Object-Oriented Programming in Java

1. Classes and Objects

- **Class**: A blueprint for creating objects. It defines attributes (fields) and behaviors (methods) that the objects instantiated from the class will have.
- **Object**: An instance of a class. When you create an object, you create a specific implementation of the class.

java

```
class Dog {
    // Attributes (fields)
    String name;
    int age;

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

    // Method
    void bark() {
        System.out.println(name + " says woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog("Buddy", 5);

        // Access object attributes and methods
        System.out.println(myDog.name);  // Output: Buddy
        myDog.bark();  // Output: Buddy says woof!
    }
```

```
}
```

- **Constructor**: A special method used to initialize objects. It is called when an object is created.
- **`this` Keyword**: Refers to the current object. It is used to distinguish between class attributes and parameters when they have the same name.

2. Inheritance

- **Inheritance** allows a new class (derived or child class) to inherit properties and behaviors (fields and methods) from an existing class (base or parent class), promoting code reuse.

java

```
class Animal {
    String name;

    void eat() {
        System.out.println(name + " is eating.");
    }
}

class Dog extends Animal {  // Dog inherits from Animal
    void bark() {
        System.out.println(name + " says woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.name = "Buddy";
        myDog.eat();  // Inherited from Animal
        myDog.bark();  // Specific to Dog
    }
}
```

- **Base (Parent) Class**: The class being inherited from.

- **Derived (Child) Class**: The class that inherits from the base class.
- **Method Overriding**: The child class can override methods from the parent class to provide specific implementations.

3. Abstraction

- **Abstraction** is the concept of hiding the internal details and showing only the essential features of an object. In Java, abstraction is achieved using abstract classes and interfaces.
- **Abstract Class**: A class that cannot be instantiated and may contain abstract methods (methods without a body) that must be implemented by subclasses.

java

```
abstract class Animal {
    abstract void sound();  // Abstract method

    void sleep() {
        System.out.println("Sleeping...");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound();  // Output: Woof woof!
        myDog.sleep();  // Output: Sleeping...
    }
}
```

- **Abstract Method**: A method declared without an implementation that must be implemented by subclasses.

4. Encapsulation

- **Encapsulation** is the concept of bundling data (fields) and methods that operate on that data within one unit (class), and restricting access to some of the object's components. This is often achieved by making fields private and providing public getter and setter methods.

java

```
class Account {
    private double balance;  // Private field

    // Getter method
    public double getBalance() {
        return balance;
    }

    // Setter method
    public void setBalance(double balance) {
        if (balance > 0) {
            this.balance = balance;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Account myAccount = new Account();
        myAccount.setBalance(1000.0);
        System.out.println("Balance: " +
myAccount.getBalance());  // Output: Balance: 1000.0
    }
}
```

- **Private Fields**: Restrict access to the data, preventing direct modification.
- **Public Getter and Setter Methods**: Provide controlled access to private fields.

5. Interfaces

- **Interface**: An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors.
- **Implementation**: A class implements an interface by providing the code for all the methods declared in the interface.

java

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

class Dog implements Animal {  // Dog implements the Animal
interface
    public void sound() {
        System.out.println("Woof woof!");
    }
}

class Cat implements Animal {  // Cat implements the Animal
interface
    public void sound() {
        System.out.println("Meow!");
    }
}

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

        myDog.sound();  // Output: Woof woof!
        myCat.sound();  // Output: Meow!
    }
}
```

- **Multiple Inheritance**: Java does not support multiple inheritance with classes, but a class can implement multiple interfaces, allowing a form of multiple inheritance.

- **Default Methods**: Introduced in Java 8, interfaces can have methods with a body using the `default` keyword. These methods do not need to be overridden in the implementing class.

java

```
interface Animal {
    void sound();

    default void sleep() {
        System.out.println("Sleeping...");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound();  // Output: Woof woof!
        myDog.sleep();  // Output: Sleeping...
    }
}
```

Summary

- **Classes and Objects**: Classes are blueprints for objects, encapsulating data and behavior.
- **Inheritance**: Allows a class to inherit fields and methods from another class, promoting code reuse.
- **Abstraction**: Hides complex implementation details and shows only the essential features through abstract classes and interfaces.
- **Encapsulation**: Protects an object's data by restricting access to it and bundling it with the methods that modify it.

- **Interfaces**: Define a contract that implementing classes must follow, allowing for polymorphism and multiple inheritance.