NAME ::Sikandar Ayoub

ID :: 16524

SECTION ::A

DEPARTMENT :: SOFTWARING ENGINEERING

SUBJECT :: ORIENTED PROGRAMMING

/…………………………………………………………………………………/

Question 1

Why access modifiers are used in jave ?

Access modifier in jave

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

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.

Four main type of jave access modifier….

- Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

- 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.

- 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.

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.

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:

2) 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.

/.../

QUESTION 1

PART B

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

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
   }
}
```

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:

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

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.

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.

```
//save by A.java
package pack;
class A{
  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();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

/…………………………………………………………………………………………………/

QUESTION 2

Protected

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 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.

Public modifier.

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

/……………………………………………………………………………/

Question 2

Part b

Program of public :

- `<?php`
- `class Car {`
- `// public methods and properties.`
- `public $model;`
- `public function getModel()`
- `{`
- `return "The car model is " . $this -> model;`
- `}`
- `}`
- `$mercedes = new Car();`
- `//Here we access a property from outside the class`
- `$mercedes -> model = "Mercedes";`
- `//Here we access a method from outside the class`
- `echo $mercedes -> getModel();`

access modifiers:

```
<?php

class Car {

 //private
 private $model;

 public function getModel()
 {
  return "The car model is " . $this -> model;
 }
}

$mercedes = new Car();

// We try to access a private property from outside the class.
```

```
$mercedes -> model = "Mercedes benz";
echo $mercedes -> getModel();

?>
```

/…………………………………………………………………………../

Question 3

What is inheritance and why it is used?

<u>Inheritance:</u>

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. most knowledge is made manageable by hierarchical (that is, top-down) classifications.

For example

A Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let’s take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a subclass of animals, where animals are referred to as mammals’ superclass.

Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

/…………………………………………………………………………/

Part b:

```
//program.cs
class Program
{
    static void Main(string[] args)
    {
        Dog oDog = new Dog();
        Console.WriteLine(oDog.Cry());

        Cat oCat = new Cat();
        Console.WriteLine(oCat.Cry());

        Console.ReadKey();
    }

//IAnimal.cs
```

```
interface IAnimal
{
    string Cry();
}


//Dog.cs
class Dog : IAnimal
{
    public string Cry()
    {
        return "Woof!";
    }

/.........................................................................../
```

QUESTION 4

Part a:

Polymorphism:

In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

There are several different kinds of polymorphism.

- A variable with a given name may be allowed to have different forms and the program can determine which form of the variable to use at the time of execution.

  For example, a variable named USERID may be capable of being either an integer (whole number) or a string of characters (perhaps because the programmer wants to allow a user to enter a user ID as either an employee number - an integer - or with a name - a string of characters). By giving the program a way to distinguish which form is being handled in each case, either kind can be recognized and handled.

- A named function can also vary depending on the parameters it is given.
  For example, if given a variable that is an integer, the function chosen would be to seek a match against a list of employee numbers; if the variable were a string, it would seek a match against a list of names. In either case, both functions would be known in the program by the same name. This type of polymorphism is sometimes known as overloading.

- Polymorphism can mean, as in the ML language, a data type of "any," such that when specified for a list, a list containing any data types can be processed by a function.

For example, if a function simply determines the length of a list, it doesn't matter what data types are in the list.

/…………………………………………………………………………./

Part b :

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

/………………………………………………………………………/

Question 5

Abstraction

Java Abstraction Example

Abstraction, in general, is a fundamental concept to computer science and software development. The process of abstraction can also be referred to as modeling and is closely related to the concepts of theory and design. Models can also be considered types of abstractions per their generalization of aspects of reality.

OR

Abstraction is one of the important concepts in OOPs. Humans manage complexity through abstraction.

For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications.This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object.

Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In t urn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Used in Oops

Object-oriented concepts form the heart of OOP just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

/………………………………………………………………………………………………/

Question 5::

Part b

```
package net.javatutorial;

public abstract class Employee {

private String name;
private int paymentPerHour;

public Employee(String name, int paymentPerHour) {
this.name = name;
this.paymentPerHour = paymentPerHour;
}

public abstract int calculateSalary();

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getPaymentPerHour() {
return paymentPerHour;
}

public void setPaymentPerHour(int paymentPerHour) {
this.paymentPerHour = paymentPerHour;
}

}
```

The `Contractor` class inherits all properties from its parent `Employee` but have to provide it's own implementation of `calculateSalary()` method. In this case we multiply the value of payment per hour with given working hours.

```
package net.javatutorial;

public class Contractor extends Employee {

private int workingHours;

public Contractor(String name, int paymentPerHour, int
    workingHours) {
```

```
super(name, paymentPerHour);
this.workingHours = workingHours;
}

@Override
public int calculateSalary() {
return getPaymentPerHour() * workingHours;
}

}
```

The `FullTimeEmployee` also has it's own implementation of`calculateSalary()`method. In this case we just multiply by constant 8 hours.

```
package net.javatutorial;

public class FullTimeEmployee extends Employee {

public FullTimeEmployee(String name, int paymentPerHour) {
super(name, paymentPerHour);
}

@Override
public int calculateSalary() {
return getPaymentPerHour() * 8;
}

}
```

/………………………………………………………………………………/