 implements Example1{

public void display1(){

System.out.println("display1 method");

}

public void display2(){

System.out.println("display2 method");

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

### Difference No.6: Abstract class can have static, final or static final variables with any access specifier

abstract class Example1{

private int numOne=10;

protected final int numTwo=20;

public static final int numThree=500;

public void display1(){

System.out.println("Num1="+numOne);

}

}

class Example2 extends Example1{

public void display2(){

System.out.println("Num2="+numTwo);

System.out.println("Num2="+numThree);

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

obj.display2();

}

}

**Interface can have only public static final (constant) variable**

interface Example1{

int numOne=10;

}

class Example2 implements Example1{

public void display1(){

System.out.println("Num1="+numOne);

}

}

class Demo{

public static void main(String args[]){

Example2 obj=new Example2();

obj.display1();

}

}

## What is encapsulation?

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

However if we setup public getter and setter methods to update (for example void setSSN(int ssn))and read (for example  int getSSN()) the private data fields then the outside class can access those private data fields via public methods.

This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That’s why encapsulation is known as **data hiding.**Lets see an example to understand this concept better.

## Example of Encapsulation in Java

How to implement encapsulation in java:  
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.  
2) Have getter and setter methods in the class to set and get the values of the fields.

class EncapsulationDemo{

private int ssn;

private String empName;

private int empAge;

//Getter and Setter methods

public int getEmpSSN(){

return ssn;

}

public String getEmpName(){

return empName;

}

public int getEmpAge(){

return empAge;

}

public void setEmpAge(int newValue){

empAge = newValue;

}

public void setEmpName(String newValue){

empName = newValue;

}

public void setEmpSSN(int newValue){

ssn = newValue;

}

}

public class EncapsTest{

public static void main(String args[]){

EncapsulationDemo obj = new EncapsulationDemo();

obj.setEmpName("Mario");

obj.setEmpAge(32);

obj.setEmpSSN(112233);

System.out.println("Employee Name: " + obj.getEmpName());

System.out.println("Employee SSN: " + obj.getEmpSSN());

System.out.println("Employee Age: " + obj.getEmpAge());

}

}

## Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of void setEmpName(String name) and String getEmpName() can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (setEmpName(String name) and getEmpName()). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
2. The fields can be made read-only (If we don’t define setter methods in the class) or write-only (If we don’t define the getter methods in the class). For e.g. If we have a field(or variable) that we don’t want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Encapsulation is also known as “**data Hiding**“.

## What is Constructor and Destructor in Java?

A constructor is used to initialize a variable that means it allocates memory for the same A constructor is nothing but automatic initialization of the object. Whenever the program creates an object at that time constructor is gets called automatically. You don’t need to call this method explicitly. Destructor is used to free that memory allocated while initialization. Generally, in java, we don’t need to call the destructor explicitly. Java has a feature of automatic garbage collection.

## Types of Constructor

There are two types of constructors depending upon the type we can add and remove variables.

* Default Constructor
* Parameterized Constructor

##### 1. Default Constructor

* This is the one type of constructor. By default without any parameters, this constructor takes place. This constructor does not have any parameters in it.
* **Example:**
* Class Abc{  
  Abc(){  
  System.out.println(“This is the example of  default constructor.”);  
  }  
  }

##### 2. Parameterized Constructor

As the name suggest parameterized constructor has some parameters or arguments at the time of initializing the object.

**Example:**

class Square{  
int width,height;  
Square( int a , int b){  
width = a;  
height = b;  
}  
int area(){  
return width \* height;  
}  
}  
class Cal{  
public static void main(String[] args){  
{  
Square s1 = new Square(10,20);  
int area\_of\_sqaure = s1.area();  
System.out.println("The area of square is:" + area\_of\_sqaure);  
}  
}  
}

## Lambda Expressions

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

## Syntax

The simplest lambda expression contains a single parameter and an expression:

*parameter* -> *expression*

To use more than one parameter, wrap them in parentheses:

(*parameter1*, *parameter2*) -> *expression*

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

(*parameter1*, *parameter2*) -> { *code block* }

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

### **Example**

Use Java's Consumer interface to store a lambda expression in a variable:

import java.util.ArrayList;

import java.util.function.Consumer;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

Consumer<Integer> method = (n) -> { System.out.println(n); };

numbers.forEach( method );

}

}

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_lambda2)

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

### **Example**

Create a method which takes a lambda expression as a parameter:

interface StringFunction {

String run(String str);

}

public class Main {

public static void main(String[] args) {

StringFunction exclaim = (s) -> s + "!";

StringFunction ask = (s) -> s + "?";

printFormatted("Hello", exclaim);

printFormatted("Hello", ask);

}

public static void printFormatted(String str, StringFunction format) {

String result = format.run(str);

System.out.println(result);

}

}