Unit 2
Concept of classes and Inheritance

OOPs Concepts

- Class
- Methods
- Constructors
- Overloading methods and constructors
- Garbage Collection in Java
- Static and Non static members
- Initializer and Class Initializer block

OOPs Concepts

- **Object** means a real word entity such as pen, chair, table etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

OOPs Concepts

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

OOPs Concepts

- Polymorphism
  - ✔ When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.
  - ✔ In java, we use method overloading and method overriding to achieve polymorphism.
  - ✔ Another example can be to speak something e.g. cat speaks meow, dog barks woof etc.
- Abstraction
  - √ Hiding internal details and showing functionality is known as abstraction.
  - √ For example: phone call, we don't know the internal processing.
  - √ In java, we use abstract class and interface to achieve abstraction.

OOPs Concepts

- Encapsulation
  - √ **Binding (or wrapping) code and data together into a single unit is known as encapsulation.** For example: capsule, it is wrapped with different medicines.
  - ✔ A java class is the example of encapsulation.

class

- A class is a blueprint from which individual objects are created.
- A class in Java can contain:
  I. **fields**
  II. **methods**
  III. **constructors**
  IV. **blocks**
  V. **nested class and interface**

- A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Syntax to declare a class

```
class <class_name>{
    field;
    method;
}
```

- A variable which is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object(instance) is created. That is why, it is known as instance variable.

class

Example of class :

```
class Student{
 int id;
 String name;
}
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Object

- An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical
- For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
- **Object Definitions:**
  1. Object is *a real world entity*.
  2. Object is *a run time entity*.
  3. Object is *an entity which has state and behavior*.
  4. Object is *an instance of a class*.

Ways to initialize object

- There are 3 ways to initialize object in java.
  1. By reference variable
  2. By method
  3. By constructor

Object and Class Example: Initialization through reference

- Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

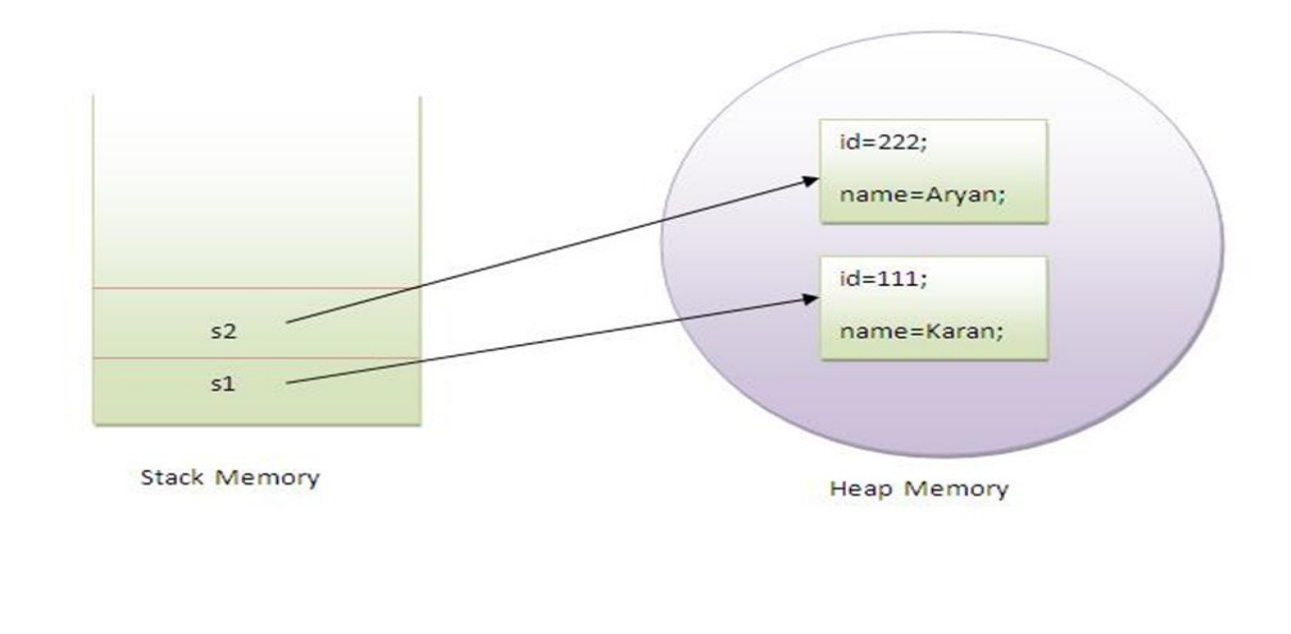
```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white spac
e
 }
}
```

Object and Class Example: Initialization through method

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Object and Class Example: Initialization through method

- As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

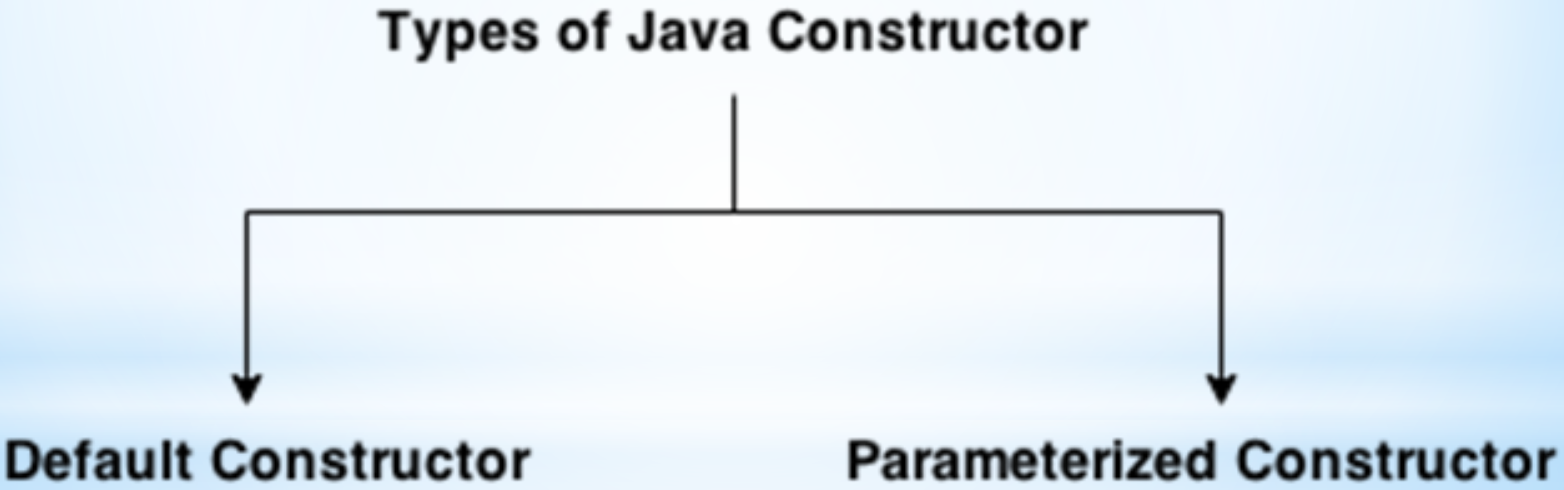


Constructor

- **Constructor in java** is a *special type of method* that is used to initialize the object.
- Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.
- Rules for creating java constructor
  1. Constructor name must be same as its class name
  2. Constructor must have no explicit return type

Constructor

- There are two types of constructors:
  I. Default constructor (no-arg constructor)
  II. Parameterized constructor



Constructor

1. Default Constructor :

- A constructor is called "Default Constructor" when it doesn't have any parameter.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

- **Rule: If there is no constructor in a class, compiler automatically creates a default constructor**

Constructor

2. Parameterized Constructor :

- A constructor which has a specific number of parameters is called parameterized constructor.
- Parameterized constructor is used to provide different values to the distinct objects.

Constructor

Example of parameterized constructor :

```
class Student{
    int id;
    String name;

    Student(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

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

Constructor

- Constructor Overloading in Java :

- ✔ Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- ✔ They are arranged in a way that each constructor performs a different task.
- ✔ They are differentiated by the compiler by the number of parameters in the list and their types.

Constructor

Example :constructor overloading :

```
class Student{
    int id;
    String name;
    int age;
    Student(int i,String n){
    id = i;
    name = n;
    }
    Student(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 Student(111,"Karan");
    Student5 s2 = new Student(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Java static keyword

- The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class.
- The static keyword belongs to the class than instance of the class.
- **Java static property is shared to all objects**

Method overloading

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.
- **Usage of Java Method Overriding**
  1. Method overriding is used to provide specific implementation of a method that is already provided by its super class.
  2. Method overriding is used for runtime polymorphism
- **Rules for Java Method Overriding**
  1. method must have same name as in the parent class
  2. method must have same parameter as in the parent class.
  3. must be IS-A relationship (inheritance).

Method overloading

Example:method overriding :

```
// 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()"); }
}
```

```
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();
    }
}
```

Inheritance

- Inheritance is the process by which one object acquires the properties of another object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Why use inheritance in java

  I. For Method Overriding (so runtime polymorphism can be achieved).

  II. For Code Reusability.

Inheritance

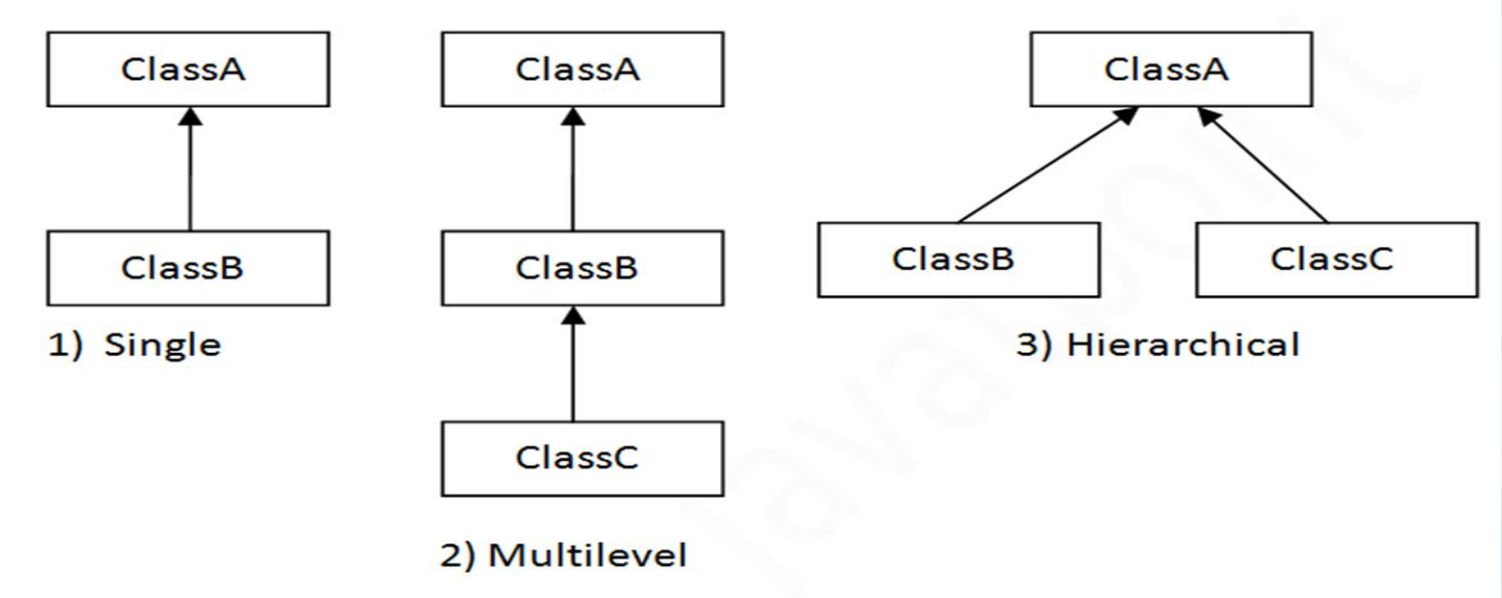
- Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

Inheritance

Types of inheritance in java :

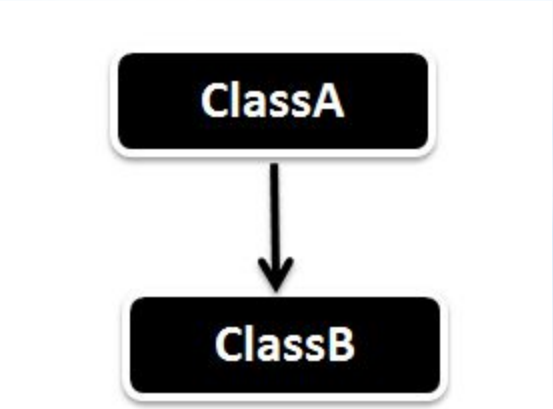


Multiple inheritance is not supported in java through class.

Inheritance

1. Single inheritance :

- Single Inheritance is the simple inheritance of all, When a class extends another class(Only one class) then we call it as **Single inheritance.**
- The below diagram represents the single inheritance in java where **Class B** extends only one class **Class A.** Here **Class B** will be the **Sub class** and **Class A** will be one and only **Super class**



Inheritance

1. Single inheritance :

```
class Animal
{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal
{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

Inheritance

1. Example Single inheritance :

```
public class Shape
{
    int length;
    int breadth;
}
public class Rectangle extends Shape
{
    int area;
    public void calcualteArea()
    {
        area = length*breadth;
    }
```

Inheritance

```
public static void main(String args[])
  {
      Rectangle r = new Rectangle();
      //Assigning values to Shape class attributes
      r.length = 10;
      r.breadth = 20;
      //Calculate the area
      r.calcualteArea();
      System.out.println("The Area of rectangle of length "  +r.length+" and breadth "+r.breadth+"is "+r.area+"");
  }
```

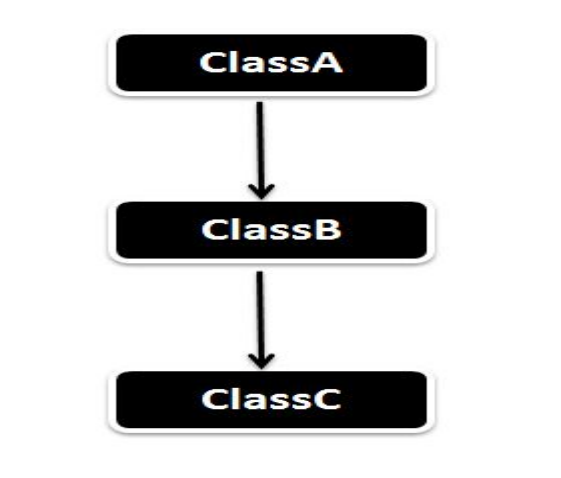
Inheritance

2. Multilevel inheritance :

- In **Multilevel Inheritance** a derived class will be **inheriting a parent class** and as well as the derived class **act as the parent class** to other class.
- As seen in the below diagram. **ClassB** inherits the property of **ClassA** and again **ClassB** act as a parent for **ClassC**. In Short **ClassA** parent for **ClassB** and **ClassB** parent for **ClassC**.
- For ex:We have3 class **Person, Staff and TemporaryStaff** Here **Person** class will be inherited by **Staff** class and **Staff** class will be again inherited by the **TemporaryStaff** class. **TemporaryStaff** class can access the members of both **Staff** class(**directly**) and **Person** class(**indirectly**).

Inheritance

2. Multilevel inheritance :



Inheritance

Example 1:Multilevel Inheritance

```
Class X
{
   public void methodX()
   {
     System.out.println("Class X method");
   }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
```

```
Class Z extends Y
{
   public void methodZ()
   {
     System.out.println("class Z method");
   }
   public static void main(String args[])
   {
     Z obj = new Z();
     obj.methodX(); //calling grand parent
class method
     obj.methodY(); //calling parent class
method
     obj.methodZ(); //calling local method
  }
}
```

Inheritance

Example2: Multilevel Inheritance

```
class Person
{
    private String name;
    Person(String s)
    {
         setName(s);
    }
    public void setName(String s)
    {
      name = s;
    }
    public String getName()
    {
      return name;
    }
    public void display()
    {
      System.out.println("Name of Person = " + name);
    }
}
```

Inheritance

Example2: Multilevel Inheritance

```
class Staff extends Person
{
    private int staffId;
    Staff(String name,int staffId)
    {
      super(name);
      setStaffId(staffId);
    }
    public int getStaffId() {
        return staffId;
    }
    public void setStaffId(int staffId) {
        this.staffId = staffId;
    }
    public void display()
    {
      super.display();
      System.out.println("Staff Id is  = " + staffId);
    }
}
```

Inheritance

Example2: Multilevel Inheritance

```
 class TemporaryStaff extends Staff
{
    private int days;
    private int hoursWorked;
    TemporaryStaff(String sname,int id,int days,int hoursWorked)
    {
       super(sname,id);
       this.days  = days;
       this.hoursWorked = hoursWorked;
    }
    public int Salary()
    {
        int salary = days * hoursWorked * 50;
```

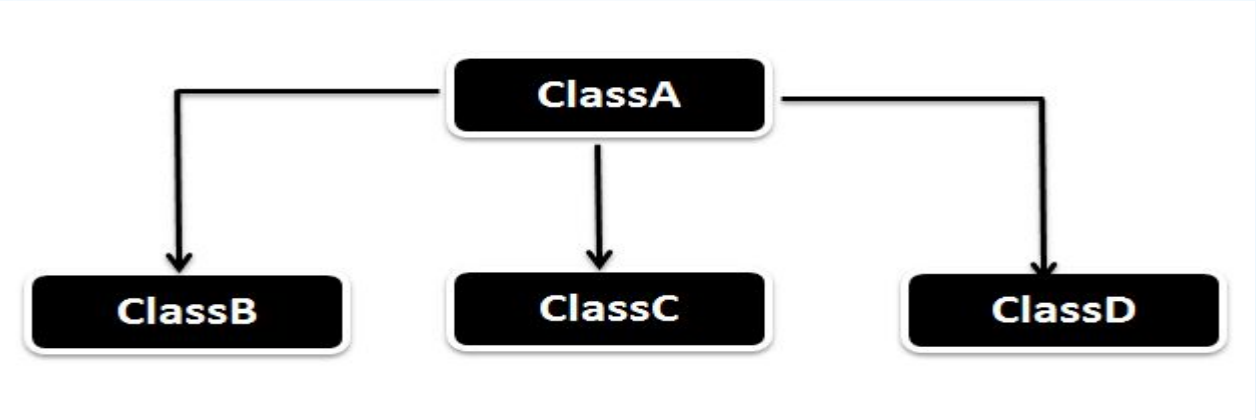
Inheritance

3. Hierarchical Inheritance :

- Hierarchical inheritance multiple classes inherits from a single class i.e there is one super class and multiple sub classes.
- As we can see from the below diagram when a same class is having more than one sub class (or) more than one sub class has the same parent is called as Hierarchical Inheritance.
- Here ClassA acts as the parent for sub classes ClassB, ClassC and ClassD.

Inheritance

3. Hierarchical Inheritance :



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

I. super can be used to refer immediate parent class instance variable.

II. super can be used to invoke immediate parent class method.

III. super() can be used to invoke immediate parent class constructor.

Super keyword in java

Example 1:super is used to refer immediate parent class instance variable :

```
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();
}}
```

Super keyword in java

Example 2:super can be used to invoke parent class method :

```
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();
}}
```

Super keyword in java

Example 3:super is used to invoke parent class constructor :

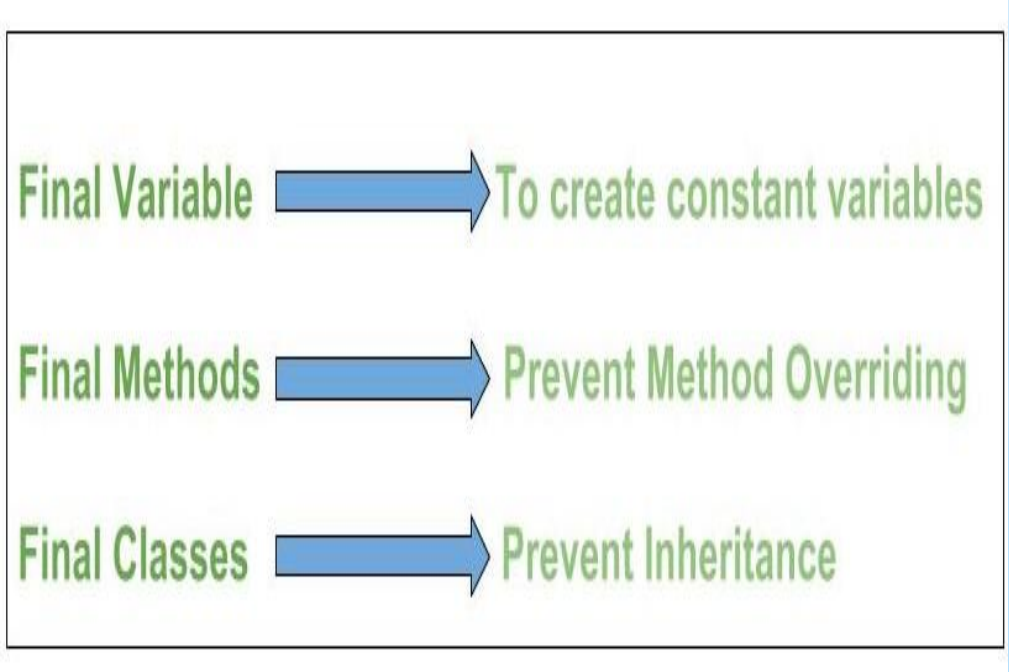
```
class Person
{
    int id;
     String name;
    Person(int id,String name)
    {
    this.id=id;
    this.name=name;
    }
}
class Emp extends Person
{
float salary;
    Emp(int id,String name,float salary)
    {
```

```
class TestSuper5
{
        public static void main(String[] args)
        {
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
        }
}
```

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 with:

I. variable
II. method
III. Class



Final keyword

Example 1:Final variable

- final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
public class Circle
 {
    public static final double PI=3.14159;
    public static void main(String[] args)
     {
      PI=5.54;// changing the value c
     System.out.println(PI);
    }
 }
```

```
D:\>javac circle.java
circle.java:6: error: cannot assign a
value to final variable PI
  PI=5.54;// changing the value of PI
  ^
1 error
```

Final keyword

Example 2:Final method

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

```
class stud
{
 final void show() {
  System.out.println("Class - stud : method defined");
 }
}
class books extends stud {
 void show() {
  System.out.println("Class - b
 }
```

```
: error: show() in books cannot override
show() in stud
 void show() {
      ^
  overridden method is final
1 error
```

Final keyword

Example 3:

- It makes a class final, meaning that the class can not be inheriting by other classes. When we want to restrict inheritance then make class as a final.

```
final class stud
{ // Methods cannot be extended to its sub class
}
class books extends stud
 { void show()
{ System.out.println("Book-Class method"); }
public static void main(String args[])
 {
```

```
error: cannot inherit from final
stud
class books extends stud
                    ^
1 error
```

Abstract class

Abstract method and abstract class in Java

- Method without body (no implementation) is known as abstract method.
- A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well.
- This is how an abstract method looks in java:

```
public abstract int myMethod(int n1, int n2);
```

Abstract class

Rules of Abstract Method :

1. Abstract methods don't have body, they just have method signature as shown above.

```
public abstract int myMethod(int n1, int n2);
```

2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Abstract class

Why we need an abstract class?

- Lets say we have a class Animal that has a method sound() and the subclasses(see inheritance) of it like Dog, Lion, Horse, Cat etc.
- Since the animal sound differs from one animal to another, there is no point to implement this method in parent class.
- This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

Abstract class

```
//abstract parent class
abstract class Animal
{ //abstract method
public abstract void sound();
 }
//Dog class extends Animal class
public class Dog extends Animal
{
	public void sound()
		{
			System.out.println("Woof");
		}
 }
public class Lion extends Animal
{
	public void sound()
		{
			System.out.println(“Roar");
		}
 }
```

Abstract class

```
classAnimaldemo
{
public static void main(String args[])

{
      Animal obj = new Dog();
      obj.sound();
   Animal obj2=new Lion()
     obj2.sound();
}
}
```

Abstract class

- An abstract class, in the context of Java, is a superclass that cannot be instantiated and is used to state or define general characteristics.
- An object cannot be formed from a Java abstract class; trying to instantiate an abstract class only produces a compiler error. The abstract class is declared using the keyword abstract.

Interface

- An **interface in java** is a blueprint of a class. It has static constants(final) and abstract methods.
- The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also **represents IS-A relationship**.
- It cannot be instantiated just like abstract class.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

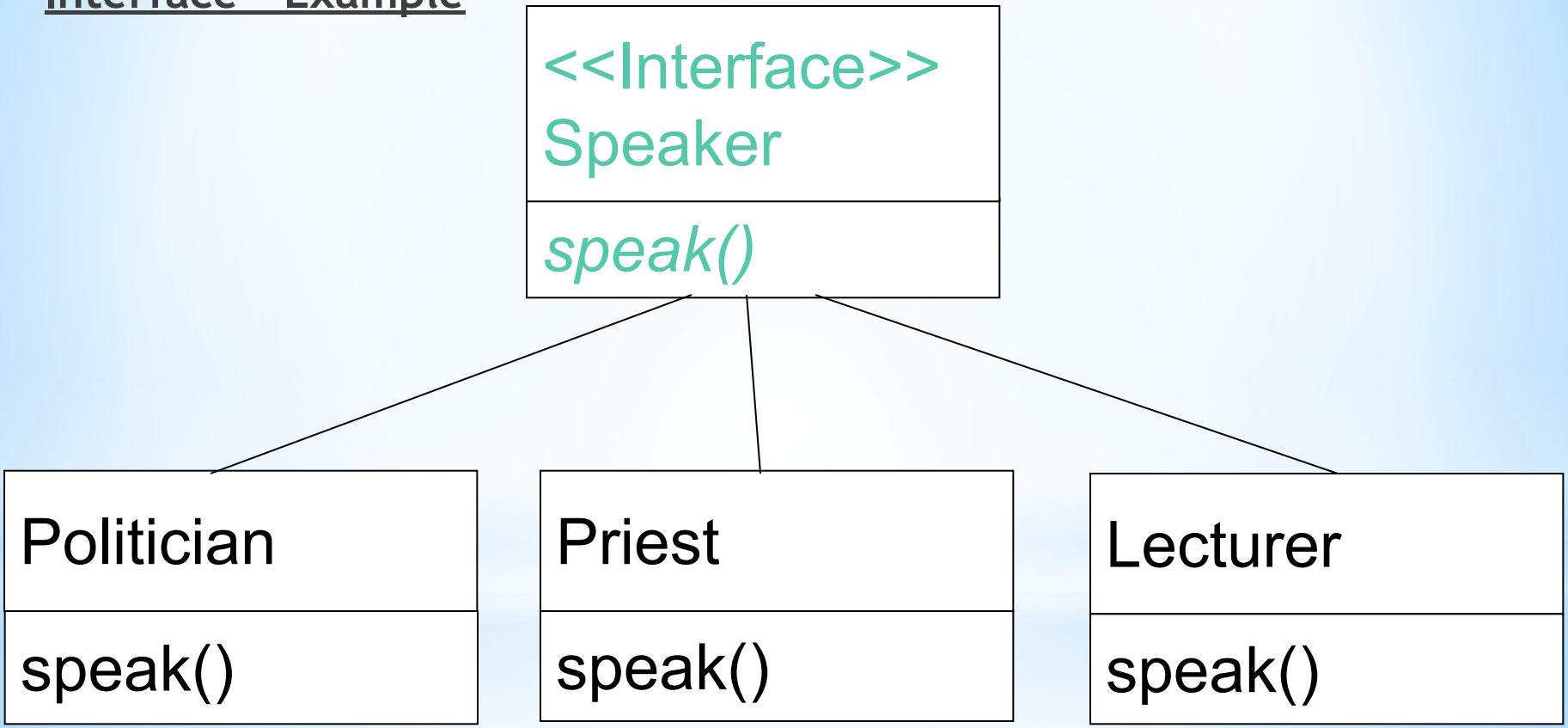
I. It is used to achieve abstraction.

II. By interface, we can support the functionality of multiple inheritance.

III. It can be used to achieve loose coupling.

Interface

Interface - Example



Interface

Interface - Defition

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration – no method body
}
```

```
interface Speaker {
    public void speak( );
}
```

Interface

Implementing Interfaces :

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, ...]
{
        // Body of Class
}
```

Interface

Implementing Interfaces Example :

```
class  Politician implements Speaker {
        public void speak(){
            System.out.println("Talk politics");
        }
}
```

```
class  Priest implements Speaker {
        public void speak(){
            System.out.println("Religious Talks");
        }
}
```

```
class  Lecturer implements Speaker {
        public void speak(){
            System.out.println("Talks Object Oriented Design and Programming!");
        }
}
```

Interface

Interface with multiple inheritance

- multiple inheritance is not supported in case of class because of ambiguity.

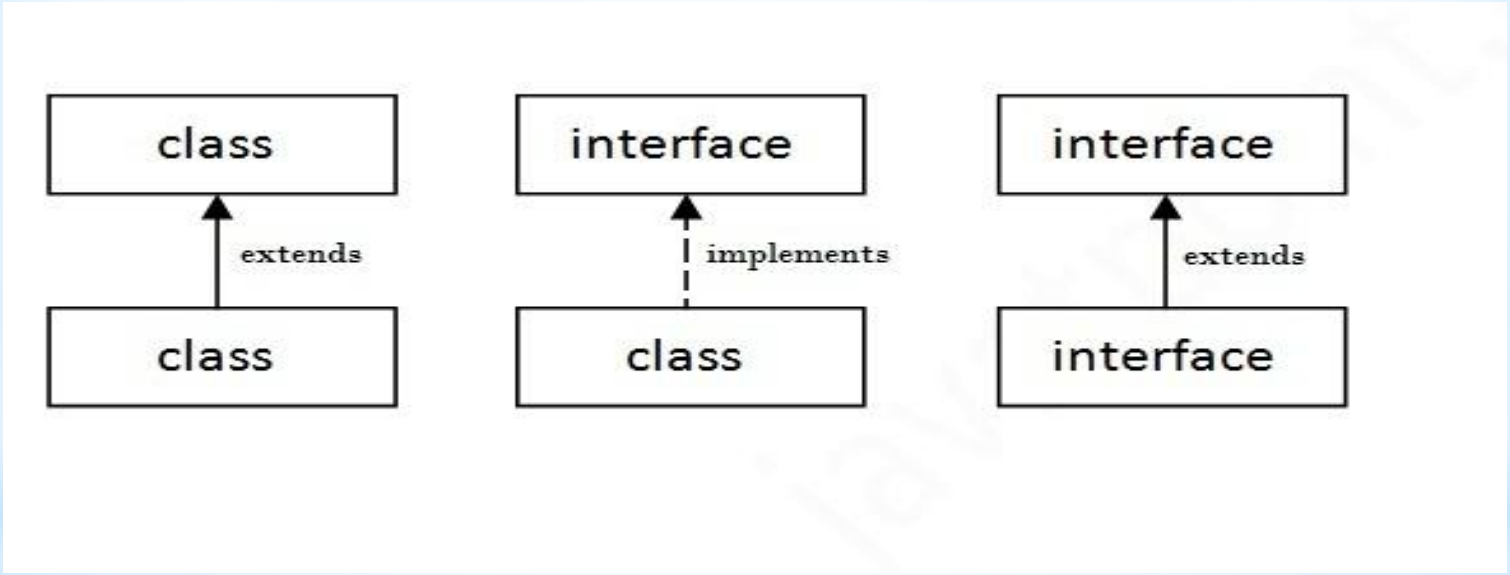
But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Interface

Understanding relationship between classes and interfaces :



Interface

Extending Interfaces

- Like classes, interfaces can also be extended.
- The new sub-interface will inherit all the members of the super interface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

Interface

Inheritance and Interface Implementation

- A general form of interface implementation:

```
class ClassName extends SuperClass implements InterfaceName [,
InterfaceName2, ...]
{
      // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Interface

Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks. A class diagram representing this scenario is as follow: