Worksheet - 4

Q1.Write in brief about OOPS Concept in java with Examples.
Object-oriented programming contains many significant features, such as encapsulation, inheritance, polymorphism and abstraction. We analyze each feature separately in the following sections.

Encapsulation
Encapsulation provides objects with the ability to hide their internal characteristics and behavior. Each object provides a number of methods, which can be accessed by other objects and change its internal data. In Java, there are three access modifiers: public, private and protected. Each modifier imposes different access rights to other classes, either in the same or in external packages.
Eg :
```
class Account {
    // Private data member
    private double balance;

    // Constructor
    public Account(double balance) {
        this.balance = balance;
    }

    // Public method to provide access to the private data
    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Account account = new Account(1000);
        account.deposit(500);
        System.out.println("Account Balance: " + account.getBalance());
    }
}
```

Inheritance

Inheritance provides an object with the ability to acquire the fields and methods of another class, called base class. Inheritance provides reusability of code and can be used to add additional features to an existing class, without modifying it.

Eg: // Superclass

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Subclass method
    }
}
```

Polymorphism

Polymorphism is the ability of programming languages to present the same interface for differing underlying data types. A polymorphic type is a type whose operations can also be applied to values of some other type.

Eg: Overloading

```
class MathOperation {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperation math = new MathOperation();
        System.out.println(math.add(2, 3));      // Output: 5
        System.out.println(math.add(2.5, 3.5));  // Output: 6.0
    }
}
```

Overriding

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

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

public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();  // Upcasting
        shape.draw();  // Output: Drawing a circle.
    }
}
```

Abstraction

Abstraction is the process of separating ideas from specific instances and thus, developing classes in terms of their own functionality, instead of their implementation details. Java supports the creation and existence of abstract classes that expose interfaces, without including the actual implementation of all methods. The abstraction technique aims to separate the implementation details of a class from its behavior

Eg: Abstract class

```
abstract class Animal {
    abstract void sound();

    void sleep() {
        System.out.println("This animal sleeps.");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("The cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal cat = new Cat();
        cat.sound();
        cat.sleep();
    }
}
```

Interface class

```
interface Drawable {
    void draw();
}

class Rectangle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class Main {
    public static void main(String[] args) {
        Drawable rect = new Rectangle();
        rect.draw();
    }
}
```

Multiple choice question

Q1. Which of the following is used to make an Abstract class? A. Making at least one member function as pure virtual function B. Making at least one member function as virtual function C. Declaring as Abstract class using virtual keyword D. Declaring as Abstract class using static keyword

Abstract class should have at least one pure virtual function. Therefore to declare an abstract class one should declare a pure virtual function in a class.
So the output is **A. Making at least one member function as pure virtual function**

Q2. Which of the following is true about interfaces in java. 1) An interface can contain the following type of members. ....public, static, final fields (i.e., constants) ....default and static methods with bodies 2) An instance of the interface can be created. 3) A class can implement multiple interfaces. 4) Many classes can implement the same interface. A. 1, 3 and 4 B. 1, 2 and 4 C. 2, 3 and 4 D. 1,2,3 and 4

The instance of an interface can't be created because it acts as an abstract class.
So the output is **A. 1, 3 and 4**

Q3. When does method overloading is determined? A. At run time B. At compile time C. At coding time D. At execution time

Overloading is determined at compile time. Hence, it is also known as compile time polymorphism.
So the output is **B. At compile time**

Q4.What is the number of parameters that a default constructor requires? A. 0 B. 1 C. 2 D. 3
Default constructors have no parameters, but they can have parameters with default values.
So the output is **A. 0**

Q5.To access data members of a class, which of the following is used? A. Dot Operator B. Arrow Operator C. A and B both as required D. Direct call

The data members and member functions of the class can be accessed using the dot('. ') operator with the object.
So the output is **A. Dot Operator**

Q6.Objects are the variables of the type ____? A. String B. Boolean C. Class D. All data types can be included

Once the class has been defined, we can create any number of objects belonging to that class.
So the output is **C. Class**

Q7.A non-member function cannot access which data of the class? A. Private data B. Public data C. Protected data D. All of the above

A member function can access private data of the class but a non-member function cannot do that.
So the output is **A. Private data**

Q8. Predict the output of following Java program class Test { int i; } class Main { public static void main(String args[]) { Test t = new Test(); System.out.println(t.i); } } A. garbage value B. 0 C. compiler error D. runtime Error

In java, when an object is created, its instance variables are automatically initialized to default values if they are not explicitly initialized. For the int data type, the default value is 0.
So the output is **B. 0**

Q9.Which of the following is/are true about packages in Java? 1) Every class is part of some package. 2) All classes in a file are part of the same package. 3) If no package is specified, the classes in the file go into a special unnamed package 4) If no package is specified, a new package is created with the folder name of class and the class is put in this package. A. Only 1, 2 and 3 B. Only 1, 2 and 4 C. Only 4 D. Only 1, 3 and 4

In Java, a package can be considered as equivalent to C++ language\ 's namespace.
So the output is **A. Only 1, 2 and 3**

Q10.Predict the Output of following Java Program. class Base { public void show() { System.out.println("Base::show() called"); } } class Derived extends Base { public void show() { System.out.println("Derived::show() called"); } } public class Main { public static void main(String[] args) { Base b = new Derived();; b.show(); } }

Base Class: Contains a method show() that prints "Base::show() called".
Derived Class: Inherits from Base and overrides the show() method to print "Derived::show() called".
In java, method calls are resolved at runtime based on the actual object's type, not the reference type. This feature is known as dynamic method dispatch or runtime polymorphism. Here, the object b is of the type Derived, so the overridden method show() in the Derived class
So the output is "**Derived::show() called**"

Q11. What is the output of the below Java program? class Base { final public void show() { System.out.println("Base::show() called"); } } class Derived extends Base { public void show() { System.out.println("Derived::show() called"); } } class Main { public static void main(String[] args) { Base b = new Derived();; b.show(); } }

Base class: The final keyword in java means that the method cannot be overridden by subclasses.
Derived class : The Derived class attempts to override the show() method from the Base class. However, since the show() method in Base is final, this is not allowed, leading to a compile-time error.

So the output is when we compile this code, we get a compilation error.
Compilation error: error: show() in Derived cannot override show() in Base

```
   public void show() {
               ^
 overridden method is final
```

Q12.Find output of the program. class Base { public static void show() { System.out.println("Base::show() called"); } } class Derived extends Base { public static void show() { System.out.println("Derived::show() called"); } } class Main { public static void main(String[] args) { Base b = new Derived(); b.show(); } }

The show() methods in both Base and Derived classes are declared as static..
In Java, static methods are not overridden; instead, they are hidden. When you declare a static method with the same name and parameters in a subclass, it hides the method in the superclass. The method that gets called is determined by the type of the reference variable, not the object it points to
The reason for this behavior is that static methods are resolved at compile time based on the reference type, not at runtime based on the object type. Thus, b.show() calls Base.show(),So the output is "**Base::show() called**".

Q13.What is the output of the following program? class Derived { public void getDetails() { System.out.printf("Derived class "); } } public class Test extends Derived { public void getDetails() { System.out.printf("Test class "); super.getDetails(); } public static void main(String[] args) { Derived obj = new Test(); obj.getDetails(); } }

Class Derived: It contains a method getDetails() which prints "Derived class ".
Class Test: It overrides the getDetails() method. Inside this method, it prints "Test class " and then calls super.getDetails(), which invokes the getDetails() method of the superclass (Derived).
An object obj of type Derived is instantiated, but it is actually a Test object (Derived obj = new Test();).
When obj.getDetails() is called, the overridden method in Test is executed because the actual object type is Test, even though the reference type is Derived.

So the output is **Test class Derived class**

Q14. What is the output of the following program? class Derived { public void getDetails(String temp) { System.out.println("Derived class " + temp); } } public class Test extends Derived { public int getDetails(String temp) { System.out.println("Test class " + temp); return 0; } public static void main(String[] args) { Test obj = new Test(); obj.getDetails("Name"); } }

Class Derived: It contains a method getDetails(String temp) which prints "Derived class " followed by the value of the temp parameter.
Class Test: It defines a method getDetails(String temp) that has the same name and parameter list as the method in Derived, but it has a different return type (int instead of void).
In Java, method overriding requires that the overridden method in the subclass has the same return type (or a subtype) and the same parameter list as the method in the superclass. Thus, if the return types differ, they are considered different methods, and the method in Test does not override the method in Derived.
When we compile the code, we will get this **error** :

```
error: getDetails(String) in Test cannot override getDetails(String) in Derived
public int getDetails(String temp)
          ^
  return type int is not compatible with void
```

Q15.What will be the output of the following Java program? class test { public static int y = 0; } class HasStatic { private static int x = 100; public static void

main(String[] args) { HasStatic hs1 = new HasStatic(); hs1.x++; HasStatic hs2 = new HasStatic(); hs2.x++; hs1 = new HasStatic(); hs1.x++; HasStatic.x++; System.out.println("Adding to 100, x = " + x); test t1 = new test(); t1.y++; test t2 = new test(); t2.y++; t1 = new test(); t1.y++; System.out.print("Adding to 0, "); System.out.println("y = " + t1.y + " " + t2.y + " " + test.y); } }

First Instance of HasStatic (hs1): hs1.x++; increments x from 100 to 101.
Second Instance of HasStatic (hs2): hs2.x++; increments x from 101 to 102.
Reassigning hs1: hs1 is assigned a new HasStatic object. However, x is static, so it remains shared. hs1.x++; increments x from 102 to 103.
Direct Access to Static Variable x: HasStatic.x++; increments x from 103 to 104.
Printing the Value of x: The output will display x = 104.
First Instance of test (t1): t1.y++; increments y from 0 to 1.
Second Instance of test (t2): t2.y++; increments y from 1 to 2.
Reassigning t1: t1 is assigned a new test object. However, y is static, so it remains shared. t1.y++; increments y from 2 to 3.
Printing the Value of y: The output will display y = 3 3 3

So the output is **Adding to 100, x = 104**
`` **Adding to 0, y = 3 3 3**

Q16.Predict the output class San { public void m1 (int i,float f) { System.out.println(" int float method"); } public void m1(float f,int i); { System.out.println("float int method");} public static void main(String[]args) { San s=new San(); s.m1(20,20); } }

The provided code attempts to define two methods named m1 in the San class, which have different parameter lists. However, there are some issues in the code that will cause compilation errors.
Errors :
Method Signatures: The first method m1 is defined correctly with the parameters int i, float f. The second method m1 is intended to have the parameters float f, int i, but there is a syntax error with an extra semicolon (;) after the method signature. The correct syntax should not have this semicolon.
Calling m1 Method: The call s.m1(20,20); tries to call the method m1 with two integer arguments, which will not match either of the defined method signatures because there are no methods defined that accept two integer parameters. In Java, there is no implicit conversion from int to float when determining which method to call.

Q17.What is the output of the following program? public class Test { public static void main(String[] args) { int temp = null; Integer data = null; System.out.println(temp + " " + data); } }

When we compile this code we got an error :
incompatible types: <null> cannot be converted to int
int temp = null;
This error indicates that null cannot be assigned to a variable of type int.

Q18.What is the output of the following code ? class Test { protected int x, y; } class Main { public static void main(String args[]) { Test t = new Test(); System.out.println(t.x + " " + t.y); } }

Class Test: This class has two protected integer fields, x and y. In Java, the protected access modifier means that these fields can be accessed within the same package and by subclasses.
Class Main: The main method in this class creates an instance of Test and prints the values of x and y.

In Java, instance variables that are not explicitly initialized are given default values: Numeric types (e.g., int, double, float) are initialized to 0. boolean is initialized to false. Object references are initialized to null. So the output is **0 0**

Q19.Find output // filename: Test2.java class Test1 { Test1(int x) { System.out.println("Constructor called " + x); } } class Test2 { Test1 t1 = new Test1(10); Test2(int i) { t1 = new Test1(i); } public static void main(String[] args) { Test2 t2 = new Test2(5); } }

Class Test1: This class has a constructor that takes an integer x and prints "Constructor called " followed by the value of x.

Class Test2:This class has an instance variable t1 of type Test1 that is initialized with a Test1 object using the value 10.The Test2 class has a constructor that takes an integer i and reassigns t1 to a new Test1 object using the value of i.The main method creates an instance of Test2 using the value 5.

So the output is **Constructor called 10**

Constructor called 5

Q20.What will be the output of the following Java program? class Main { public static void main(String[] args) { int []x[] = {{1,2}, {3,4,5}, {6,7,8,9}}; int [][]y = x; System.out.println(y[2][1]); } }

The above code has given a 2D array with variable x. Another variable is y is also a 2D array which is equal to x . So the value of y is also equal to the value of x .value at the index of y[2][1] is equal to the value at index of x[2][1] which is 7. So the output is **7.**

Q21.What will be the output of the following Java program? class A { int i; public void display() { System.out.println(i); } } class B extends A { int j; public void display() { System.out.println(j); } } class Dynamic_dispatch { public static void main(String args[]) { B obj2 = new B(); obj2.i = 1; obj2.j = 2; A r; r = obj2; r.display(); } }

1. **Class A**:
   - Contains an integer variable i.
   - Has a method display() that prints the value of i.
2. **Class B**:
   - Extends class A.
   - Contains an additional integer variable j.
   - Overrides the display() method to print the value of j.
3. **Class Dynamic_dispatch**:
   - The main method creates an instance of class B and assigns values to i and j.
   - A reference of type A (r) is assigned to refer to the instance of B.
   - The display() method is called using the reference r.

Important Concepts

- **Inheritance**: Class B inherits from class A, so B objects have access to members of A.
- **Method Overriding**: The display() method in B overrides the display() method in A.
- **Dynamic Method Dispatch**: Also known as runtime polymorphism, where the method to be called is determined at runtime based on the object being referred to by the reference variable, not the type of the reference variable itself.

Execution Steps

1. **Object Creation**:
   - B obj2 = new B(); creates an instance of B.

2. **Setting Values**:
    - obj2.i = 1; sets the value of i (inherited from A) to 1.
    - obj2.j = 2; sets the value of j to 2.
3. **Reference Assignment**:
    - A r; declares a reference variable of type A.
    - r = obj2; assigns the reference of obj2 (an instance of B) to r.
4. **Method Call**:
    - r.display(); calls the display() method.
    - Due to dynamic method dispatch, the display() method of class B is called because r refers to an object of type B.

The output is **2.**

Q22. What will be the output of the following Java code? class A { int i; void display() { System.out.println(i); } } class B extends A { int j; void display() { System.out.println(j); } } class method_overriding { public static void main(String args[]) { B obj = new B(); obj.i=1; obj.j=2; obj.display(); } }

1. **Class A**:
    - Contains an integer variable `i`.
    - Has a method `display()` that prints the value of `i`.
2. **Class B**:
    - Extends class `A`.
    - Contains an additional integer variable `j`.
    - Overrides the `display()` method to print the value of `j`.
3. **Class `method_overriding`**:
    - The `main` method creates an instance of class `B` and assigns values to `i` and `j`.
    - The `display()` method is called using the instance of `B`.

Execution Steps

1. **Object Creation**:
    - `B obj = new B();` creates an instance of `B`.
2. **Setting Values**:
    - `obj.i = 1;` sets the value of `i` (inherited from `A`) to 1.
    - `obj.j = 2;` sets the value of `j` to 2.
3. **Method Call**:
    - `obj.display();` calls the `display()` method.

- Since `obj` is an instance of `B`, the overridden `display()` method in class `B` is called.

The output is 2 because the `display()` method of class `B` is called, and it prints the value of `j`, which is 2.

Q23.What will be the output of the following Java code? class A { public int i; protected int j; } class B extends A { int j; void display() { super.j = 3; System.out.println(i + " " + j); } } class Output { public static void main(String args[]) { B obj = new B(); obj.i=1; obj.j=2; obj.display(); } }

1. **Class A**:
    - Contains a public integer variable `i`.
    - Contains a protected integer variable `j`.
2. **Class B**:
    - Extends class `A`.
    - Contains an additional integer variable `j`.
    - Has a method `display()` that sets the `j` variable in the superclass `A` to 3 and prints the value of `i` and the local `j`.
3. **Class `Output`**:
    - The `main` method creates an instance of class `B` and assigns values to `i` and the local `j`.
    - The `display()` method is called using the instance of `B`.

Execution Steps

1. **Object Creation**:
    - `B obj = new B();` creates an instance of `B`.
2. **Setting Values**:
    - `obj.i = 1;` sets the value of `i` (inherited from `A`) to 1.
    - `obj.j = 2;` sets the value of the local `j` in `B` to 2.
3. **Method Call**:
    - `obj.display();` calls the `display()` method.
    - Inside `display()`, `super.j = 3;` sets the value of `j` in the superclass `A` to 3.
    - `System.out.println(i + " " + j);` prints the value of `i` (inherited from `A`) and the local `j` in `B`.

The output is **1 2** because the `display()` method prints the value of `i` from class `A` (which is 1) and the local `j` from class `B` (which is 2). The statement `super.j = 3;` affects only the `j` variable in class `A` and not the local `j` in class `B`.

Q24.What will be the output of the following Java program? class A { public int i; public int j; A() { i = 1; j = 2; } } class B extends A { int a; B() { super(); } } class super_use { public static void main(String args[]) { B obj = new B(); System.out.println(obj.i + " " + obj.j) } }

1. Class `A`:
   - Contains two public integer variables `i` and `j`.
   - Has a constructor that initializes `i` to 1 and `j` to 2.
2. Class `B`:
   - Extends class `A`.
   - Contains an integer variable `a`.
   - Has a constructor that calls the superclass constructor using `super()`.
3. Class `super_use`:
   - The `main` method creates an instance of class `B` and prints the values of `i` and `j`.

Execution Steps

1. Object Creation:
   - `B obj = new B();` creates an instance of `B`.
   - The constructor of `B` calls the constructor of `A` using `super()`.
   - The constructor of `A` initializes `i` to 1 and `j` to 2.
2. Printing Values:
   - `System.out.println(obj.i + " " + obj.j);` prints the values of `i` and `j`.

The output is 1 2 because the constructor of `A` initializes `i` to 1 and `j` to 2, and these values are inherited by the instance of `B`.

Q 25. Find the output of the following program. class Test { int a = 1; int b = 2; Test func(Test obj) { Test obj3 = new Test(); obj3 = obj; obj3.a = obj.a++ + ++obj.b; obj.b = obj.b; return obj3; } public static void main(String[] args) { Test obj1 = new Test(); Test obj2 = obj1.func(obj1); System.out.println("obj1.a = " + obj1.a + " obj1.b = " + obj1.b); System.out.println("obj2.a = " + obj2.a + " obj1.b = " + obj2.b); } }

1. **Class `Test`**:
   - Contains two integer variables a and b.
   - Has a method `func(Test obj)` that performs operations on the passed `Test` object and returns a new `Test` object.
2. **Method `func(Test obj)`**:
   - Creates a new `Test` object `obj3`.
   - Assigns `obj3` to `obj`, meaning `obj3` and `obj` now reference the same object.
   - Updates `obj3.a` using the expression `obj.a++ + ++obj.b`.
   - Returns the modified `obj3`.
3. **Class `main` Method**:
   - Creates a `Test` object `obj1`.
   - Calls `func(obj1)` and assigns the result to `obj2`.
   - Prints the values of a and b for `obj1` and `obj2`.

Execution Steps

1. **Object Creation**:
   - `Test obj1 = new Test();` creates a `Test` object `obj1` with a `= 1` and b `= 2`.
2. **Method Call**:
   - `Test obj2 = obj1.func(obj1);` calls `func(obj1)` and assigns the result to `obj2`.
   - Inside `func`:
   - `Test obj3 = new Test();` creates a new `Test` object `obj3`.
   - `obj3 = obj;` assigns `obj3` to reference the same object as `obj` (`obj1`).
   - `obj3.a = obj.a++ + ++obj.b;` updates `obj3.a` and `obj.a` and `obj.b` as follows:
   - `++obj.b` increments `obj.b` to 3.
   - `obj.a++` returns the current value of `obj.a` (1) and then increments `obj.a` to 2.
   - `obj3.a = 1 + 3` sets `obj3.a` to 4.
   - `obj.b = obj.b;` does not change the value of `obj.b`, which remains 3.
   - `func` returns the modified `obj3` (which is the same as `obj1`).
3. **Printing Values**:

- After the method call, both `obj1` and `obj2` reference the same object, so their values will be identical.
- `System.out.println("obj1.a = " + obj1.a + " obj1.b = " + obj1.b);` prints the values of a and b for `obj1`.
- `System.out.println("obj2.a = " + obj2.a + " obj1.b = " + obj2.b);` prints the values of a and b for `obj2`.

The output is **`obj1.a = 4 obj1.b = 3`** and **`obj2.a = 4 obj1.b = 3`** because both `obj1` and `obj2` reference the same object after the `func` method call, and their values were updated within the `func` method.