# **OOPs concepts in Java**

<https://beginnersbook.com/2013/04/oops-concepts/>

#### **OOPs Concepts**

* [OOPs Concepts](https://beginnersbook.com/2013/04/oops-concepts/)
* [Constructor](https://beginnersbook.com/2013/03/constructors-in-java/)
* Abtraction
* [Polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/)
* [Inheritance](https://beginnersbook.com/2013/03/inheritance-in-java/)
* [Encapsulation](https://beginnersbook.com/2013/05/encapsulation-in-java/)
* [Static keyword](https://beginnersbook.com/2013/04/java-static-class-block-methods-variables/)
* [Types of inheritance](https://beginnersbook.com/2013/05/java-inheritance-types/)
* [Aggregation](https://beginnersbook.com/2013/05/aggregation/)
* [Association](https://beginnersbook.com/2013/05/association/)
* [Super Keyword](https://beginnersbook.com/2014/07/super-keyword-in-java-with-example/)
* [Method overloading](https://beginnersbook.com/2013/05/method-overloading/)
* [Method overriding](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/)
* [Overloading vs Overriding](https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/)
* [Polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/)
* [Types of polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)
* [Static and dynamic binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/)
* [Abstract class and methods](https://beginnersbook.com/2013/05/java-abstract-class-method/)
* [Interface](https://beginnersbook.com/2013/05/java-interface/)
* [Abstract class vs interface](https://beginnersbook.com/2013/05/abstract-class-vs-interface-in-java/)
* [Packages](https://beginnersbook.com/2013/03/packages-in-java/)
* [Access modifiers](https://beginnersbook.com/2013/05/java-access-modifiers/)
* [Garbage Collection](https://beginnersbook.com/2013/04/java-garbage-collection/)
* [Inner classes](https://beginnersbook.com/2013/05/inner-class/)
* [Static import](https://beginnersbook.com/2013/05/java-static-import/)
* [Static constructor](https://beginnersbook.com/2013/05/static-constructor/)
* [Java Interview Q](https://beginnersbook.com/2013/05/java-interview-questions/)

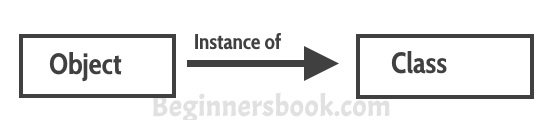
# **OOPs concepts in Java**

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. We will cover each and every feature of OOPs in detail so that you won’t face any difficultly understanding **OOPs Concepts**.

## **OOPs Concepts – Table of Contents**

1. [What is an Object](https://beginnersbook.com/2013/04/oops-concepts/#1)
2. [What is a class](https://beginnersbook.com/2013/04/oops-concepts/#2)
3. [Constructor in Java](https://beginnersbook.com/2013/04/oops-concepts/#3)
4. [Object Oriented Programming Features](https://beginnersbook.com/2013/04/oops-concepts/#4)
   * [Abstraction](https://beginnersbook.com/2013/04/oops-concepts/#5)
   * [Encapsulation](https://beginnersbook.com/2013/04/oops-concepts/#6)
   * [Inheritance](https://beginnersbook.com/2013/04/oops-concepts/#7)
   * [Polymorphism](https://beginnersbook.com/2013/04/oops-concepts/#8)
5. [Abstract Class and Methods](https://beginnersbook.com/2013/04/oops-concepts/#9)
6. [Interfaces in Java](https://beginnersbook.com/2013/04/oops-concepts/#10)

## **What is an Object**



**Object:** is a bundle of data and its behaviour(often known as methods).

Objects have two characteristics: They have **states** and **behaviors**.

**Examples of states and behaviors**

**Example 1:**

**Object**: House

**State**: Address, Color, Area

**Behavior**: Open door, close door

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class. We will see how to create classes in the next section of this guide.

**Example 2:**

Let’s take another example.

**Object**: Car

**State**: Color, Brand, Weight, Model

**Behavior**: Break, Accelerate, Slow Down, Gear change.

## **What is a Class in OOPs Concepts**

A class can be considered as a blueprint using which you can create as many objects as you like. For example, here we have a class Website that has two data members (also known as fields, instance variables and object states). This is just a blueprint, it does not represent any website, however using this we can create Website objects (or instances) that represents the websites. We have created two objects, while creating objects we provided separate properties to the objects using constructor.

## **What is a Constructor**

[Constructor](https://beginnersbook.com/2013/03/constructors-in-java/) looks like a method but it is in fact not a method. It’s name is same as class name and it does not return any value. You must have seen this statement in almost all the programs I have shared above:

MyClass obj = new MyClass();

If you look at the right side of this statement, we are calling the default constructor of class myClass to create a new object (or instance).

### **Four Pillars of OOP:**

**Abstraction**

**Abstraction**: Abstraction is a process where you **show only “relevant” data** and **“hide” unnecessary details of an object from the user**. For example, when you login to your bank account online, you enter your user\_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you. Read more about it here: [Abstraction in Java](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/).

Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/java/java_interface.asp)

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

**To access the abstract class, it must be inherited from another class.**

Code:

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class MyMainClass {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

Output:

The pig says: wee wee

Zzz

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

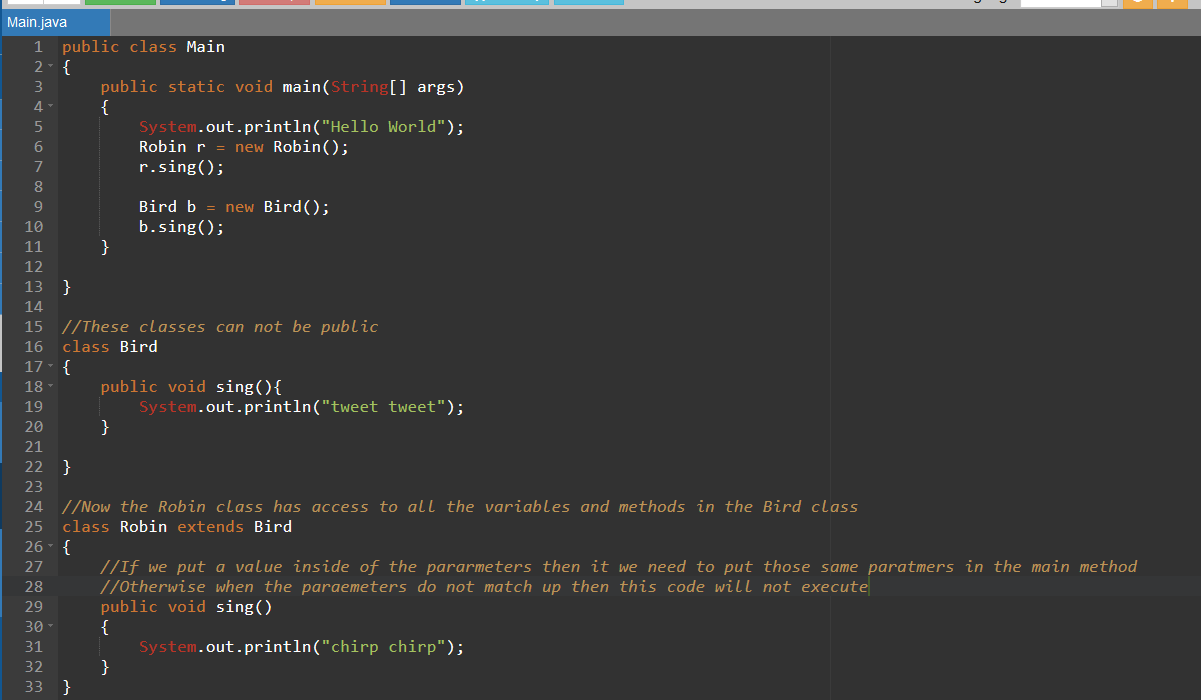
**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. **Avoids code duplication and increases reusability.**
3. **Helps to increase security of an application or program as only important details are provided to the user.**

Poly morf ism

### **Polymorphism**

[Polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/) is an object oriented programming feature that **allows us to perform a single action in different ways. When we have many classes that are related to each other by inheritance**. [**Inheritance**](https://www.w3schools.com/cs/cs_inheritance.asp) lets us inherit fields and methods from another class. **Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.**



[Types of Polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)

1) Static Polymorphism

2) Dynamic Polymorphism

#### **Static Polymorphism:**

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.

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

**Method Overloading**: This allows us to have more than one methods with same name in a class that differs in signature.

#### **Dynamic Polymorphism or (Runtime 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 rather, thats why it is called runtime polymorphism.

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

## **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 runtime 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

**Inheritance**

The process by which one class acquires the properties and functionalities of another class. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

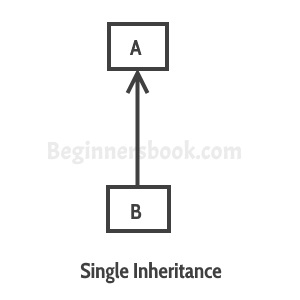
1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
2. Inheritance allows us to reuse of code, it improves reusability in your java application.
3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

**Note:** The biggest advantage of Inheritance is that the code in base class need not be rewritten in the child class.

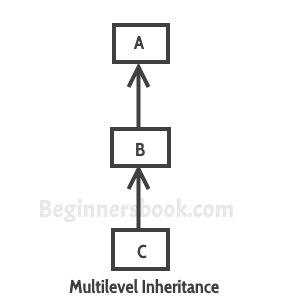
The **variables** and **methods** of the base class can be used in the **child class** as well.

[Types of Inheritance](https://beginnersbook.com/2013/05/java-inheritance-types/):

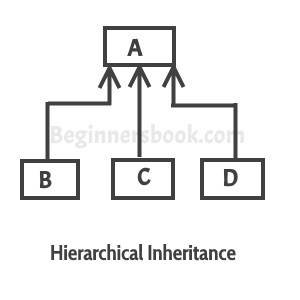
**Single Inheritance**: refers to a child and parent class relationship where a class extends the another class. **Single inheritance** is damn easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



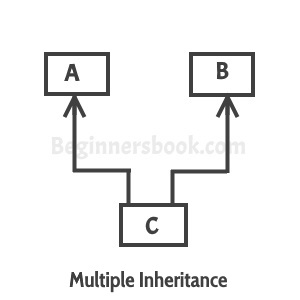
**Multilevel inheritance**: refers to a child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C. **Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer – [Multilevel inheritance in Java](https://beginnersbook.com/2013/12/multilevel-inheritance-in-java-with-example/).



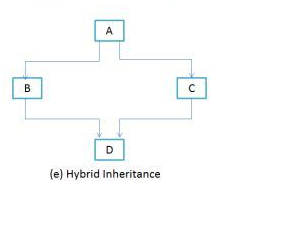
**Hierarchical inheritance**: refers to a child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A. In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D. Read More at – [Hierarchical Inheritance in java with example program](https://beginnersbook.com/2013/10/hierarchical-inheritance-java-program/).



**Multiple Inheritance**: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. Java doesn’t support multiple inheritance, read more about it [here](https://beginnersbook.com/2013/05/java-multiple-inheritance/). “**Multiple Inheritance**” refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.



### **Hybrid Inheritance: I**n simple terms you can say that Hybrid inheritance is a combination of Single and Multiple inheritance. A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using interfaces you can have multiple as well as hybrid inheritance in Java.



**Child Class:**

The class that extends the features of another class is known as child class, sub class or derived class.

**Parent Class:**

The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

## **Syntax: Inheritance in Java**

To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

class XYZ extends ABC

{

}

class Teacher {

String designation = "Teacher";

String collegeName = "Beginnersbook";

void does(){

System.out.println("Teaching");

}

}

public class PhysicsTeacher extends Teacher{

String mainSubject = "Physics";

public static void main(String args[]){

PhysicsTeacher obj = new PhysicsTeacher();

System.out.println(obj.collegeName);

System.out.println(obj.designation);

System.out.println(obj.mainSubject);

obj.does();

}

}

Output:

Beginnersbook

Teacher

Physics

Teaching

## **Inheritance and Method Overriding**

When we declare the same method in child class which is already present in the parent class the this is called [method overriding](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/). In this case when we call the method from child class object, the child class version of the method is called. However we can call the parent class method using super keyword as I have shown in the example below:

class ParentClass{

//Parent class constructor

ParentClass(){

System.out.println("Constructor of Parent");

}

void disp(){

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

}

}

class JavaExample extends ParentClass{

JavaExample(){

System.out.println("Constructor of Child");

}

void disp(){

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

//Calling the disp() method of parent class

super.disp();

}

public static void main(String args[]){

//Creating the object of child class

JavaExample obj = new JavaExample();

obj.disp();

}

}

The output is :

Constructor of Parent

Constructor of Child

Child Method

Parent Method

**Encapsulation**

**Encapsulation**: Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating a class, you are doing encapsulation.

#### **Encapsulation example in Java**

How to do it?

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 EmployeeCount

{

private int numOfEmployees = 0;

public void setNoOfEmployees (int count)

{

numOfEmployees = count;

}

public double getNoOfEmployees ()

{

return numOfEmployees;

}

}

public class EncapsulationExample

{

public static void main(String args[])

{

EmployeeCount obj = new EmployeeCount ();

obj.setNoOfEmployees(5613);

System.out.println("No Of Employees: "+(int)obj.getNoOfEmployees());

}

}

**Output:**

No Of Employees: 5613

**So what is the benefit of encapsulation in java programming**

Well, at some point of time, if you want to change the implementation details of the class EmployeeCount, you can freely do so without affecting the classes that are using it.

## **What is a Class in OOPs Concepts**

A blueprint using which you can create as many objects

## **Constructor**

[Constructor](https://beginnersbook.com/2013/03/constructors-in-java/) looks like a method but it is in fact not a method. It’s name is same as class name and it does not return any value. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type.

Example of constructor in Java

public class MyClass{

//This is the constructor

MyClass(){

}

..

}

### **A simple constructor program in java**

Here we have created an object obj of class Hello and then we displayed the instance variable name of the object. As you can see that the output is BeginnersBook.com which is what we have passed to the name during initialization in constructor. This shows that when we created the object obj the constructor got invoked. In this example we have used **this keyword**, which refers to the current object, object obj in this example. We will cover this keyword in detail in the next tutorial.

public class Hello {

String name;

//Constructor

Hello(){

this.name = "BeginnersBook.com";

}

public static void main(String[] args) {

Hello obj = new Hello();

System.out.println(obj.name);

}

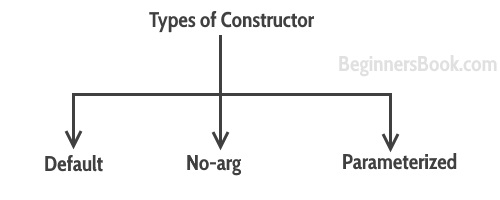
}

**Output:**

BeginnersBook.com

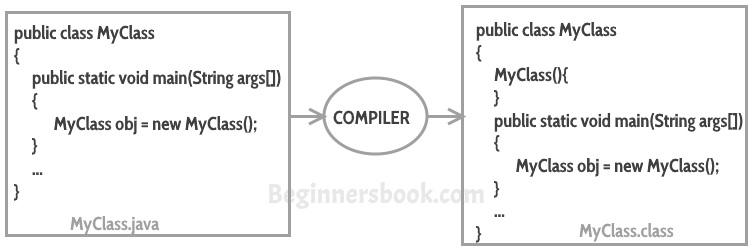
## **Types of Constructors**

There are three types of constructors: Default, No-arg constructor and Parameterized.



### **Default constructor**

If you do not implement any constructor in your class, Java compiler inserts a [default constructor](https://beginnersbook.com/2014/01/default-constructor-java-example/) into your code on your behalf. This constructor is known as default constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. If you implement any constructor then you no longer receive a default constructor from Java compiler. This process is shown in the diagram below:



### **no-arg constructor:**

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty. Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

class Demo

{

public Demo()

{

System.out.println("This is a no argument constructor");

}

public static void main(String args[]) {

new Demo();

}

}

Output:

This is a no argument constructor

### **Parameterized constructor:**

Constructor with arguments(or you can say parameters) is known as [Parameterized constructor](https://beginnersbook.com/2014/01/parameterized-constructor-in-java-example/).

#### **Example: parameterized constructor**

In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2.

public class Employee {

int empId;

String empName;

//parameterized constructor with two parameters

Employee(int id, String name){

this.empId = id;

this.empName = name;

}

void info(){

System.out.println("Id: "+empId+" Name: "+empName);

}

public static void main(String args[]){

Employee obj1 = new Employee(10245,"Chaitanya");

Employee obj2 = new Employee(92232,"Negan");

obj1.info();

obj2.info();

}

}

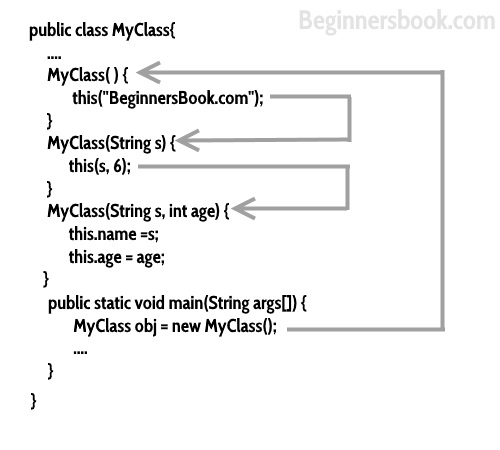
**Output:**

**Id: 10245 Name: Chaitanya**

**Id: 92232 Name: Negan**

## **Constructor Chaining**

When A constructor calls another constructor of same class then this is called constructor chaining. Read more about it [here](https://beginnersbook.com/2013/12/java-constructor-chaining-with-example/).

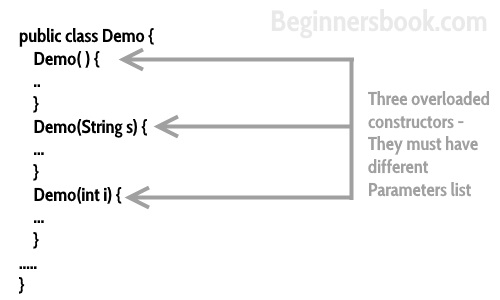


## **Super()**

Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class. You can also say that the compiler inserts a super(); statement at the beginning of child class constructor.

## **Constructor Overloading**

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.



## **Quick Recap**

1. Every class has a constructor whether it’s a normal class or a abstract class.
2. Constructors are not methods and they don’t have any return type.
3. Constructor name should match with class name .
4. Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
5. **Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**
6. If you don’t implement any constructor within the class, compiler will do it for.
7. **this() and super() should be the first statement in the constructor code.** If you don’t mention them, compiler does it for you accordingly.
8. Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can’t override a constructor.
9. Constructors can not be inherited.
10. If Super class doesn’t have a no-arg(default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario.
11. Interfaces [do not have constructors](https://beginnersbook.com/2013/12/java-constructor-in-interface/).
12. Abstract class can have constructor and it gets invoked when a class, which implements interface, is instantiated. (i.e. object creation of concrete class).
13. A constructor can also invoke another constructor of the same class – By using this(). If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.

## **Difference between Constructor and Method**

1. The purpose of constructor is to initialize the object of a class while the purpose of a method is to perform a task by executing java code.
2. Constructors cannot be abstract, final, static and synchronised while methods can be.
3. Constructors do not have return types while methods do.

Static Keyword

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object.

Here we have a static method myMethod(), we can call this method without any object because when we make a member static it becomes class level. If we remove the static keyword and make it non-static then we must need to create an object of the class in order to call it.

Static members are common for all the instances(objects) of the class but non-static members are separate for each instance of class.

class SimpleStaticExample

{

// This is a static method

static void myMethod()

{

System.out.println("myMethod");

}

public static void main(String[] args)

{

/\* You can see that we are calling this

\* method without creating any object.

\*/

myMethod();

}

}

Output:

myMethod

### **Example 1: Single static block**

As you can see that both the static variables were intialized before we accessed them in the main method.

class JavaExample{

static int num;

static String mystr;

static{

num = 97;

mystr = "Static keyword in Java";

}

public static void main(String args[])

{

System.out.println("Value of num: "+num);

System.out.println("Value of mystr: "+mystr);

}

}

Output:

Value of num: 97

Value of mystr: Static keyword in Java

### **Example 2: Multiple Static blocks**

Lets see how multiple static blocks work in Java. They execute in the given order which means the first static block executes before second static block. That’s the reason, values initialized by first block are overwritten by second block.

class JavaExample2{

static int num;

static String mystr;

//First Static block

static{

System.out.println("Static Block 1");

num = 68;

mystr = "Block1";

}

//Second static block

static{

System.out.println("Static Block 2");

num = 98;

mystr = "Block2";

}

public static void main(String args[])

{

System.out.println("Value of num: "+num);

System.out.println("Value of mystr: "+mystr);

}

}

**Output:**

Static Block 1

Static Block 2

Value of num: 98

Value of mystr: Block2

## **Java Static Variables**

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Few Important Points:

* Static variables are also known as Class Variables.
* Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

### **Example 1: Static variables can be accessed directly in Static method**

Here we have a static method disp() and two static variables var1 and var2. Both the variables are accessed directly in the static method.

class JavaExample3{

static int var1;

static String var2;

//This is a Static Method

static void disp(){

System.out.println("Var1 is: "+var1);

System.out.println("Var2 is: "+var2);

}

public static void main(String args[])

{

disp();

}

}

**Output:**

Var1 is: 0

Var2 is: null

### **Example 2: Static variables are shared among all the instances of class**

In this example, String variable is non-static and integer variable is Static. As you can see in the output that the non-static variable is different for both the objects but the static variable is shared among them, thats the reason the changes made to the static variable by object ob2 reflects in both the objects.

class JavaExample{

//Static integer variable

static int var1=77;

//non-static string variable

String var2;

public static void main(String args[])

{

JavaExample ob1 = new JavaExample();

JavaExample ob2 = new JavaExample();

/\* static variables can be accessed directly without

\* any instances. Just to demonstrate that static variables

\* are shared, I am accessing them using objects so that

\* we can check that the changes made to static variables

\* by one object, reflects when we access them using other

\* objects

\*/

//Assigning the value to static variable using object ob1

ob1.var1=88;

ob1.var2="I'm Object1";

/\* This will overwrite the value of var1 because var1 has a single

\* copy shared among both the objects.

\*/

ob2.var1=99;

ob2.var2="I'm Object2";

System.out.println("ob1 integer:"+ob1.var1);

System.out.println("ob1 String:"+ob1.var2);

System.out.println("ob2 integer:"+ob2.var1);

System.out.println("ob2 STring:"+ob2.var2);

}

}

Output:

ob1 integer:99

ob1 String:I'm Object1

ob2 integer:99

ob2 STring:I'm Object2

## **Java Static Methods**

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.

Static methods can be accessed directly in static and non-static methods.

**Syntax:**

Static keyword followed by return type, followed by method name.

static return\_type method\_name();

### **Example 1: static method main is accessing static variables without object**

class JavaExample{

static int i = 10;

static String s = "Beginnersbook";

//This is a static method

public static void main(String args[])

{

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

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

}

}

**Output:**

i:10

s:Beginnersbook

### **Example 2: Static method accessed directly in static and non-static method**

class JavaExample{

static int i = 100;

static String s = "Beginnersbook";

//Static method

static void display()

{

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

System.out.println("i:"+s);

}

//non-static method

void funcn()

{

//Static method called in non-static method

display();

}

//static method

public static void main(String args[])

{

JavaExample obj = new JavaExample();

//You need to have object to call this non-static method

obj.funcn();

//Static method called in another static method

display();

}

}

Output:

i:100

i:Beginnersbook

i:100

i:Beginnersbook

## **Static Class**

A class can be made **static** only if it is a nested class.

1. Nested static class doesn’t need reference of Outer class
2. A static class cannot access non-static members of the Outer class

We will see these two points with the help of an example:

### **Static class Example**

class JavaExample{

private static String str = "BeginnersBook";

//Static class

static class MyNestedClass{

//non-static method

public void disp() {

/\* If you make the str variable of outer class

\* non-static then you will get compilation error

\* because: a nested static class cannot access non-

\* static members of the outer class.

\*/

System.out.println(str);

}

}

public static void main(String args[])

{

/\* To create instance of nested class we didn't need the outer

\* class instance but for a regular nested class you would need

\* to create an instance of outer class first

\*/

JavaExample.MyNestedClass obj = new JavaExample.MyNestedClass();

obj.disp();

}

}

Output:

BeginnersBook

[Aggregation](https://beginnersbook.com/2013/05/aggregation/)

Aggregation is a special form of association. It is a relationship between two classes like [association](https://beginnersbook.com/2013/05/association/), however its a **directional** association, which means it is strictly a **one way association.** It represents a **HAS-A** relationship.

## **Aggregation Example in Java**

For example consider two classes Student class and Address class. Every student has an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make any sense as an Address doesn’t need to have a Student necessarily. Lets write this example in a java program.

Student Has-A Address

class Address

{

int streetNum;

String city;

String state;

String country;

Address(int street, String c, String st, String coun)

{

this.streetNum=street;

this.city =c;

this.state = st;

this.country = coun;

}

}

class StudentClass

{

int rollNum;

String studentName;

//Creating HAS-A relationship with Address class

Address studentAddr;

StudentClass(int roll, String name, Address addr){

this.rollNum=roll;

this.studentName=name;

this.studentAddr = addr;

}

public static void main(String args[]){

Address ad = new Address(55, "Agra", "UP", "India");

StudentClass obj = new StudentClass(123, "Chaitanya", ad);

System.out.println(obj.rollNum);

System.out.println(obj.studentName);

System.out.println(obj.studentAddr.streetNum);

System.out.println(obj.studentAddr.city);

System.out.println(obj.studentAddr.state);

System.out.println(obj.studentAddr.country);

}

}

Output:

123

Chaitanya

55

Agra

UP

India

The above example shows the **Aggregation** between Student and Address classes. You can see that in Student class I have declared a property of type Address to obtain student address. Its a typical example of Aggregation in Java.

## **Why we need Aggregation?**

**To maintain code re-usability**. To understand this lets take the same example again. Suppose there are two other classes College and Staff along with above two classes Student and Address. In order to maintain Student’s address, College Address and Staff’s address we don’t need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:

Student Has-A Address (Has-a relationship between student and address)

College Has-A Address (Has-a relationship between college and address)

Staff Has-A Address (Has-a relationship between staff and address)

# **Association**

In this article we will discuss **Association in Java**. Association establishes relationship between two separate **classes** through their **objects**. The relationship can be one to one, One to many, many to one and many to many.

### **Association Example**

class CarClass{

String carName;

int carId;

CarClass(String name, int id)

{

this.carName = name;

this.carId = id;

}

}

class Driver extends CarClass{

String driverName;

Driver(String name, String cname, int cid){

super(cname, cid);

this.driverName=name;

}

}

class TransportCompany{

public static void main(String args[])

{

Driver obj = new Driver("Andy", "Ford", 9988);

System.out.println(obj.driverName+" is a driver of car Id: "+obj.carId);

}

}

Output:

Andy is a driver of car Id: 9988

In the above example, there is a one to one relationship(**Association**) between two classes: CarClass and Driver. Both the classes represent two separate entities.

## **Association vs Aggregation vs Composition**

Lets discuss **difference between Association, Aggregation and Composition**:

Although all three are related terms, there are some major differences in the way they relate two classes. **Association** is a relationship between two separate classes and the association can be of any type say one to one, one to may etc. It joins two entirely separate entities.

[Aggregation](https://beginnersbook.com/2013/05/aggregation/) is a special form of association which is a unidirectional one way relationship between classes (or entities), for e.g. Wallet and Money classes. Wallet has Money but money doesn’t need to have Wallet necessarily so its a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if Wallet class is not present, it does not mean that the Money class cannot exist.

**Composition** is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.

# **Super keyword**

Refers to the objects of immediate parent class. Before learning super keyword you must have the knowledge of [inheritance in Java](https://beginnersbook.com/2013/03/inheritance-in-java/) so that you can understand the examples given in this guide.

## **The use of super keyword**

1) To access the data members of parent class when both parent and child class have member with same name

2) To explicitly call the no-arg and parameterized constructor of parent class

3) To access the method of parent class when child class has overridden that method.

### **1) How to use super keyword to access the variables of parent class**

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

Lets take an example to understand this: In the following program, we have a data member num declared in the child class, the member with the same name is already present in the parent class. There is no way you can access the num variable of parent class without using super keyword. .

//Parent class or Superclass or base class

class Superclass

{

int num = 100;

}

//Child class or subclass or derived class

class Subclass extends Superclass

{

/\* The same variable num is declared in the Subclass

\* which is already present in the Superclass

\*/

int num = 110;

void printNumber(){

System.out.println(num);

}

public static void main(String args[]){

Subclass obj= new Subclass();

obj.printNumber();

}

}

Output:

110

**Accessing the num variable of parent class:**

By calling a variable like this, we can access the variable of parent class if both the classes (parent and child) have same variable.

super.variable\_name

Let’s take the same example that we have seen above, this time in print statement we are passing super.num instead of num. As you can see by using super.num we accessed the num variable of parent class.

class Superclass

{

int num = 100;

}

class Subclass extends Superclass

{

int num = 110;

void printNumber(){

/\* Note that instead of writing num we are

\* writing super.num in the print statement

\* this refers to the num variable of Superclass

\*/

System.out.println(super.num);

}

public static void main(String args[]){

Subclass obj= new Subclass();

obj.printNumber();

}

}

Output:

100

### **2) Use of super keyword to invoke constructor of parent class**

When we create the object of sub class, the new keyword invokes the [constructor](https://beginnersbook.com/2013/03/constructors-in-java/) of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

Let’s see an example to understand what I have explained above:

class Parentclass

{

Parentclass(){

System.out.println("Constructor of parent class");

}

}

class Subclass extends Parentclass

{

Subclass(){

/\* Compile implicitly adds super() here as the

\* first statement of this constructor.

\*/

System.out.println("Constructor of child class");

}

Subclass(int num){

/\* Even though it is a parameterized constructor.

\* The compiler still adds the no-arg super() here

\*/

System.out.println("arg constructor of child class");

}

void display(){

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

}

public static void main(String args[]){

/\* Creating object using default constructor. This

\* will invoke child class constructor, which will

\* invoke parent class constructor

\*/

Subclass obj= new Subclass();

//Calling sub class method

obj.display();

/\* Creating second object using arg constructor

\* it will invoke arg constructor of child class which will

\* invoke no-arg constructor of parent class automatically

\*/

Subclass obj2= new Subclass(10);

obj2.display();

}

}

**Output:**

Constructor of parent class

Constructor of child class

Hello!

Constructor of parent class

arg constructor of child class

Hello!

#### **Parameterized super() call to invoke parameterized constructor of parent class**

We can call super() explicitly in the constructor of child class, but it would not make any sense because it would be redundant. It’s like explicitly doing something which would be implicitly done otherwise.

However when we have a constructor in parent class that takes arguments then we can use parameterized super, like super(100); to invoke [parameterized constructor](https://beginnersbook.com/2014/01/parameterized-constructor-in-java-example/) of parent class from the constructor of child class.

Let’s see an example to understand this:

class Parentclass

{

//no-arg constructor

Parentclass(){

System.out.println("no-arg constructor of parent class");

}

//arg or parameterized constructor

Parentclass(String str){

System.out.println("parameterized constructor of parent class");

}

}

class Subclass extends Parentclass

{

Subclass(){

/\* super() must be added to the first statement of constructor

\* otherwise you will get a compilation error. Another important

\* point to note is that when we explicitly use super in constructor

\* the compiler doesn't invoke the parent constructor automatically.

\*/

super("Hahaha");

System.out.println("Constructor of child class");

}

void display(){

System.out.println("Hello");

}

public static void main(String args[]){

Subclass obj= new Subclass();

obj.display();

}

}

**Output:**

parameterized constructor of parent class

Constructor of child class

Hello

There are few important points to note in this example:

1) super()(or parameterized super must be the first statement in constructor otherwise you will get the compilation error: “Constructor call must be the first statement in a constructor”

2) When we explicitly placed super in the constructor, the java compiler didn’t call the default no-arg constructor of parent class.

**Method Overloading**

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.

## **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)

### **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

## **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.

# **Method overriding**

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

## **Method Overriding Example**

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

class Human{

//Overridden method

public void eat()

{

System.out.println("Human is eating");

}

}

class Boy extends Human{

//Overriding method

public void eat(){

System.out.println("Boy is eating");

}

public static void main( String args[]) {

Boy obj = new Boy();

//This will call the child class version of eat()

obj.eat();

}

}

Output:

Boy is eating

## **Advantage of method overriding**

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

## **Rules of method overriding in Java**

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.

[Access Modifier](https://beginnersbook.com/2013/05/java-access-modifiers/) of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method ) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.  
 For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)  
 class MyBaseClass{

public void disp()

{

System.out.println("Parent class method");

}

}

class MyChildClass extends MyBaseClass{

protected void disp(){

System.out.println("Child class method");

}

public static void main( String args[]) {

MyChildClass obj = new MyChildClass();

obj.disp();

}

}  
  
 Output:  
  
 Exception in thread "main" java.lang.Error: Unresolved compilation

problem: Cannot reduce the visibility of the inherited method from MyBaseClass  
  
 However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.  
  
 class MyBaseClass{

protected void disp()

{

System.out.println("Parent class method");

}

}

class MyChildClass extends MyBaseClass{

public void disp(){

System.out.println("Child class method");

}

public static void main( String args[]) {

MyChildClass obj = new MyChildClass();

obj.disp();

}

1. }  
     
    Output:  
     
    Child class method
2. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
3. Overriding method (method of child class) can throw [unchecked exceptions](https://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/), regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw [checked exceptions](https://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/) that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
4. Binding of overridden methods happen at runtime which is known as [dynamic binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/).
5. If a class is extending an [abstract class](https://beginnersbook.com/2013/05/java-abstract-class-method/) or implementing an [interface](https://beginnersbook.com/2013/05/java-interface/) then it has to override all the abstract methods unless the class itself is a abstract class.

## **Super keyword in Method Overriding**

The [super keyword](https://beginnersbook.com/2014/07/super-keyword-in-java-with-example/) is used for calling the parent class method/constructor. super.myMethod() calls the myMethod() method of base class while super() calls the [constructor](https://beginnersbook.com/2013/03/constructors-in-java/) of base class. Let’s see the use of super in method Overriding.

As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

class ABC{

public void myMethod()

{

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

}

}

class Demo extends ABC{

public void myMethod(){

//This will call the myMethod() of parent class

super.myMethod();

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

}

public static void main( String args[]) {

Demo obj = new Demo();

obj.myMethod();

}

}

Output:

Class ABC: mymethod()

Class Test: mymethod()

As you see using super keyword, we can access the overriden method.

# **Difference between method Overloading and Overriding**

## **Overloading vs Overriding in Java**

1. Overloading happens at [compile-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) while Overriding happens at [runtime](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/): The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overloaded methods and [dynamic binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type ([refer this](https://stackoverflow.com/questions/14694852/can-overridden-methods-differ-in-return-type)).
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

### **Overloading example**

Here we have 4 versions of same method add. We are overloading the method add() here.

//A class for adding upto 5 numbers

class Sum

{

int add(int n1, int n2)

{

return n1+n2;

}

int add(int n1, int n2, int n3)

{

return n1+n2+n3;

}

int add(int n1, int n2, int n3, int n4)

{

return n1+n2+n3+n4;

}

int add(int n1, int n2, int n3, int n4, int n5)

{

return n1+n2+n3+n4+n5;

}

public static void main(String args[])

{

Sum obj = new Sum();

System.out.println("Sum of two numbers: "+obj.add(20, 21));

System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));

System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));

System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));

}

}

Output:

Sum of two numbers: 41

Sum of three numbers: 63

Sum of four numbers: 86

Sum of five numbers: 110

### **Overriding example**

Here speedLimit() method of class Ford is overriding the speedLimit() method of class CarClass.

package beginnersbook.com;

class CarClass

{

public int speedLimit()

{

return 100;

}

}

class Ford extends CarClass

{

public int speedLimit()

{

return 150;

}

public static void main(String args[])

{

CarClass obj = new Ford();

int num= obj.speedLimit();

System.out.println("Speed Limit is: "+num);

}

}

Output:

Speed Limit is: 150

# **Static and dynamic binding in java**

Association of method call to the method body is known as binding. There are two types of binding: **Static Binding** that happens at compile time and **Dynamic Binding** that happens at runtime

### **Static Binding or Early Binding**

The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is [compile-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/). **Why?** The reason is that the these method cannot be overridden and the type of the class is determined at the compile time. Lets see an example to understand this:

### **Static binding example**

Here we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overriden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there.

class Human{

public static void walk()

{

System.out.println("Human walks");

}

}

class Boy extends Human{

public static void walk(){

System.out.println("Boy walks");

}

public static void main( String args[]) {

/\* Reference is of Human type and object is

\* Boy type

\*/

Human obj = new Boy();

/\* Reference is of HUman type and object is

\* of Human type.

\*/

Human obj2 = new Human();

obj.walk();

obj2.walk();

}

}

Output:

Human walks

Human walks

### **Dynamic Binding or Late Binding**

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. [Method Overriding](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

### **Dynamic binding example**

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens. Lets see an example to understand this:

class Human{

//Overridden Method

public void walk()

{

System.out.println("Human walks");

}

}

class Demo extends Human{

//Overriding Method

public void walk(){

System.out.println("Boy walks");

}

public static void main( String args[]) {

/\* Reference is of Human type and object is

\* Boy type

\*/

Human obj = new Demo();

/\* Reference is of HUman type and object is

\* of Human type.

\*/

Human obj2 = new Human();

obj.walk();

obj2.walk();

}

}

Output:

Boy walks

Human walks

As you can see that the output is different than what we saw in the static binding example, because in this case while creation of object obj the type of the object is determined as a Boy type so method of Boy class is called. Remember the type of the object is determined at the runtime.

## **Static Binding vs Dynamic Binding**

Lets discuss the **difference between static and dynamic binding in Java**.

1. Static binding happens at compile-time while dynamic binding happens at runtime.
2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
3. The binding of [overloaded methods](https://beginnersbook.com/2013/05/method-overloading/) is static and the binding of overridden methods is dynamic

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **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). A normal class(non-abstract class) cannot have abstract methods. In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.

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. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method( otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

## **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();

}

}

Output:

Woof

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

## **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.

**Key Points:**

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can’t have abstract method in a concrete class. It’s vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

For now lets just see some basics and example of abstract method.

1) Abstract method has no body.

2) Always end the declaration with a **semicolon**(;).

3) It must be [overridden](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/). An abstract class must be extended and in a same way abstract method must be overridden.

4) A class has to be declared abstract to have abstract methods.

Abstract method is just a signature without any implementation block inside.Abstract method must be overridden in the subclasses to make use for the object to invoke.

An Abstract Method is just a prototype for the method with the following attributes:-

1) A return type

2) A name

3) A list of Parameters

4) A throws clause which is optional

Example:- public abstract int salary(int empNo)

# **Interface**

In the last tutorial we discussed [abstract class](https://beginnersbook.com/2013/05/java-abstract-class-method/) which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user(See: [Abstraction](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/)). In this guide, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in [Java Programming](https://beginnersbook.com/java-tutorial-for-beginners-with-examples/).

## **What is an interface in Java?**

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. We will cover this in detail, later in this guide

## **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.

**Syntax:**

Interfaces are declared by specifying a keyword “interface”. E.g.:

interface MyInterface

{

/\* All the methods are public abstract by default

\* As you see they have no body

\*/

public void method1();

public void method2();

}

## **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();

}

}

Output:

implementation of method1

**Key points:** Here are the key points to remember about interfaces:

1) We can’t instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) implements keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

interface Try

{

int a=10;

public int a=10;

public static final int a=10;

final int a=10;

static int a=0;

}

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

interface Try

{

int x;//Compile-time error

}

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

class Sample implements Try

{

public static void main(String args[])

{

x=20; //compile time error

}

}

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

interface A

{

public void aaa();

}

interface B

{

public void aaa();

}

class Central implements A,B

{

public void aaa()

{

//Any Code here

}

public static void main(String args[])

{

//Statements

}

}

14) A class cannot implement two interfaces that have methods with same name but different return type.

interface A

{

public void aaa();

}

interface B

{

public int aaa();

}

class Central implements A,B

{

public void aaa() // error

{

}

public int aaa() // error

{

}

public static void main(String args[])

{

}

}

15) Variable names conflicts can be resolved by interface name.

interface A

{

int x=10;

}

interface B

{

int x=100;

}

class Hello implements A,B

{

public static void Main(String args[])

{

/\* reference to x is ambiguous both variables are x

\* so we are using interface name to resolve the

\* variable

\*/

System.out.println(x);

System.out.println(A.x);

System.out.println(B.x);

}

}

# **Difference Between Abstract Class and Interface in Java**

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| **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** |
| **An abstract class can extend another concrete (regular) class or abstract class** | **An interface can only extend another interface** |
| **An abstract class can have both abstract and concrete methods** | **An interface can have only abstract methods** |
| **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** |
| **An abstract class can have protected and public abstract methods** | **An interface can have only have public abstract methods** |
| **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** |

# **Java Access Modifiers – Public, Private, Protected & Default**

You must have seen public, private and protected keywords while practising java programs, these are called access modifiers. An access modifier restricts the access of a class, constructor, data member and method in another class. In java we have four access modifiers:

1. default

2. private

3. protected

4. public

## **1. Default access modifier**

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

## **2. Private access modifier**

The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and [Interface](https://beginnersbook.com/2013/05/java-interface/) cannot be declared as private
3. If a class has [private constructor](https://beginnersbook.com/2013/12/java-private-constructor-example/) then you cannot create the object of that class from outside of the class.

Let’s see an example to understand this:

### **Private access modifier example in java**

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

## **3. Protected Access Modifier**

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.

Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

### **Protected access modifier example in Java**

In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

## **4. Public access modifier**

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

### **public access modifier example in java**

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Addition.java

## **The scope of access modifiers in tabular form**

------------+-------+---------+--------------+--------------+--------

| Class | Package | Subclass | Subclass |Outside|

| | |(same package)|(diff package)|Class |

————————————+———————+—————————+——————————----+—————————----—+————————

public | Yes | Yes | Yes | Yes | Yes |

————————————+———————+—————————+—————————----—+—————————----—+————————

protected | Yes | Yes | Yes | Yes | No |

————————————+———————+—————————+————————----——+————————----——+————————

default | Yes | Yes | Yes | No | No |

————————————+———————+—————————+————————----——+————————----——+————————

private | Yes | No | No | No | No |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Class | Package | Subclass | Subclass | Outside Class |
| public |  |  |  |  |  |
| protected |  |  |  |  |  |
| default |  |  |  |  |  |
| private |  |  |  |  |  |

Random programming stuff

Lambda Expression: A Anonymous Expression, functions that do not belong to a class. Ashort 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.

Python

def f(x):

return 3+x

f(2)

Output: 5

g = lambda x: 3+x

g(2)

Output:5

Java

//This is the normal way we write a method in Java

aBlockOfCode{

…

...

}

//What is we want to assign that method to a variable

//When you code a Java expression you do not need things like the access type, return type, or the name of the method

//For multiple lines of code

aBlockOfCode =() -> {

System.out.println(“Hello World1”);

System.out.println(“Hello World2”);

}

//For single lines of code

aBlockOfCode =() -> System.out.println(“Hello World1”);

