Java OOPS Concepts

1) Encapsulation

Encapsulation is the concept of wrapping data (variables) and methods (functions) together as a single unit, usually by defining them in a class.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

It also involves restricting direct access to some of the object's components (using access modifiers like private).

Encapsulation=Data Hiding+Abstraction

In an encapsulated class we have to maintain getter and setter methods for every data member. •

The main advantages of encapsulation are:

1) We can achieve security.

2) Enhancement will become very easy.

3) It improves maintainability of the application.

4) It provides flexibility to the user to use the system very easily. •

The main disadvantage of encapsulation is it increases the length of the code and slows down execution.

Tightly encapsulated class:

A class is said to be tightly encapsulated if and only if every variable of that class declared as private whether the variable has getter and setter methods are not and whether these methods declared as public or not, not required to check.

```
package oopsconcepts;
class Student {
        // Private variables for encapsulation
        private String name;
        private int age;
        // Public methods to set and get the values of private variables
        public void setName(String name) {
                this.name = name;
```

```
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
public class EncapsulationEx {
    public static void main(String[] args) {
        Student student = new Student();
        student.setName("John");
        student.setAge(20);
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
    }
}
```

Output

Name: John

Age: 20

2) Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features. It can be achieved through abstract classes or interfaces. Hiding internal implementation and just highlighting the set of services, is called abstraction.

By using abstract classes and interfaces we can implement abstraction.

Examples

By using ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation.

• **The main advantages of Abstraction are:**

1) We can achieve security as we are not highlighting our internal implementation.

2) Enhancement will become very easy because without affecting the end user we are able to perform any type of changes in our internal system.

3) It provides more flexibility to the end user to use the system very easily.

4) It improves maintainability of the application.

Using Abstract Class:

```
package oopsconcepts;
abstract class Shape {
	abstract void draw(); // Abstract method
}
class Circle extends Shape {
	@Override
	void draw() {
		System.out.println("Drawing a circle");
	}
}
public class AbstractionEx {
	public static void main(String[] args) {
		Shape shape = new Circle();
		shape.draw();
	}
}
```

Output

Drawing a circle

Using Interface:

```
package oopsconcepts;
```

```
interface Animal {
        void eat();
        void sleep();
}
class Cat implements Animal {
        @Override
        public void eat() {
                System.out.println("Cat eats fish");
        }
        @Override
        public void sleep() {
                System.out.println("Cat sleeps");
        }
}
public class AbstractionUsingInterface {
        public static void main(String[] args) {
                Animal myCat = new Cat();
                myCat.eat();
                myCat.sleep();
        }
}
```

Output

Cat eats fish

Cat sleeps

3) Inheritance

Inheritance allows one class (child or subclass) to inherit fields and methods from another class (parent or superclass). This promotes code reusability.

The **extends** keyword is used to inherit a class.

i. Single Inheritance: In single inheritance, a class inherits from one superclass.

```
class Parent {
    void display() {
        System.out.println("This is the parent class");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("This is the child class");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.display(); // Inherited method
        c.show();
    }
}
```

Output:

```
This is the parent class
This is the child class
```

---

ii. Multilevel Inheritance: In multilevel inheritance, a class inherits from a class, which itself inherits from another class.

```
class GrandParent {
    void greet() {
```

```
        System.out.println("Hello from GrandParent");
    }
}

class Parent extends GrandParent {
    void welcome() {
        System.out.println("Welcome from Parent");
    }
}

class Child extends Parent {
    void sayHi() {
        System.out.println("Hi from Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.greet();   // Inherited from GrandParent
        c.welcome(); // Inherited from Parent
        c.sayHi();
    }
}
```

Output:

```
Hello from GrandParent
Welcome from Parent
Hi from Child
```

iii. Hierarchical Inheritance: In hierarchical inheritance, multiple child classes inherit from a single parent class.

```
class Parent {
    void commonMethod() {
        System.out.println("This method is common to all");
    }
}

class Child1 extends Parent {
    void specificMethod1() {
        System.out.println("Child1-specific method");
    }
}

class Child2 extends Parent {
    void specificMethod2() {
        System.out.println("Child2-specific method");
    }
}

public class Main {
    public static void main(String[] args) {
        Child1 c1 = new Child1();
        c1.commonMethod();
        c1.specificMethod1();

        Child2 c2 = new Child2();
        c2.commonMethod();
        c2.specificMethod2();
```

```
    }
}
```

Output:

```
This method is common to all
Child1-specific method
This method is common to all
Child2-specific method
```

iv. Multiple Inheritance (Not Supported with Classes in Java)

Java does not support multiple inheritance with classes to avoid **diamond problems** (ambiguity when methods with the same name exist in parent classes). However, it is achievable using **interfaces**.

Example with Interfaces:

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

interface B {
    void methodB();
}

class C implements A, B {
    @Override
    public void methodA() {
        System.out.println("MethodA from Interface A");
    }
```

```
    @Override
    public void methodB() {
        System.out.println("MethodB from Interface B");
    }
}

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

Output:

```
MethodA from Interface A
MethodB from Interface B
```

4) Polymorphism

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.

Polymorphism allows you to perform the same action in different ways. It can be achieved through method overloading (compile-time polymorphism) or method overriding (run-time polymorphism).

Types of Java Polymorphism

In Java Polymorphism is mainly divided into two types:

- **Compile-time Polymorphism**

- **Runtime Polymorphism**

Benefits of Polymorphism

1. **Code Reusability**: Polymorphism allows methods to work for different data types or subclasses.
2. **Flexibility**: The same interface can be used for different types of objects.
3. **Extensibility**: Makes it easier to introduce new behaviors without altering existing code.
4. **Readability**: Reduces code complexity by allowing the same name for different purposes.

1) Method overloading(Compile-time Polymorphism)

Compile-time polymorphism occurs when multiple methods in the same class share the same method name but differ in the number of parameters, type of parameters. and sequence of parameters.

```
package com.oopsconcepts;
class Calculator {
	// Method to add two integers
	int add(int a, int b) {
		return a + b;
	}
	// Overloaded method to add three integers
	int add(int a, int b, int c) {
		return a + b + c;
	}
	// Overloaded method to add two double values
	double add(double a, double b) {
		return a + b;
	}
```

```
}
public class Overloading {
	public static void main(String[] args) {
		Calculator calc = new Calculator();
		System.out.println(calc.add(2, 3)); // Calls the first method
		System.out.println(calc.add(2, 3, 4)); // Calls the second method
		System.out.println(calc.add(2.5, 3.5)); // Calls the third method
	}
}
```

Output

```
5
9
6.0
```

2) Method overriding (run-time polymorphism).

The concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class.

```
	package com.oopsconcepts;
class Animals {
	public void move() {
		System.out.println("Animals can move");
	}
}
class Dog extends Animals {
	public void move() {
		System.out.println("Dogs can walk and run");
	}
}
public class OverridingEx {
```

```
    public static void main(String args[]) {
        Animals a = new Animals(); // Animal reference and object
        Animals b = new Dog(); // Animal reference but Dog object
        a.move(); // runs the method in Animal class
        b.move(); // runs the method in Dog class
    }
}
```

Output

Animals can move

Dogs can walk and run

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
The argument list should be different while doing method overloading.	The argument list should be the same in method overriding.