Department of Computer Science

Final Term Exam Spring 2020

Subject: Object Oriented Programming

**BS (CS,SE)** Instructor: M.Ayub Khan

There are total **5** questions in this paper. Max Marks: 50

***Note:***

***At the top of the answer sheet there must be the ID, Name and semester of the concerned Student.***

***Students must have to provide the output of their respective programs. Students have same answers or programs will be considered fail. Programs in Java or codes should be explained clearly.***

***As this paper is online so incase of any ambiguity my Whatsapp no. is 034499121116.***

**Each question carry equal marks.**
**Please answer briefly.**

Q1. a. Why access modifiers are used in java, explain in detail Private and Default access modifiers?
b. Write a specific program of the above mentioned access modifiers in java.

Q2. a. Explain in detail Public and Protected access modifiers?
b. Write a specific program of the above mentioned access modifiers in java.

Q3. a. What is inheritance and why it is used, discuss in detail ?
b. Write a program using Inheritance class on Animal in java.

Q4. a. What is polymorphism and why it is used, discuss in detail ?
b. Write a program using polymorphism in a class on Employee in java.

Q5. a. Why abstraction is used in OOP, discuss in detail ?
b. Write a program on abstraction in java.

Student name :haider saleem / father name : saleem khan
Roll no :16368 \ program: bs(cs)2

Q1. a. Why access modifiers are used in java, explain in detail Private and Default access modifiers?
b. Write a specific program of the above mentioned access modifiers in java.

Ans: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.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **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.
4. **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.

Private access modifier:

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.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
```

```
9.     System.out.println(obj.data);//Compile Time Error
10.    obj.msg();//Compile Time Error
11.    }
12. }
```

## Role of Private Constructor

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

```
1.  class A{
2.  private A(){}//private constructor
3.  void msg(){System.out.println("Hello java");}
4.  }
5.  public class Simple{
6.   public static void main(String args[]){
7.     A obj=new A();//Compile Time Error
8.   }
9.  }
```

: A class cannot be private or protected except nested class.

Default access modifier:

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.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.    void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
```

```
5.   public static void main(String args[]){
6.    A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.   }
9. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Q2. a. Explain in detail Public and Protected access modifiers?

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

Ans: **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. The **protected access modifier** is accessible within package and outside the package but through inheritance only. Variables, methods, and constructors, which are declared **protected** in a superclass can be accessed only by the subclasses in other package or any class within the package of the **protected** members' class. The **protected access modifier** cannot be applied to class and interfaces

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 of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
```

```
5. class B extends A{
6.   public static void main(String args[]){
7.    B obj = new B();
8.    obj.msg();
9.   }
10. }
```

```
Output:Hello
```

**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. 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

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

**Example of public access modifier**

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

```
Output:Hello
```

Q3. a. What is inheritance and why it is used, discuss in detail ?

b. Write a program using Inheritance class on Animal in java.

Ans: **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

**One of the most important concepts in object-oriented programming is that of** inheritance**.** Inheritance **allows us to define a class in terms of another class, which makes it easier to create and maintain an** application**. This also provides an opportunity to reuse the code functionality and fast implementation time.**

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

## 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.

## The syntax of Java Inheritance

1. **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.

## Inheritance Example

In this example, we have a base class `Teacher` and a sub class `PhysicsTeacher`. Since class `PhysicsTeacher` extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class.
Here we have collegeName, designation and work() method which are common to all the teachers so we have declared them in the base class, this way the child classes like `MathTeacher`, `MusicTeacher` and `PhysicsTeacher` do not need to write this code and can be used directly from base class.

```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
        System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);
        System.out.println(obj.mainSubject);
        obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Based on the above example we can say that `PhysicsTeacher` **IS-A** `Teacher`. This means that a child class has IS-A relationship with the parent class. This is inheritance is known as **IS-A relationship** between child and parent class

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*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

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

```
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();
    }
}
```

**Output**

```
I can eat
I can sleep
I can bark
```

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

Q4. a. What is polymorphism and why it is used, discuss in detail ?

b. Write a program using polymorphism in a class on Employee in java.

Ans: Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.
Which sound() method will be called is determined at runtime .

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. For example, lets say we have a class Animal that has a method sound(). Since this is a generic class so we can't give it a implementation like: Roar, Meow, Oink etc.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:
1. Method Overloading in Java – This is an example of compile time (or static polymorphism)
2. Method Overriding in Java – This is an example of runtime time (or dynamic polymorphism)
3. Types of Polymorphism – Runtime and compile time – This is our next tutorial where we have covered the types of polymorphism in detail.

```
public class Animal{
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Horse.java

```
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
        Animal obj = new Horse();
        obj.sound();
    }
}
```

Output:

```
Neigh
```

B :Example

```
/* 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();
   }
}
```

This will produce the following result −

## 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
```

Q5. a. Why abstraction is used in OOP, discuss in detail ?

b. Write a program on abstraction in java.

Its main goal is to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

That's a very generic concept that's not limited to object-oriented programming. You can find it everywhere in the real world.

## Abstraction in the real world

I'm a coffee addict. So, when I wake up in the morning, I go into my kitchen, switch on the coffee machine and make coffee. Sounds familiar?

Making coffee with a coffee machine is a good example of abstraction.

You need to know how to use your coffee machine to make coffee. You need to provide water and coffee beans, switch it on and select the kind of coffee you want to get.

The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use.

Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details. You just interact with a simple interface that doesn't require any knowledge about the internal implementation.

You can use the same concept in object-oriented programming languages like Java.

## Abstraction in OOP

Objects in an OOP language provide an abstraction that hides the internal implementation details. Similar to the coffee machine in your kitchen, you just need to know which methods of the object are available to call and which input parameters are needed to trigger a specific operation. But you don't need to understand how this method is implemented and which kinds of actions it has to perform to create the expected result.

Let's implement the coffee machine example in Java. You do the same in any other object-oriented programming language. The syntax might be a little bit different, but the general concept is the same.

### Use abstraction to implement a coffee machine

Modern coffee machines have become pretty complex. Depending on your choice of coffee, they decide which of the available coffee beans to use and how to grind them. They also use the right amount of water and heat it to the required temperature to brew a huge cup of filter coffee or a small and strong espresso.

### Implementing the *CoffeeMachine* abstraction

Using the concept of abstraction, you can hide all these decisions and processing steps within your *CoffeeMachine* class. If you want to keep it as simple as possible, you just need a constructor method that takes a *Map* of *CoffeeBean* objects to create a new *CoffeeMachine* object and a *brewCoffee* method that expects your *CoffeeSelection* and returns a *Coffee* object.

```
import org.thoughts.on.java.coffee.CoffeeException;

import java.utils.Map;
```

```
public class CoffeeMachine {

    private Map<CoffeeSelection, CoffeeBean> beans;



    public CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

         this.beans = beans

    }



    public Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException {

        Coffee coffee = new Coffee();

        System.out.println("Making coffee ...");

        return coffee;

    }

}
```

*CoffeeSelection* is a simple enum providing a set of predefined values for the different kinds of coffees.

```
public enum CoffeeSelection {

    FILTER_COFFEE, ESPRESSO, CAPPUCCINO;

}
```

And the classes *CoffeeBean* and *Coffee* are simple POJOs (plain old Java objects) that only store a set of attributes without providing any logic.

```
public class CoffeeBean {

     private String name;

     private double quantity;



     public CoffeeBean(String name, double quantity) {
```

```
         this.name = name;

        this.quantity;

    }

}

public class Coffee {

    private CoffeeSelection selection;

    private double quantity;



    public Coffee(CoffeeSelection, double quantity) {

        this.selection = selection;

        this. quantity = quantity;

    }

}
```

## Using the *CoffeeMachine* abstraction

Using the *CoffeeMachine* class is almost as easy as making your morning coffee. You just need to prepare a *Map* of the available *CoffeeBeans*, instantiate a new *CoffeeMachine* object, and call the *brewCoffee* method with your preferred *CoffeeSelection*.

```
import org.thoughts.on.java.coffee.CoffeeException;

import java.util.HashMap;

import java.util.Map;



public class CoffeeApp {

    public static void main(String[] args) {

        // create a Map of available coffee beans
```

```
        Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeBean>();

        beans.put(CoffeeSelection.ESPRESSO,

            new CoffeeBean("My favorite espresso bean", 1000));

        beans.put(CoffeeSelection.FILTER_COFFEE,

            new CoffeeBean("My favorite filter coffee bean", 1000));



        // get a new CoffeeMachine object

        CoffeeMachine machine = new CoffeeMachine(beans);



        // brew a fresh coffee

        try {

             Coffee espresso = machine.brewCoffee(CoffeeSelection.ESPRESSO);

         } catch(CoffeeException  e) {

             e.printStackTrace();

        }

    } // end main

} // end CoffeeApp
```

You can see in this example that the abstraction provided by the *CoffeeMachine* class hides all the details of the brewing process. That makes it easy to use and allows each developer to focus on a specific class.

If you implement the *CoffeeMachine*, you don't need to worry about any external tasks, like providing cups, accepting orders or serving the coffee. Someone else will work on that. Your job is to create a *CoffeeMachine* that makes good coffee.

And if you implement a client that uses the *CoffeeMachine*, you don't need to know anything about its internal processes. Someone else already implemented it so that you can rely on its abstraction to use it within your application or system.

That makes the implementation of a complex application a lot easier. And this concept is not limited to the public methods of your class. Each system, component, class, and method provides a different level of abstraction. You can use that on all levels of your system to implement software that's highly reusable and easy to understand.