**Polymorphism in Java**

We can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism allows you to define one interface and have multiple implementations. The method called through a polymorphic reference can change from one invocation to the next. All object references in Java are potentially Polymorphic.

**Dynamic Polymorphism:**

The polymorphism exhibited at runtime is called dynamic polymorphism. This means when a method is called, the method call is bound to the method body at the time of running the program, dynamically. This is also called as ‘runtime polymorphism’ or ‘dynamic polymorphism’.

E.g. we are creating two instance methods having same name as follows,

Class Sample {

void add(int a, int b)

{ System.out.println("Sum of two="+(a+b)); }

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

{ System.out.println("Sum of three="+(a+b+c)); } }

In this example to call these methods we use same method name like,

s.add(10,20); //s object of Sample class

**Dynamic Method Overridden:**

Writing two or more methods in super and sub classes such that the methods have same name and same signature – is called Method Overriding. It is a process in which a function call to the overridden method is resolved at Runtime.

E.g. class One{

Void calc(double a) {

System.out.Println(“Square of a: “ + (a \* a) }

class Two extends One {

Void calc(double a) {

System.out.Println(“Square root of a: “ + (Math.sqrt(a)) }

class poly{ public static void main(String args[]) {

Two t=new Two();

t.calc(25); } }

**Dynamic Method Overloading:**

Writing two or methods in the same class in such a way that each method has same name but with different method signatures – is called method overloading.

E.g: class MethOver{

void opr(String s1, String s2) {

System.out.Println(“Combine string: “ + (s1 + s2) }

Void opr(int a, int b) {

System.out.Println(“Sum of 2 no’s: “ + (a + b) }

class Main { public static void main(String[] args)

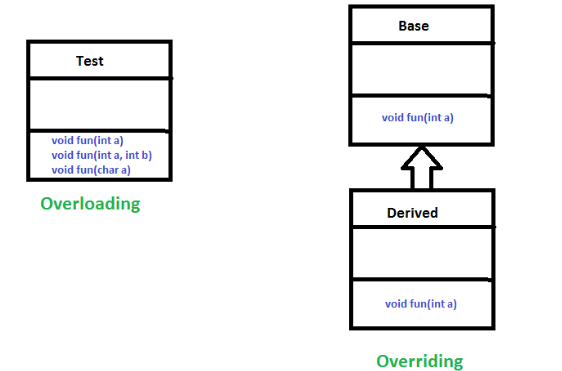
    {   MethOver obj = new MethOver ();

        obj.opr(2, 3);

        obj.opr("my", "name");     } }

**Difference between Method Overloading and Overriding:**

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| **→**Writing two or more methods with the same name but with different signatures is called method overloading. | **→**Writing two or more methods with the same name and same signature is called method overriding. |
| **→**Method overloading is done in the same class. | **→**Method Overriding is done in super and sub classes. |
| **→**Method overloading is done when the programmer wants to extend the already available feature. | **→**Method overriding is done when the programmer wants to provide a different implementation (body) for the same feature. |
| **→**Method overloading is code refinement. Same method is refined to perform a different task. | **→**Method overriding is code replacement. The sub class method overrides (replaces) the super class method. |
| **→**Method Overloading gives better performance compared to overriding, reason is binding of overridden methods is being done at runtime. | **→** Method overriding performance is slow than method overloading |
| **→**Return type of method does not matter in case of method overloading it can be same or different. | **→**Method overriding the overriding method can have more specific return type. |
| **→**Argument list should be different while doing method overloading. | **→**Argument list should be same in method Overriding. |



**Static Polymorphism:**

The Polymorphism exhibited at compilation time is called static polymorphism. The compiler knows and can bind the method call with method code (body) at the time of compilation. So, it is also called ‘static’ or ‘compile’ time polymorphism. In Static polymorphism method overloading by using static methods, private methods, and final methods.

**Polymorphism with Static Methods:**

A static method is a method whose single copy in memory is shared by all the objects of the class. Static methods belong to the class rather than to the objects. So they are also called class methods.

E.g: class One {

static void calculate(double x) {

System.out.println("Square value="+(x\*x)); } }

class Two extends One

{ static void calculate(double x) {

System.out.println("Square root="+Math.sqrt(x)); } }

public class Helloword {

public static void main(String[] args) {

One o=new Two();

o.calculate(25);

Two t=new Two();

t.calculate(25); } }

here,

* ‘o’ is super class reference variable, the super class method is called. we created super class reference to sub class object, as: One o=new Two();
* ‘t’ is sub class reference variable,  t.caclualte(25) call the sub class calculate() and executed.

when static methods are overridden, the JVM decides which method is to be depending on the reference type used to call the method.

**Polymorphism with Private Methods:**

Private methods are declared by using the access specifier ‘private’. This access specifier makes the method not to be available outside the class. There is no possibility to override the private methods of the super class in its sub classes. So only method overloading is possible in case of private methods.

**Polymorphism with Final Methods:**

Methods which are declared as ‘final’ are called final methods. Final methods can’t be overridden, because they are not available to the sub classes. Therefore, only method overloading is possible with final methods.

The advantage of final method,

1. When a method is declared as final, the performance will be better.
2. When the programmer does not want others to override his method, he should declare his method as ‘final’.

E.g. class A {

final void method1() {

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

class B extends A {

void method2() {

method1(); //call the final method } }

Note: Neither the private methods nor the final variables cannot be modified at runtime so it cannot be overridden.

**Final class:** Final keyword before a class prevents inheritance. This means sub classes cannot be created to a final class.

**final class A**

**class B extends A  //invalid**

**Static and Dynamic binding Java:**

Association of method call to the method body is known as binding. There are 2 types of binding,

1. Static Binding (also known as Early Binding) – type of object determined at compile time
2. Dynamic Binding (also known as Late Binding) – type of object determined at run time

To understand these binding we need to understand the type of instances.

**Variable:** Each variable has a type, it may be primitive and non-primitive.

E.g. int data **=** 30; here data variable is a type int.

**Reference:** A reference is an address that indicates where an object's variables and methods are stored.

E.g class Dog{   public static void main(String args[]){

  Dog d1;//Here d1 is a type of Dog   }  }

**Object:** An object is an instance of particular java class, but it is also an instance of its superclass.

class Animal{}

class Dog extends Animal{

 public static void main(String args[]){

  Dog d1=new Dog();  }   }

Here d1 is an instance of Dog class, but it is also an instance of Animal.

**Static 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 cannot be overridden and the type of the class is determined at the compile time.

**E.g.** class Human {

Public static void walk() {

System.out.println(“Its human”); } }

Class Boy extends Human {

public static void walk() {

System.out.println(“Its boy”); }

Public static void main(String args[]){

Human obj = new Boy();

Human obj1 = new Human();

Obj.walk();

Obj.walk(); } }

Output:

Its human

Its human

**Dynamic 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.

**E.g.** class Human {

Public void walk() {

System.out.println(“Its human”); } }

Class Boy extends Human {

public void walk() {

System.out.println(“Its boy”); }

Public static void main(String args[]){

Human obj = new Boy();

Human obj1 = new Human();

Obj.walk();

Obj.walk(); } }

Output:

Its boy

Its human

In this example the only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. 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.

**Difference between static and dynamic binding in Java.**

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| **→**Static binding happens at compile-time. | **→**dynamic binding happens at runtime. |
| **→**Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. | →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. |
| **→**Method overloading is Static | **→**Method overriding is dynamic |