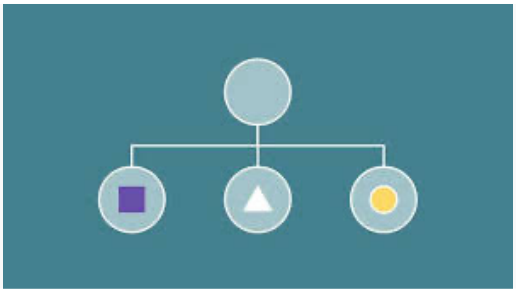
객체 지향 프로그래밍



객체지향 프로그래밍(Object-Oriented Programming, OOP)은 컴퓨터 프로그래밍 패러다임 중 하나로, 현실 세계의 객체들을 모델링하여 프로그래밍하는 방법입니다.

객체지향 프로그래밍의 주요 특징

1. **캡슐화**: 데이터와 그 데이터를 다루는 코드를 함께 묶어서 외부의 접근을 제한하는 것을 의미합니다.

```
public class Car {
    private String model;
    private int price;

    // getter, setter
    public String getModel() { return model; }
    public void setModel(String model) { this.model = model; }
    public int getPrice() { return price; }
    public void setPrice(int price) { this.price = price; }

    public void start() {
        System.out.println("Car started.");
    }
}
```

위 코드에서는 `model`과 `price` 필드를 private으로 선언하고, 이를 접근하는 getter와 setter 메서드를 public으로 제공합니다. 이렇게 함으로써 외부에서는 `Car` 객체의 `model`과 `price` 필드에 직접적인 접근을 할 수 없습니다. 대신, `getModel()`, `setModel()`, `getPrice()`, `setPrice()` 메서드를 통해 간접적으로 접근할 수 있습니다. 이렇게 함으로써 `Car` 객체 내부의 데이터 무결성과 보안이 보장됩니다.

2. **상속**: 이미 정의된 클래스에서 새로운 클래스를 생성할 때, 기존 클래스의 속성과 기능을 상속받아 확장하는 것을 의미합니다.

```java
// 동물 클래스
class Animal {
  String name;

  public Animal(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("동물이 소리를 냅니다.");
  }
}

// 개 클래스 (Animal 클래스를 상속)
class Dog extends Animal {
  public Dog(String name) {
    super(name);
  }

  public void makeSound() {
    System.out.println(name + "이(가) 멍멍 소리를 냅니다.");
  }
}

// 고양이 클래스 (Animal 클래스를 상속)
class Cat extends Animal {
  public Cat(String name) {
    super(name);
  }

  public void makeSound() {
    System.out.println(name + "이(가) 야옹 소리를 냅니다.");
  }
}

// 실행 클래스
public class Main {
  public static void main(String[] args) {
    Animal animal = new Animal("동물");
    Dog dog = new Dog("개");
    Cat cat = new Cat("고양이");

    animal.makeSound(); // "동물이 소리를 냅니다." 출력
    dog.makeSound(); // "개이(가) 멍멍 소리를 냅니다." 출력
    cat.makeSound(); // "고양이이(가) 야옹 소리를 냅니다." 출력
  }
}
```

위 예제에서는 Animal 클래스를 상속받아 Dog 클래스와 Cat 클래스를 구현했습니다. Animal 클래스에는 동물의 이름을 저장하는 name 멤버 변수와 makeSound() 메서드가 정의되어 있고, Dog 클래스와 Cat 클래스에서는 makeSound() 메서드를 오버라이딩하여 동물의 울음 소리를 출력하도록 하였습니다.

이를 실행하면, 동물은 기본적으로 "동물이 소리를 냅니다."를 출력하고, 개는 "개이(가) 멍멍 소리를 냅니다.", 고양이는 "고양이이(가) 야옹 소리를 냅니다."를 출력합니다. 이처럼, 상속을 통해 부모 클래스의 기능을 그대로 물려받으면서 자식 클래스에서 필요한 기능을 추가, 수정하거나 뺄 수 있습니다.

3. **다형성**: 동일한 메서드를 호출해도, 객체에 따라 다르게 동작 할 수 있도록 하는 것을 의미합니다.

```
public class Animal {
    public void sound() {
        System.out.println("동물 소리");
    }
}

public class Cat extends Animal {
    public void sound() {
        System.out.println("야옹");
    }
}

public class Dog extends Animal {
    public void sound() {
        System.out.println("멍멍");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.sound();  // "동물 소리" 출력

        Animal cat = new Cat();
        cat.sound();  // "야옹" 출력

        Animal dog = new Dog();
        dog.sound();  // "멍멍" 출력
    }
}
```

위 예제에서 Animal 클래스는 기본적인 동물 클래스이며, Cat과 Dog 클래스는 Animal 클래스를 상속받아 각각의 동물 소리를 출력하는 sound 메서드를 오버라이드합니다. Main 클래스에서는 다형성을 이용하여 Animal 타입의 변수에 Cat과 Dog 객체를 할당하고, 각 객체의 sound 메서드를 호출합니다. 이때, Animal 타입의 변수에 할당된 객체의 실제 타입에 따라 호출되는 메서드가 결정됩니다.

4. **추상화**: 여러 객체들이 가지고 있는 공통적인 특징을 추출하여 하나의 개념으로 다루는 것을 의미합니다.

```
abstract class Animal {
  protected int age;
  protected String name;

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

  // 추상 메소드로 선언하여 하위 클래스에서 구현하도록 강제한다.
  public abstract void move();
}

class Dog extends Animal {
  public Dog(int age, String name) {
    super(age, name);
  }

  @Override
  public void move() {
    System.out.println("강아지가 네 발로 뛰어갑니다.");
  }
}

class Bird extends Animal {
  public Bird(int age, String name) {
    super(age, name);
  }

  @Override
  public void move() {
    System.out.println("새가 날아갑니다.");
  }
}
```

위 예제에서 `Animal` 클래스는 추상 클래스로 선언되어 있고, `move()` 메소드가 추상 메소드로 선언되어 하위 클래스에서 반드시 구현되어야 합니다. `Dog` 클래스와 `Bird` 클래스는 `Animal` 클래스를 상속받고, `move()` 메소드를 각각의 방식으로 구현합니다.

이렇게 추상화를 사용하면, 상위 클래스에서 공통적인 메소드나 필드를 정의할 수 있고, 하위 클래스에서 이를 구현함으로써 중복을 제거 할 수 있습니다. 또한, 추상 메소드를 이용하여 하위 클래스에서 반드시 구현되도록 강제함으로써 코드의 안정성을 높일 수 있습니다.

객체지향 프로그래밍의 장점

1. 코드의 재사용성: 상속을 통해 기존 코드를 재활용할 수 있고, 캡슐화를 통해 코드의 재사용성을 높일 수 있습니다.
2. 유지보수 용이성: 객체 지향 프로그래밍은 모듈화가 잘 되어 있기 때문에, 코드 수정이나 추가가 용이합니다.
3. 대규모 프로젝트 관리 용이성: 객체 지향 프로그래밍은 각 객체별로 독립적이기 때문에, 대규모 프로젝트에서도 쉽게 관리할 수 있습니다.

객체지향 프로그래밍의 단점

1. 설계가 어려움: 객체지향 프로그래밍은 설계 단계에서 객체들의 상호작용과 관계를 결정해야 하기 때문에 설계가 어려울 수 있습니다.
2. 실행 속도가 느릴 수 있음: 상속과 다형성 등의 기능은 실행 시간에 결정되기 때문에, 실행 속도가 느릴 수 있습니다.

객체지향 프로그래밍의 원칙

1. SOLID 원칙
    - SRP (단일 책임 원칙) : 하나의 클래스는 하나의 책임만 가져야 한다.
    - OCP (개방-폐쇄 원칙) : 기존 코드를 변경하지 않으면서 기능을 추가할 수 있게 설계되어야 한다.
    - LSP (리스코프 치환 원칙) : 자식 클래스는 언제나 부모 클래스의 역할을 대체할 수 있어야 한다.
    - ISP (인터페이스 분리 원칙) : 사용하지 않는 인터페이스는 구현하지 않아야 한다.
    - DIP (의존 역전 원칙) : 추상화된 것은 구체적인 것에 의존하면 안된다.
2. DRY 원칙
    - Don't Repeat Yourself : 중복 코드를 피하고 한 번만 정의하도록 한다.