JAVA Mid 1 QnA

Set-1

1. a) What is a constructor? What is its requirement in programming?

3. **Constructors:** Constructors are special methods in a class used for initializing objects. They have the same name as the class and are called when an object is created. Constructors can be used to set initial values for object properties.

- **Definition**: A constructor is a special type of method used for automatically initializing an object.
- **Invocation**: Constructors are invoked whenever an object is created using the `new` keyword.
- **Purpose**: Mainly used to initialize the variables of an object.

Rules for Creating Constructors

1. Constructor name must be the same as its class name.
2. No explicit return type.
3. Constructors are defined inside the class.

Syntax

```
class ClassName {
    ClassName() {
        // Constructor body
    }
}
```

Types of Constructors

1. **Default Constructor**
    - No-argument constructor.
    - Automatically inserted by the Java compiler if no other constructors are present.

```
class A {
    A() {
        System.out.println("Default constructor");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A obj = new A();
    }
}
```

2. **Parameterized Constructor**
    - Constructor with a specific number of parameters.
    - Allows different values to be provided to distinct objects.

```
class Student {
    int id;
    String name;

    Student(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println("ID: " + id + ", Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(143, "Surya");
        s1.display();

        Student s2 = new Student(225, "Reba");
        s2.display();
    }
}
```

Example: Calculate Area of a Rectangle Using Constructor

```
class Rectangle {
    int length;
    int breadth;

    // Constructor
    Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }

    void calculateArea() {
        int area = length * breadth;
        System.out.println("Area of Rectangle: " + area);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10, 5);
        rect.calculateArea();
    }
}
```

1. b) Design a java program to find the factorial of a given number.

```
public int factorial(int n) {
         if (n == 0)
                 return 1;
         else
                 return n * factorial(n - 1);
}
```

2. Define a package. How to import packages? Explain with illustrations

- A package is a container for a group of related classes and interfaces.
- The package name and the directory name should be the same.
- Packages promote reusability and encapsulation.
- They also help in categorizing classes and interfaces for easier maintenance.

Creating a Package

- **In an Integrated Development Environment (IDE)**:
  - IDEs like Eclipse allow you to create packages easily.
  - Define the package name at the beginning of the source file using the `package` keyword.
- **In a source file**:
  - Declare the package name at the top of your Java source file using the `package` keyword.
  - The package name should match the directory structure.
  - Example:

```
package mypackage;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello from package!");
    }
}
```

- **Compiling**:

    - Use the `-d` flag with the `javac` command to specify the output directory for compiled classes.
    - Example:
        - `javac -d bin MyClass.java` to compile a class and place the class file in the `bin` directory.
- **Running**:
    - Run the compiled class using the package name and the class name.
    - Example:
        - `java mypackage.MyClass` to run the `MyClass` class in the `mypackage` package.

3. Differentiate between interface and abstract class.

Aspect	Class	Interface
Definition	A blueprint for creating objects.	A contract specifying methods and fields.
Implementation	Instantiated to create objects.	Cannot be instantiated directly.
Declaration	Declared using the `class` keyword.	Declared using the `interface` keyword.
Methods	Can have concrete, abstract, static, and final methods.	Only abstract methods, unless specified as default or static.
Variables	Can have instance, class, and local variables with different access modifiers.	Only constants (`public` , `static` , and `final` by default).
Access Modifiers	Methods and variables can have various access modifiers (e.g., `public` , `private` , `protected`).	Methods are `public` by default; variables are `public` , `static` , and `final` .
Inheritance	Supports single inheritance (`extends` one superclass).	Supports multiple inheritance (can implement multiple interfaces).
Extending	Extends another class using the `extends` keyword.	Can extend other interfaces using the `extends` keyword.
Implementing	Implements an interface using the `implements` keyword.	Cannot implement other classes or interfaces.
Use Cases	Defines objects and their behaviors.	Defines a contract for classes to implement specific methods and fields.
Abstraction	Can provide partial abstraction (with abstract methods).	Used primarily for achieving full abstraction.
Usage in Program Structure	Used to define objects and encapsulate code.	Provides a way to separate interface from implementation.

Aspect	Class	Interface
Multiple Inheritance	Not supported in classes directly.	Supported, as a class can implement multiple interfaces.

4. With suitable code segments illustrate various uses of ‘final’ keyword.

5. Explain interfaces and implements

interface

- **Keyword**: Interfaces are declared using the `interface` keyword.
- **Purpose**: Interfaces are used to achieve data abstraction by defining a contract or blueprint for classes to implement.
- **Structure**: An interface consists of abstract methods and may include constant variables, static methods, and default methods.
- **Abstraction**: Interfaces are similar to classes but do not contain method bodies (unless they are default or static methods).
- **Multiple Inheritance**: Interfaces can be used to support multiple inheritance in Java.
- **Syntax**:

```
interface Animal {
    void eat();  // Abstract method
    void move(); // Abstract method
}
```

implements

- **Usage**: To use an interface in a class, use the `implements` keyword followed by the interface name.
- **Method Implementation**: When a class implements an interface, it must provide concrete implementations for all the abstract methods defined in the interface.
- **Example**:

```
public class Mammals implements Animal {
    public void eat() {
        System.out.println("Mammals eat food");
    }

    public void move() {
        System.out.println("Mammals move around");
    }
}
```

6. With example show extends keyword usage.

The `extends` keyword in Java is used to establish inheritance relationships between classes. Let's dissect the example provided:

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

This class `Vehicle` has a method `drive()`.

```
class Car extends Vehicle {
    @Override
    public void drive() {
        System.out.println("Car is driving");
    }
}
```

Here, `Car` is a subclass of `Vehicle`, indicated by `extends Vehicle`. This means that `Car` inherits all the properties and methods of `Vehicle`, including the `drive()` method. However, `Car` overrides the `drive()` method with its own implementation. This is evident with the `@Override` annotation, indicating that the method is overriding a superclass method.

```
public class InheritanceExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.drive(); // Calls the overridden method in Car class
    }
}
```

In the `main` method, an instance of `Car` is created. When `car.drive()` is called, it invokes the `drive()` method of the `Car` class, not the one in the `Vehicle` class, due to method overriding.

Set-2 Q.No Question

1. Explain method overriding with a suitable example program.

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to define the behavior for a method in a way that is specific to the subclass. Method overriding is characterized by:

- **Same method name**: The method in the subclass has the same name as the method in the superclass.
- **Same parameters**: The method in the subclass has the same parameters (signature) as the method in the superclass.
- **Different class**: The method is defined in a subclass that extends the superclass.

Example:

```
class Test {
    public void eat() {
        System.out.println("Hi");
    }
}

class Second extends Test {
    public void eat() {
        System.out.println("Hello");
    }
}

public class Main {
    public static void main(String[] args) {
        Second s = new Second();
        s.eat(); // Output: Hello

        Test t = new Test();
        t.eat(); // Output: Hi
    }
}
```

- **Polymorphism**: Method overriding supports runtime polymorphism, allowing methods to be invoked based on the actual object's type.
- **Dynamic binding**: Method overriding relies on dynamic binding (late binding), where the method to be called is determined at runtime based on the object's type.
- **Access level**: In method overriding, the subclass method should have the same or broader access level (e.g., public, protected) as the superclass method.
- **Return type**: The return type of the subclass method must be the same as, or a subtype of, the return type of the superclass method.
- **Use with inheritance**: Method overriding occurs in the context of inheritance, where a subclass extends a superclass.
- **Annotations**: The `@Override` annotation can be used in Java to explicitly indicate that a method in a subclass is intended to override a method in the superclass. This helps prevent errors due to typos or mismatches in method signatures.

Example:

```
class Vehicle {
    void run() {
```

```
        System.out.println("Vehicle is running");
    }
}

class Bike extends Vehicle {
    @Override
    void run() {
        System.out.println("Bike is riding");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike bike = new Bike();
        bike.run(); // Output: Bike is riding
    }
}
```

2. What is inheritance? Explain different forms of inheritance with suitable program segments and real world example classes.

Inheritance

- **Definition**:
  - Inheritance is a process of creating a new class (called a derived class or subclass) from an existing class (called a base class, parent class, or superclass).
  - The derived class inherits properties and methods from the base class while adding its own properties and methods.
- **Syntax**:
  - The keyword `extends` is used to create a derived class from a base class.
  - Syntax: `class SubClassName extends SuperClassName { // methods and fields }`
  - Example:

```
class Employee {
    void display() {
        System.out.println("Salary = 1000000");
    }
}

class Programmer extends Employee {
    void bonus() {
        System.out.println("Bonus = 50000");
    }
}
```

Types of Inheritance

1. **Single Inheritance**:
    - In single inheritance, a subclass is derived from a single parent class.
    - Example:

```
class Employee {
    void display() {
        System.out.println("Salary = 1000000");
    }
}

class Programmer extends Employee {
    void bonus() {
        System.out.println("Bonus = 50000");
    }
}
```

2. **Multiple Inheritance**:
    - In multiple inheritance, a subclass can have more than one parent class.
    - Java does not support multiple inheritance directly but achieves it through interfaces.
    - Example:

```
interface Animal {
    void eat();
}

interface Bird {
    void fly();
}

class Bat implements Animal, Bird {
    public void eat() {
        System.out.println("Bat is eating");
    }

    public void fly() {
        System.out.println("Bat is flying");
    }
}
```

3. **Multilevel Inheritance**:
    - In multilevel inheritance, a class is derived from a derived class.
    - Example:

```
class Person {
    void show() {
        System.out.println("I am a person");
    }
}
```

```
class Student extends Person {
    void show() {
        System.out.println("I am a student");
    }
}

class Engineer extends Student {
    void show() {
        System.out.println("I am an engineer");
    }
}
```

4. **Hierarchical Inheritance**:
    - In hierarchical inheritance, multiple derived classes inherit from a single parent class.
    - Example:

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

class Lion extends Animal {
    void roar() {
        System.out.println("Lion roars");
    }
}

class Leopard extends Animal {
    void run() {
        System.out.println("Leopard runs fast");
    }
}
```

5. **Hybrid Inheritance**:

- Hybrid inheritance is a combination of more than two types of inheritance (e.g., single, multiple, hierarchical).
- This form of inheritance is not directly supported in Java.

3. Justify the concept of variables interface and extending interfaces with example

- **Keyword**: Interfaces are declared using the `interface` keyword.
- **Purpose**: Interfaces are used to achieve data abstraction by defining a contract or blueprint for classes to implement.
- **Structure**: An interface consists of abstract methods and may include constant variables, static methods, and default methods.

- **Abstraction**: Interfaces are similar to classes but do not contain method bodies (unless they are default or static methods).
- **Multiple Inheritance**: Interfaces can be used to support multiple inheritance in Java.
- **Syntax**:

```
interface Animal {
    void eat();  // Abstract method
    void move(); // Abstract method
}
```

Variables in Interfaces:

- **Constant Variables**: Variables in interfaces are implicitly `public`, `static`, and `final` (constants).
- **Example**:

```
interface Constants {
    int MAX_VALUE = 100;  // public static final int MAX_VALUE = 100;
}
```

Extending Interfaces:

- **Inheritance**: Interfaces can extend other interfaces using the `extends` keyword.
- **Multiple Inheritance**: An interface can extend multiple interfaces, allowing for a form of multiple inheritance.
- **Example**:

```
interface A {
    void methodA();
}

interface B extends A {
    void methodB();
}
```

4. Explain stream I/O with programming examples.

5. Explain polymorphism with example

- **Definition**:
  - Polymorphism means "many forms."
  - It refers to the ability of a function to operate differently based on the inputs or data types.

    - Polymorphism allows an object to take on many forms and behave differently depending on the context.
- **Types of Polymorphism**:
    - **Compile-Time Polymorphism (Ad Hoc or Static Polymorphism)**:
        - Achieved through method overloading and operator overloading.
        - In method overloading, methods with the same name can have different parameters.
        - Example:

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(3, 5)); // Output: 8 (int)
        System.out.println(calc.add(3.5, 2.5)); // Output: 6.0
(double)
    }
}
```

    - **Run-Time Polymorphism (Pure or Dynamic Polymorphism)**:
        - Achieved through method overriding.
        - In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass.
        - This allows the subclass to define its behavior for a method in a way that is specific to the subclass.

6. Write a note on Exception

Exceptions Handling:

- Java uses exceptions to handle errors and unexpected situations during program execution.
- Exceptions are caught and handled using try-catch blocks.
- The `try` block contains the code that may throw an exception, and the `catch` block handles the exception if it occurs.
- Example:

```java
try {
    // Code that may throw an exception
    int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
    // Handling the exception
    System.out.println("Error: " + e.getMessage());
}
```