Name: Muhammad Ammar

Std.Id: 16602

BSSE- 2nd Semester

Department of Computer Science

Final Term Assignment

Object Oriented Programming

Q#1 (a):

Why access modifiers are used in java, explain in detail Private and Default access modifiers?

ANS:

Access Modifiers in Java

- As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, and method or data member. There are four types of access modifiers available in java:
- A Java access modifier specifies which classes can access a given class and its fields, constructors and methods. Access modifiers can be specified separately for a class, its constructors, fields and methods. Java access modifiers are also sometimes referred to in daily speech as Java access specifiers, but the correct name is Java access modifiers. Classes, fields, constructors and methods can have one of four different Java access modifiers.
- The use of modifiers goes to the core concepts of encapsulation, aka 'data hiding' in object-oriented development.
- Variables should never be public. That is the whole point of private/protected modifiers on them: to prevent direct access to the variables themselves.
- You provide methods to manipulate variables. A method has to be public in order to allow other programmers (or classes) access to that data. But by using methods, you can control how a variable is manipulated. Which is the entire point because you've hidden the details of the variable behind the method. This is the fundamental concept of OO encapsulation.
- Access Modifiers are like entry gates for other classes i.e. a class can control what information or data can be accessible by other classes.
- Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified.

Private Access Modifiers

- 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 cannot be declared as private because
  - ✓ Private means “only visible within the enclosing class”.
  - ✓ 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.
- a method or variable is marked as private (has the private access modifier assigned to it), then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.
- Classes cannot be marked with the private access modifier. Marking a class with the private access modifier would mean that no other class could access it, which means that you could not really use the class at all. Therefore the private access modifier is not allowed for classes.

Example

Here is an example of assigning the private access modifier to a field:

```
public class Clock {
   private long time = 0;
}
```

The member variable time has been marked as private. That means, that the member variable time inside the Clock class cannot be accessed from code outside the Clock class.

Default Access Modifiers

- The default Java access modifier is declared by not writing any access modifier at all.
- The default access modifier means that code inside the class itself as well as code inside classes in the same package as this class, can access the class, field, constructor or method which the default access modifier is assigned to.
- Therefore, the default access modifier is also sometimes referred to as the package access modifier. If you don't know what a Java package is, I have explained that in my Java packages tutorial.

- Subclasses cannot access methods and member variables (fields) in the superclass, if they these methods and fields are marked with the default access modifier, unless the subclass is located in the same package as the superclass.

- Here is an default / package access modifier example:

```
public class Clock {
   long time = 0;
}
public class ClockReader {
   Clock clock = new Clock();
   public long readClock{
      return clock.time;  } }
```

- The time field in the Clock class has no access modifier, which means that it is implicitly assigned the default / package access modifier. Therefore, the ClockReader class can read the time member variable of the Clock object, provided that ClockReader and Clock are located in the same Java package.

Q#1 (b):

Write a specific program of the above mentioned access modifiers in java.

Ans:

Default Access Modifier Program

```
//Java program to illustrate default modifier

package p1;

//Class Geeks is having Default access modifier

class Geek

{

 void display()

 {

        System.out.println("Hello World!");

} }
```

```
//Java program to illustrate error while
//using class from different package with
//default modifier
package p2;
import p1.*;

//This class is having default access modifier
class GeekNew
{
   public static void main(String args[])
     {
       //accessing class Geek from package p1
       Geeks obj = new Geek();

       obj.display();
     }
}
```

Output Program

Compile time error

Private Access Modifier Program

```
//Java program to illustrate error while

//using class from different package with

//private modifier

package p1;

class A

{

  private void display()

 {

      System.out.println("GeeksforGeeks");

 } }

class B

{

  public static void main(String args[])

    {

      A obj = new A();

      //trying to access private method of another class

      obj.display();

    } }
```

Output Program

```
error: display() has private access in A

obj.display()
```

Q#2 (a): Explain in detail Public and Protected access modifiers?

Ans:

Public Access Modifiers

- The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.
- The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.
  Here is a public access modifier example:

```
public class Clock {
   public long time = 0;
}
public class ClockReader {
   Clock clock = new Clock();

   public long readClock{
      return clock.time;
   }
}
```

- The time field in the Clock class is marked with the public Java access modifier. Therefore, the ClockReader class can access the time field in the Clock no matter what package the ClockReader is located in.

Protected Access Modifiers

- 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 modifier.
- The protected access modifier provides the same access as the default access modifier, with the addition that subclasses can access protected methods and member variables (fields) of the superclass.
- This is true even if the subclass is not located in the same package as the superclass.

- Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.
- Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.
- Here is a protected access modifier example:

```
public class Clock {
   protected long time = 0;    // time in milliseconds
}
public class SmartClock() extends Clock{

   public long getTimeInSeconds() {
      return this.time / 1000;
   }
}
```

- In the above example the subclass SmartClock has a method called getTimeInSeconds() which accesses the time variable of the superclass Clock. This is possible even if Clock and SmartClock are not located in the same package, because the time field is marked with the protected Java access modifier.

Q#2 (b): Write a specific program of the above mentioned access modifiers in java.

Ans:

Public Access Modifier Program

```
//Java program to illustrate

//public modifier

package p1;

public class A
```

```
{
   public void display()
     {
        System.out.println("Muhammad Ammar");
     } }
package p2;
import p1.*;
class B
{
    public static void main(String args[])
     {
        A obj = new A;
        obj.display();
     }
}
```

Output Program

Muhammad Ammar

Protected Access Modifier Program

```
//Java program to illustrate
//protected modifier
```

```
package p1;

//Class A

public class A

{

  protected void display()

   {

      System.out.println("IqraUniversity");

   }

}

//Java program to illustrate

//protected modifier

package p2;

import p1.*; //importing all classes in package p1

//Class B is subclass of A

class B extends A

{

  public static void main(String args[])

{    B obj = new B();

     obj.display();

  }  }
```

Output Program

IqraUniversity

Q#3 (a): What is inheritance and why it is used, discuss in detail ?

Ans:

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 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.

Inheritance is used for:

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

- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called inheritance.
- The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Syntax is:

```
class XYZ extends ABC
{
}
```

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **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.
- **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.
- **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.

Uses of Inheritance

There are various advantages of using inheritance in Java that is given below.

- Inheritance provides code reusability. The derived class does not need to redefine the method of base class unless it needs to provide the specific implementation of the method.
- Runtime polymorphism cannot be achieved without using inheritance.
- We can simulate the inheritance of classes with the real-time objects which makes OOPs more realistic.
- Inheritance provides data hiding. The base class can hide some data from the derived class by making it private.
- Method overriding cannot be achieved without inheritance. By method overriding, we can give a specific implementation of some basic method contained by the base class.

Q#3 (b): Write a program using Inheritance class on Animal in java.

Ans:

Single Inheritance Example

```
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 Program

barking...

eating...

Multilevel Inheritance program

```
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 Program

weeping...

barking...

eating...

Hierarchical Inheritance Example

```
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();
}}
```

Output Program

```
meowing...
eating...
```

Q#4 (a): What is polymorphism and why it is used, discuss in detail?

Ans:

Polymorphism

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

```
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 –

```
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.

Uses of Polymorphism

- The good reason for why Polymorphism is need in java is because the concept is extensively used in implementing inheritance.
- It plays an important role in allowing objects having different internal structures to share the same external interface.
- It helps programmers reuse the code and classes once written, tested and implemented. They can be reused in many ways.
- Single variable name can be used to store variables of multiple data types(Float, double, Long, Int etc).
- Polymorphism helps in reducing the coupling between different functionalities
- Faster code at runtime
- More efficient code at runtime
- More dynamic code at runtime
- More flexible and reusable code
- Code that is protected from extension by other classes

Q#4 (b): Write a program using polymorphism in a class on Employee in java.

Ans:

```
/* 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;
  } }
```

Now suppose we extend Employee class as follows −

```
/* 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);    }
   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 setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      } }
   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   } }
```

Now, you study the following program carefully and try to determine its output −

```
/* 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 Program

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

Q#5 (a): Why abstraction is used in OOP, discuss in detail?

Ans:

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.

Uses of Abstraction

- Abstract methods are mostly declared where two or more subclasses are also doing the same thing in different ways through different implementations.
- It also extends the same Abstract class and offers different implementations of the abstract methods.
- Abstract classes help to describe generic types of behaviors and object-oriented programming class hierarchy. It also describes subclasses to offer implementation details of the abstract class.
- 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 using abstract classes if any of these statements apply to your situation:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Q#5 (b): Write a program on abstraction in java.

Ans:

In this Program, 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 Honda4 extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

Bike obj = new Honda4();

obj.run();

}  }
```

Output Program

```
running safely
```

END...