Name Irfan Ullah

ID 16332

Class BS Software Engineering

Section (B)

Subject Object Oriented Programming

Semester 2nd

Date jun 29th, 2020

Submitted to Sir M Ayub khan

Q NO 1 PART A)

ANSWER:

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

USE OF ACCESS MODIFIER:

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Java provides a rich set of modifiers. They are used to control access mechanism and also provide information about class functionalities to JVM. They are divided into two categories :

ACCESS MODIFIERS :

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level (called package-private).

HOW THEY WORK:

PUBLIC:

When a member of a class is modified by **public**, then that member can be accessed by any other code.

PRIVATE:

When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
Now you can understand why main ( ) has always been preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

PROTECTED:

Protected applies only when inheritance **is involved.**

PRIVATE:

The private access modifier is accessible only within the class.

The private access modifier is specified using the keyword private.

- The methods or data members declared as private are accessible only within the class in which they are declared.

- Any other class of same package will not be able to access these members.
- Top level Classes or interface can not be declared as private because
    1. Private means “only visible within the enclosing class”.
    2. Protected means “only visible within the enclosing class and any subclasses”

  Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

PROGRAM:

```
package paper;


public class A {
        private int data=30;
        private void msg() {
                System.out.println("hello world");
        }

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

OUTPUT:

Compilation time error because we access the private class in another class.

EXPLANATION:

In this example, we will create two classes A and B within same package paper. We will declare a method in class A as private and try to access this method from class B and in the result error occur because private access modifier not access outside the private class.

ROLE OF PRIVATE CONSTRUCTOR:

If you make any class constructor private, you cannot create the instance of that class from outside the class.

PROGRAM:

```
package paper;


class A {
        private A() {}
        //private constructor
        void msg() {
                System.out.println("hello world");
        }

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

OUTPUT:

Compilation error.

EXPLANATION:

Error occur in the above program because we access the private access modifier in another class.

DEFAULT:

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

When no access modifier is specified for a class , method or data member – It is said to be having the default access modifier by default.

- The data members, class or methods which are not declared using any access modifiers i.e. Having default access modifier are accessible only within the same package.

PROGRAM:

```
package p1;


class irfan {


        void display() {
                System.out.println("hello world");
        }


}
package p2;
class simplenew{
        public static void main(String args[])
        {
                REHMAT obj=new irfan();
                                obj.display();
        }
}
```

OUTPUT:

Compilation error.

EXPLANATION:

In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second package.in the result compilation error.

ANSWER PART B):

PROGRAM:

DEFAULT ACCESS MODIFIER:

```
package paper1;


public class paper {
	//this is default access modifier
	    int a=50;
	    void display() {
	        System.out.println("HELLO REHMAT");
	}

}
package paper1;


public class subclass {
	public static void main(String[] args) {
		// TODO Auto-generated method stub

		subclass sub1=new subclass();
		sub1.display();
}
```

OUTPUT:

Compilation error.

PRIVATE ACCESS MODIFIER:

```
package paper;


public class A {
        private int data=60;
        private void msg() {
                System.out.println("hello world");
        }

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

OUTPUT:

Compilation error.

EXPLANATION:

The above program show how to use default and access modifier in program and how they work. First I create default access modifier which access out side the class and the other is private which cannot access outside the class. In the result compilation error occur.

ANSWER NO 2:

PUBLIC ACCESS MODIFIER:

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

The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods or data members which are declared as public are accessible from every where in the program. There is no restriction on the scope of a public data members.

EXAMPLE PROGRAM:

```
package paper2;


public class A {
	public void msg()
	{
		System.out.println("hello rehmat khan  ");
	}


}
package subpaper2;


import paper2.A;


public class B {


	public static void main(String[] args) {
		// TODO Auto-generated method stub
```

```
        A obj=new A ();
        obj.msg();
    }

}
```

Output:

hello rehmat khan

EXPLANATION:

In the above program I create two classes one is A and other is B with different packages. In A class I use public access modifier which means to access this class method to another class by simple calling this method. In the result hello sami is print on the screen.

PROTECTED ACCESS MODIFIER:

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

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

EXAMPLE PROGRAM:

```
package p3;


public class irfan {


    protected void display()
    {
        System.out.println("hello world");
    }
```

```
}
package p4;

import p3.irfan;

class khan extends irfan {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		khan obj=new khan();
		obj.display();
	}

}
```

OUTPUT:

Hello world

EXPLANATION:

The above program how to use protected access modifier. For this I create two classes one is sami and other is khan with different packages. In sami class I use protected access modifier method which access in other class. In the result Hello world print on the screen.

Part b answer:

PROGRAM:

```
package p3;

public class irfan {

	protected void display()
	{
		System.out.println("hello world");
	}
	public void message()
	{
		System.out.println("Hello irfan");
	}

}
package p4;

import p3.irfan;

class saad extends irfan  {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
```

```
        saad obj=new saad();
        obj.display();
        saad mss=new saad();
        mss.message();
    }

}
```

OUTPUT:

Hello world

Hello irfan

EXPLANATION:

The above program how to use protected and access modifier. For this I create a java project which name is p3. After this I create two class one is rehmat and other is khan with different packages. In sami class I used two method one method is protected access modifier and the other is public access modifier one name is display and the other is message and both of them is void type which not return anything else. These method access in other class which name is khan by simple calling the method. In the result Hello world and Hello rehmat khan is print on the screen.

Answer no 3 :

INHERITANCE:

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

EXTENDS KEYWORD:

Extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

SYNTAX:

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

IMPORTANT TERMINOLOGY:

- **SUPER CLASS:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **SUB CLASS:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **REUSABILITY:** Inheritance supports the concept of "reusability", i.e. When we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

EXAMPLE:

```
package p6;
```

```
//Java program to illustrate the
//concept of inheritance

//base class
class Bicycle
{
 // the Bicycle class has two fields
 public int gear;
 public int speed;

 // the Bicycle class has one constructor
 public Bicycle(int gear, int speed)
 {
    this.gear = gear;
    this.speed = speed;
 }

 // the Bicycle class has three methods
 public void applyBrake(int decrement)
 {
    speed -= decrement;
 }

 public void speedUp(int increment)
 {
    speed += increment;
 }
```

```
 // toString() method to print info of Bicycle
 public String toString()
 {
    return("No of gears are "+gear
           +"\n"
           + "speed of bicycle is "+speed);
 }
}

//derived class
class MountainBike extends Bicycle
{

 // the MountainBike subclass adds one more field
 public int seatHeight;

 // the MountainBike subclass has one constructor
 public MountainBike(int gear,int speed,
                  int startHeight)
 {
    // invoking base-class(Bicycle) constructor
    super(gear, speed);
    seatHeight = startHeight;
 }

 // the MountainBike subclass adds one more method
```

```
 public void setHeight(int newValue)
 {
    seatHeight = newValue;
 }

 // overriding toString() method
 // of Bicycle to print more info
 @Override
 public String toString()
 {
    return (super.toString()+
          "\nseat height is "+seatHeight);
 }

}

//driver class
public class Test
{
 public static void main(String args[])
 {

    MountainBike mb = new MountainBike(3, 100, 25);
    System.out.println(mb.toString());

 }
}
```

OUTPUT:

No of gears are 3
speed of bicycle is 100

seat height is 25

EXPLANATION:

In above program, when an object of mountainbike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why, by using the object of the subclass we can also access the members of a superclass.
Please note that during inheritance only object of subclass is created, not the superclass.

IMPORTANT FACTS ABOUT INHERITANCE IN JAVA:

- **DEFAULT SUPERCLASS:** Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **SUPERCLASS CAN ONLY BE ONE:** A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- **INHERITING CONSTRUCTORS:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **PRIVATE MEMBER INHERITANCE:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

Why use inheritance:

- The most important use is the reusability of code. The code that is present in the parent class doesn't need to be written again in the child class.
- To achieve runtime polymorphism through method overriding.

Part b 3:

PROGRAM:

Inheritance is one of the key features of OOP (Object-oriented Programming) that allows us to define a new class from an existing class. For example,

```
class Animal
{
   // eat() method
   // sleep() method
}
class Dog extends Animal
{
   // bark() method
}
```

In Java, we use the `extends` keyword to inherit from a class. Here, we have inherited the `Dog` class from the `Animal` class.

The `Animal` is the superclass (parent class or base class), and the `Dog` is a subclass (child class or derived class). The subclass inherits the fields and methods of the superclass.

IS-a relationship:

Inheritance is an **is-a** relationship. We use inheritance only if an **is-a** relationship is present between the two classes.

Here are some examples:

- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal

EXAMPLE PROGRAM:

```
package animal;
class Animal {

    public void eat() {
      System.out.println("I can eat");
    }

    public void sleep() {
      System.out.println("I can sleep");
    }
  }

  class Dog extends Animal {
    public void bark() {
      System.out.println("I can bark");
    }
  }

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

      Dog dog1 = new Dog();

      dog1.eat();
      dog1.sleep();
```

```
        dog1.bark();
    dog1.type = "mammal";
        dog1.setColor("black");
        dog1.displayInfo(dog1.getColor());
      }
    }
```

Output:

```
I can eat
I can sleep
I can bark
I am a mammal
My color is black
```

Explanation:

Here, we have inherited a subclass `Dog` from superclass `Animal`. The `Dog` class inherits the methods `eat()` and `sleep()` from the `Animal` class.

Hence, objects of the `Dog` class can access the members of both the `Dog` class and the `Animal` class.

2ND PROGRAM:

```
package animal;
class Animal {
}

class Mammal extends Animal {
}
```

```
class Reptile extends Animal {
}

public class Dog extends Mammal {

  public static void main(String args[]) {
    Animal a = new Animal();
    Mammal m = new Mammal();
    Dog d = new Dog();

    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}

interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

  public static void main(String args[]) {
    Mammal m = new Mammal();
    Dog d = new Dog();

    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
```

```
  }
}
```

Output:

true

true

true

Explanation:

in the above program how to inherit animal in java for this I create three classes one is main class which name is animal and other is subclasses which name is mamel, reptile and last one is public class dog. Inheritance show the features of the dog.

ANSWER NO 4 PART A:

POLYMORPHISM:

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

EXAMPLE PROGRAM:

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal –

Example

```
Deer d = new Deer();

Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

REAL LIFE EXAMPLE OF POLYMORPHISM:

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

PART B Q NO4:

PROGRAM:

```
package paper2;
/* File name : Employee.java */
public class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }
```

```
  public String getAddress() {
    return address;
  }

  public void setAddress(String newAddress) {
    address = newAddress;
  }
  public int getNumber() {
            return number;
          }
        }
/* File name : Salary.java */
public class Salary extends Employee {
  private double salary; // Annual salary

  public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
  }

  private void setSalary(double salary2) {
        // TODO Auto-generated method stub

}

public void mailCheck() {
```

```
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName()
    + " with salary " + salary);
  }

  public double getSalary() {
    return salary;
  }
  public void setSalary1(double newSalary) {
            if(newSalary >= 0.0) {
              salary = newSalary;
            }
          }

          public double computePay() {
            System.out.println("Computing salary pay for " + getName());
            return salary/52;
          }
        }
/* File name : VirtualDemo.java */
public class VirtualDemo {

  public static void main(String [] args) {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();
```

```
    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

Output:

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Explanation:

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

ANSWER NO 5 PART A;

Abstraction:

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the

protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class :

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstraction in Real Life:

Abstraction is present in almost all the real life machines.

- Your car is a great example of abstraction. You can start a car by turning the key or pressing the start button. You don't need to know how the engine is getting started, what all components your car has. The car internal implementation and complex logic is completely hidden from the user.
- We can heat our food in Microwave. We press some buttons to set the timer and type of food. Finally, we get a hot and delicious meal. The microwave internal details are hidden from us. We have been given access to the functionality in a very simple manner.

Abstraction in OOPS:

Objects are the building blocks of Object-Oriented Programming. An object contains some properties and methods. We can hide them from the outer world through access modifiers. We can provide access only for required functions and properties to the other programs. This is the general procedure to implement abstraction in OOPS.

Example:

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name : Employee.java */
public abstract class Employee {
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public double computePay() {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

PART B Q NO 5:

PROGRAM:

```
package paper55;
//Java program to illustrate the
//concept of Abstraction
abstract class Shape
{
 String color;

 // these are abstract methods
 abstract double area();
 public abstract String toString();

 // abstract class can have constructor
 public Shape(String color) {
    System.out.println("Shape constructor called");
    this.color = color;
 }

 // this is a concrete method
 public String getColor() {
    return color;
 }
}
class Circle extends Shape
{
 double radius;
```

```
public Circle(String color,double radius) {

    // calling Shape constructor
    super(color);
    System.out.println("Circle constructor called");
    this.radius = radius;
}

@Override
double area() {
    return Math.PI * Math.pow(radius, 2);
}

@Override
public String toString() {
    return "Circle color is " + super.color +
                "and area is : " + area();
}

}
class Rectangle extends Shape{

double length;
double width;

public Rectangle(String color,double length,double width) {
    // calling Shape constructor
    super(color);
```

```
    System.out.println("Rectangle constructor called");
    this.length = length;
    this.width = width;
  }

  @Override
  double area() {
    return length*width;
  }

  @Override
  public String toString() {
    return "Rectangle color is " + super.color +
                  "and area is : " + area();
  }

}
public class Test
{
 public static void main(String[] args)
 {
    Shape s1 = new Circle("Red", 2.2);
    Shape s2 = new Rectangle("Yellow", 2, 4);

    System.out.println(s1.toString());
    System.out.println(s2.toString());
 }
}
```

OUTPUT:

Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is: 15.205308443374602
Rectangle color is Yellowand area is: 8.0

EXPLANATION:

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived (inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.