MODULE-3

Basic Characteristics of OOP, Class fundamentals. Declaration objects, Introducing Methods, Constructors, this keyword. Inheritance, Types of inheritance, Member access rules, Abstract Classes, Super and final keywords. Method overloading and overriding. Defining an interface, implementing interface, Accessing interface properties

1. Explain the basic concepts of OOPS?

Basic Characteristics of OOP/Basic Concepts of Object Oriented Programming/OOP's Principles

Object-Oriented Programming supports the following concepts.

1. Objects
2. Classes
3. Data Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic Binding
8. Message Communication

Objects: Objects are primary run-time entities in an Object-Oriented System. An object can be any thing that exists in the real world. Any programming problem is analyzed in terms of objects. An object is a composition of properties and actions. Properties of an object can be represented with variables and actions are represented with methods.

Ex: Chair, Car, Pen etc.,

Classes: A Class defines the abstract characteristics of an object, including the characteristics (Properties) and the behaviors. A Class is a collection of related objects that share common properties and actions. Once a class is defined, we can create any number of objects belonging to that class.

Ex : Person, Vehicle, Furniture, etc.,

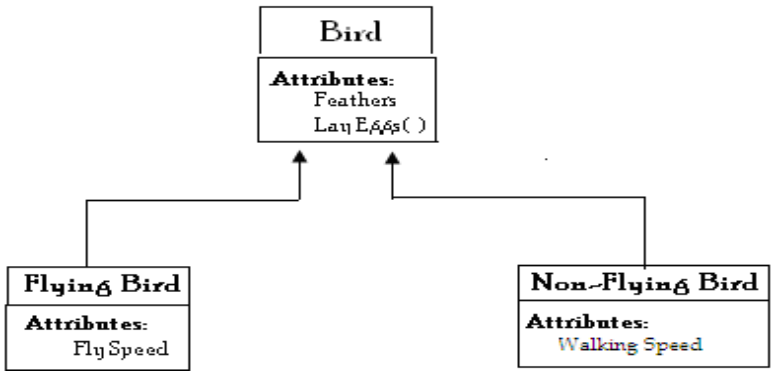
Data Abstraction: Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. The concept of hiding the unnecessary contents from the user is called "***Abstraction***".

Encapsulation: The wrapping up of data and methods into a single unit called Class is known as *"**Encapsulation**"*. The data is not accessible to the other methods and the methods which are wrapped in the class can only access the data items. This insulation of the data from direct access by the program is called *"**Data Hiding**"*.

Inheritance:

The process of inheriting attributes and behaviors from one class to other is called *"**Inheritance**"*. Here the objects of one class acquire the properties of another class. We can derive as many classes based on a class. The class which is used as base for deriving a new class is called "Super/Parent/Base Class". The newly developed class is called as "Sub/Child/Derived Class". By using inheritance, we achieve reusability of the code and develop more specialized classes.

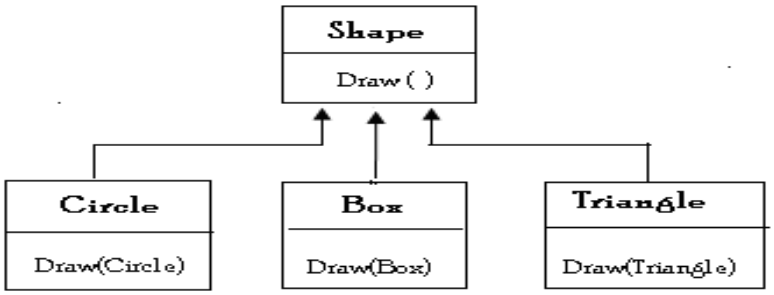
Ex:



In the above example, the class ***Bird*** is called as Super class and the classes ***FlyingBird*** and ***Non-FlyingBird*** are called as Sub classes to Bird.

Polymorphism: Poly means Several and Morphose means Forms. Polymorphism means the ability to take more than one form. If something exists in different forms in different situations it is called as

*"**Polymorphism**"*. Polymorphism allows the programmer to treat derived class members just like their parent class' members.
Ex:



Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic Binding means that the code associated with a given procedure call is not known until the time of the call at runtime. Dynamic Binding is associated with the Polymorphism and Inheritance.

Message Communication:

The process by which an object sends data to another object to invoke a method is called as "***Message Communication***". Message Communication involves the following basic steps:

a. Creating Classes that define objects and their behavior
b. Creating objects from class definition
c. Establishing communication among objects

(*Optional Question)Class fundamentals

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

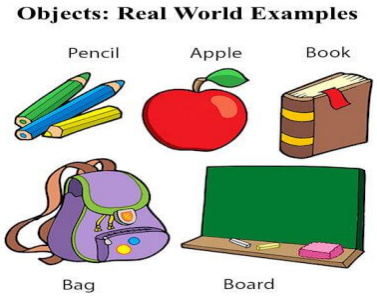
- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

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

(*Optional Question)Declaration objects

Object: An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only. 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 (tangible and intangible). The example of an intangible object is the banking system.



An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Create an object in Java:

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Initialize object:

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference: Initializing an object means storing data into the object. The below example illustrates how to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[])
{
  Student s1=new Student();
  s1.id=1000;
  s1.name="Narayana";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

OUTPUT:

1000

2) Object and Class Example: Initialization through method: In this example, need to create the two objects of Student class and initializing the value to these objects by invoking the insert method. Here, displaying the state (data) of the objects by invoking the display() method.

```
class Student{
 int rollno;
 String name;
 void insert(int r, String n){
  rollno=r;
  name=n;
 }
 void display()
{
System.out.println(rollno+" "+name);
}
```

```
    }
    class TestStudent{
     public static void main(String args[]){
      Student s1=new Student();
      Student s2=new Student();
      s1.insert(1000,"Ramesh");
      s2.insert(2000,"Chakry");
      s1.display();
      s2.display();
     }
    }
```

OUTPUT:

1000 Ramesh
2000 Chakry

3) Object and Class Example: Initialization through a constructor

```
    class Employee{
       int id;
       String name;
       float salary;
       void Employee(int i, String n, float s) {
          id=i;
          name=n;
          salary=s;
       }
       void display(){System.out.println(id+" "+name+" "+salary);}
    }
    public class TestEmployee {
    public static void main(String[] args) {
       Employee e1=new Employee(101,"Raju",45000);
       Employee e2=new Employee(102,"Ram",25000);
       Employee e3=new Employee(103,"Ramesh",55000);
       e1.display();
       e2.display();
       e3.display();
    } }
```

OUTPUT:

```
101 Raju 45000.0
102 Ram 25000.0
103 Ramesh 55000.0
```

(*Optional Question)ASSIGNING OBJECT REFERENCE VARIABLES

Use an object reference variable to create, manage, and delete objects. It has the data type of a class and, like other variables, is a named area in storage. However, unlike other variables, the value stored in the area is not the object itself but a 4-byte pointer to the object data, called an object reference.Starting by taking two variables var1 and var2 of integer type, such as

```
int var1 = 5;
int var2 = var1;
```

Simply, 5 is assigned to var1 and the value of var1 is assigned to var2.

Changing the value of var2 as

```
var2 = 10;
```

If var1 and var2 are printed, the output is 5 and 10 respectively. But, in case of object reference variables, user may marvel, when assigning one object reference variable to another.

Ex:

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

```
{
 Demo obj = new Demo();
 obj.a = 30;
 obj.b = 50;
 Demo obj1 = obj;
 System.out.println(obj.add(obj.a, obj.b));
 System.out.println(obj.add(obj1.a, obj1.b));
 obj.a = 50;
 obj.b = 50;
 System.out.println(obj.add(obj.a, obj.b));
 System.out.println(obj.add(obj1.a, obj1.b));
 obj1.a = 10;
 obj1.b = 20;
 System.out.println(obj.add(obj.a, obj.b));
 System.out.println(obj.add(obj1.a, obj1.b));
 Demo obj2 = new Demo();
 obj2.a = 5;
 obj2.b = 6;
 System.out.println(obj.add(obj2.a, obj2.b));
 obj2 = obj1;
 System.out.println(obj.add(obj2.a, obj2.b));
 obj2.a = 15;
 obj2.b = 75;
 System.out.println(obj.add(obj.a, obj.b));
 System.out.println(obj.add(obj1.a, obj1.b));
 System.out.println(obj.add(obj2.a, obj2.b));
}
}
class Demo
{
 int a, b;
 public int add(int a, int b)
 {
  return a + b;
 }
}
```

Output:

80 ~~80100~~
80
100 1003030113090 9090

User can observe that after assigning one object reference variable to another object reference variable. The output of "a+b" is same whether the user is using any of object reference variable. This happens, because the assignment of one object reference variable to another didn't create any memory, user will refer to the same object. In other words, any copy of the object is not created, but the copy of the reference is created.

For example, obj1 = obj;

The above line of code just defines that obj1 is referring to the object, obj is referring. So, when user make changes to object using obj1 will also affect the object, b1 is referring because they are referring to the same object.

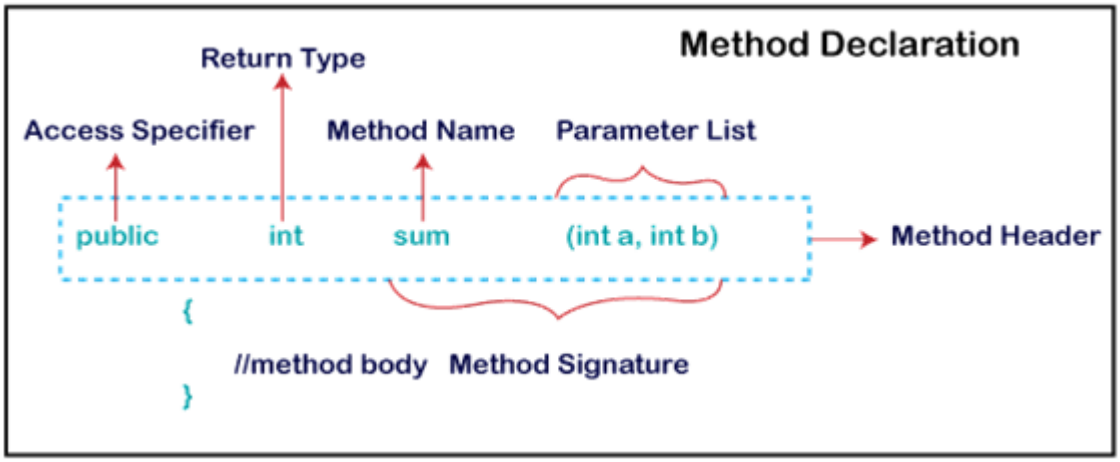
2.a) What is a method? Explain types of methods in java.
b) Explain how to declare and access methods in java.

Method :

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code.

Method: A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. Write a method once and use it many times. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it. The most important method in Java is the **main()** method.

Method Declaration: The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method: While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Methods :

There are two types of methods in Java:

- *Predefined Method*
- *User-defined Method*

Predefined Method: In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. User can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt()etc. When user call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that user write inside the method. For example, **print("Java")**, it prints Java on the console.

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

Output:

The maximum number is: 9

In the above example, we have used three predefined methods **main(), print(),** and **max()**.

User-defined Method: *The* method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Create a User-defined Method: Create a user defined method that checks the number is even or odd.

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

The user have defined the above method named findevenodd(). It has a parameter **num** of type int. The method does not return any value that's why the user has to used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

Call or Invoke a User-defined Method

Once the user has defined a method, it should be called. The calling of a method in a program is simple. When user call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user
```

```
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
```

In the above code snippet, as soon as the compiler reaches at line **findEvenOdd(num),** the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

EvenOdd.java

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
}
```

Output 1:

Enter the number: 12
12 is even

Output 2:

Enter the number: 99
99 is odd

Program that return a value to the calling method is illustrated below:.

In the following program, user have defined a method named **add()** that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b);   //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2)   //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
```

```
    return s; //returning the sum  }
    }
```

Output: The sum of a and b is= 24

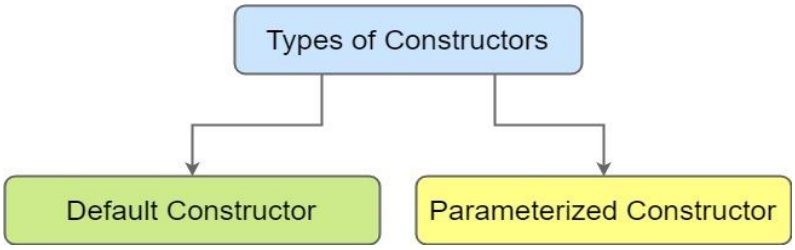
3. What is Constructor? Explain types of Constructors with example?

Constructors :

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.



It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor: There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized.

Java Default Constructor: A constructor is called "Default Constructor" when it doesn't have any parameter.The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Syntax of default constructor:

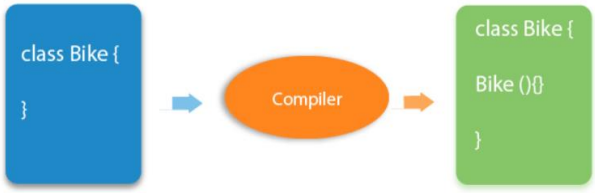
```
<class_name>()
        {
        }
```

Example:

In this example, we are creating the no-arg constructor in the Bike class.
It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1()
{
System.out.println("Bike is created");
}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:
Bike is created



Parameterized Constructor: A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example: In this example, user created the constructor of Student class that have two parameters. User can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
   //creating a parameterized constructor
   Student4(int i, String n)
   {
   id = i;
   name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   //creating objects and passing values
   Student4 s1 = new Student4(1000,"Kiran");
   Student4 s2 = new Student4(2000,"Arun");
   //calling method to display the values of object
   s1.display();
   s2.display();
   }
}
```

OUTPUT:

1000 Kiran
2000 Arun

Constructor Overloading in Java: In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

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.

Example:

```
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();
       }
    }
```

Output:

111 Karan 0
222 Aryan 25

4. Explain the following with an example.

1. **this keyword**
2. **super keyword**
3. **final keyword**

1. 'this' keyword:

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly).
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

The main purpose of using this keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use this keyword for the following purpose.

- this keyword is used to refer to current object.
- this is always a reference to the object on which method was invoked.
- this can be used to invoke current class constructor.

this can be passed as an argument to another method.

In this example, User has three instance variables and a constructor that have three parameters with same name as instance variables. Now use this to assign values of parameters to instance variables.

```
class Demo
{
 Double width, height, depth;
 Demo (double w, double h, double d)
 {
  this.width = w;
  this.height = h;
  this.depth = d;
 }
 public static void main(String[] args) {
        Demo d = new Demo(10,20,30);
        System.out.println("width = "+d.width);
        System.out.println("height = "+d.height);
        System.out.println("depth = "+d.depth);
 }
}
```

Output:

```
width = 10.0
height = 20.0
depth = 30.0
```

Here this is used to initialize member of current object. Such as, this.width refers to the variable of the current object and width only refers to the parameter received in the constructor i.e the argument passed while calling the constructor.

Calling Constructor using this keyword: Call a constructor from inside the another function by using **this** keyword. In this example, user calling a parameterized constructor from the non-parameterized constructor using the **this** keyword along with argument.

```
class Demo
{

 Demo ()
 {
        // Calling constructor
  this("Studies");
 }

 Demo(String str){

        System.out.println(str);

 }

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

Accessing Method using this keyword: This is another use of this keyword that allows to access method. We can access method using object reference too but if we want to use implicit object provided by Java then use this keyword.

In this example, user accessing getName() method using this and it works fine as works with object reference.

```
class Demo
{
        public void getName()
        {
         System.out.println("Studytonight");
        }

        public void display()
        {
         this.getName();
        }

        public static void main(String[] args) {
                Demo d = new Demo();
                d.display();
        }
}
```

Return Current Object from a Method: In such scenario, where user want to return current object from a method then user can use this to solve this problem. In this example, user created a method display that returns the object of Demo class. To return the object, used this keyword and stored the returned object into Demo type reference variable. The user used that returned object to call getName() method and it works fine.

```
class Demo
{
        public void getName()
```

```
        {
         System.out.println("Studies");
        }

        public Demo display()
        {
         // return current object
         return this;
        }

        public static void main(String[] args) {
                Demo d = new Demo();
                Demo d1 = d.display();
                d1.getName();
    }}
```

2. 'Super' Keyword

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 can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable: Super keyword is used 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.

2) 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.

3) 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

3. 'final' Keyword

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

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 speedlimit, 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

Is final method inherited?

Ans) 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...

5. Demonstrate the types of inheritance with example?

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs. (Object Oriented programming system). The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When inheritance is used from an existing class, reuse methods and fields of the parent class

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Use of inheritance in java

- For Method Overriding
- For Code Reusability.

The syntax of Java Inheritance

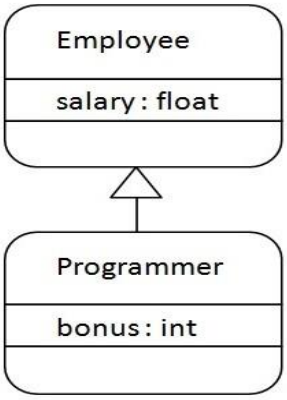
```
class Subclass-name extends Superclass-name
{
  //methods and fields
```

```
}
```

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 a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

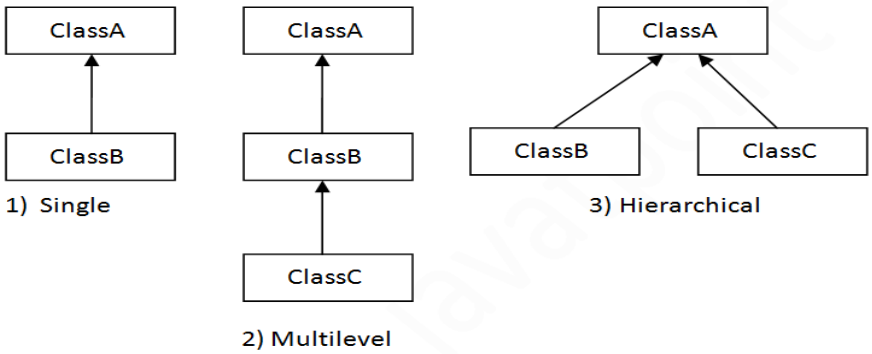
Output:

```
Programmer salary is: 40000.0
Bonus of programmer is: 10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

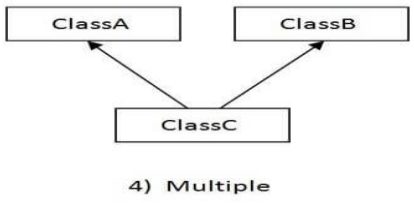
Types of inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only.

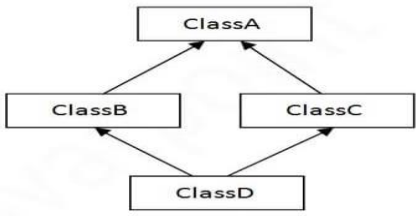


Multiple inheritance is not supported in Java through class. When one class inherits multiple classes, it is known as multiple inheritance.

For Example:



4) Multiple



5) Hybrid

Single Inheritance Example: When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the 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();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example: When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example: When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
```

```
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

meowing...
eating...

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
 }
}
```

Output: Compile Time Error

The inheritance is the core and more useful concept Object Oriented Programming. It proWith inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.vides lot of benefits and few of them are listed below.

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

6.a) what is an abstract class in java
b) what are the member access rules

Member Access Rules

The member access rules determine whether a sub class can use a property of it's super class or it can only access or it can neither access nor access. There are two level of access control:

- At the top level: public or package-private
- At the member level: public, private, protected

A class may be declared with the 'public' modifier, in that case that class is visible to all classes everywhere. At the member level, there are three different access modifiers are there: 'private', 'protected' and 'public'.

private : If **private access modifier** is applied to an instance variable, method or with a constructor in side a class then they will be accessed inside that class only not out side of the class.

for example:
```
class A
{
        private int x=10;
}
class B extends A
{
        int y=20;
        System.out.println(x);//Illegal access to x ;
}
```

If you make any class constructor private, you can't create the instance/object of that class from outside the class. for example:

```
class A
{
        int x;
        private A(int k) // private constructor
        {
                x=k;
        }
}
class Test
{
        public static void main(String args[])
        {
                A ob=new A(10); //Compile time error
        }
}
```

protected: If **protected access modifier** is applied to an instance variable, method or with a constructor in side a class then they will be accessed inside the package only in which class is present and, in addition, by a sub class in another package.

public: The class, variable, method or a constructor with public access modifier can be accessed from anywhere.

Access Modifiers	Within a Class	Within a Package	Outside package by sub class only	Outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Abstract Classes

Abstract class in Java: **class which is declared as abstract is known as an** abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.**

Points to Remember

Following are some important observations about abstract classes in Java.

- An instance of an abstract class cannot be created.
- Constructors are allowed.
- We can have an abstract class without any abstract method.
- Abstract classes cannot have final methods because when you make a method final you cannot override it but the abstract methods are meant for overriding.
- We are not allowed to create object for any abstract class.
- We can define static methods in an abstract class

Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda extends Bike
{
void run(){System.out.println("running safely");
}
public static void main(String args[]){
Honda h = new Honda();
 h.run();
}
}
```

Output: running safely

(*Optional Question)CONSTRUCTOR AND CALLING SEQUENCE

Order of Execution of Constructors in Java Inheritance

Constructors in Java

A constructor in Java is similar to a method with a few differences. Constructor has the same name as the class name. A constructor doesn't have a return type.

A Java program will automatically create a constructor if it is not already defined in the program. It is executed when an instance of the class is created.

A constructor cannot be static, abstract, final or synchronized. It cannot be overridden.

Java has two types of constructors:

- Default constructor
- Parameterized constructor

What is the order of execution of constructor in Java inheritance?

While implementing inheritance in a Java program, every class has its own constructor. Therefore the execution of the constructors starts after the object initialization. It follows a certain sequence according to the class hierarchy. There can be different orders of execution depending on the type of inheritance.

Different ways of the order of constructor execution in Java

1. Order of execution of constructor in Single inheritance

In single level inheritance, the constructor of the base class is executed first.

OrderofExecution1.java

```
/* Parent Class */
class ParentClass
{
   /* Constructor */
   ParentClass()
   {
      System.out.println("ParentClass constructor executed.");
   }
}

/* Child Class */
class ChildClass extends ParentClass
{
   /* Constructor */
   ChildClass()
   {
      System.out.println("ChildClass constructor executed.");
   }
}

public class OrderofExecution1
{
   /* Driver Code */
   public static void main(String ar[])
   {
      /* Create instance of ChildClass */
      System.out.println("Order of constructor execution...");
      new ChildClass();
   }
}
```

Output:

```
Order of constructor execution...
ParentClass constructor executed.
ChildClass constructor executed.
```

In the above code, after creating an instance of ***ChildClass*** the ***ParentClass*** constructor is invoked first and then the ***ChildClass.***

2. Order of execution of constructor in Multilevel inheritance

In multilevel inheritance, all the upper class constructors are executed when an instance of bottom most child class is created.

OrderofExecution2.java

```
class College
{
   /* Constructor */
   College()
   {
      System.out.println("College constructor executed");
   }
}

class Department extends College
{
   /* Constructor */
```

```
    Department()
    {
        System.out.println("Department constructor executed");
    }
}

class Student extends Department
{
    /* Constructor */
    Student()
    {
    System.out.println("Student constructor executed");
    }
}
public class OrderofExecution2
{
        /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of Student class */
        System.out.println("Order of constructor execution in Multilevel inheritance...");
        new Student();
    }
}
```

Output:

```
Order of constructor execution in Multilevel inheritance...
College constructor executed
Department constructor executed
Student constructor executed
```

In the above code, an instance of ***Student*** class is created and it invokes the constructors of ***College, Department*** and ***Student*** accordingly.

3. Calling same class constructor using this keyword

Here, inheritance is not implemented. But there can be multiple constructors of a single class and those constructors can be accessed using **this** keyword.

OrderofExecution3.java

```
public class OrderofExecution3
{
    /* Default constructor */
    OrderofExecution3()
    {
        this("CallParam");
        System.out.println("Default constructor executed.");
    }
    /* Parameterized constructor */
    OrderofExecution3(String str)
    {
        System.out.println("Parameterized constructor executed.");
    }
    /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of the class */
        System.out.println("Order of constructor execution...");
        OrderofExecution3 obj = new OrderofExecution3();
    }
}
```

Output:

```
Order of constructor execution...
Parameterized constructor executed.
Default constructor executed.
```

In the above code, the parameterized constructor is called first even when the default constructor is called while object creation. It happens because ***this*** keyword is used as the first line of the default constructor.

4. Calling superclass constructor using super keyword

A child class constructor or method can access the base class constructor or method using the super keyword.

OrderofExecution4.java

```
/* Parent Class */
class ParentClass
{
   int a;
   ParentClass(int x)
   {
      a = x;
   }
}

/* Child Class */
class ChildClass extends ParentClass
{
   int b;
   ChildClass(int x, int y)
   {
      /* Accessing ParentClass Constructor */
      super(x);
      b = y;
   }
   /* Method to show value of a and b */
   void Show()
   {
      System.out.println("Value of a : "+a+"\nValue of b : "+b);
   }
}

public class OrderofExecution4
{
   /* Driver Code */
   public static void main(String ar[])
   {
      System.out.println("Order of constructor execution...");
      ChildClass d = new ChildClass(79, 89);
      d.Show();
   }
}
```

Output:

```
Order of constructor execution...
Value of a : 79
Value of b : 89
```

In the above code, the ***ChildClass*** calls the ***ParentClass*** constructor using a ***super*** keyword that determines the order of execution of constructors.

7. What is Method overloading explain with example

Method overloading

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.

Advantage of method overloading: *Method overloading* increases the readability of the program.

Different ways to overload the method: There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. 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.In this example, 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));
}}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments: In this example, we have created two methods that differs 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
```

Why Method Overloading is not possible by changing the return type of method only?

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

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.

```
class TestOverloading4{
public static void main(String[] args)
        {System.out.println("main with String[]");}
public static void main(String args)
        {System.out.println("main with String");}
public static void main()
        {System.out.println("main without args");}
}
```

Output: main with String[]

8. What is Method overriding explain with example

Method overriding

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 a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding: Understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output: Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding: In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}
  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
   }
}
```

Output: Bike is running safely

9. What are the differences between Method overloading and overriding?

Differences between method overloading and method overriding:

Method overloading	Method overriding
1. Writing 2 or more methods with same name but with different signatures is called method overloading.	1. Writing 2 or more methods with same name and same signatures is called method overriding.
2. Method overloading is done in the same class	2. Method overriding is done in super and sub classes.
3. In method overloading, method return type can be same or different.	3. In method overriding, method return types should also be same.
4. Method overloading is code refinement. Same method is refined to perform a different task.	4. Method overriding is code replacement. The sub class method overrides (replaces) the super class method.
5. Accessibility mode of the methods has no bearing to implement method overloading.	5. Accessibility mode of the methods has bearing to implement method overloading.
6. Number of exceptions thrown does not influence overloading.	6. Number of exceptions thrown does influence overloading.
7. It implements static (compile time) polymorphism.	7. It can participate in the implementation of dynamic (run time) polymorphism.

10. a) Define an interface?
b) Explain Implementing and Accessing interface with example

INTERFACES :

An interface is a class like java language construct. An interface is similar to a class in the following areas.

1. It is compiled to .class file
2. Its reference can be created.
3. It is a user defined data type.
4. It contains variables and methods.
5. It can act as a parent to a class.

Defining an interface

It is created with a keyword 'interface'. The general form of an interface definition is:

```
interface interfacename
{
Variables declaration;
Methods declaration;
}
```

- An interface can have only constants and abstract methods.
- Interface members implicitly public. No other mode is allowed.
- Interface methods are implicitly abstract.
- Interface variables are implicitly static and final.

E.g.:

```
interface Item
{
public static final int code=1001;
public static final String name=”fan”;
public abstract void display();
}
```

In the above example code and name are the constants. We need not use the 3 keywords for it. They are implicitly public, static and final. For the display() method public and abstract are implicitly available.

Implementing interface

A class can inherit from an interface. When a class is inheriting from an interface "implements" keyword is used instead of "extends" keyword. If a class is inheriting from an interface, it is the responsibility of that sub class to implement (override) all the abstract methods of the interface. Otherwise, the class should be declared abstract. One class can inherit from any number of interfaces.

1. E.g.:

```
interface I{void y();}
class A implements I{
public void y()
{
System.out.println("Some functionality");
}
}//sub class
class Interface
{
public static void main(String args[])
{
A a=new A();
a.y();
}
}
```

O/P:

Some functionality

2. E.g.:

```
interface I1
{
void x();
}
interface I2
{
void y();
}
class A implements I1,I2
{
public void x()
```

```
{
System.out.println("Some functionality done in x");
}
public void y()
{System.out.println("Some functionality done in y");
}
}//sub class
class Interface
{
public static void main(String args[])
{
A a=new A();
a.x();
a.y();
}}
```

O/P:

Some functionality done in x

Some functionality done in y

Accessing interface properties

Accessing implementations through interface references:

An interface reference can be created. But its instance cannot be created. Interface reference can refer to sub class objects. Through the interface reference we can access the abstract methods.

1. E.g.:

```
interface I
{
void y();
}
class A implements I
{
public void y()
{
System.out.println("Some functionality");
}
}//sub class
class Interface
{
public static void main(String args[])
{
I i=new A();
i.y();
}
}
```

O/P:

Some functionality

2. E.g.:

```
interface I1
{
void x();
}
interface I2
{
void y();
}
class A implements I1,I2
{
public void x()
{
System.out.println("Some functionality done in x");
}
public void y()
{
System.out.println("Some functionality done in y");
}
}//sub class
class Interface
{
public static void main(String args[])
{
I1 i1=new A();
i1.x();
I2 i2=new A();
i2.y();
}}
```

O/P:

Some functionality done in x

Some functionality done in y

Extending interfaces

Like classes, interfaces can also be extended. That is, an interface can be sub interfaced from other interfaces. The new sub interface will inherit all the members of the super interface in the manner similar to sub classes. This is achieved using the 'extends' keyword as shown below.

1. E.g.:

```
interface I1
{
void x();
}
interface I2 extends I1
{
void y();
}
class A implements I2
{
public void x()
{
System.out.println("Some functionality done in x");
}
public void y()
{
System.out.println("Some functionality done in y");
}
}//sub class
class Interface
{
public static void main(String args[])
{
I2 i=new A();
i.x();
i.y();
}
}
```

O/P:

Some functionality done in x
Some functionality done in y

2. E.g.:

```
interface I1
{
void x();
}
interface I2 extends I1
{
void y();
}
interface I3
{
void z();
}
class A implements I2,I3
{
public void x()
{
System.out.println("Some functionality done in x");
}
public void y()
{
System.out.println("Some functionality done in y");
}
public void z()
{
System.out.println("Some functionality done in z");
}
}//sub class
class Interface
{
public static void main(String args[])
{
I2 i=new A();
i.x();
i.y();
I3 i3=new A();
i3.z();
}
}
```

O/P:

Some functionality done in x
Some functionality done in y
Some functionality done in z

11. How to implement multiple inheritance in java. Explain with an example program

Multiple inheritance: Producing sub classes from multiple super classes are called multiple inheritance. In this case, there will be more than one super class and there can be one or more sub classes. There is no multiple inheritance in java. The following are the reasons:

Multiple inheritance leads to confusion to the programmer. For example, class A has got a member x and class B has also got a member x. When another class C extends both the classes, then there is a confusion regarding which copy of x is available in C

The programmer can achieve multiple inheritance by using interfaces.

1. E.g.:

```
// multiple inheritance using interfaces
interface I1
{
void x();
}
interface I2
{
void x();
}
class A implements I1,I2
{
public void x()
{
System.out.println("Some functionality done in x");
}
}//sub class
class Interface
{
public static void main(String args[])
{
A a=new A();
a.x();
}
}
```

O/P:

Some functionality done in x

2. E.g.:

```
interface Father
{
float ht=6.2f;
void height();
}
interface Mother
{
float ht=5.8f;
void height();
}
class Child implements Father,Mother
{
public void height()
{
//child got average height of its parents
float ht=(Father.ht+Mother.ht)/2;
System.out.println("child's height="+ht);
}
}
class Multi
{
public static void main(String args[])
{
Child ch=new Child();
ch.height();
}
}
```

O/P:

Child's height=6.0

12. Explain the uses of final keyword with example programs.

'final' Keyword

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

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.

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 speedlimit, 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

Is final method inherited?

Ans) 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...

(*Optional Question)

Interfaces v/s abstract classes:

Similarities between an abstract class and an interface:

1. Both cannot be instantiated, i.e. object cannot be created.
2. Both are compiled to .class files and hence act as user defined data types. Consequently their references can be created.
3. Both can zero or more abstract methods.
4. Both can be inherited.

Differences between an abstract class and an interface:

Abstract class	Interface
It is partially abstract.	It is completely abstract.
It can have constructor.	It cannot have constructor.
Members are implicitly default. Other accessibility mode also allowed.	Members are implicitly public. Other accessibility mode not allowed.
Abstract class cannot support multiple inheritance.	It can support multiple inheritance.
An abstract class contains some abstract methods and some concrete methods.	An interface contains only abstract methods.
An abstract class can contain instance variables also.	An interface cannot contain instance variables. It contains only constants.