What is the purpose of copy constructor?

Copy Constructors is a type of constructor which is **used to create a copy of an already existing object of a class type**. It is usually of the form X (X&), where X is the class name. The compiler provides a default Copy Constructor to all the classes.

Q.2:How to differentiate a class from an object?

Class is a **blueprint or template from which objects are created**. Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. Class is a group of similar objects. Object is a physical entity.

Q.3:What is polymorphism?

Polymorphism in Java is **the ability of an object to take many forms**. To simply put, polymorphism in java allows us to perform the same action in many different ways. ... Polymorphism is a feature of the object-oriented programming language, Java, which allows a single task to be performed in different ways.

Q.4:

JavaExceptionExample.java

```
public class JavaExceptionExample{
  public static void main(String args[]){
        try{
           //code that may raise exception
           int data=100/0;
        }catch(ArithmeticException e){System.out.println(e);}
        //rest code of the program
        System.out.println("rest of the code...");
      }
    }
```

Q.5:

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Q.7:

A class doesn't need a constructor. A default constructor is not needed if the object doesn't need initialization.

Under some circumstances a constructor won't be called, period. As soon as you write a constructor you don't have a POD class, so now the constructor will be called.
Whether it is a good or bad thing to have an object running around that contains random, uninitialized data is a different question entirely.

Q.8:

The class members declared as private can be accessed only by the functions inside the class. Protected access modifier is similar to that of private access modifiers. Only the member functions or the friend functions are allowed to access the private data members of a class.

Q.9:

Information hiding is **the primary criteria of system modularization** and should be concerned with hiding the critical decisions of OOP designing. Information hiding isolates the end users from the requirement of intimating knowledge of the design for the usage of a module.

Q.10:

The **assignment operator (=)** can be used to assign an object to another of the same type. The assignment operator (=) can be used to assign an object to another of the same type. Objectsmay be passed by value to or returned by value from functions.

Q.11:

Sequence Containers

In standard template library they refer to the group of container class template, we use to them store data. One common property as the name suggests is that elements can be accessed sequentially.
Each of the following containers use different algorithm for data storage thus for different operations they have *different speed.* And all the elements in the containers should be of same type.

Associative containers

In standard template libraries, they refer to the group of class templates used to implement associative arrays. They are used to store elements but have some constraints placed on their elements. And two important characteristics of these containers are

Q.12:

Dynamic Memory Allocation

11. We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
12. For this reason, dynamic allocation requires two steps:
    1. Creating the dynamic space.
    2. Storing its **address** in a **pointer** (so that the space can be accesed)
13. To dynamically allocate memory in C++, we use the **new** operator.
14. De-allocation:

    1. Deallocation is the "clean-up" of space being used for variables or other data storage
    2. Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
    3. It is the programmer's job to deallocate dynamically created space
    4. To de-allocate dynamic memory, we use the `delete` operator
    5.

Long Question:

Q.1:Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.Example 1: Prefix ++ Increment Operator Overloading with no return type

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

```
package pack;

public class A{
protected void msg(){System.out.println("Hello");}

}
```

```
//save by B.java
package mypack;

import pack.*;


class B extends A{
  public static void main(String args[]){

   B obj = new B();
   obj.msg();

  }
}
```

Public

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
save by A.java


package pack;
public class A{

public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Q(b).

```
package Amjad;

public class Operator {

        static int Opr(int a)
        {
        return ++a;
        }
        static double Opr(double b){
            return ++b;
        }
    public static void main(String args[]) {
        System.out.println(Operator.Opr(1));
        System.out.println(Operator.Opr(2.1));

    }
}
```

Postfix:

```
package Amjad;


public class Operator {


     static int Opr(int a)

     {

     return --a;

     }

     static double Opr(double b){

         return --b;
```

```
    }

  public static void main(String args[]) {

    System.out.println(Operator.Opr(1));

    System.out.println(Operator.Opr(2.1));



  }
}
```

Q.3

Inheritance in Java

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 you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

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

Why use inheritance in java

58. For Method Overriding (so runtime polymorphism can be achieved).
59. For Code Reusability.

Terms used in Inheritance

60. Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
61. Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
62. Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

63. Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

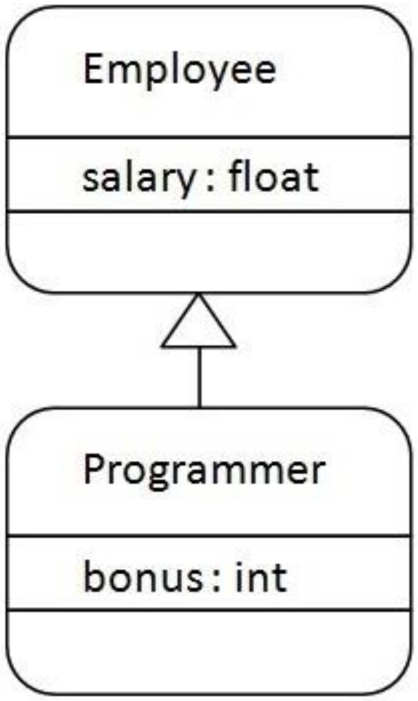
The syntax of Java Inheritance

```
64. class Subclass-name extends Superclass-name
65. {
66.    //methods and fields
67. }
```

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);
}

}
```

Types of inheritance in java

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. We will learn about interfaces later.

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.
File: TestInheritance.java

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

}
class Dog extends Animal{

void bark(){System.out.println("barking...");}
}
```

```java
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

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.

```java
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.

File: TestInheritance3.java

```java
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 3(b):

Difference between Virtual function and Pure virtual function in C++

Virtual Function in Java

A virtual function or virtual method in an OOP language is a function or method used to override the behavior of the function in an inherited class with the same signature to achieve the polymorphism.

When the programmers switch the technology from C++ to Java, they think about where is the virtual function in Java. In C++, the virtual function is defined using the **virtual** keyword, but in Java, it is achieved using different techniques. See Virtual function in C++.

Java is an object-oriented programming language; it supports OOPs features such as polymorphism, abstraction, inheritance, etc. These concepts are based on objects, classes, and member functions.

By default, all the instance methods in Java are considered as the Virtual function except final, static, and private methods as these methods can be used to achieve polymorphism.

How to use Virtual function in Java

The virtual keyword is not used in Java to define the virtual function; instead, the virtual functions and methods are achieved using the following techniques:

- We can override the virtual function with the inheriting class function using the same function name. Generally, the virtual function is defined in the parent class and override it in the inherited class.
- The virtual function is supposed to be defined in the derived class. We can call it by referring to the derived class's object using the reference or pointer of the base class.

- A virtual function should have the same name and parameters in the base and derived class.
- For the virtual function, an IS-A relationship is necessary, which is used to define the class hierarchy in inheritance.
- The Virtual function cannot be private, as the private functions cannot be overridden.
- A virtual function or method also cannot be final, as the final methods also cannot be overridden.
- Static functions are also cannot be overridden; so, a virtual function should not be static.
- By default, Every non-static method in Java is a virtual function.
- The virtual functions can be used to achieve oops concepts like runtime polymorphism.

Let's understand it with some examples:

Parent.Java:

```java
  class Parent {
void v1() //Declaring function
{
System.out.println("Inside the Parent Class");
}
}
```

Child.java:

```java
public class Child extends Parent{
          void v1() // Overriding function from the Parent class
          {
          System.out.println("Inside the Child Class");
```

```
    }
    public static void main(String args[]){
    Parent ob1 = new Child(); //Refering the child class object using the parent class
    ob1.v1();
    }
    }
```

11. From the above example, we have used the function v1() as the virtual function by overriding it in the Child class.
12. In Java Every non-static, non-final, and public method is a virtual function. These methods can be used to achieve polymorphism. The methods that can not be used to achieve the polymorphism never be virtual function.
13. static, final, and private method never be the virtual function. We can not call the static method by using the object name or class name. Even if we try, it will not be able to achieve the polymorphism.

14. Java Interfaces as Virtual Function

15. An interface in Java is a blueprint of a class; it holds static constants and abstract methods. All Java Interfaces are considered virtual functions, as they depend on the implementing classes to provide the method implementation

Consider the below example to understand the behavior of the interface:

```
interface Car{
void print();
}
class BMW implements Car{
public void print(){System.out.println("BMW X7");}
public static void main(String args[]){
BMW obj = new BMW();
obj.print();
 }
}
```

Output:

From the above example, we can see the interface's method is executed using the implementing class BMW.

Hence, we can also achieve polymorphism using the Interfaces.

Pure Virtual Function

Pure Virtual Function

A virtual function for which we are not required implementation is considered as pure virtual function. For example, Abstract method in Java is a pure virtual function. Consider the below example:

```
abstract class Animal {
        final void bark(){
        System.out.println("Dog");
        }
        abstract void jump(); // Abstract Method (Pure Virtual Function)
}
    class MyPet extends Animal{
        void jump(){
        System.out.println("MyPet is so sweet");
        }
}
public class Demo {
        public static void main(String args[]){
        Animal ob1 = new MyPet();
        ob1.jump();
        }
        }
```

18. From the above example, the jump() method is a pure virtual function.

19. Runtime Polymorphism

20. Runtime polymorphism is a process in which a call to an overridden method is resolved at runtime rather than compile time.

In runtime polymorphism, we will use the reference variable to call a method instead of a reference variable.

The virtual functions can be used to achieve runtime polymorphism.

Consider the below example to understand how the virtual function is used to achieve the runtime polymorphism:

```
class Javatpoint{
public void show() //Virtual Function
{
System.out.println("welcome to JavaTpoint");
}
}
class Training extends Javatpoint{
public void show(){
System.out.println("Best Java Training Institute");
}
public static void main(String args[]){
Javatpoint ob1 = new Training();
ob1.show();
}}
```

Difference between virtual function and pure virtual function in C++

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`    // code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`    = 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Q.4

```
package Amjad;


public class Distance {
private int feet;
private int inch;
public Distance()
{
  this(0,0);
}
public Distance(int f,int i)
{
  feet=f;
  inch=i;
}
public void setFeet(int f)
```

```
{
  feet=f;
}
public void setInch(int i)
{
  inch=i;
}
public int getFeet()
{
  return feet;
}
public int getInch()
{
  return inch;
}
public void subtract(Distance d1, Distance d2)
```

```
{
  this.feet=d1.feet-d2.feet;
  this.inch=d1.inch-d2.inch;

if(this.inch<1)
{
  int f=this.inch/12;
  this.inch=this.inch%12;
  feet -=f;
}
}
public void PrintDistance()
{

System.out.println("Distance:"+feet+"Feet"+inc
h+"inch");
}
```

}

Q.5(a)

```
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));
}}
```