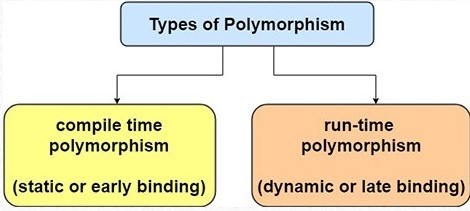
<https://www.sitepoint.com/quick-guide-to-polymorphism-in-java/>

**Quick Guide to Polymorphism in Java**

What is Polymorphism?

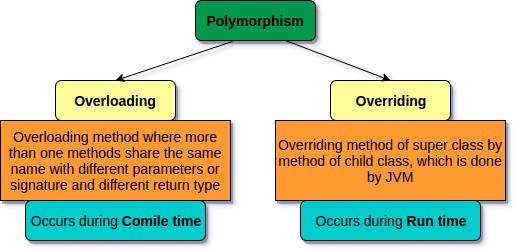
Polymorphism is the ability of an object to take more than one forms. It is one of the important concept of object-oriented programming language.**[JAVA](https://www.learntek.org/advanced-java-training/)** is object-oriented programming language which support the concept of polymorphisms.

Alternatively, it is defined as the ability of a reference variable to change behavior according to what object instance it is holding.



There are two types of polymorphism in Java:

1) Compile-time polymorphism (static binding)  
2) Runtime polymorphism (dynamic binding)

Method overloading is an example of compile time polymorphism, while method overriding is an example of runtime polymorphism.  


**1. Compile time Polymorphism:**

The type of polymorphism that is implemented when the compiler compiles a program is called compile-time polymorphism. This type of polymorphism is also called as static polymorphism or early binding.

**Method Overloading:**

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading. Method overloading is performed within class.

If we must perform only one operation, having same name of the methods increases the readability of the program.

There are three ways to overload the method in java  
• By changing number of arguments  
• By changing the data type  
• By changing both number of arguments and data type

**a. Method overloading by changing number of argument:**

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as x(int,int) for two parameters, and y(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

**Polymorphism in [Java](https://www.learntek.org/core-java-training/) – Example1:**  
In the following program, two methods are created, first add1() performs addition of two numbers and second add1() performs addition of three numbers.

In following example, we are creating static methods so that we don’t need to create instance for calling methods.

==============================================================

The word ‘polymorphism’ literally means ‘a state of having many shapes’ or ‘the capacity to take on different forms’. When applied to object oriented programming languages like Java, it describes a language’s ability to process objects of various types and classes through a single, uniform interface.

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

An important example of polymorphism is how a parent class refers to a child class object.  In fact, any object that satisfies more than one IS-A relationship is polymorphic in nature.

For instance, let’s consider a class Animal and let Cat be a subclass of Animal. So, any cat **IS** animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

**Note:** It’s also legal to say every object in Java is polymorphic in nature, as each one passes an IS-A test for itself and also for Object class.

**Static Polymorphism:**

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures.  So, this is called **compile time polymorphism** or **static binding**. The concept will be clear from the following example: METHOD OVERLOADING

class DemoOverload{

public int add(int x, int y){ //method 1

return x+y;

}

public int add(int x, int y, int z){ //method 2

return x+y+z; }

public int add(double x, int y){ //method 3

return (int)x+y;

}

public int add(int x, double y){ //method 4

return x+(int)y;

}

}

class Test{

public static void main(String[] args){

DemoOverload demo=new DemoOverload();

System.out.println(demo.add(2,3)); //method 1 called

System.out.println(demo.add(2,3,4)); //method 2 called

System.out.println(demo.add(2,3.4)); //method 4 called

System.out.println(demo.add(2.5,3)); //method 3 called

}

}

In the above example, there are four versions of add methods. The first method takes two parameters while the second one takes three. For the third and fourth methods there is a change of order of parameters.  The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

**Dynamic Polymorphism:**

Suppose a sub class overrides a particular method of the super class. Let’s say, in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called.

Have a look at the following example.

class Vehicle{

public void move(){

System.out.println(“Vehicles can move!!”);

}

}

class MotorBike extends Vehicle{

public void move(){

System.out.println(“MotorBike can move and accelerate too!!”);

}}

class Test{

public static void main(String[] args){

//parent class reference holds the child class object

Vehicle vh=new MotorBike();

vh.move(); // prints MotorBike can move and accelerate too!!

vh=new Vehicle();

vh.move(); // prints Vehicles can move!! }

}

It should be noted that in the first call to move(), the reference type is Vehicle and the object being referenced is MotorBike. So, when a call to move() is made, Java waits until runtime to determine which object is actually being pointed to by the reference.  In this case, the object is of the class MotorBike. So, the move() method of MotorBike class will be called. In the second call to move(), the object is of the class Vehicle. So, the move() method of Vehicle will be called.

As the method to call is determined at runtime, this is called **dynamic binding** or **late binding**.

**Summary:**

*An object in Java that passes more than one IS-A tests is polymorphic in nature*

*Every object in Java passes a minimum of two IS-A tests: one for itself and one for Object class*

*Static/compile-time polymorphism in Java is achieved by method overloading*

*Dynamic/runtime polymorphism in Java is achieved by method overriding*

ABSTRACTION <https://beginnersbook.com/2013/05/java-abstract-class-method/>

# Abstract Class in Java with example

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

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 **cannot** be **instantiated**, which means you are not allowed to create an **object** of it. Why? We will discuss that later in this guide.

## 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.

**Do you know?** Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

[Interfaces](https://beginnersbook.com/2013/05/java-interface/" \o "Interface in java with example programs) on the other hand are used for 100% abstraction (See more about [abstraction](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/) here).  
You may also want to read this: [Difference between abstract class and Interface in Java](https://beginnersbook.com/2013/05/abstract-class-vs-interface-in-java/)

### 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.

### Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

abstract class AbstractDemo{

public void myMethod(){

System.out.println("Hello");

}

abstract public void anotherMethod();

}

public class Demo extends AbstractDemo{

public void anotherMethod() {

System.out.print("Abstract method");

}

public static void main(String args[])

{

//error: You can't create object of it

AbstractDemo obj = new AbstractDemo();

obj.anotherMethod();

}

}

Output:

Unresolved compilation problem: Cannot instantiate the type AbstractDemo

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

## Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, Animal is a abstract class and Cat, Dog & Lion are concrete classes.

**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.

I have covered the rules and examples of abstract methods in a separate tutorial, You can find the guide here: [Abstract method in Java](https://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/)  
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/" \t "_blank). 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.

**Note:** The class which is extending abstract class must override all the abstract methods.

## Example of Abstract class and method

abstract class MyClass{

public void disp(){

System.out.println("Concrete method of parent class");

}

abstract public void disp2();

}

class Demo extends MyClass{

/\* Must Override this method while extending

\* MyClas

\*/

public void disp2()

{

System.out.println("overriding abstract method");

}

public static void main(String args[]){

Demo obj = new Demo();

obj.disp2();

}

}

Output:

overriding abstract method