Lec No. (11)

Java Polymorphism

Instance initializer block

Instance Initializer block is used to initialize the instance data member. It runs each time when object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
class Bike{
   int speed=100;
}
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. as for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block that performs initialization.

```
class Bike7{
   int speed;

   Bike7(){System.out.println("speed is "+speed);}

   {speed=100;}

   public static void main(String args[]){
   Bike7 b1=new Bike7();
   Bike7 b2=new Bike7();
   }
}
```

```
Output:speed is 100
       speed is 100
```

- **There are three places in java where you can perform operations:**
  1. **method**
  2. **constructor**
  3. **block**

Q) What is invoked first, instance initializer block or constructor?

```
class Bike8{
   int speed;

   Bike8(){System.out.println("constructor is invoked");}

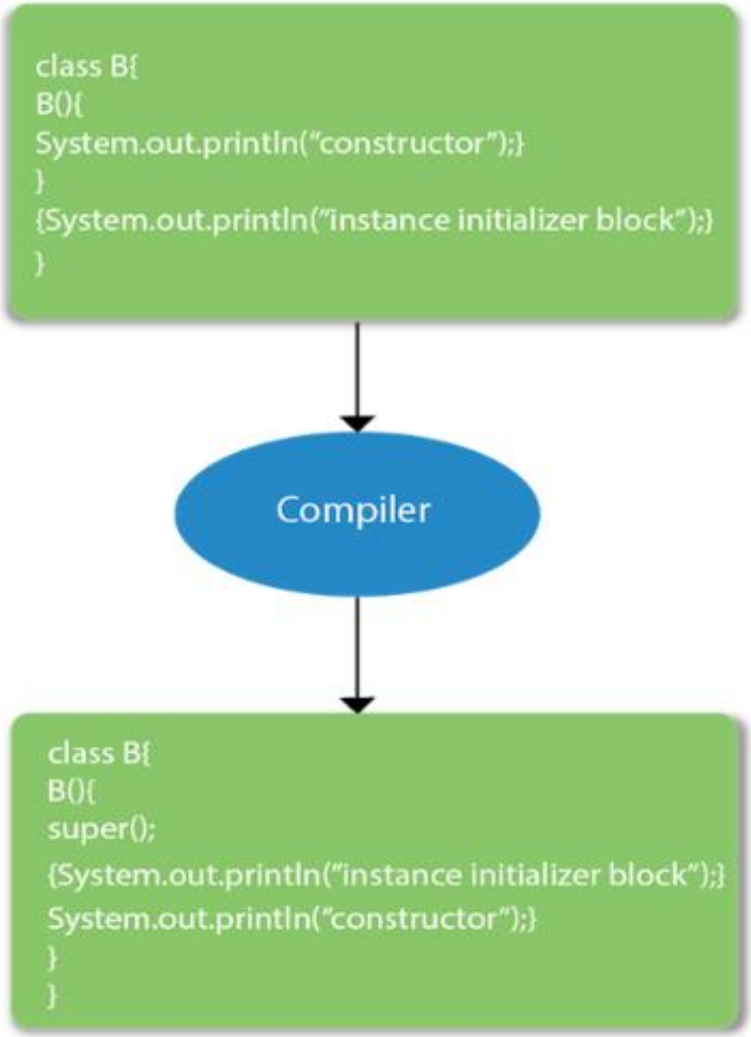
   {System.out.println("instance initializer block invoked");}

   public static void main(String args[]){
   Bike8 b1=new Bike8();
   Bike8 b2=new Bike8();
   }
}
```

```
Output:instance initializer block invoked
       constructor is invoked
       instance initializer block invoked
       constructor is invoked
```

In the above example, **it seems that instance initializer block is firstly invoked but NO.** Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super ( ). **So firstly, constructor is invoked**. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



➢ Rules for instance initializer block:

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked **after** the parent class constructor is invoked (i.e. after super ( ) constructor call).
3. The instance initializer block comes in the order in which they appear.

- Program of instance initializer block that is invoked after super ( )

```
class A{
A(){
System.out.println("parent class constructor invoked");
}
}
class B2 extends A{
B2(){
super();
System.out.println("child class constructor invoked");
}

{System.out.println("instance initializer block is invoked");}

public static void main(String args[]){
B2 b=new B2();
}
}
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```

➢Final Keyword in Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

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

There is a final variable speed limit, we are going to change the value of this variable, but it can't be changed because final variable once assigned a value can never be changed.

```
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**.

Example of final method

```
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.**

Example of final class

```
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? Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

```
Output:running...
```

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.

Example of blank final variable

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Q) Can we initialize blank final variable? Yes, but only in constructor. For example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

O/P: 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

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

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

➢ Runtime Polymorphism in Java

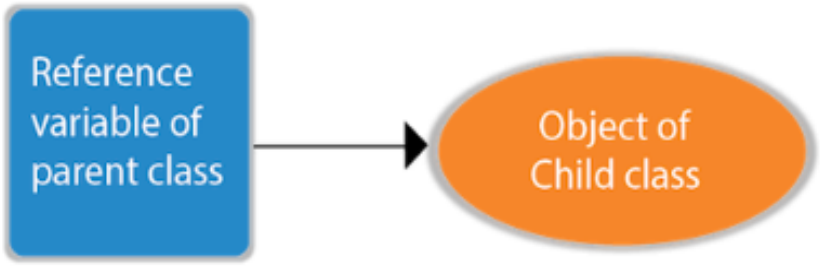
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

➢ Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}
class B extends A{}
```

```
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

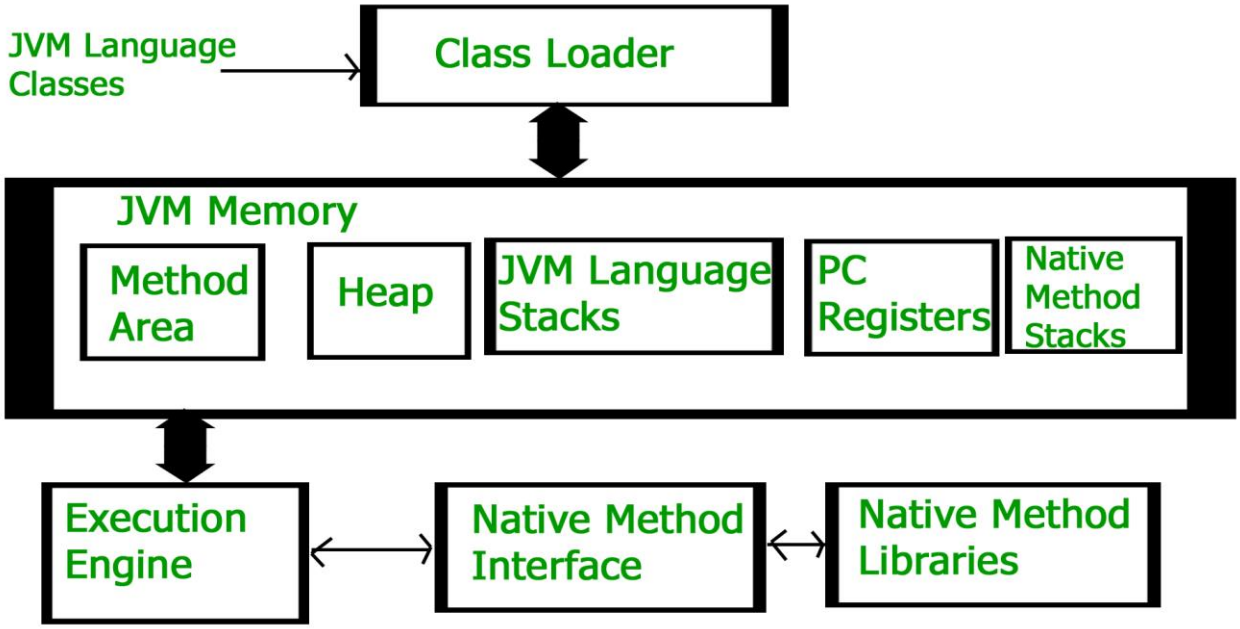
Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run ( ) method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

❖ **Java Virtual Machine (JVM)** is the engine that converts Java bytecode into machine language. JVM architecture in Java contains class loader, method area, heap, JVM language stacks, PC registers, native method stacks, execution engine, native methods interface, and native methods libraries.



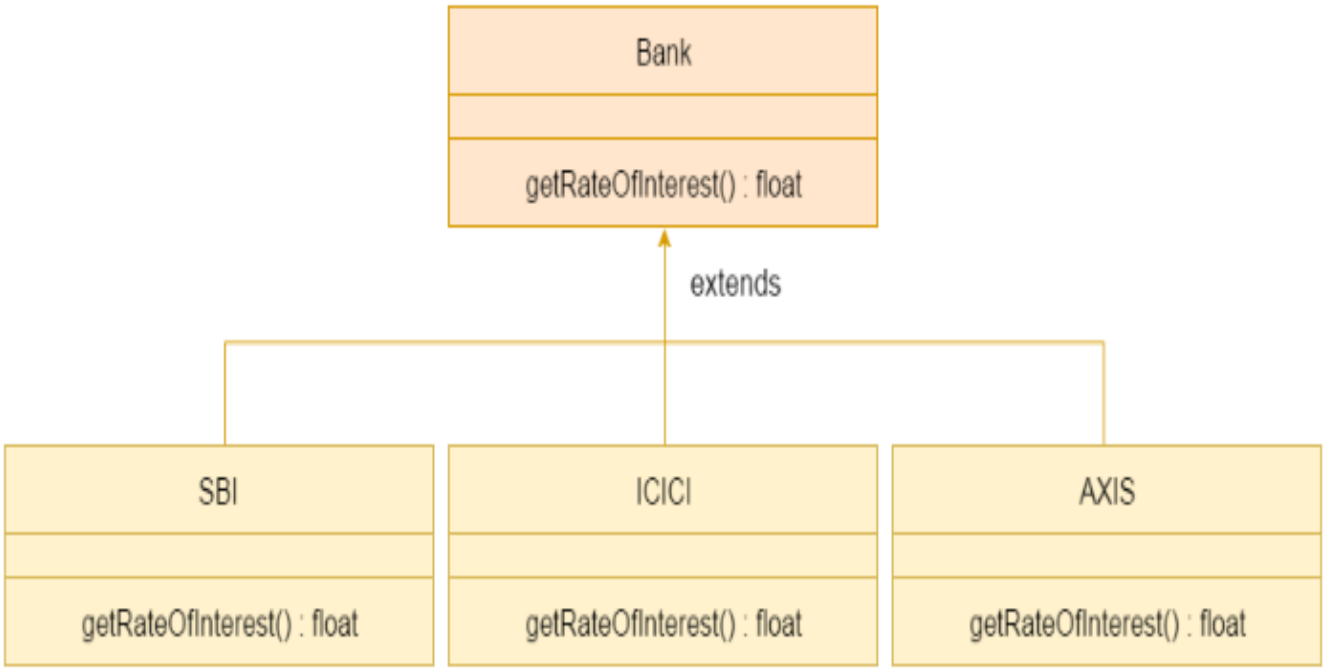

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output: running safely with 60km.

➢ Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

➢ Java Runtime Polymorphism Example: Animal

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
class Cat extends Animal{
void eat(){System.out.println("eating rat...");}
}
class Lion extends Animal{
void eat(){System.out.println("eating meat...");}
}
class TestPolymorphism3{
public static void main(String[] args){
Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();
}}
```

Output:

```
eating bread...
eating rat...
eating meat...
```

➢ Java Runtime Polymorphism with Data Member

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

In the example given below, both the classes have a data member speed limit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: 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

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
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
```

Try for Output

```
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, Baby Dog is not overriding the eat ( ) method, so eat ( ) method of Dog class is invoked.