OOPs with Java

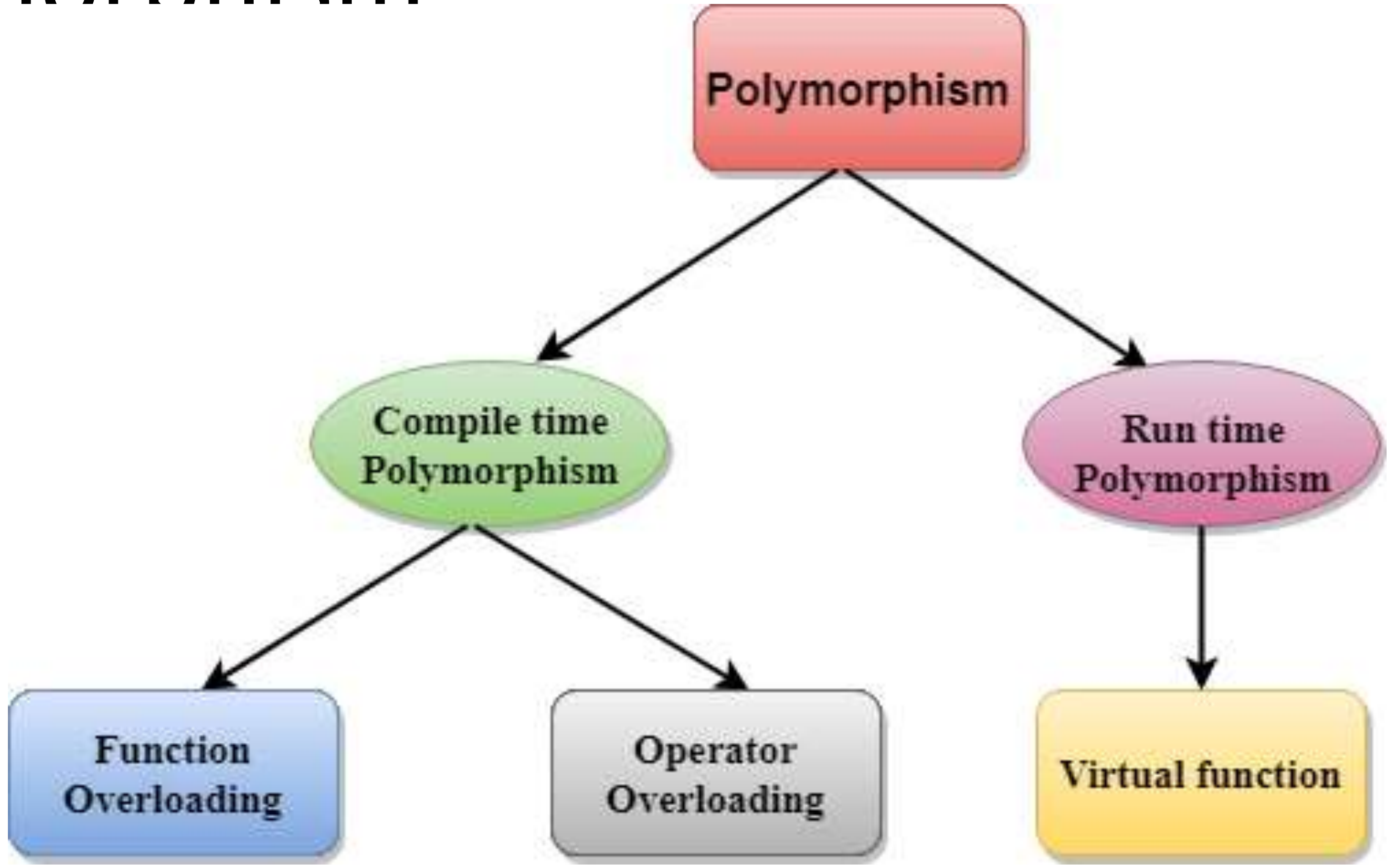
Method Overloading & Constructor

References:

- Java: The Complete Reference, Book & other online resources

Contents

Polymorphism

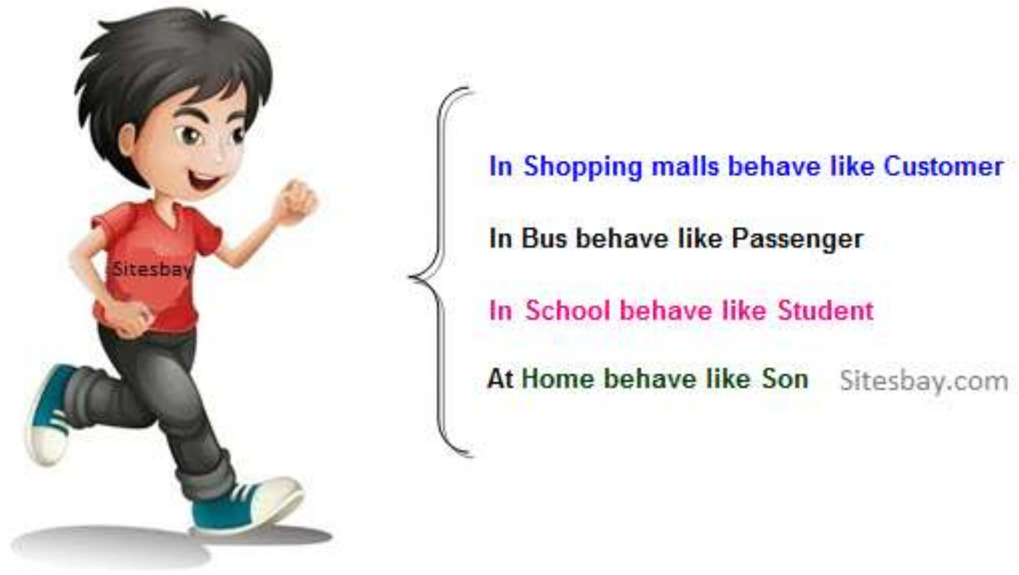


Compile Time Polymorphism: Whenever an object is bound with its functionality at the compile time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading. Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology and which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time. The following is an example where compile-time polymorphism can be observed.

Run-Time Polymorphism: Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. Java virtual machine determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says the child class has the same method as declared in the parent class. It means if the child class provides the specific implementation of the method that has been provided by one of its parent classes then it is known as method overriding. The following is an example where runtime polymorphism can be observed.

Compile time Polymorphism	**Run time Polymorphism**
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator**overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

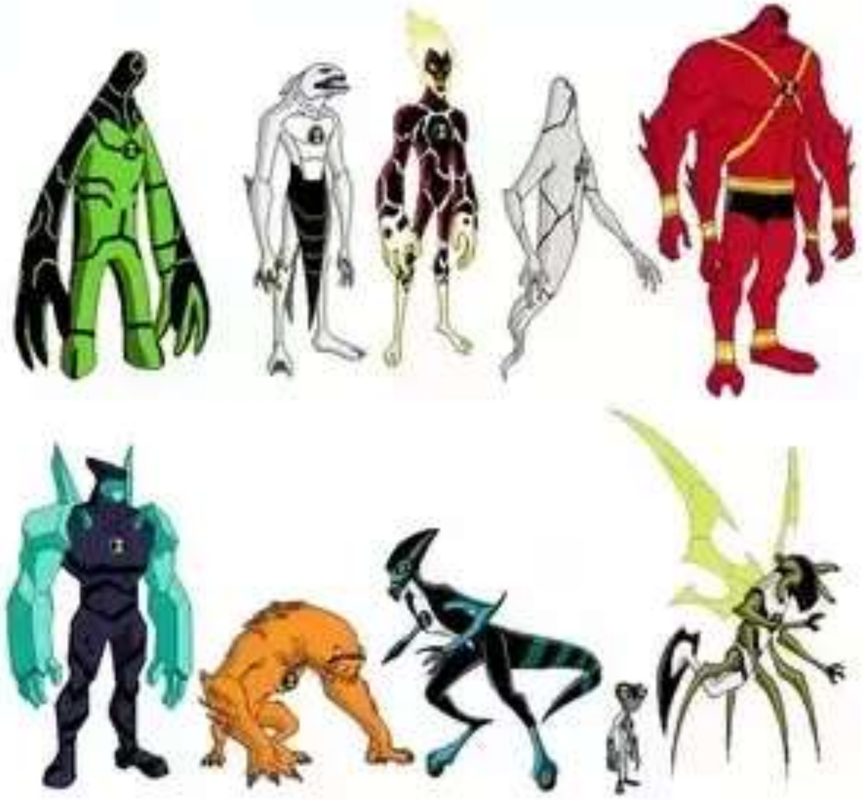
Real Time Examples


Sitesbay
In Shopping malls behave like Customer
In Bus behave like Passenger
In School behave like Student
At Home behave like Son
Sitesbay.com

Hello Rupesh,
We have cabs with facilities like AC, Non-AC
Which one do you like?
TAXI
TAXI
TAXI
TAXI

Transform Form of Ben ten

Ben ten



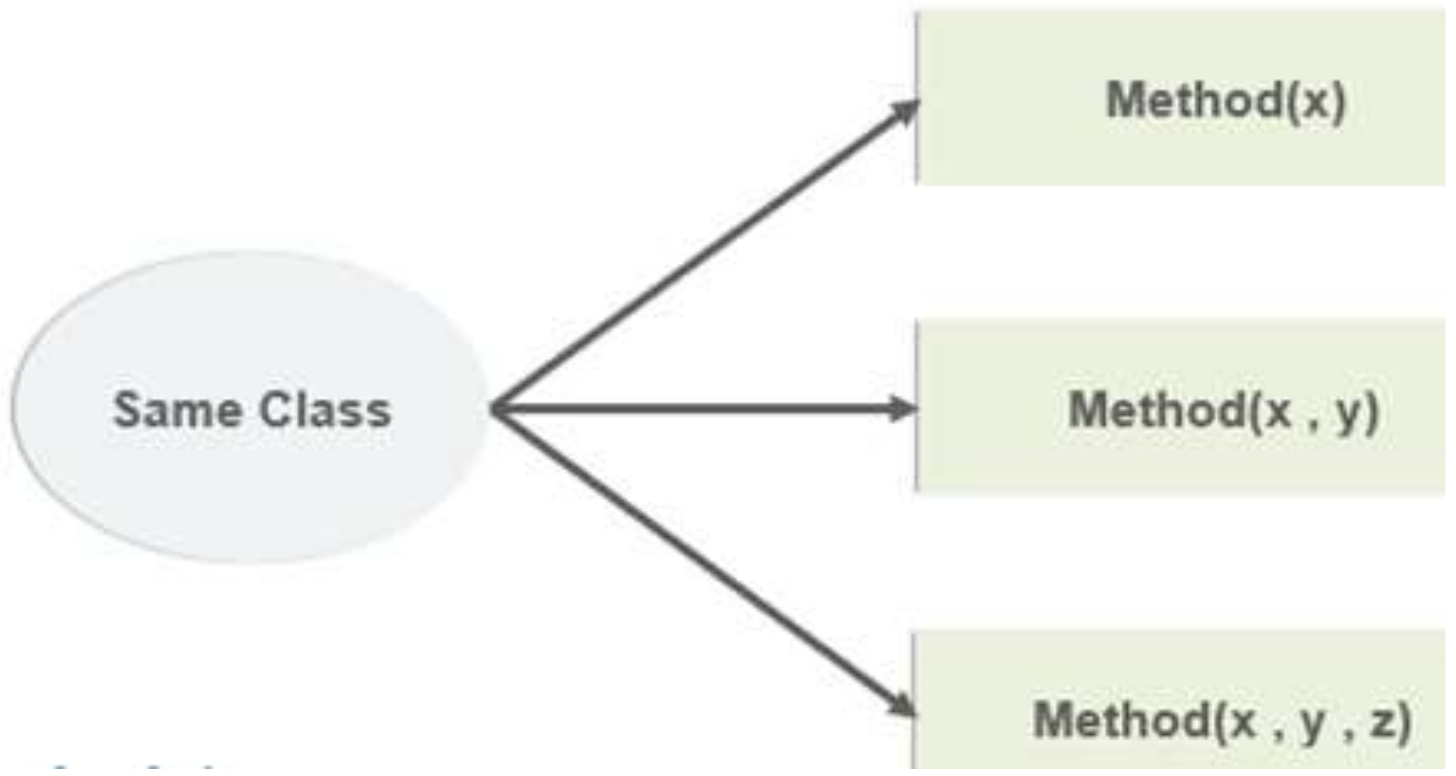
Overloading Methods

- Overload: To give excessive work, responsibility, or information
- Method: A way to imply a manner in which a thing is done or in which it happens

Overloading Methods

- To define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- This process is referred to as method overloading
- It is one of the ways that Java supports polymorphism

- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.


Same Class
Method(x)
Method(x , y)
Method(x , y , z)

Program 1

```
public class Div{

                public int div(int a , int b){
                                return (a / b); }

                public int div(int a , int b , int c){
                                 return ((a + b ) / c); }

public static void main(String args[]){
                Div ob = new Div();
                ob.div(10 , 2);
                ob.div(10, 2 , 3);
}
}
```

Output: 5 4

Program 2

```
public class Add{
public int add(int a , int b){
        return (a + b);
}
public int add(int a , int b , int c){
        return (a + b + c) ;
}
public double add(double a , double b){
        return (a + b);
}
public static void main( String args[]){
Add ob = new Add();
ob.add(15,25);
ob.add(15,25,35);
ob.add(11.5 , 22.5);
}
}
```

Output:
```
40
75
34
```

Program 3

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}

// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of
ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

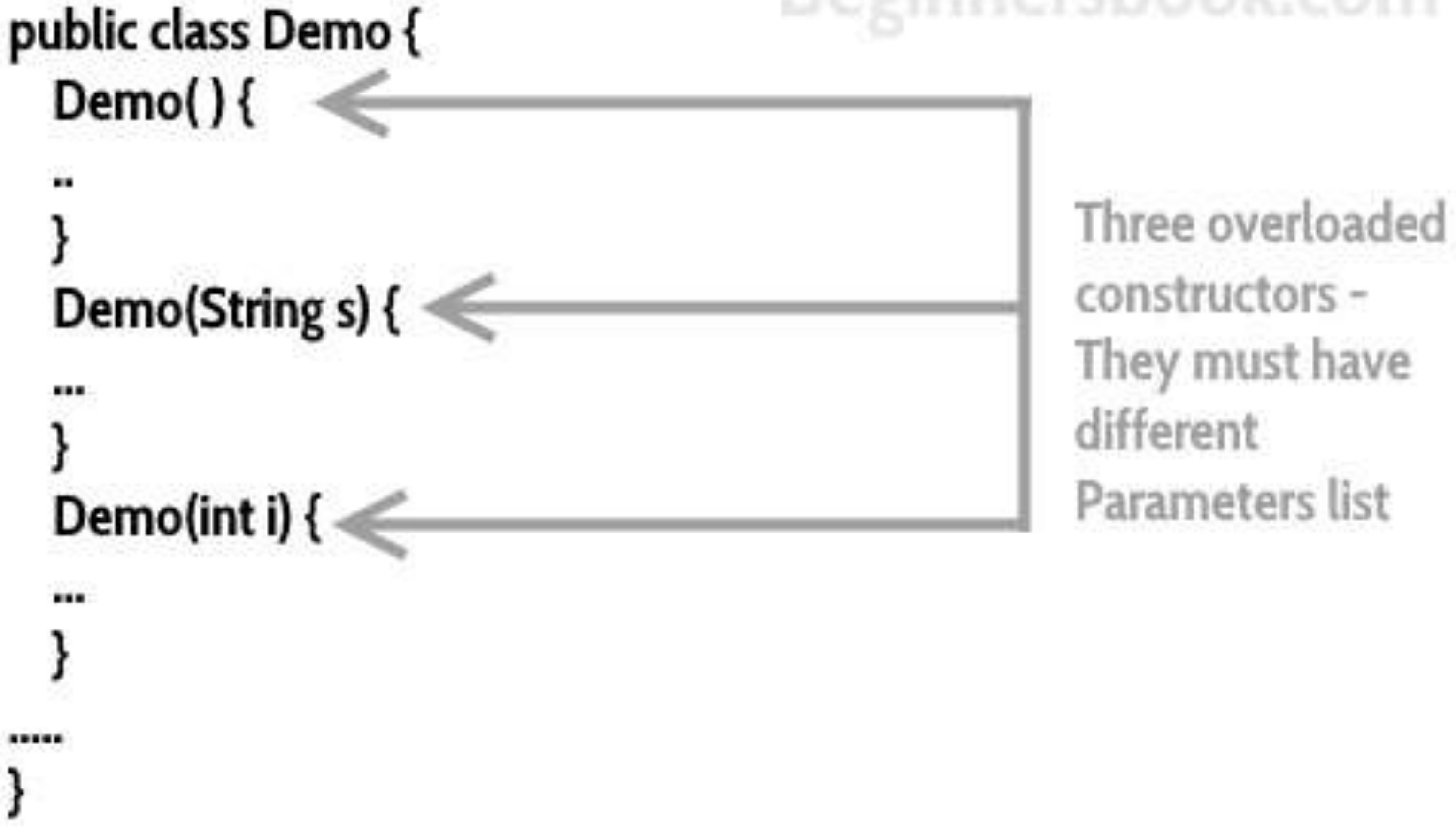
Constructors

- A constructor is a block of codes similar to the method. It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- Every time an object is created using the new() keyword, at least one constructor is called.

- Rules defined for the constructor.
    - Constructor name must be the same as its class name
    - A Constructor must have no explicit return type
    - A Java constructor cannot be abstract, static, final, and synchronized

Constructor Overloading

- We can have more than one constructor in a class with same name, as long as each has a different list of arguments.
  - Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
  - A constructor is called depending upon the number and type of arguments passed.
  - While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.



```
public class Demo {
   Demo( ) {
   ..
   }
   Demo(String s) {
   ...
   }
   Demo(int i) {
   ...
   }
.....
}
```

Sample example

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

Program

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
```

```
  void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

o/p

```
111 Karan 0
 222 Aryan 25
```