5. 객체지향 프로그래밍 (OOP)

클래스와 객체

✅ 1. 클래스(Class)란?

> **객체를 생성하기 위한 설계도** 또는 **템플릿**

- 변수(필드)와 메서드를 포함하는 사용자 정의 데이터 타입
- 현실 세계의 개념(사람, 동물, 자동차 등)을 **프로그래밍적으로 모델링**

✅ 2. 객체(Object)란?

> **클래스를 기반으로 만들어진 실체(instance)**
> 클래스의 변수, 메서드 등을 **구체적으로 사용할 수 있는 상태**

- 객체는 **상태(state)**와 **행동(behavior)**를 가짐
  - 상태: 멤버 변수(필드)
  - 행동: 메서드

✅ 3. 클래스와 객체의 관계

```
클래스 (설계도) → new 연산자 → 객체 (구체적 실체)
```

클래스	객체
Car	`new Car()`
Dog	`new Dog()`
Student	`new Student()`

✅ 4. 클래스 기본 구조

```java
public class Student {
    // 필드 (속성)
    String name;
    int age;

    // 메서드 (기능)
    void study() {
        System.out.println(name + " is studying.");
    }
}
```

✅ 5. 객체 생성 및 사용

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();     // 객체 생성
        s1.name = "Alice";              // 필드 설정
        s1.age = 20;
        s1.study();                     // 메서드 호출
    }
}
```

> `s1`은 `Student` 클래스의 인스턴스

✅ 6. 생성자(Constructor)

> 객체를 생성할 때 **초기화 작업을 수행**하는 메서드
> 클래스 이름과 동일하며 반환형 없음

🔹 기본 생성자

```
public Student() {
    System.out.println("Student 객체 생성됨!");
}
```

🔹 매개변수 생성자

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

✅ 7. this 키워드

> 현재 객체 자신을 가리키는 참조 변수

```
this.name = name;  // 필드와 매개변수를 구분
```

✅ 8. 여러 객체의 독립성

```
Student s1 = new Student("Alice", 20);
Student s2 = new Student("Bob", 25);
```

→ `s1`, `s2`는 서로 **완전히 독립적인 메모리 공간**에 생성됨
→ 필드 변경도 서로 영향을 주지 않음

✅ 9. 클래스와 객체의 메모리 구조

요소	위치
클래스 정의	메서드 영역(Method Area)
객체 데이터	힙(Heap)
참조 변수	스택(Stack)

🔹 예시 도식 (힙 & 스택)

```
Student s1 = new Student("Alice", 20);
```

```
Stack:                  Heap:
┌──────────────┐        ┌─────────────────┐
| s1           |──────▶| name: "Alice"|
└──────────────┘        | age: 20       |
                        └─────────────────┘
```

✅ 10. 예제: 간단한 자동차 클래스

```
public class Car {
    String model;
    int speed;

    void accelerate() {
        speed += 10;
        System.out.println(model + " speed: " + speed);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.model = "Tesla";
        c.accelerate();  // Tesla speed: 10
    }
}
```

✅ 11. 클래스와 객체 개념 요약

항목	클래스	객체
의미	설계도	실체

항목	클래스	객체
역할	멤버 정의	메모리상 인스턴스
생성 방법	`class` 키워드	`new` 연산자
위치	메서드 영역	힙 메모리
예시	`class Dog {}`	`Dog d = new Dog();`

✅ 12. 다음으로 확장할 수 있는 주제

- 캡슐화(Encapsulation)
- 접근 제어자 (`private`, `public`, `protected`)
- static 멤버와 인스턴스 멤버 차이
- 객체 배열, 객체 배열 초기화
- equals(), toString(), hashCode() 메서드 오버라이딩
- 참조 vs 얕은 복사 vs 깊은 복사

생성자와 초기화 블록

✅ 1. 생성자(Constructor)란?

> **객체가 생성될 때 자동으로 호출되는 특수한 메서드**

- 클래스 이름과 동일한 이름
- 반환 타입 없음 (`void`도 안 씀)
- 주로 객체의 **필드를 초기화**하는 데 사용

🔹 기본 생성자

```
public class Person {
    String name;

    // 기본 생성자 (인자가 없음)
    public Person() {
        this.name = "Unknown";
    }
}
```

🔹 매개변수 생성자

```
public class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }
}
```

🔹 생성자 오버로딩

```
public class Person {
    String name;
    int age;

    public Person() {
        this("Unknown", 0);
    }

    public Person(String name) {
        this(name, 0);
    }

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

> `this(...)`를 사용하여 **다른 생성자 호출 가능**
> 단, 반드시 **첫 줄**에 있어야 함.

✅ 2. 초기화 블록(Initialization Block)이란?

> **생성자보다 먼저 실행되는 코드 블록으로, 인스턴스 필드 초기화를 위한 보조 수단**

🔹 인스턴스 초기화 블록 (`{ ... }`)

```
public class Person {
    String name;

    {
        name = "기본이름";
        System.out.println("초기화 블록 실행");
    }

    public Person() {
        System.out.println("생성자 실행");
    }
}
```

> 출력 순서:

```
초기화 블록 실행
생성자 실행
```

🔹 정적 초기화 블록 (`static { ... }`)

> 클래스 로딩 시 단 한 번 실행되는 블록

```
public class Person {
    static {
        System.out.println("정적 초기화 블록 실행");
    }
}
```

> 객체를 생성하지 않아도 **클래스 로딩 시점에 실행됨**

✅ 3. 실행 순서

순서	단계	예시
1	클래스 로딩	`static 초기화 블록` 실행
2	객체 생성	`인스턴스 초기화 블록` 실행
3	생성자 실행	생성자 내부 코드 실행

🔹 예시: 순서 확인

```
public class Test {
    static {
        System.out.println("1. 정적 초기화 블록");
    }

    {
        System.out.println("2. 인스턴스 초기화 블록");
    }

    public Test() {
        System.out.println("3. 생성자");
    }

    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();
    }
}
```

> 출력 결과:

```
1. 정적 초기화 블록
2. 인스턴스 초기화 블록
3. 생성자
2. 인스턴스 초기화 블록
3. 생성자
```

✅ 4. 언제 생성자 vs 초기화 블록을 쓸까?

항목	생성자	초기화 블록
역할	매개변수 기반 객체 초기화	공통 초기화 로직 작성
위치	클래스 내부 메서드	클래스 내부 코드 블록
실행 시점	`new` 시점	생성자 실행 직전
오버로딩	가능	불가
호출 순서	초기화 블록 후	초기화 블록 선행 후 생성자

🔹 사용 예시: 여러 생성자에 공통 초기화가 필요한 경우

```
public class Config {
    String env;

    {
```

```
        // 모든 생성자에서 공통 초기화
        System.out.println("공통 초기화");
    }

    public Config() {
        System.out.println("기본 설정");
    }

    public Config(String env) {
        this.env = env;
        System.out.println("환경 설정: " + env);
    }
}
```

✅ 5. 초기화 블록을 남용하면 안 되는 이유

- 생성자에 넣을 수 있는 초기화를 굳이 블록에 넣으면 **코드 가독성 저하**
- **무조건 호출되는 코드**이므로 예외처리나 조건 분기 제한됨
- 일반적으로는 생성자에서 모든 초기화를 처리하는 게 더 **명시적**

✅ 6. 요약 정리

항목	생성자	인스턴스 초기화 블록	static 초기화 블록
실행 시점	객체 생성 시	생성자 전에 실행	클래스 로딩 시
역할	인스턴스 변수 초기화	공통 초기화	정적 필드 초기화
선언 위치	클래스 내부	클래스 내부	클래스 내부
사용 여부	보통 필수	선택적	클래스 당 1회
오버로딩	가능	불가	불가

접근제한자 (`public`, `private`, `protected`, default)

✅ 1. 접근 제한자란?

> 클래스, 메서드, 변수 등에 붙여서 **외부 접근 허용 범위**를 제어하는 키워드

- 목적: **정보 은닉**, **캡슐화**, **보안성**, **모듈 간 결합도 최소화**
- 4가지 종류:
  - `public`
  - `private`
  - `protected`
  - **default** (아무 것도 쓰지 않았을 때)

✅ 2. 접근 제한자 비교표

제한자	동일 클래스	동일 패키지	하위 클래스(상속)	외부 클래스
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
(default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

> 가장 개방적인 것은 `public`, 가장 제한적인 것은 `private`

✅ 3. 각 접근 제한자 설명

🔹 `public`

- **어디서든 접근 가능**
- 클래스, 생성자, 메서드, 필드에 사용
- 보통 **라이브러리 외부 제공 API**, **메인 클래스**, **public getter/setter** 등에서 사용

```
public class Student {
    public String name;

    public void study() {
        System.out.println("Studying");
    }
}
```

🔹 `private`

- **오직 동일 클래스 내부에서만 접근 가능**
- 외부 접근 **완전 차단**
- 캡슐화 핵심: 내부 로직 보호, **getter/setter로만 간접 접근 유도**

```
public class Student {
    private String name;

    public void setName(String n) {
        name = n;
    }

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

🔹 `protected`

- **같은 패키지** 또는 **다른 패키지의 하위 클래스**에서 접근 가능
- 상속 구조에서 **조심해서 사용**해야 함

```
public class Animal {
    protected void makeSound() {
        System.out.println("소리 냄");
    }
}

class Dog extends Animal {
    public void bark() {
        makeSound(); // 가능
    }
}
```

> 단, 다른 패키지에서 **상속만으로 접근 가능**, 그냥 객체 생성해서는 접근 불가

🔹 default (package-private)

- 아무 접근 제한자를 **명시하지 않으면 default**
- **동일 패키지에서만 접근 가능**
- 다른 패키지에선 접근 불가 (상속도 불가)

```
class Book {
    String title;   // default
}
```

> `Book`은 **동일 패키지 클래스에서만 사용 가능**

✅ 4. 적용 대상별 사용 가능 제한자

대상	`public`	`protected`	`default`	`private`
클래스	✅	❌	✅	❌
필드	✅	✅	✅	✅
메서드	✅	✅	✅	✅
생성자	✅	✅	✅	✅
내부 클래스	✅	✅	✅	✅

> **클래스 자체에는 `private`, `protected` 불가**

✅ 5. 예시 비교

```java
package pkg1;
public class Parent {
    public int a = 1;
    protected int b = 2;
    int c = 3; // default
    private int d = 4;
}
```

```java
package pkg2;
import pkg1.Parent;

public class Child extends Parent {
    public void accessTest() {
        System.out.println(a); // ✅ public
        System.out.println(b); // ✅ protected (상속)
        // System.out.println(c); ❌ default
        // System.out.println(d); ❌ private
    }
}
```

✅ 6. 실무에서의 사용 가이드라인

상황	추천 제한자
클래스 전체 공개	`public`
내부 구현 보호	`private`
상속만 허용	`protected`
같은 패키지 전용	`default`
멤버 변수	`private` + getter/setter
메서드	외부 사용 시 `public`, 내부 유틸은 `private` or `protected`

✅ 7. 요약 정리

제한자	외부 접근성	상속 접근성	실무 용도
public	전체 허용	전체 허용	API 제공
protected	같은 패키지 + 상속 허용	O	상속 시 재사용
default	같은 패키지	X	패키지 내부 구현 공유
private	오직 내부	X	은닉, 보안, 내부 전용

`this` 키워드

✅ 1. `this`란?

> 현재 메서드나 생성자가 속한 객체 자신을 참조하는 키워드

- 객체 내부에서 자기 자신을 가리킬 때 사용
- 주로 **인스턴스 변수와 지역 변수의 구분**, **생성자 간 호출**, **현재 객체 전달** 등에 사용됨

✅ 2. `this` 키워드의 주요 사용 목적

목적	설명
🔹 필드와 지역 변수 구분	매개변수 이름이 필드와 같을 때 구분
🔹 생성자 간 호출	다른 생성자를 호출해 코드 중복 제거
🔹 현재 객체 전달	메서드 인자로 자신을 넘길 때
🔹 현재 객체 반환	메서드 체이닝 가능

✅ 3. 필드와 지역 변수 구분

❌ 잘못된 예:

```
public class Person {
    String name;

    public void setName(String name) {
        name = name;  // ❌ 지역 변수 name이 자신과 대입됨
    }
}
```

✅ `this` 사용:

```
public class Person {
    String name;

    public void setName(String name) {
        this.name = name;  // ✅ 필드 name에 지역 변수 name을 대입
    }
}
```

✅ 4. 생성자에서 다른 생성자 호출

> `this(...)`를 통해 **같은 클래스의 다른 생성자**를 호출

```
public class Car {
    String model;
    int year;

    public Car() {
        this("Unknown", 2000); // 🔁 다른 생성자 호출
    }

    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

> 주의: `this(...)`는 반드시 **생성자의 첫 줄**에만 사용 가능

✅ 5. 현재 객체를 메서드 인자로 전달

```
public class Student {
    String name;

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

    public void register() {
        SchoolSystem.register(this);  // 현재 객체 자신을 넘김
    }
}
```

```
public class SchoolSystem {
    public static void register(Student s) {
        System.out.println(s.name + " 등록됨");
    }
}
```

✅ 6. 현재 객체 반환 (`return this;`)

> **메서드 체이닝(method chaining)**을 구현할 때 유용

```
public class Builder {
    String field1;
    int field2;

```

```
    public Builder setField1(String value) {
        this.field1 = value;
        return this;
    }

    public Builder setField2(int value) {
        this.field2 = value;
        return this;
    }
}
```

```
Builder b = new Builder()
                .setField1("hello")
                .setField2(42);
```

✅ 7. `this` 없이도 되는 경우

```
public class Student {
    String name;

    public void setName(String n) {
        name = n;  // 필드와 지역변수 이름이 다르면 this 생략 가능
    }
}
```

✅ 8. `this` 키워드의 동작 원리

- 인스턴스 메서드 내부에는 **컴파일러가 자동으로 `this`를 삽입**
- 정적 메서드(static)에서는 `this`를 사용할 수 없음

```
public static void someStatic() {
    // System.out.println(this);  // ❌ 컴파일 에러
}
```

✅ 9. 요약 정리

사용 방식	의미 및 용도	예시
`this.필드명`	필드와 지역 변수 구분	`this.name = name;`
`this(...)`	생성자 간 호출	`this("John", 20);`
`this`	자기 자신을 인자로 전달	`otherMethod(this);`
`return this;`	체이닝용 반환	`return this;`

✅ 10. 연습 문제

```java
public class Book {
    String title;

    public Book(String title) {
        this.title = title;
    }

    public Book printTitle() {
        System.out.println("제목: " + title);
        return this;
    }

    public Book updateTitle(String newTitle) {
        this.title = newTitle;
        return this;
    }

    public static void main(String[] args) {
        new Book("Java").printTitle().updateTitle("Advanced Java").printTitle();
    }
}
```

> 출력:

```
제목: Java
제목: Advanced Java
```

객체 간 협력 및 관계 (has-a, is-a)

✅ 1. 객체 간 협력이란?

> 하나의 객체가 **혼자서 모든 일을 하지 않고,**
> **다른 객체와 상호작용하며 역할을 나누는 구조**

- 현실 세계도 마찬가지야: 사람이 주문하고, 점원이 받고, 요리사가 조리하고...
- Java에서는 각 객체가 **필드를 통해 다른 객체를 참조하거나,**
  메서드를 호출함으로써 협력함

✅ 2. 객체 간 관계: 두 가지 핵심

관계	설명	키워드
is-a	상속 관계	`상속(Inheritance)`
has-a	포함 관계	`합성(Composition)` or `집합(Aggregation)`

✅ 3. is-a 관계 (상속)

> **A is a B**: A는 B의 일종이다
> → 자식 클래스가 부모 클래스의 특성을 **상속**받는 구조

🔹 예시:

```
class Animal {
    void eat() {
        System.out.println("먹는다");
    }
}

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

```
Dog d = new Dog();
d.eat();   // 부모 메서드 사용 가능
d.bark();  // 자식 메서드 사용
```

> Dog is an Animal ✅ (올바른 상속)

✅ 핵심 특징

- `is-a` 관계는 **다형성(polymorphism)**의 기반
- 모든 `Dog`는 `Animal`이지만, 모든 `Animal`이 `Dog`는 아님

✅ 4. has-a 관계 (합성/집합)

> **A has a B**: A는 B를 **필드로 가지고 있다**

- **구성 요소**를 포함하는 구조
- 객체 간의 **협력**은 대부분 has-a 관계로 구현

🔹 예시:

```
class Engine {
    void start() {
        System.out.println("엔진이 켜짐");
    }
}

class Car {
    private Engine engine = new Engine(); // has-a 관계

    void drive() {
        engine.start();
```

```
        System.out.println("차가 움직임");
    }
}
```

> Car has-a Engine ✅

✅ 5. Composition vs Aggregation (합성 vs 집합)

관계 유형	설명	생명 주기
Composition	강한 포함	A 없으면 B도 소멸
Aggregation	약한 포함	A 없어도 B는 독립 존재

🔹 Composition 예:

```
class Heart {
    void beat() {
        System.out.println("심장이 뛴다");
    }
}

class Human {
    private Heart heart = new Heart();
}
```

> Human이 죽으면 Heart도 끝

🔹 Aggregation 예:

```
class Student {}

class Classroom {
    List<Student> students;

    public Classroom(List<Student> students) {
        this.students = students;  // 외부에서 주입
    }
}
```

> Classroom 없어져도 Student는 존재 가능

✅ 6. 실제 협력 구조 예시

🔹 시나리오: 주문 → 결제

```
class Customer {
    void order(Food food, Clerk clerk) {
        clerk.takeOrder(food);
    }
}

class Clerk {
    void takeOrder(Food food) {
        System.out.println(food.name + " 주문 받음");
        food.prepare();
    }
}

class Food {
    String name = "라면";
    void prepare() {
        System.out.println("음식 준비됨");
    }
}
```

```
Customer c = new Customer();
Clerk k = new Clerk();
Food f = new Food();

c.order(f, k);
```

> 객체들끼리 **직접 기능을 수행하지 않고,**
> **서로 협력하여 전체 로직 수행**

✅ 7. UML로 보는 관계 요약

관계	UML 기호	설명
상속	빈 삼각형 →	일반화 (`is-a`)
합성	● 선	전체와 부분 (전체 소멸 시 부분도 소멸)
집합	◇ 선	느슨한 포함 관계
연관	→ 선	협력 관계 (단방향, 양방향)

✅ 8. 실무 설계 시 판단 기준

판단 기준	관계 형태
"A는 B의 일종인가?"	`is-a` → 상속
"A는 B를 가지고 있는가?"	`has-a` → 합성 or 집합
"A 없이도 B가 의미 있나?"	의미 있음 → 집합 없음 → 합성
"서로 어떤 역할을 맡는가?"	협력 구조로 설계 (역할별 분리)

✅ 9. 요약 정리

관계	키워드	예시	설명
is-a	상속	Dog → Animal	공통 기능 재사용
has-a	합성/집합	Car → Engine	부품 조립 구조
협력	메서드 호출	Customer → Clerk	역할 분리 & 연계

캡슐화, 추상화, 상속, 다형성

✅ 1. 캡슐화 (Encapsulation)

> **데이터(필드)**와 **동작(메서드)**을 하나로 묶고,
> 외부에서는 **제공된 인터페이스로만 접근**하도록 제한하는 설계 원칙.

🔹 목적

- 내부 구현 은닉
- 잘못된 접근 방지 (보안성)
- 유지보수 용이

🔹 구현 방법

- 필드를 `private`으로 선언
- 외부에 `getter/setter`를 통해서만 접근

```java
public class Account {
    private int balance;

    public void deposit(int amount) {
        if (amount > 0) balance += amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

> 외부는 `balance`를 직접 수정할 수 없고, 메서드를 통해서만 접근 가능함

✅ 2. 추상화 (Abstraction)

> **복잡한 내부 로직은 숨기고,**
> **필요한 기능만 인터페이스로 외부에 노출**하는 것.

🔹 목적

- 공통 기능 정의
- 사용자와 구현을 분리
- 인터페이스 기반 설계 가능

🔹 구현 방법

- 추상 클래스 (`abstract class`)
- 인터페이스 (`interface`)

```java
interface RemoteControl {
    void turnOn();
    void turnOff();
}
```

```java
class TV implements RemoteControl {
    public void turnOn() { System.out.println("TV 켬"); }
    public void turnOff() { System.out.println("TV 끔"); }
}
```

> TV가 어떻게 켜지는지는 몰라도, `turnOn()`이라는 명령으로 동작만 알 수 있음

✅ 3. 상속 (Inheritance)

> 기존 클래스를 **재사용하여 새로운 클래스를 만드는 기법**.
> 부모 클래스의 속성과 메서드를 **자식 클래스가 물려받음**.

🔹 목적

- 코드 재사용
- 계층적 구조 설계
- 다형성의 기반

```
class Animal {
    void eat() {
        System.out.println("먹는다");
    }
}

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

> `Dog`는 `Animal`의 `eat()` 메서드를 상속받아 사용할 수 있음

✅ 4. 다형성 (Polymorphism)

> **하나의 타입(부모 또는 인터페이스)**으로 여러 객체를 **동일하게 다룰 수 있는 능력**

🔹 목적

- 코드 확장성과 유연성
- 객체 교체 및 대체 가능성

🔹 구현 방법

- 상속과 오버라이딩
- 인터페이스 구현

```
class Animal {
    void speak() {
        System.out.println("...");
    }
}

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

```
12
13 class Cat extends Animal {
14     void speak() {
15         System.out.println("야옹");
16     }
17 }
```

```
1 Animal a1 = new Dog();
2 Animal a2 = new Cat();
3
4 a1.speak();  // 멍멍
5 a2.speak();  // 야옹
```

> `Animal` **타입**으로 통일해서 다룰 수 있으나,
> 실제 실행 결과는 **객체의 타입에 따라 달라짐 (동적 바인딩)**

✅ 핵심 비교 요약

원칙	설명	Java 문법 요소
캡슐화	필드 보호, 인터페이스 제공	`private`, `getter/setter`
추상화	핵심만 보여주기	`abstract class`, `interface`
상속	코드 재사용	`extends`, `super`
다형성	하나의 타입, 다양한 실행	오버라이딩, 인터페이스, 동적 바인딩

✅ 객체지향 4원칙 통합 예제

```
1  interface Shape {                      // 추상화
2      double area();
3  }
4
5  class Circle implements Shape {        // 다형성 + 추상화
6      private double radius;             // 캡슐화
7
8      public Circle(double r) {
9          this.radius = r;
10     }
11
12     public double area() {
13         return Math.PI * radius * radius;
14     }
15 }
```

```java
public class Test {
    public static void main(String[] args) {
        Shape s = new Circle(5);       // 다형성
        System.out.println(s.area());  // 실제는 Circle.area() 실행
    }
}
```

✅ 객체지향 4원칙 적용의 이점

- 복잡도 감소
- 변경에 강한 구조
- 유연한 확장
- 팀 개발에 적합한 역할 분리 가능