Worksheet 4

Q1.

OOP - Object-Oriented Programming (OOP) in Java is a way to design and structure your code so that it's organized into objects. Objects are like real-world entities that combine data (attributes) and actions (methods).

Following are the core concepts of OOP with simple examples:

Class:

- **Definition**: A blueprint for creating objects. It defines the data and methods that the objects created from it will have.

Example:

```
public class Car {
   String color;
   int speed;
   void accelerate() {
      speed += 10;
   }
}
```

Object:

- **Definition**: An instance of a class. It represents a specific entity with its own set of attributes and behaviors.

Example:

```
Car myCar = new Car(); // Create an object of the Car class
myCar.color = "Red";   // Set the color attribute
myCar.accelerate();   // Call the accelerate method
```

Encapsulation:

- **Definition**: Hiding the internal state of an object and requiring all interaction to be performed through an object's methods.
- **Example**:

```
public class Person {
    private String name;

    public String getName() {
        return name;
    }

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

Inheritance:

- **Definition**: A way to create a new class that is based on an existing class. The new class inherits attributes and methods from the old class.
- **Example**:

```
public class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
}

public class Bike extends Vehicle {
    void pedal() {
        System.out.println("Bike is pedaling");
    }
}
```

Polymorphism:

- **Definition**: The ability of different objects to respond to the same method in different ways.

Example:

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

public class Dog extends Animal {
   void makeSound() {
      System.out.println("Dog barks");
   }
}

public class Cat extends Animal {
   void makeSound() {
      System.out.println("Cat meows");
   }
}

Animal myPet = new Dog();
myPet.makeSound();
 // Output: Dog barks
```

Abstraction:

- **Definition**: The concept of hiding the complex implementation details and showing only the essential features of an object.

Example:

```
abstract class Shape {
```

```
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Q2. MCQ

Q1. None of the mentioned options as: In Java, an abstract class is declared using the abstract keyword. Therefore, the answer related to Java would be:

Declaring a class as abstract using the abstract keyword.

Q2. Option A – 1), 3), and 4).

Q3. Option B – At compile time. The compiler decides which overloaded method to invoke based on the method signature (the number and type of parameters).

Q4. Option A. 0 – A default constructor in Java does not take any parameters. It is either provided by the compiler if no constructors are explicitly defined, or it can be explicitly defined with an empty parameter list.

Q5. Option A. Dot Operator

In Java, the dot operator (.) is used to access the data members (fields) and methods of a class through its object.

For example:

MyClass obj = new MyClass();

obj.dataMember; // Accessing a data member

Q6. Option C. Class

Objects are instances of a class in Java, so they are variables of the type Class.

Q7. Option A. Private Data

In Java, friend functions or classes don't exist, so private members are accessible only within the class they are defined in.

Q8. Option B. 0

‘I’ is an integer (int), so it is initialized to 0. Therefore, the program will output 0.

Q9. Option A. Only 1, 2 and 3

1, 2, 3 statements are correct but 4^{th} statement is false because java does not automatically create a package with the folder name if no package is specified. Instead, the class is placed in the unnamed package.

Q10. Output : Derived::show() called

Explanation:

In this program, there are two classes: Base and Derived, where Derived extends Base. Both classes have a show() method. The key concept here is method overriding and polymorphism.

- When you create an object of Derived and assign it to a reference of type Base (i.e., Base b = new Derived();), the reference type is Base, but the actual object is of type Derived.
- In Java, the method that gets called is determined by the actual object type at runtime, not by the reference type. This is known as **dynamic method dispatch** or **runtime polymorphism**.

So, when b.show() is called, the show() method of the Derived class is executed, not the one in the Base class, even though b is of type Base.

Q11. Output: The program will result in a compile-time error.

Explanation:

- The Base class has a final method show().
- The Derived class attempts to override the show() method, but because the method in the Base class is final, this is not allowed.

Q12. Output: Base::show() called

Explanation:

- The show() method in Base is static, and it is called with Base as the reference type in the main method.
- When you write b.show(), since b is of type Base, the Base class’s static method show() is called, not the Derived class’s static method.

So, despite b actually referring to an instance of Derived, the static method that gets called is determined by the reference type (Base).

Q13. Output: Test class Derived class

Explanation:

- **Method Resolution:** obj is a reference of type Derived but actually an instance of Test. Due to dynamic method dispatch, the getDetails() method of the Test class is called.
- **Execution of Test Class Method:** The getDetails() method in the Test class prints "Test class " and then calls super.getDetails().
- **Calling Superclass Method:** super.getDetails() refers to the getDetails() method in the Derived class, which prints "Derived class ".

Q14. Output: Test class Name

Explanation:

- In the Test class, the method getDetails(String temp) is overridden with a different return type (int), but this is actually a method signature mismatch and is not considered an override in Java. Instead, it is treated as a new method specific to the Test class.
- In the main method, obj is an instance of Test, and obj.getDetails("Name") calls the getDetails method defined in the Test class, because the reference type is Test.

Q15. Output:

Adding to 100, x = 103

Adding to 0, y = 3 3 3

Explanation:

Part 1:

- HasStatic hs1 = new HasStatic();
  hs1.x++ increments x to 101.
- HasStatic hs2 = new HasStatic();
  hs2.x++ increments x to 102.
- hs1 = new HasStatic();
  hs1.x++ increments x to 103.
- HasStatic.x++;
  Directly increments x to 104.

The final value of x is 104, but the output string shows 103, which is due to the post-increment operation not reflecting in the output statement.

Part 2:

- test t1 = new test();
  t1.y++ increments y to 1.
- test t2 = new test();
  t2.y++ increments y to 2.

- t1 = new test();
  t1.y++ increments y to 3.

All references to test.y will reflect this final value of 3.

Q16. Output: The program will result in a compile-time error.

Explanation:

There is a syntax error in the program. The method signature public void m1(float f,int i); contains a semicolon (;) at the end, which is incorrect. In Java, method declarations should have a method body, or if it's an abstract method, it should be declared within an abstract class or interface.

Q17. Output : Compilation error

Explanation:

int is a primitive type and cannot be assigned null. The statement int temp = null; will cause a compilation error because null cannot be assigned to a primitive type.

Q18. Output: 0 0

- x and y are protected instance variables in the Test class. In Java, instance variables are initialized to their default values if not explicitly initialized.
- For int, the default value is 0.

So, when t.x and t.y are accessed, they will both be 0, resulting in the output:

Q19. Output:

Constructor called 10

Constructor called 5

Explanation:

- **Creating an Instance of Test2:** When new Test2(5) is executed in main(), it creates an instance of Test2.
- **Field Initialization:** During the construction of a Test2 instance, the fields are initialized before the constructor body is executed. So, the field t1 is initialized with new Test1(10) This prints: Constructor called 10
- **Constructor Execution:** After the field initialization, the Test2(int i) constructor is executed. This constructor creates a new Test1 object with i (which is 5). This prints: Constructor called 5

Q20. Output: 7

Explanation:

- int []x[] = {{1,2}, {3,4,5}, {6,7,8,9}}; initializes a 2D array x with 3 rows. Each row is an array of integers:

    - x[0] is {1, 2}
    - x[1] is {3, 4, 5}
    - x[2] is {6, 7, 8, 9}
- int [][]y = x; makes y a reference to the same 2D array as x.
- y[2][1] accesses the element in the third row and second column of the array, which is 7.

Q21. Output : 2

Explanation:

- B extends A, so B inherits i from A and has its own variable j.
- Both A and B have a display() method, but B overrides the method from A.
- r is a reference of type A but points to an instance of B (obj2).
- When r.display() is called, the overridden display() method in B is executed because Java uses dynamic method dispatch to determine the method to call based on the actual object type at runtime, not the reference type.

Thus, r.display() calls B's display() method, which prints the value of j: 2

Q22. Output: 2

Explanation:

- B extends A, so B inherits the i field and display() method from A.
- B also has its own j field and overrides the display() method from A.
- In main(), an instance of B (obj) is created. The i field is set to 1, and the j field is set to 2.
- When obj.display() is called, it invokes the overridden display() method in B.

Since B's display() method prints j, and j was set to 2, the output is: 2

Q23. Output : 1 2

Explanation:

- super.j = 3; sets the protected j from A to 3, but B has its own j field.
- System.out.println(i + " " + j); prints i from A (which is 1) and j from B (which is 2).

Q24. Output: 1 2

Explanation:

- class A initializes i to 1 and j to 2 in its constructor.
- class B extends A and calls super() in its constructor, which invokes the constructor of A, setting i to 1 and j to 2.

- B does not modify i and j but inherits these values from A.

In main(), obj is an instance of B, and obj.i and obj.j are accessed, showing:

Q25. Output:

obj1.a = 4 obj1.b = 4

obj2.a = 4 obj2.b = 4

Explanation:

- **Initial Values:** obj1.a = 1, obj1.b = 2.
- **Method Call:** obj1.func(obj1) assigns obj3 to obj1, so obj3 and obj1 are the same object.
- **Update Values:** obj3.a is set to 1 (obj.a++) + 3 (++obj.b) resulting in 4. obj.b is updated to 4.
- **Results:** Both obj1 and obj2 refer to the same object, so obj1.a = 4, obj1.b = 4, obj2.a = 4, and obj2.b = 4.