OOPS:

Java Object-Oriented Programming (OOPs) Concepts

1. Introduction to OOPs

- **Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain both data (attributes) and methods (functions).
- **Key Principles of OOP:**
  - **Encapsulation**
  - **Abstraction**
  - **Inheritance**
  - **Polymorphism**

These concepts help to manage large codebases, improve reusability, and ensure scalability.

---

2. Classes and Objects

- **Class**: A blueprint or template for creating objects.
  - Defines variables (fields) and methods (functions).
- **Object**: An instance of a class.
  - Created using the `new` keyword and a constructor.

Example:

```
java
Copy
class Car {
    String model;
    int year;
    void start() {
        System.out.println("The car is starting.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Car
        Car car1 = new Car();
        car1.model = "Toyota";
        car1.year = 2022;
        car1.start();  // Calls the method
    }
}
```

3. Encapsulation

- **Definition**: The bundling of data (variables) and methods that operate on the data into a single unit (class), restricting direct access to some of an object's components.
- **Purpose**: To protect object integrity by preventing unintended interference and misuse of data.

Access Modifiers:

- `private`: Accessible only within the same class.
- `default` (package-private): Accessible within the same package.
- `protected`: Accessible within the same package and by subclasses.
- `public`: Accessible from anywhere.

Example:

```
java
Copy
```

```
class Person {
    private String name;

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

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

4. Abstraction

- **Definition**: Hiding the implementation details and showing only the functionality.
    - Achieved using **abstract classes** or **interfaces**.

Abstract Class:

- A class that cannot be instantiated on its own and may contain abstract methods (without implementation) and concrete methods (with implementation).
- Used when classes share common functionality but may have different implementations for certain behaviors.

Abstract Class Example:

```
java
Copy
abstract class Animal {
    // Abstract method (no implementation)
    abstract void sound();
    // Concrete method (has implementation)
    void eat() {
        System.out.println("The animal is eating.");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Woof!");
    }
}
```

Abstract Class Features:

- Can have both abstract and non-abstract methods.
- Can have member variables.
- Can have constructors (used by subclasses).

5. Interfaces

- **Definition**: A contract that defines a set of methods that the implementing class must provide, but without any method implementation.
- **Key Points:**
    - Interfaces cannot have concrete methods (except default methods from Java 8).
    - A class can implement multiple interfaces, promoting **multiple inheritance**.

Interface Example:

```
java
Copy
interface Animal {
    void sound();  // Abstract method
    default void eat() {  // Default method (Java 8)
        System.out.println("The animal is eating.");
    }
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Woof!");
    }
}
```

Interface Features:

- Can be implemented by multiple classes.
- Cannot have instance variables (only constants).
- Supports multiple inheritance.

---

6. Inheritance

- **Definition**: A mechanism where one class inherits fields and methods from another class.
- **Types of Inheritance:**
    - **Single Inheritance**: One class inherits from another.
    - **Multilevel Inheritance**: A class inherits from a derived class, forming a chain.
    - **Hierarchical Inheritance**: Multiple classes inherit from the same base class.
    - **Multiple Inheritance** (via interfaces): A class can implement multiple interfaces (not supported directly by classes in Java).

Inheritance Example:

```
java
Copy
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

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

Key Benefits of Inheritance:

- Reusability of code.
- Method overriding (dynamic polymorphism).

---

7. Polymorphism

- **Definition**: The ability of an object to take many forms. It allows objects of different classes to be treated as objects of a common superclass.
    - **Compile-time Polymorphism** (Method Overloading).
    - **Runtime Polymorphism** (Method Overriding).

Method Overloading (Compile-time Polymorphism)

- Same method name, but different parameters.

Overloading Example:

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

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

Method Overriding (Runtime Polymorphism)

- A subclass provides a specific implementation of a method that is already defined in its superclass.

Overriding Example:

```
java
Copy
class Animal {
    void sound() {
        System.out.println("Some sound...");
    }
}

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

8. Constructor

- **Definition**: A special method used to initialize objects.
    - **Default Constructor**: Provided automatically if no constructor is defined by the programmer.
    - **Parameterized Constructor**: Defined by the programmer to initialize an object with specific values.

Constructor Example:

```
java
Copy
class Car {
    String model;
    int year;

    // Parameterized constructor
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object with parameters
        Car car1 = new Car("Toyota", 2022);
        System.out.println("Car Model: " + car1.model + ", Year: " + car1.year);
    }
}
```

9. Static Keyword

- **Static** methods and variables belong to the class rather than an instance of the class.
    - **Static Method**: Can be called without creating an instance of the class.
    - **Static Variable**: Shared by all instances of the class.

Static Example:

```
java
Copy
class Counter {
    static int count = 0;  // Static variable

    // Static method
    static void increment() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter.increment();
        System.out.println("Count: " + Counter.count);
    }
}
```

10. Final Keyword

- **Final** is used to define constants, prevent method overriding, and prevent inheritance.

  - **final variable**: A constant that cannot be changed.
  - **final method**: Cannot be overridden.
  - **final class**: Cannot be subclassed.

Final Example:

```
java
Copy
final class FinalClass {
    // This class cannot be inherited
}

class ChildClass extends FinalClass {  // Compile-time error
}
```

---

11. Super Keyword

- **Super** refers to the superclass (parent class) of the current object.
  - Used to call superclass methods and constructors.

Super Example:

```
java
Copy
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void eat() {
        super.eat();  // Calls the superclass method
        System.out.println("Dog is eating...");
    }
}
```

12. Object Class

- **Object class** is the root class of all classes in Java.
- Every class inherits from the `Object` class, which provides methods such as:
    - `toString()` : Returns a string representation of the object.
    - `equals()` : Compares two objects for equality.
    - `hashCode()` : Returns a hash code value for the object.

Object Class Example:

```
java
Copy
class Person {
    String name;

    @Override
    public String toString() {
        return "Person Name: " + name;
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "John";
        System.out.println(p);  // Calls toString() implicitly
    }
}
```