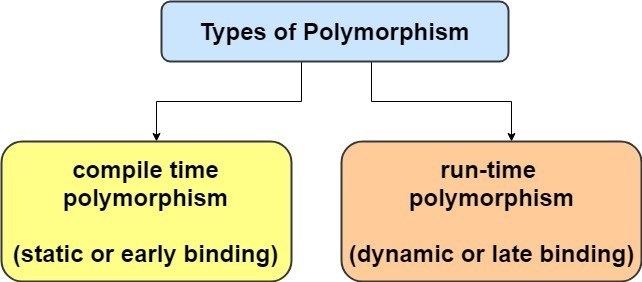
Polymorphism

Polymorphism is derived from 2 greek words: poly and morphs. The word “poly” means many and “morphs” means forms. So Polymorphism means the ability to take many forms. is a concept by which we can perform a ***single task****in****different ways***. It is one of the most striking features of Object Oriented Programming in Java.

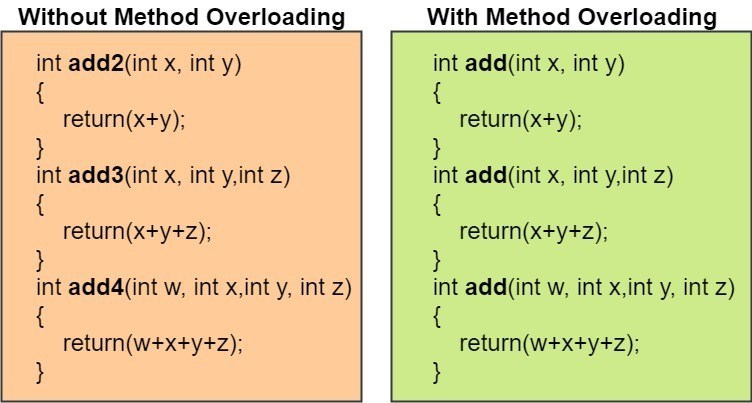
In terms of Java Programming Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

There are two types of polymorphism in java:**compile time polymorphism (static or early binding)** and **runtime polymorphism(dynamic or late binding)**.



1. Compile Time Polymorphism – [Method Overloading](https://simplesnippets.tech/java-method-overloading-in-detail-with-examples/)
2. Run Time Polymorophism – Method Overriding

## Method Overloading in Java with Examples



Method overloading in Java is a feature which makes it possible to use the**same method** name to perform **different tasks**. In this tutorial post we will understand the concept of **Java Method Overloading**.

If a class has **multiple** methods having **same name** but **different** in **parameters**, it is known as **Method Overloading**. Overloading allows different methods to have same name, but**different signatures** where signature can differ by**number of input parameters** or **type of input parameters** or **order of input parameters**. Overloading is related to **compile time (or static) polymorphism**(we will understand polymorphism in other tutorial).

Lets, take an example situation to understand –

Suppose you have a program to perform addition of numbers and you create a method to add 2 numbers as – add(int x, int y). Now you have a new requirement to add 3 numbers so you again create a new method as – add3(int x, int y, int z). But now again you have one more new requirement to add 4 numbers. You see the hassle here right ? every time you get this requirement you have to find a new method name. But the underlying logical operation is the same right ? cause in the end we are anyways going to perform addition. So why use different method names. So that is where method overloading comes to the rescue.

Method Overloading in Java

#### Advantage of method overloading

Method overloading increases the readability of the program.

#### Different ways to overload the method

There are 3 ways to overload the method in java

* By changing **number of arguments**
* By changing the **data type**
* By changing the **order** of arguments (if different datatypes are involved)

public class MethodOverloading {

    //1) 2 arguments int & double

    double add(int x, double y)

    {

        return(x+y);

    }

    //2) 2 arguments but different order - double & int

    double add(double x, int y)

    {

        return(x+y);

    }

    //3) 3 arguments

    double add(double x, int y, float z)

    {

        return(x+y+z);

    }

    public static void main(String[] args) {

        // TODO code application logic here

        int a=5;

        double b = 7.5;

        float c = 4.5f;

        double result;

        MethodOverloading obj = new MethodOverloading();

        result = obj.add(b, a); // 1st add function

        System.out.println("Addtion is: "+result);

        result = obj.add(a, b); // 2nd add function

        System.out.println("Addtion is: "+result);

        result = obj.add(b,a,c); // 3rd add function

        System.out.println("Addtion is: "+result);

    }

}

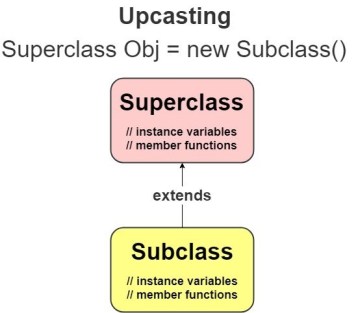
**Output**

Addtion is: 12.5  
Addtion is: 12.5  
Addtion is: 17.0

## Run Time Polymorphism

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a **superclass**. Thus this happens only when [Inheritance](https://simplesnippets.tech/inheritance-in-java-types-of-inheritance/) is implemented. The method to be called is based on the object being referred to by the reference variable.

* When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
* At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
* A superclass reference variable can refer to a subclass object. This is also known as **upcasting**. Java uses this fact to resolve calls to overridden methods at run time.



Lets see a program example to understand the working of run-time polymorphism –

// A Simple Java program to demonstrate

// method overriding in java

// Base Class

class Parent

{

void show() { System.out.println("Parent's show()"); }

}

// Inherited class

class Child extends Parent

{

// This method overrides show() of Parent

@Override

void show() { System.out.println("Child's show()"); }

}

// Driver class

class Main

{

public static void main(String[] args)

{

// If a Parent type reference refers

// to a Parent object, then Parent's

// show is called

Parent obj1 = new Parent();

obj1.show();

// If a Parent type reference refers

// to a Child object Child's show()

// is called. This is called RUN TIME

// POLYMORPHISM.

Parent obj2 = new Child();

obj2.show();

}

}

Output

Parent’s show()  
Child’s show()

##### **Some Rules to follow in Method Overriding –**

* **Overriding and Access-Modifiers :** The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass.
* **Final methods can not be overridden.**
* **Static methods can not be overridden(Method Overriding vs Method Hiding).**
* **Private methods can not be overridden.**
* **The overriding method must have same return type (or subtype)**
* **Invoking overridden method from sub-class :**We can call parent class method in overriding method using **super** keyword.
* **Overriding and constructor :** We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
* **Overriding and abstract method :**Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise compile-time error will be thrown.

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |