1. **Java Polymorphism**

**Method Overloading in Java:**

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

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 a (int,int) for two parameters, and b(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.

So, we perform method overloading to figure out the program quickly. Method overloading increases the readability of the program.

**Note: In java, Method Overloading is not possible by changing the return type of the method only.**

**There are two ways to overload the method in java**

* **By changing number of arguments**

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

We are creating static methods so that we don't need to create instance for calling methods.

**class** Adder {

**static** **int** add(**int** a, **int** b) {

**return** a + b;

}

**static** **int** add(**int** a, **int** b, **int** c) {

**return** a + b + c;

}

}

**class** TestOverloading1 {

**public** **static** **void** main(String[] args) {

System.***out***.println(Adder.*add*(11, 11));

System.***out***.println(Adder.*add*(11, 11, 11));

}

}

* **By changing data type of arguments**

In this example, we have created two methods that differ in data type. The first add method receives two integer arguments and second add method receives two double arguments.

**class** Adder {

**static** **int** add(**int** a, **int** b) {

**return** a + b;

}

**static** **double** add(**double** a, **double** b) {

**return** a + b;

}

}

**class** TestOverloading2 {

**public** **static** **void** main(String[] args) {

System.***out***.println(Adder.*add*(11, 11));

System.***out***.println(Adder.*add*(12.3, 12.6));

}

}

Output:

22

24.9

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

**class** Adder {

**static** **int** add(**int** a, **int** b) {

**return** a + b;

}

**static** **double** add(**int** a, **int** b) {

**return** a + b;

}

}

**class** TestOverloading3 {

**public** **static** **void** main(String[] args) {

System.***out***.println(Adder.*add*(11, 11));// ambiguity

}

}

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

**Method Overriding in Java:**

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.

In other words if subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

**Rules for Java Method Overriding**

* Method must have same name as in the parent class
* Method must have same parameter as in the parent class.
* Must be IS-A relationship (inheritance).

**File: Test2.java**

**class** Bank {

**int** getRateOfInterest() {

**return** 0;

}

}

**class** SBI **extends** Bank {

**int** getRateOfInterest() {

**return** 8;

}

}

**class** ICICI **extends** Bank {

**int** getRateOfInterest() {

**return** 7;

}

}

**class** AXIS **extends** Bank {

**int** getRateOfInterest() {

**return** 9;

}

}

**class** Test2 {

**public** **static** **void** main(String args[]) {

SBI s = **new** SBI();

ICICI i = **new** ICICI();

AXIS a = **new** AXIS();

System.***out***.println("SBI Rate of Interest: " + s.getRateOfInterest());

System.***out***.println("ICICI Rate of Interest: " + i.getRateOfInterest());

System.***out***.println("AXIS Rate of Interest: " + a.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

Note: static method cannot be overridden because static method is bound with class whereas instance method is bound with object.

**‘super’ keyword in java**

The super keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

**1) ‘super’ is used to refer immediate parent class instance variable.**

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

**class** Animal {

String color = "white";

}

**class** Dog **extends** Animal {

String color = "black";

**void** printColor() {

System.***out***.println(color);// prints color of Dog class

System.***out***.println(**super**.color);// prints color of Animal class

}

}

**class** TestSuper1 {

**public** **static** **void** main(String args[]) {

Dog d = **new** Dog();

d.printColor();

}

}

Output:

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

1. **super can be used to invoke parent class method**

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

**class** Animal {

**void** eat() {

System.***out***.println("eating...");

}

}

**class** Dog **extends** Animal {

**void** eat() {

System.***out***.println("eating bread...");

}

**void** bark() {

System.***out***.println("barking...");

}

**void** work() {

**super**.eat();

bark();

}

}

**class** TestSuper2 {

**public** **static** **void** main(String args[]) {

Dog d = **new** Dog();

d.work();

}

}

**Output:**

eating...

barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

1. **super is used to invoke parent class constructor**

The super keyword can also be used to invoke the parent class constructor.

**class** Animal {

Animal() {

System.***out***.println("animal is created");

}

}

**class** Dog **extends** Animal {

Dog() {

**super**();

System.***out***.println("dog is created");

}

}

**class** TestSuper3 {

**public** **static** **void** main(String args[]) {

Dog d = **new** Dog();

}

}

**Output:**

animal is created

dog is created

**super example: real use**

**class** Person {

**int** id;

String name;

Person(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

}

**class** Emp **extends** Person {

**float** salary;

Emp(**int** id, String name, **float** salary) {

**super**(id, name);// reusing parent constructor

**this**.salary = salary;

}

**void** display() {

System.***out***.println(id + " " + name + " " + salary);

}

}

**class** TestSuper5 {

**public** **static** **void** main(String[] args) {

Emp e1 = **new** Emp(1, "ankit", 45000f);

e1.display();

}

}

**Output:**

1 ankit 45000

**Final Keyword:**

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

* variable
* method
* class

**1) Java final variable**

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example of final variable

**class** Bike9 {

**final** **int** speedlimit = 90;// final variable

**void** run() {

speedlimit = 400;

}

**public** **static** **void** main(String args[]) {

Bike9 obj = **new** Bike9();

obj.run();

}

}// end of class

Output:Compile Time Error

**2) Java final method**

If you make any method as final, you cannot override it.

**class** Bike {

**final** **void** run() {

System.***out***.println("running");

}

}

**class** Honda **extends** Bike {

**void** run() {

System.***out***.println("running safely with 100kmph");

}

**public** **static** **void** main(String args[]) {

Honda honda = **new** Honda();

honda.run();

}

}

Output:Compile Time Error

**3) Java final class**

If you make any class as final, you cannot extend it.

**final** **class** Bike {

}

**class** Honda1 **extends** Bike {

**void** run() {

System.***out***.println("running safely with 100kmph");

}

**public** **static** **void** main(String args[]) {

Honda1 honda = **new** Honda1();

honda.run();

}

}

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it.

**Q) What is blank or uninitialized final variable?**

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

**class** Bike10 {

**final** **int** speedlimit;// blank final variable

Bike10() {

speedlimit = 70;

System.***out***.println(speedlimit);

}

**public** **static** **void** main(String args[]) {

**new** Bike10();

}

}

Output: 70

**static blank final variable**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

**class** A {

**static** **final** **int** ***data***;// static blank final variable

**static** {

***data*** = 50;

}

**public** **static** **void** main(String args[]) {

System.***out***.println(A.***data***);

}

}

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

**class** Bike11 {

**int** cube(**final** **int** n){

n=n+2;//can't be changed as n is final

n\*n\*n;

}

**public** **static** **void** main(String args[]) {

Bike11 b = **new** Bike11();

b.cube(5);

}

}

Output: Compile Time Error

**Q) Can we declare a constructor final?**

No, because constructor is never inherited.

**Polymorphism in Java:**

There are two types of polymorphism in java

1. Compile time polymorphism
2. Runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime polymorphism 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. The determination of the method to be called is based on the object being referred to by the reference variable.

When reference variable of Parent class refers to the object of Child class, it is known as upcasting.

**class** Bike {

**void** run() {

System.***out***.println("running");

}

}

**class** Splender **extends** Bike {

**void** run() {

System.***out***.println("running safely with 60km");

}

**public** **static** **void** main(String args[]) {

Bike b = **new** Splender();// upcasting // this is also called dynamic binding

b.run();

}

}

Output:

running safely with 60km.

**Java Runtime Polymorphism with Data Member**

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

**class** Bike {

**int** speedlimit = 90;

}

**class** Honda3 **extends** Bike {

**int** speedlimit = 150;

**public** **static** **void** main(String args[]){

Bike obj=**new** Honda3();

System.out.println(obj.speedlimit);//90

}

Output:

90

**Java Runtime Polymorphism with Multilevel Inheritance**

File: BabyDog.java

**class** Animal {

**void** eat() {

System.***out***.println("eating");

}

}

**class** Dog **extends** Animal {

**void** eat() {

System.***out***.println("eating fruits");

}

}

**class** BabyDog **extends** Dog {

**void** eat() {

System.***out***.println("drinking milk");

}

**public** **static** **void** main(String args[]) {

Animal a1, a2, a3;

a1 = **new** Animal();

a2 = **new** Dog();

a3 = **new** BabyDog();

a1.eat();

a2.eat();

a3.eat();

}

}

Output:

eating

eating fruits

drinking Milk

**class** Animal {

**void** eat() {

System.***out***.println("animal is eating...");

}

}

**class** Dog **extends** Animal {

**void** eat() {

System.***out***.println("dog is eating...");

}

}

**class** BabyDog1 **extends** Dog {

**public** **static** **void** main(String args[]) {

Animal a = **new** BabyDog1();

a.eat();

}

}

Output:

Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

**Static binding**

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

class Dog{

private void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){

Dog d1=new Dog();

d1.eat();

}

}

**Dynamic binding**

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

class Animal{

void eat(){System.out.println("animal is eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){

Animal a=new Dog();

a.eat();

}

}

Output:

dog is eating...