Day 02: 객체 지향 프로그래밍 (OOP)

1. **상속과 다형성**
2. **캡슐화와 접근 지정자**
3. **추상 클래스와 인터페이스**
4. **내부 클래스**

---

1. 상속과 다형성

상속 (Inheritance)

- 객체 지향 프로그래밍의 핵심 원칙 중 하나.
- 기존의 클래스를 기반으로 새로운 클래스를 생성하는 것.
- 코드 재사용성을 높이며, 확장성을 향상시킨다.
- 부모 클래스(Base or Super class)와 자식 클래스(Derived or Subclass)로 구분.
- 자식 클래스는 부모 클래스의 속성 및 메서드를 상속받아 사용 가능.

예제

1. **부모 클래스 (Base or Super class)**

```
class Animal {
    private String species;

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

    public void eat() {
        System.out.println("The " + species + " is eating.");
    }

    public void sleep() {
```

```
        System.out.println("The " + species + " is sleeping.");
    }
}
```

이 `Animal` 클래스는 동물의 일반적인 행동인 먹기와 자기를 메서드로 가지고 있습니다.

2. **자식 클래스 (Derived or Subclass)**

```
class Dog extends Animal {
    private String name;

    public Dog(String name) {
        super("Dog");  // 부모 클래스의 생성자 호출
        this.name = name;
    }

    public void bark() {
        System.out.println(name + " is barking!");
    }
}
```

`Dog` 클래스는 `Animal` 클래스를 상속받습니다. 따라서 `Dog` 클래스는 `Animal`의 `eat`와 `sleep` 메서드를 상속받아 사용할 수 있습니다. 또한, `bark`라는 개만의 특별한 메서드를 추가로 가지게 됩니다.

3. **사용 예제**

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");

        myDog.eat();      // 출력: The Dog is eating.
        myDog.sleep();    // 출력: The Dog is sleeping.
        myDog.bark();     // 출력: Buddy is barking!
    }
}
```

이 예제에서 `Dog` 객체를 생성하고, 상속받은 `eat`와 `sleep` 메서드를 사용하며, 자신만의 `bark` 메서드도 사용합니다. 이러한 방식으로 Java의 상속은 기존의 클래스를 재사용하고, 필요한 기능을 추가하여 새로운 클래스를 생성하는 데 활용됩니다.

다형성 (Polymorphism)

- 한 타입의 참조 변수로 여러 타입의 객체를 참조할 수 있도록 하는 특징.

- 오버라이딩(Overriding) 또는 오버로딩(Overloading)을 통해 구현.
- 코드의 유연성 및 확장성을 향상시킨다.

오버로딩 (Overloading)

오버로딩은 클래스 내에서 같은 이름의 메서드를 여러 개 정의하는 것을 의미합니다. 이 메서드들은 매개변수의 유형, 순서 또는 매개변수의 수가 달라야 합니다. 오버로딩은 메서드의 이름을 재사용하되 다른 매개변수 목록을 제공하여 다양한 상황에서 해당 메서드를 호출할 수 있게 합니다.

오버로딩의 주요 규칙 및 특징은 다음과 같습니다:

1. **메서드 이름**: 오버로딩된 메서드들은 반드시 같은 이름을 가져야 합니다.
2. **매개변수**: 오버로딩된 메서드들은 매개변수의 유형, 순서, 또는 수가 달라야 합니다.
3. **반환 유형**: 오버로딩에서 반환 유형만이 다른 경우는 허용되지 않습니다. 즉, 반환 유형만 다르게 해서는 메서드를 오버로드 할 수 없습니다.

예제

```
class MathOperations {

    // 오버로딩된 메서드 #1: 두 정수의 합을 반환
    int sum(int a, int b) {
        return a + b;
    }

    // 오버로딩된 메서드 #2: 세 정수의 합을 반환
    int sum(int a, int b, int c) {
        return a + b + c;
    }

    // 오버로딩된 메서드 #3: 두 실수의 합을 반환
    double sum(double a, double b) {
        return a + b;
    }
}

public class OverloadingExample {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();

        System.out.println(math.sum(10, 20));       // 출력: 30
        System.out.println(math.sum(10, 20, 30));  // 출력: 60
        System.out.println(math.sum(10.5, 20.5));  // 출력: 31.0
    }
}
```

위 예제에서 `MathOperations` 클래스는 `sum`이라는 메서드를 세 번 오버로드하였습니다. 각 오버로드된 메서드는 다른 타입 및 수의 매개변수를 받습니다. 따라서 메서드를 호출할 때 제공하는 인수에 따라 적절한 메서드 버전이 실행됩니다.

오버라이딩 (Overriding)

오버라이딩은 객체 지향 프로그래밍에서 중요한 개념 중 하나로, Java에서도 자주 사용됩니다. 이것은 특정 메서드를 상위 클래스(부모 클래스)에서 이미 정의하였지만, 그 메서드를 하위 클래스(자식 클래스)에서 다시 정의하는 것을 의미합니다.

오버라이딩의 주요 특징 및 규칙은 다음과 같습니다:

1. **메서드 시그니처**: 오버라이딩하는 메서드는 부모 클래스에서의 메서드와 동일한 이름, 매개변수 목록 및 반환 유형을 가져야 합니다.
2. **접근 제어자**: 오버라이딩하는 메서드는 부모 클래스의 메서드보다 더 제한적인 접근 제어자를 가질 수 없습니다. 예를 들어, 부모 클래스의 메서드가 `public`이면, 오버라이딩하는 메서드도 `public`이어야 합니다.
3. **예외**: 오버라이딩하는 메서드는 부모 클래스의 메서드보다 더 많은 예외를 `throws` 키워드로 선언할 수 없습니다.
4. `@Override` **어노테이션**: Java에서는 오버라이딩하는 메서드 앞에 `@Override` 어노테이션을 사용하여 해당 메서드가 오버라이딩되었다는 것을 명시적으로 나타낼 수 있습니다. 이것은 컴파일러에게 이 메서드가 부모 클래스의 메서드를 오버라이드한다는 것을 알려줍니다. 만약 메서드가 올바르게 오버라이드되지 않았다면, 컴파일러는 오류를 발생시킵니다.

예제

```
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 Test {
    public static void main(String[] args) {
        Animal myDog = new Dog();
```

```
        myDog.sound(); // 출력: Dog barks
    }
}
```

위의 예제에서, `Dog` 클래스는 `Animal` 클래스의 `sound` 메서드를 오버라이드합니다. 따라서 `Dog` 객체를 사용하여 `sound` 메서드를 호출하면, "Dog barks"가 출력됩니다.

예를 들어 휴대폰에는 기본적으로 "전화 받기"라는 기능이 있습니다. 이 기능은 통화 버튼을 누르면 상대방과 연결되는 기본 기능을 가지고 있습닏

---

2. 캡슐화와 접근 지정자

캡슐화 (Encapsulation)

- 객체의 상태(state)와 행동(behavior)을 하나의 단위로 묶는 것.
- 객체의 상세한 내용을 숨기고, 필요한 기능만 외부에 제공.
- 데이터의 직접적인 접근을 제한하여 객체의 안정성을 보장.

캡슐화는 객체 지향 프로그래밍의 핵심 원칙 중 하나로, 객체의 내부 데이터와 해당 데이터를 조작하는 메서드를 하나의 '캡슐'처럼 묶는 것을 의미합니다. 이를 통해 객체의 내부 구조를 외부에서 알 수 없게 하며, 오직 정의된 메서드를 통해서만 데이터에 접근하거나 변경할 수 있게 합니다.

예제: 은행 계좌

```
public class BankAccount {
    // private 변수로 데이터를 캡슐화
    private double balance;

    // 생성자
    public BankAccount(double initialBalance) {
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;
            System.out.println("Initial balance cannot be negative. Setting balance to
0.");
        }
    }

    // public 메서드를 통해 balance에 접근
```

```
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Deposit amount should be positive.");
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }

    // balance 값을 가져오는 메서드
    public double getBalance() {
        return balance;
    }
}

public class TestBankAccount {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);
        account.deposit(500);         // balance: 1500
        account.withdraw(300);       // balance: 1200
        System.out.println(account.getBalance());  // 출력: 1200.0
    }
}
```

위 예제에서, `BankAccount` 클래스는 `balance`라는 private 변수를 가지고 있습니다. 이 변수는 외부에서 직접 접근할 수 없습니다. 대신, `deposit`, `withdraw`, 및 `getBalance`와 같은 public 메서드를 통해 `balance`에 안전하게 접근하거나 수정할 수 있습니다. 이렇게 객체의 상태와 행동을 캡슐화하면, 잘못된 방식으로 객체의 상태를 변경하거나 객체의 상태에 직접 접근하는 것을 방지하여 객체의 안정성을 보장할 수 있습니다.

접근 지정자 (Access Modifiers)

- 클래스, 변수, 메서드의 접근 범위를 설정.
- 예: `private`, `protected`, `public` 등.
- 정보 은닉 및 캡슐화의 구현에 도움을 준다.

Java에서의 접근 지정자는 캡슐화의 원칙을 구현하는 데 중요한 역할을 합니다.

접근 지정자	동일 클래스	동일 패키지	하위 클래스	다른 패키지

`private`	O	X	X	X
(default)	O	O	X	X
`protected`	O	O	O	X
`public`	O	O	O	O

설명:

1. `private`: 같은 클래스 내에서만 접근 가능합니다.
2. (default): 아무런 접근 지정자를 지정하지 않으면 기본 접근 지정자가 적용되며, 같은 패키지 내의 클래스들만 접근할 수 있습니다.
3. `protected`: 같은 패키지 내의 클래스나 다른 패키지의 하위 클래스에서 접근이 가능합니다.
4. `public`: 어떤 클래스에서든 접근이 가능합니다.

예제

간단한 `Person` 클래스를 만들어보겠습니다. 이 클래스에는 이름, 나이, 그리고 비밀번호가 있습니다. 이름과 나이는 `public`과 `protected`로 설정하여 외부에서 접근이 가능하도록 하고, 비밀번호는 `private`로 설정하여 외부에서 직접 접근이 불가능하게 합니다.

```
// Person.java
package com.example;

public class Person {

    public String name;          // 모든 곳에서 접근 가능
    protected int age;          // 같은 패키지나 하위 클래스에서 접근 가능
    private String password;    // 오직 Person 클래스 내에서만 접근 가능

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

    // 비밀번호를 확인하는 메서드: 이를 통해 password에 간접적으로 접근 가능
    public boolean checkPassword(String inputPassword) {
        return this.password.equals(inputPassword);
    }
}

// Main.java
package com.example;

public class Main {
```

```
    public static void main(String[] args) {
        Person person = new Person("Alice", 25, "secret123");

        System.out.println(person.name);  // 출력: Alice
        // System.out.println(person.password);  // 컴파일 오류: private 접근 지정자 때문에
접근 불가
        System.out.println(person.checkPassword("secret123"));  // 출력: true
    }
}
```

위 예제에서 `name`은 `public` 접근 지정자로 선언되어 있어 어디서든 접근이 가능하고, `age`는 `protected` 접근 지정자로 선언되어 같은 패키지나 하위 클래스에서 접근이 가능합니다. 그러나 `password`는 `private` 접근 지정자로 선언되어 있어 `Person` 클래스 외부에서는 직접 접근할 수 없습니다. 그렇기 때문에 `checkPassword` 메서드를 통해 간접적으로 비밀번호를 확인할 수 있습니다.

3. 추상 클래스와 인터페이스

추상 클래스 (Abstract Class)

- 직접 객체화할 수 없는 클래스.
- 하나 이상의 추상 메서드(정의되지 않은 메서드)를 포함.
- 상속을 통해 자식 클래스에서 구현해야 함.

예제
동물을 나타내는 추상 클래스 `Animal`을 정의하고, 이를 상속받는 구체 클래스 `Dog`와 `Cat`을 생성해보겠습니다. `Animal` 클래스는 `makeSound`라는 추상 메서드를 가집니다. `Dog`와 `Cat` 클래스는 이 메서드를 오버라이드하여 구체적인 구현을 제공합니다.

```
// Animal.java
public abstract class Animal {
    private String name;

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

    // 추상 메서드
    public abstract void makeSound();

    public String getName() {
        return name;
    }
}
```

```
// Dog.java
public class Dog extends Animal {

    public Dog(String name) {
        super(name);
    }

    // 추상 메서드 구현
    @Override
    public void makeSound() {
        System.out.println(getName() + " says: Woof!");
    }
}

// Cat.java
public class Cat extends Animal {

    public Cat(String name) {
        super(name);
    }

    // 추상 메서드 구현
    @Override
    public void makeSound() {
        System.out.println(getName() + " says: Meow!");
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Kitty");

        dog.makeSound();  // 출력: Buddy says: Woof!
        cat.makeSound();  // 출력: Kitty says: Meow!
    }
}
```

위의 예제에서 `Animal`은 추상 클래스입니다. 이는 `makeSound`라는 추상 메서드를 포함하고 있어, 이를 상속받는 모든 자식 클래스는 이 메서드를 구체적으로 구현해야 합니다. 따라서 `Dog`와 `Cat` 클래스는 각각 `makeSound` 메서드를 구체적으로 구현합니다.

인터페이스 (Interface)

- 모든 메서드가 추상 메서드인 것처럼 구현되지 않은 상태.
- 다중 상속의 문제점을 해결하기 위해 사용.
- 구현 객체가 어떤 기능을 제공하는지 명시적으로 알 수 있게 해 줌.

예제

비행과 수영 능력을 나타내는 두 개의 인터페이스 `Flyable`과 `Swimmable`을 정의하겠습니다. 그 후, `Duck` 클래스를 생성하여 이 두 인터페이스를 모두 구현합니다.

```
// Flyable.java
public interface Flyable {
    void fly();  // 추상 메서드
}

// Swimmable.java
public interface Swimmable {
    void swim();  // 추상 메서드
}

// Duck.java
public class Duck implements Flyable, Swimmable {

    @Override
    public void fly() {
        System.out.println("The duck flies in the sky.");
    }

    @Override
    public void swim() {
        System.out.println("The duck swims in the pond.");
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Duck duck = new Duck();

        duck.fly();   // 출력: The duck flies in the sky.
        duck.swim();  // 출력: The duck swims in the pond.
    }
}
```

이 예제에서 `Flyable`과 `Swimmable`은 두 개의 인터페이스입니다. 각각 `fly`와 `swim`이라는 추상 메서드를 가지고 있습니다. `Duck` 클래스는 이 두 인터페이스를 모두 구현하므로 `fly`와 `swim` 메서드를 모두 오버라이드하여 구체적으로 구현해야 합니다.

인터페이스를 사용함으로써, `Duck` 클래스가 어떤 기능(비행 및 수영)을 제공하는지 명시적으로 알 수 있게 됩니다. 또한, 자바에서는 하나의 클래스만 상속받을 수 있지만 여러 인터페이스를 구현할 수 있기 때문에 다중 상속의 문제점을 해결하면서도 다양한 기능을 클래스에 추가할 수 있습니다.

추상 클래스와 인터페이스 비교

항목	추상 클래스 (Abstract Class)	인터페이스 (Interface)
정의	클래스이지만, 직접 인스턴스화 할 수 없는 클래스입니다.	모든 메서드가 추상 메서드인 구조입니다.
상속/구현	`extends` 키워드를 사용하여 상속받습니다.	`implements` 키워드를 사용하여 구현합니다.
메서드 구현	추상 메서드와 일반 메서드 모두 포함 가능합니다.	오직 추상 메서드만 포함 (단, Java 8부터 default, static 메서드 허용)
변수	인스턴스 변수, 스태틱 변수 등 모든 변수를 가질 수 있습니다.	오직 상수 (static final 변수)만 선언 가능
접근 제한자	모든 접근 제한자 (public, protected, private) 사용 가능	메서드는 대부분 public. 변수는 public static final
다중 상속	지원하지 않습니다. 하나의 클래스만 상속 가능	지원합니다. 여러 인터페이스를 동시에 구현 가능
사용하는 이유	공통된 기능을 재사용하면서 확장성을 가지고자 할 때	여러 구현을 갖는 스펙(specification)을 정의하거나 다중 상속의 문제점을 해결하고자 할 때
추가적인 특징	생성자, 초기화 블록 사용 가능	생성자나 초기화 블록은 없습니다.

이 표는 추상 클래스와 인터페이스의 주요 차이점을 간략하게 나타내고 있습니다. 실제 프로그래밍에서는 상황 및 필요에 따라 적절한 선택을 해야 합니다.

동적 바인딩 (Dynamic Binding)

동적 바인딩은 실행 시간에 메서드를 객체와 연결하는 프로세스를 의미합니다. 자바에서 모든 비정적(non-static) 메서드 호출은 기본적으로 동적 바인딩을 사용합니다. 동적 바인딩의 주된 이유는 메서드 오버라이딩 때문입니다.

컴파일 시간에는 참조 변수의 타입에 따라 메서드를 호출할 것처럼 보이지만, 실행 시간에는 객체의 실제 타입에 따라 오버라이드된 메서드가 호출됩니다. 이러한 방식으로 자바는 다형성을 구현하게 됩니다.

예제:

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

class Dog extends Animal {
```

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

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

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

        myDog.sound();  // Outputs: Dog barks
        myCat.sound();  // Outputs: Cat meows
    }
}
```

위의 예제에서 `Animal` 클래스에는 `sound`라는 메서드가 있고, `Dog`와 `Cat`은 `Animal`을 상속받아 `sound` 메서드를 오버라이드합니다.

`TestDynamicBinding`의 `main` 메서드에서 `Animal` 타입의 참조 변수 `myDog`와 `myCat`를 사용하여 각각 `Dog`와 `Cat` 객체를 생성합니다. 여기서 동적 바인딩이 발생하는데, 컴파일 시간에는 `sound` 메서드는 `Animal` 클래스의 메서드처럼 보입니다. 그러나 실행 시간에는 실제 객체의 타입에 따른 오버라이드된 `sound` 메서드가 호출됩니다. 따라서, `myDog.sound()`는 `Dog` 클래스의 `sound` 메서드를 호출하고, `myCat.sound()`는 `Cat` 클래스의 `sound` 메서드를 호출합니다.

자바의 동적 바인딩을 활용한 디자인 패턴

1. **전략 패턴 (Strategy Pattern)**
2. **상태 패턴 (State Pattern)**
3. **템플릿 메서드 패턴 (Template Method Pattern)**
4. **방문자 패턴 (Visitor Pattern)**

템플릿 메서드 패턴

템플릿 메서드 패턴은 알고리즘의 구조와 실행 순서를 슈퍼클래스에서 정의하되, 알고리즘의 일부 단계는 서브클래스에서 구체적으로 구현될 수 있도록 하는 디자인 패턴입니다. 이 패턴은 알고리즘의 변경 없이 알고리즘의 특정 부분만을 수정하거나 확장하는 데 유용합니다.

예제

```
// 슈퍼클래스
abstract class CaffeineBeverage {

    // 템플릿 메서드
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();           // 서브클래스에서 구체화될 메서드
    abstract void addCondiments();  // 서브클래스에서 구체화될 메서드

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

// 서브클래스 1: 커피
class Coffee extends CaffeineBeverage {

    @Override
    void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

// 서브클래스 2: 차
class Tea extends CaffeineBeverage {

    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }

    @Override
```

```
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}

public class TestTemplateMethod {
    public static void main(String[] args) {
        Coffee coffee = new Coffee();
        Tea tea = new Tea();

        System.out.println("\\nMaking coffee...");
        coffee.prepareRecipe();

        System.out.println("\\nMaking tea...");
        tea.prepareRecipe();
    }
}
```

위 예제에서 `CaffeineBeverage`는 템플릿 메서드인 `prepareRecipe`를 정의하고 있습니다. 이 메서드에서는 커피나 차를 만드는 일반적인 과정을 정의하며, 구체적인 내용은 서브클래스인 `Coffee`와 `Tea`에서 구현됩니다.

`TestTemplateMethod` 클래스를 실행하면 커피와 차를 만드는 과정을 볼 수 있습니다. 이를 통해 동일한 알고리즘의 흐름 내에서 서브클래스마다 다르게 동작하는 부분들을 쉽게 처리할 수 있습니다.

4. 내부 클래스

- 클래스 내부에 정의된 클래스.
- 주로 외부 클래스와 밀접한 관계를 가지는 경우 사용.
- 다양한 유형: `static` 내부 클래스, 비스타틱 내부 클래스, 지역 내부 클래스, 익명 내부 클래스 등.
- 내부 클래스는 외부 클래스의 private 멤버에도 접근이 가능하다.

예제

```
// 외부 클래스
public class OuterClass {
    private String outerField = "Outer field";

    // 1. 비스타틱 내부 클래스
```

```
    public class InnerClass {
        public void display() {
            System.out.println("InnerClass: " + outerField);
            System.out.println("OuterClass's this from InnerClass: " + OuterClass.thi
s);
        }
    }

    // 2. static 내부 클래스
    public static class StaticNestedClass {
        public void display() {
            System.out.println("StaticNestedClass");
            // Static 내부 클래스는 외부 클래스의 non-static 멤버에 직접 접근할 수 없습니다.
        }
    }

    // 메서드 내에서 정의하는 지역 내부 클래스
    public void methodWithLocalInnerClass() {
        String localVariable = "Local variable";

        // 3. 지역 내부 클래스
        class LocalInnerClass {
            public void display() {
                System.out.println("LocalInnerClass: " + localVariable);
            }
        }

        LocalInnerClass localInner = new LocalInnerClass();
        localInner.display();
    }

    public void methodWithAnonymousClass() {
        // 4. 익명 내부 클래스
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Anonymous Inner Class: " + outerField);
            }
        };
        r.run();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();

        // 비스태틱 내부 클래스의 사용
        InnerClass inner = outer.new InnerClass();
        inner.display();

        // static 내부 클래스의 사용
        StaticNestedClass staticNested = new StaticNestedClass();
        staticNested.display();

        // 지역 내부 클래스의 사용
        outer.methodWithLocalInnerClass();
```

```
        // 익명 내부 클래스의 사용
        outer.methodWithAnonymousClass();
    }
}
```

종류	설명
비스태틱 내부 클래스 (Non-static Inner Class)	- 가장 일반적인 형태의 내부 클래스. - 외부 클래스의 인스턴스 멤버와 tighly-coupled(긴밀하게 연결된) 관계를 가짐. - 외부 클래스의 private 멤버에도 접근 가능.
스태틱 내부 클래스 (Static Nested Class)	- static 키워드를 사용하여 정의됨. - 외부 클래스의 인스턴스에 종속되지 않음. - 외부 클래스의 static 멤버에만 접근 가능.
지역 내부 클래스 (Local Inner Class)	- 메서드 내부에서 정의됨. - 그 메서드 내에서만 사용할 수 있슴. - 메서드의 로컬 변수와 파라미터에 접근할 수 있지만, 이들 변수/파라미터는 final 또는 effectively final 이어야 함.
익명 내부 클래스 (Anonymous Inner Class)	- 이름이 없는 내부 클래스. - 주로 GUI 이벤트 핸들러나, Runnable 객체 등에 사용 됨. - 주로 클래스 정의와 객체 생성을 동시에 수행 함.

OuterClass.this 예제

`OuterClass.this`는 내부 클래스에서 외부 클래스의 `this` 참조를 얻기 위한 문법입니다. 이를 사용하면 내부 클래스에서 외부 클래스의 인스턴스를 참조할 수 있습니다.

```
public class OuterClass {

    private String outerField = "I'm in the OuterClass";

    class InnerClass {
        private String innerField = "I'm in the InnerClass";

        public void showFields() {
            System.out.println(innerField); // 현재 InnerClass의 멤버 변수

            // OuterClass의 this를 사용해서 OuterClass의 멤버 변수에 접근
            System.out.println(OuterClass.this.outerField);
        }
    }

    public void createAndShow() {
        InnerClass inner = new InnerClass();
        inner.showFields();
    }

    public static void main(String[] args) {
```

```
        OuterClass outer = new OuterClass();
        outer.createAndShow();
    }
}
```

위 예제에서 `InnerClass` 내의 `showFields` 메서드에서 `OuterClass.this.outerField`를 사용하여 외부 클래스 `OuterClass`의 `outerField` 멤버 변수에 접근하였습니다.

Tip: ArrayList 활용

`ArrayList`는 자바의 표준 라이브러리 중에서 컬렉션 프레임워크에 속한 클래스로, 동적 배열을 구현한 것입니다. 그렇기 때문에 크기가 고정된 배열과는 다르게, `ArrayList`는 요소를 추가하거나 제거할 때 크기가 자동으로 조절됩니다.

ArrayList의 주요 특징:

1. **동적 크기**: `ArrayList`의 크기는 자동으로 조절됩니다.
2. **순서가 있음**: `ArrayList`에 저장된 요소들은 인덱스에 의해 순서가 정해집니다.
3. **중복 허용**: 같은 요소를 여러 번 저장할 수 있습니다.
4. **null 허용**: `ArrayList`에는 `null` 요소도 저장이 가능합니다.

ArrayList 사용법:

1. **선언 및 초기화**:

```
ArrayList<Type> listName = new ArrayList<Type>();
```

예:

```
ArrayList<String> names = new ArrayList<String>();
```

1. **요소 추가**: `add` 메서드를 사용해 요소를 추가합니다.

```
names.add("John");
names.add("Jane");
```

1. **요소 검색**: `get` 메서드를 사용해 특정 인덱스의 요소를 가져옵니다.

```
String name = names.get(0);  // "John"
```

1. **요소의 개수**: `size` 메서드를 사용해 `ArrayList`의 크기(요소의 개수)를 알 수 있습니다.

```
int size = names.size();  // 2
```

1. **요소 수정**: `set` 메서드를 사용해 특정 인덱스의 요소를 수정합니다.

```
names.set(0, "Mike");  // 첫 번째 요소를 "Mike"로 변경
```

1. **요소 삭제**: `remove` 메서드를 사용해 특정 인덱스의 요소를 삭제합니다.

```
names.remove(0);  // 첫 번째 요소 삭제
```

1. **전체 요소 삭제**: `clear` 메서드를 사용해 모든 요소를 삭제합니다.

```
names.clear();
```

1. **특정 요소 포함 여부 확인**: `contains` 메서드를 사용해 `ArrayList`가 특정 요소를 포함하고 있는지 확인합니다.

```
boolean containsJane = names.contains("Jane");  // true
```

1. **반복문을 통한 요소 접근**: `ArrayList`의 요소에 접근할 때는 일반 `for` 루프나 향상된 `for` 루프(`foreach`)를 사용할 수 있습니다.

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}

// 또는

for (String name : names) {
```

```
    System.out.println(name);
}
```

실습 문제

회원 관리 프로그램

객체 지향 프로그래밍의 기본 원칙을 활용하여 회원 관리 프로그램을 구현합니다.

요구사항:

1. **회원 데이터**:
    - 아이디(ID)
    - 성명(Name)
    - 나이(Age)
    - 이메일(Email)
2. 프로그램은 다음 메뉴를 사용자에게 제공해야 합니다:
    - 입력 (Add a new member)
    - 출력 (Display all members)
    - 검색 (Search for a member by ID)
    - 수정 (Update a member's information)
    - 삭제 (Delete a member)
    - 종료 (Exit the program)
3. 회원 데이터는 `ArrayList`를 사용하여 관리합니다.
4. 상속과 다형성 특징을 적절히 활용하여 프로그램을 구현합니다.

프로그램 구성 힌트:

- `Member`라는 기본 클래스를 만듭니다.
- `VIPMember`라는 클래스를 생성하여 `Member` 클래스를 상속받습니다.

- `MemberManager` 클래스를 만들어 회원들을 관리하는 기능들을 구현합니다. 이 클래스 내에 `ArrayList<Member>` 타입의 리스트를 포함합니다.
- 사용자의 입력을 받고 프로그램의 메뉴를 관리하는 `Menu` 클래스나 메인 함수를 구현합니다.

회원 관리 프로그램의 일부 소스코드

```
import java.util.ArrayList;
import java.util.Scanner;

class Member {
    private String id;
    private String name;
    private int age;
    private String email;

    public Member(String id, String name, int age, String email) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.email = email;
    }

    // Getter and Setter 메서드는 생략하겠습니다.

    @Override
    public String toString() {
        return "ID: " + id + ", 이름: " + name + ", 나이: " + age + ", 이메일: " + email;
    }
}

public class MemberManager {
    private ArrayList<Member> members = new ArrayList<>();
    private Scanner scanner = new Scanner(System.in);

    public void addMember() {
        System.out.print("아이디 입력: ");
        String id = scanner.nextLine();
        System.out.print("이름 입력: ");
        String name = scanner.nextLine();
        System.out.print("나이 입력: ");
        int age = scanner.nextInt();
        scanner.nextLine();  // consume newline
        System.out.print("이메일 입력: ");
        String email = scanner.nextLine();

        members.add(new Member(id, name, age, email));
        System.out.println("회원이 추가되었습니다.");
    }

    // 출력, 검색, 수정, 삭제 등의 메서드는 이어서 구현...
```

```
    public static void main(String[] args) {
        MemberManager manager = new MemberManager();
        boolean running = true;

        while (running) {
            System.out.println("---- 회원 관리 시스템 ----");
            System.out.println("1. 입력");
            System.out.println("2. 출력");
            System.out.println("3. 검색");
            System.out.println("4. 수정");
            System.out.println("5. 삭제");
            System.out.println("6. 종료");
            System.out.print("선택: ");

            int choice = manager.scanner.nextInt();
            manager.scanner.nextLine();  // consume newline

            switch (choice) {
                case 1:
                    manager.addMember();
                    break;
                // case 2, 3, 4, 5는 여기에 구현...
                case 6:
                    running = false;
                    break;
                default:
                    System.out.println("올바른 선택이 아닙니다.");
            }
        }
    }
}
```

회원 관리 프로그램 실행 예시

```
---- 회원 관리 시스템 ----
1. 입력
2. 출력
3. 검색
4. 수정
5. 삭제
6. 종료
선택: 1

아이디 입력: user123
이름 입력: 홍길동
나이 입력: 30
이메일 입력: hong@example.com
회원이 추가되었습니다.

---- 회원 관리 시스템 ----
1. 입력
```

```
2. 출력
3. 검색
4. 수정
5. 삭제
6. 종료
선택: 2

회원 목록:
1. ID: user123, 이름: 홍길동, 나이: 30, 이메일: hong@example.com

---- 회원 관리 시스템 ----
1. 입력
2. 출력
3. 검색
4. 수정
5. 삭제
6. 종료
선택: 3

검색할 아이디 입력: user123

검색 결과:
ID: user123, 이름: 홍길동, 나이: 30, 이메일: hong@example.com

---- 회원 관리 시스템 ----
1. 입력
2. 출력
3. 검색
4. 수정
5. 삭제
6. 종료
선택: 6

프로그램을 종료합니다.
```

추가 도전

- **VIP 회원**과 **일반 회원**의 구분을 만들어서 상속을 활용하세요. 예를 들면, VIP 회원에게는 추가로 할인율이라는 필드가 있을 수 있습니다.
- 검색 기능에서 이름이나 이메일로도 회원을 검색할 수 있게 확장하세요.
- 다형성을 활용하여 회원들의 정보를 출력하는 다양한 방식(표 형태, 목록 형태 등)을 구현해보세요.

추가도전 힌트

1. **VIP 회원**과 **일반 회원**의 구분:

```
abstract class Member {
    protected String id;
    protected String name;
    protected int age;
    protected String email;

    // 생성자, getter, setter 생략...
}

class VIPMember extends Member {
    private double discountRate;

    public VIPMember(String id, String name, int age, String email, double discountRat
e) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.email = email;
        this.discountRate = discountRate;
    }

    // 생성자, getter, setter 생략...
}

class RegularMember extends Member {
    // 일반 회원은 추가적인 필드가 없음
}
```

2. 이름이나 이메일로 회원 검색 확장:

```
public Member searchMemberByName(String name) {
    for (Member member : members) {
        if (member.getName().equals(name)) {
            return member;
        }
    }
    return null;
}

public Member searchMemberByEmail(String email) {
    for (Member member : members) {
        if (member.getEmail().equals(email)) {
            return member;
        }
    }
    return null;
}
```

3. 다형성을 활용한 출력:

```
interface MemberDisplay {
    void display(Member member);
}

class TableDisplay implements MemberDisplay {
    @Override
    public void display(Member member) {
        // 표 형태로 회원 정보 출력
    }
}

class ListDisplay implements MemberDisplay {
    @Override
    public void display(Member member) {
        // 목록 형태로 회원 정보 출력
    }
}

// 사용 예:
MemberDisplay display = new TableDisplay(); // 또는 new ListDisplay();
display.display(member);
```

추가 도전을 포함한 실행 결과 예시

```
========== 회원 관리 프로그램 ==========
1. 입력
2. 출력
3. 검색
4. 수정
5. 삭제
6. 종료

선택: 1

--- 회원 정보 입력 ---
회원 유형 (1. 일반 회원, 2. VIP 회원): 2
아이디: vip001
이름: 김VIP
나이: 30
이메일: vip@example.com
할인율(%): 10

VIP 회원 '김VIP'님이 등록되었습니다.

선택: 3

--- 회원 검색 ---
1. 아이디로 검색
2. 이름으로 검색
3. 이메일로 검색
```

```
선택: 2
검색할 이름: 김VIP

아이디: vip001
이름: 김VIP
나이: 30
이메일: vip@example.com
할인율: 10%

선택: 2

--- 회원 정보 출력 ---
1. 표 형태로 출력
2. 목록 형태로 출력

선택: 1

| 아이디   | 이름   | 나이 | 이메일           | 할인율 |
| -------- | ------ | ---- | ---------------- | ------ |
| vip001   | 김VIP  | 30   | vip@example.com  | 10%    |

선택: 6

프로그램을 종료합니다.
```

회원 관리 프로그램의 추가 도전 부분에 대한 간략한 힌트입니다. 실제로는 더 많은 상세 구현 및 예외 처리, 인터페이스등을 자유롭게 확장해서 완성 해 보세요.