Contents

Definition of OOP (Object-Oriented Programming)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data (in the form of fields or attributes) and code (in the form of methods or functions). OOP emphasizes the principles of encapsulation, inheritance, abstraction, and polymorphism to design and build modular, reusable, and maintainable software systems.

Key Principles of OOP

1. **Encapsulation**:
   Bundling data (attributes) and methods (functions) that operate on the data into a single unit (class). It also involves restricting direct access to some components using access modifiers (private, protected, public).
2. **Inheritance**:
   A mechanism where one class (child) can acquire properties and behaviors of another class (parent). This promotes code reuse and establishes a hierarchical relationship between classes.
3. **Polymorphism**:
   The ability of a single function or object to take on multiple forms. This is typically achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).
4. **Abstraction**:
   The process of hiding implementation details and exposing only the necessary functionality. Abstract classes and interfaces are tools used to achieve abstraction.

Advantages of OOP

1. **Modularity**: Code is organized into classes, making it easy to understand and manage.
2. **Reusability**: Existing classes can be reused and extended via inheritance.
3. **Scalability**: Easier to scale and maintain applications as complexity grows.
4. **Data Security**: Encapsulation helps protect sensitive data by restricting access.
5. **Flexibility**: Polymorphism allows methods to handle different types of data or behavior.

Real-World Example

A "Car" class in OOP could include:

- **Attributes**: brand, color, speed.
- **Methods**: startEngine(), accelerate(), brake().

Class and Objects in Java

Definition:

- **Class**: A class is a blueprint or prototype from which objects are created. It defines properties (variables) and behaviors (methods) that the objects created from the class will have.
- **Object**: An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created.

Class Syntax:

```
class ClassName {
   // Fields (Properties)
   int field1;
   String field2;

   // Constructor
   public ClassName(int field1, String field2) {
      this.field1 = field1;
      this.field2 = field2;
   }

   // Method (Behavior)
   public void display() {
        System.out.println("Field1: " + field1 + ", Field2: " + field2);
    }
}
```

Creating Objects:

To create an object of a class, you use the new keyword followed by the constructor of the class.

```
ClassName objectName = new ClassName(arguments);
```

Example Code: Class and Objects

```
// Class definition
class Car {
    // Properties (Fields)
    String model;
    int year;
    String color;

    // Constructor to initialize the properties
    public Car(String model, int year, String color) {
        this.model = model;
        this.year = year;
        this.color = color;
    }

    // Method to display the details of the car
    public void displayDetails() {
        System.out.println("Car Model: " + model);
        System.out.println("Car Year: " + year);
        System.out.println("Car Color: " + color);
    }

    // Method to start the car
    public void start() {
        System.out.println(model + " is starting...");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating objects (instances) of the Car class
        Car car1 = new Car("Toyota Corolla", 2020, "Red");
        Car car2 = new Car("Honda Civic", 2021, "Blue");

        // Calling methods on car1 and car2 objects
        car1.displayDetails();  // Displaying details of car1
        car1.start();           // Starting car1

        System.out.println();  // Print a blank line for separation

        car2.displayDetails();  // Displaying details of car2
        car2.start();           // Starting car2
    }
}
```

Output:

yaml

```
Car Model: Toyota Corolla
Car Year: 2020
Car Color: Red
Toyota Corolla is starting...

Car Model: Honda Civic
Car Year: 2021
Car Color: Blue

Honda Civic is starting...
```

Constructors in Java

Definition:

A constructor in Java is a special method used to initialize objects. It is called automatically when an object of a class is created. The constructor's primary purpose is to assign initial values to the fields of a class.

Characteristics of Constructors:

1. **Name**: A constructor has the same name as the class.
2. **No Return Type**: It does not have a return type (not even void).
3. **Automatic Invocation**: Called automatically when an object is created using the new keyword.
4. **Overloading**: Multiple constructors can be defined in a class (constructor overloading) by varying the number or types of parameters.

Types of Constructors in Java:

1. **Default Constructor**:
    - A constructor without any parameters.
    - Automatically provided by Java if no other constructor is defined.
    - Initializes fields to their default values (e.g., 0 for numbers, null for objects, etc.).

Example:

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

  // Default Constructor
  Student() {
    System.out.println("Default constructor called");
    name = "Unknown";
```

```
        age = 0;
    }

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student(); // Default constructor called
        student.displayInfo();
    }
}
```

Output:

```
Default constructor called
Name: Unknown
Age: 0
```

---

2. **Parameterized Constructor**:
   - A constructor that accepts parameters to initialize fields with specific values.
   - Useful for creating objects with different initial states.

Example:

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

    // Parameterized Constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Anil", 20); // Parameterized constructor called
        student.displayInfo();
    }
}
```

Output:

```
Name: Anil
Age: 20
```

---

3. **Copy Constructor**:
    - A constructor that creates a new object as a copy of an existing object.
    - Java does not provide a default copy constructor; you must implement it explicitly.

Example:

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

    // Parameterized Constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy Constructor
    Student(Student other) {
        this.name = other.name;
        this.age = other.age;
    }

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student("Anil", 20);
        Student student2 = new Student(student1); // Copy constructor called
        student2.displayInfo();
    }
}
```

Output:

```
Name: Anil
Age: 20
```

Key Points to Remember:

1. **this()**:
   Used to call one constructor from another within the same class.
   Example:

```
class Student {
    String name;
    int age;
    Student() {
        this("Default Name", 0); // Calls parameterized constructor
    }

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student(); // Calls default constructor
        student.displayInfo();
    }
}
```

2. **super()**:
   Used to call the parent class constructor.
   Example:

```
class Parent {
    Parent() {
        System.out.println("Parent class constructor called");
    }
}

class Child extends Parent {
    Child() {
        super(); // Calls the Parent class constructor
        System.out.println("Child class constructor called");
    }
}

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

Output:

kotlin

Copy code

```
Parent class constructor called
Child class constructor called
```

Constructor Overloading:

Java supports defining multiple constructors with different parameter lists.

Example:

```
class Rectangle {
    int length, breadth;

    // Constructor with no parameters
    Rectangle() {
        this.length = 0;
        this.breadth = 0;
    }

    // Constructor with one parameter
    Rectangle(int side) {
        this.length = side;
        this.breadth = side;
    }

    // Constructor with two parameters
    Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
    }

    int area() {
        return length * breadth;
    }
}

public class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5);
        Rectangle r3 = new Rectangle(4, 6);

        System.out.println("Area of r1: " + r1.area());
        System.out.println("Area of r2: " + r2.area());
        System.out.println("Area of r3: " + r3.area());
    }
}
```

Output:

mathematica

Copy code

```
Area of r1: 0
Area of r2: 25
```

Area of r3: 24

Object Class in Java

Definition:

The Object class in Java is the root class of the Java class hierarchy. Every class in Java, either directly or indirectly, inherits from the Object class. This means that all Java classes automatically inherit the methods of the Object class unless explicitly overridden.

Key Characteristics:

1. **Root Class**: All classes in Java extend Object implicitly.
2. **Common Methods**: Provides methods that are available to all Java objects, such as toString(), equals(), hashCode(), etc.
3. **Universal Parent**: Even user-defined classes are derived from the Object class.

Methods of the Object Class

Method	**Description**
toString()	Returns a string representation of the object.
equals(Object obj)	Compares two objects for equality.
hashCode()	Returns an integer hash code for the object.
getClass()	Returns the runtime class of the object.
clone()	Creates and returns a copy of the object. (Used with Cloneable interface.)
finalize()	Called by the garbage collector when there are no more references to the object.
wait()	Causes the current thread to wait until another thread invokes notify() or notifyAll() on the object.
notify()	Wakes up a single thread waiting on the object's monitor.
notifyAll()	Wakes up all threads waiting on the object's monitor.

---

toString():

- Returns a string representation of the object.

- Default implementation: ClassName@HashCodeInHexadecimal.
- Typically overridden in custom classes for meaningful output.

Example:

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

Output:

```
101 Raj lucknow
102 Vijay ghaziabad
```

Abstraction in Java

Definition:

Abstraction in Java is the process of hiding implementation details and showing only essential features of an object or a system. It focuses on *what* an object does instead of *how* it does it. Abstraction is achieved in Java through abstract classes and interfaces.

Key Characteristics:

1. **Purpose**: Simplifies complexity by exposing only necessary functionality to the user.
2. **Focus**: Defines functionality rather than implementation.
3. **Achieved Through**:
    - Abstract Classes.
    - Interfaces.

Details

1. Abstract Class:

- A class declared with the abstract keyword.
- Can have both abstract methods (without implementation) and concrete methods (with implementation).
- Cannot be instantiated directly.

Syntax:

```
abstract class AbstractClass {
  // Abstract method (no body)
  abstract void abstractMethod();

  // Concrete method (has body)
  void concreteMethod() {
    System.out.println("This is a concrete method.");
  }
}
```

Example:

```
abstract class Shape {
    abstract void draw(); // Abstract method

    void display() { // Concrete method
        System.out.println("This is 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(); // Polymorphism
        shape.draw();
        shape.display();
    }
}
```

Output:

```
Drawing a circle.
This is a shape.
```

---

2. Interface:

- A blueprint for a class containing only abstract methods (until Java 7).
- From Java 8 onwards, can also include:
  - Default methods (methods with implementation).
  - Static methods.
- Supports multiple inheritance by allowing a class to implement multiple interfaces.

Syntax:

```
interface InterfaceName {
    void abstractMethod(); // Abstract method

    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
}
```

Example:

```
interface Animal {
    void sound(); // Abstract method

    default void breathe() { // Default method
        System.out.println("Animals breathe oxygen.");
    }
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.sound();
        dog.breathe();
    }
}
```

Output:

```
Dog barks.

Animals breathe oxygen.
```

Key Differences Between Abstract Classes and Interfaces

Aspect	**Abstract Class**	**Interface**
Declaration	Declared using the abstract keyword.	Declared using the interface keyword.
Methods	Can have both abstract and concrete methods.	Can have abstract, default, and static methods.
Fields	Can have instance variables.	Can only have static final variables (constants).
Multiple Inheritance	A class can extend only one abstract class.	A class can implement multiple interfaces.
Access Modifiers	Can have any access modifier.	Methods are public by default.

Benefits of Abstraction:

1. **Improved Code Maintainability**: By exposing only high-level functionality, changes in implementation do not affect the users of the abstraction.
2. **Enhanced Security**: Implementation details are hidden from the user.
3. **Code Reusability**: Abstract classes and interfaces allow sharing common functionality among related classes.
4. **Scalability**: Easier to extend or modify abstracted systems.

Encapsulation in Java

Definition:

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the process of wrapping data (fields) and methods (functions) that operate on the data into a single unit (class) and restricting direct access to the fields. Encapsulation ensures that the internal representation of an object is hidden from the outside world and accessed only through well-defined methods.

Key Characteristics:

1. **Data Hiding**: Fields (data) of a class are kept private to prevent unauthorized access.

2. **Controlled Access**: Access to the fields is provided through public getter and setter methods.
3. **Improved Security**: Only authorized methods can modify the data.
4. **Increased Maintainability**: Code changes in one part of the class do not affect other parts of the application.

Details of Encapsulation

Components of Encapsulation:

1. **Private Fields**: Data members of a class are declared as private to restrict access.
2. **Public Methods**: Public getter and setter methods are used to access and modify private fields.
3. **Validation Logic**: Setters can include validation rules to ensure data integrity.

Code Implementation

Example 1: Basic Encapsulation

```
class Student {
  private String name; // Private field
  private int age;     // Private field

  // Getter for name
  public String getName() {
    return name;
  }

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

  // Getter for age
  public int getAge() {
    return age;
    }

    // Setter for age with validation
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setName("Anil");
        student.setAge(20);
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
    }
}
```

Output:

```
Name: Anil

Age: 20
```

Inheritance in Java

Definition:

Inheritance is an Object-Oriented Programming (OOP) concept where one class (child class or subclass) can acquire the properties (fields) and behaviors (methods) of another class (parent class or superclass). It allows for code reuse and the creation of a hierarchical relationship between classes.

Key Characteristics:

1. **Code Reusability**: Enables a subclass to reuse the code from a parent class.
2. **Hierarchy**: Creates a relationship between general and specific classes.
3. **Extensibility**: Facilitates the addition of new features to existing classes.
4. **Types of Inheritance**:
    - Single Inheritance.
    - Multilevel Inheritance.
    - Hierarchical Inheritance.

Use Cases:

1. Reusing existing functionality in new classes.
2. Implementing polymorphism (method overriding).
3. Creating a hierarchical structure of related classes.

Details

Syntax:

```
class ParentClass {

   // Parent class members

}


class ChildClass extends ParentClass {

   // Child class members

}
```

Code Implementation

Example 1: Single Inheritance

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

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(); // Method of Dog class
   }
}
```

Output:

```
This animal eats food.
The dog barks.
```

Example 2: Multilevel Inheritance

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

```
class Mammal extends Animal {
    void walk() {
        System.out.println("This mammal walks.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Animal
        dog.walk(); // Inherited from Mammal
        dog.bark(); // Defined in Dog
    }
}
```

Output:

```
This animal eats food.
This mammal walks.
The dog barks.
```

Example 3: Hierarchical Inheritance

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

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

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

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.eat();
      dog.bark();

      Cat cat = new Cat();
      cat.eat();
      cat.meow();
   }
}
```

Output:

```
This animal eats food.
The dog barks.
This animal eats food.
The cat meows.
```

Types of Inheritance in Java:

1. **Single Inheritance**: A subclass inherits from a single parent class.
2. **Multilevel Inheritance**: A class inherits from a subclass, forming a chain.
3. **Hierarchical Inheritance**: Multiple subclasses inherit from a single parent class.

Method Overriding in Inheritance

When a subclass provides a specific implementation of a method that is already defined in its parent class, it is known as method overriding. This is a key feature of polymorphism.

Example:

java

Copy code

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

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Polymorphism
        animal.sound();
    }
}
```

Output:

```
Dog barks.
```

Access Specifiers in Java

Definition:

Access specifiers in Java define the visibility and accessibility of classes, methods, and variables. They determine which parts of a program can access a particular class member.

Java provides **four types of access specifiers**:

1. **Public**
2. **Private**
3. **Protected**
4. **Default** (no keyword)

Types of Access Specifiers

1. Public

- **Definition**: The public keyword makes a class, method, or field accessible from **anywhere in the program**.
- **Scope**: Visible to all classes, even outside the package.
- **Use Case**: When methods or variables need to be accessible across all packages.

Example:

```
class PublicExample {
    public void show() {
        System.out.println("This is a public method.");
    }
}


public class Main {
    public static void main(String[] args) {
        PublicExample obj = new PublicExample();
        obj.show(); // Accessible
    }
}
```

Output:

```
This is a public method.
```

---

2. Private

- **Definition**: The private keyword restricts access to the class itself.
- **Scope**:
    - Visible only within the class where it is defined.
    - Not accessible by subclasses or other classes.
- **Use Case**: When data or methods should not be accessed directly for security or encapsulation.

Example:

```
class PrivateExample {
  private int number = 10;

  private void display() {
    System.out.println("This is a private method.");
  }

  public void accessPrivate() {
    display(); // Private method accessed within the class
  }
}

public class Main {
  public static void main(String[] args) {
    PrivateExample obj = new PrivateExample();
    obj.accessPrivate(); // Indirect access to the private method
  }
}
```

Output:

```
This is a private method.
```

---

3. Protected

- **Definition**: The protected keyword allows access within the same package and to **subclasses**, even if they are in different packages.

- **Scope**:
  - Accessible in the same package.
  - Accessible in subclasses through inheritance (even in different packages).
- **Use Case**: When extending classes and allowing controlled access to variables or methods.

Example:

```
class Parent {
  protected void display() {
    System.out.println("This is a protected method.");
  }
}

class Child extends Parent {
  void show() {
    display(); // Protected method accessed in subclass
  }
}

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

Output:

```
This is a protected method.
```

Comparison Table

Access Specifier	Within Class	Same Package	Subclass (Different Package)	Outside Package
Public	Yes	Yes	Yes	Yes
Private	Yes	No	No	No
Protected	Yes	Yes	Yes (if subclassed)	No
Default	Yes	Yes	No	No

Key Points:

1. **Public**: Most open, accessible everywhere.
2. **Private**: Most restrictive, visible only within the same class.
3. **Protected**: Balanced access for inheritance and package-level use.
4. **Default**: Package-private access, no visibility outside the package.

Polymorphism in Java

Definition:

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. The term **polymorphism** means "many shapes" and it allows a single action to behave differently based on the object that it is acting upon. In Java, polymorphism can be achieved in two ways:

1. **Compile-time Polymorphism** (Method Overloading)
2. **Runtime Polymorphism** (Method Overriding)

Types of Polymorphism:

1. Compile-time Polymorphism (Method Overloading):

- **Definition**: Method overloading occurs when two or more methods in the same class have the same name but different parameters (either in number or type).
- **How it works**: The method is selected at compile time based on the method signature (number, type, and order of parameters).
- **Use Case**: Used to perform similar operations with different types or numbers of arguments.

Example:

```
class Calculator {
   // Overloaded method with two integer parameters
   public int add(int a, int b) {
      return a + b;
   }

   // Overloaded method with three integer parameters
   public int add(int a, int b, int c) {
      return a + b + c;
   }
}

public class Main {
   public static void main(String[] args) {
      Calculator calc = new Calculator();
      System.out.println(calc.add(5, 10));     // Calls method with two parameters
      System.out.println(calc.add(5, 10, 15));  // Calls method with three parameters
   }
}
```

Output:

```
15
30
```

---

2. Runtime Polymorphism (Method Overriding):

- **Definition**: Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method is called at runtime, and the subclass method is invoked based on the type of object being referred to, not the reference type.
- **How it works**: The method is selected at runtime based on the object's type.
- **Use Case**: Used to implement dynamic method dispatch, especially when you want to change or extend the behavior of methods in the superclass.

Example:

```
class Animal {
   // Method in superclass
   public void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   // Overridden method in subclass
   @Override
   public void sound() {
      System.out.println("Dog barks");
   }
}

class Cat extends Animal {
   // Overridden method in subclass
   @Override
   public void sound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal myAnimal = new Animal();  // Parent class reference
      Animal myDog = new Dog();       // Child class reference
      Animal myCat = new Cat();       // Child class reference

      myAnimal.sound();  // Calls Animal's sound method
```

```
        myDog.sound();     // Calls Dog's overridden sound method

        myCat.sound();     // Calls Cat's overridden sound method

    }

}
```

Output:

```
Animal makes a sound

Dog barks

Cat meows
```