7. 추상 클래스와 인터페이스

추상 클래스 (`abstract`)

✅ 1. 추상 클래스란?

> **하나 이상의 추상 메서드를 포함**하며,
> **직접 인스턴스화할 수 없는 클래스**

- `abstract` 키워드를 클래스 선언부에 사용
- **미완성 설계 템플릿** 역할
- **자식 클래스가 반드시 오버라이딩하여 완성**

```
abstract class Animal {
    abstract void sound(); // 추상 메서드
    void breathe() {
        System.out.println("숨을 쉰다");
    }
}
```

✅ 2. 추상 클래스 문법

```
abstract class ClassName {
    // 추상 메서드
    abstract 반환형 메서드명(매개변수);

    // 일반 메서드
    void 일반메서드() { ... }
}
```

사용 예시:

```
abstract class Shape {
    abstract double area(); // 반드시 구현해야 함
}

class Circle extends Shape {
    double radius;
    Circle(double r) { radius = r; }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}
```

✅ 3. 추상 클래스의 특징

특징	설명
인스턴스 생성 ❌	`new Shape()` 불가능
추상 메서드 포함	0개 이상 허용 (`abstract` 없는 메서드도 가질 수 있음)
상속 필수	자식 클래스가 `abstract` 메서드 구현해야 함
생성자 허용	객체 직접 생성은 안 되지만, **자식 생성 시 호출됨**
필드 가질 수 있음	일반 클래스처럼 필드 선언 가능
일반 메서드 포함 가능	공통 기능은 여기서 제공 가능

✅ 4. 왜 추상 클래스를 사용하는가?

- **공통의 틀 제공 (템플릿 역할)**
- **상속 기반 다형성 구현**
- **구현 강제화 (일관성 보장)**
- **중복 제거 및 유지보수성 향상**

✅ 5. 예제: 동물 소리 추상화

```
abstract class Animal {
    abstract void sound();
    void sleep() {
        System.out.println("잠자는 중...");
    }
}

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

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

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog();
```

```
        Animal a2 = new Cat();

        a1.sound(); // 멍멍
        a2.sound(); // 야옹
    }
}
```

✅ 6. 추상 클래스와 생성자

```
abstract class Base {
    Base() {
        System.out.println("Base 생성자");
    }
}

class Derived extends Base {
    Derived() {
        System.out.println("Derived 생성자");
    }
}
```

```
Derived d = new Derived();
// 출력:
// Base 생성자
// Derived 생성자
```

> **추상 클래스도 생성자 존재 가능!**
> 단, 직접 `new Base()`는 불가

✅ 7. 추상 클래스 vs 인터페이스

항목	추상 클래스	인터페이스 (Java 8+ 기준)
키워드	`abstract class`	`interface`
다중 상속	❌ 단일 클래스만 상속 가능	✅ 다중 구현 가능
필드	가질 수 있음	`public static final` 상수만 가능
메서드	일반 메서드 + 추상 메서드	`abstract`, `default`, `static` 메서드 가능
생성자	있음	없음
목적	**"is-a" 관계 구현** (상속 기반)	**기능 정의 및 다중 구현**
사용 시점	공통 로직 + 일부 추상화	순수 계약만 정의하거나 유틸 역할로 사용

> 일반적으로 **공통 기능이 많고 기본 구현이 필요**할 때는 추상 클래스
> **다중 구현이 필요하거나 독립 기능**이면 인터페이스

✅ 8. abstract 메서드만 있는 경우?

```
abstract class Shape {
    abstract void draw();
}
```

> 인터페이스와 유사하지만, 추상 클래스는 필드와 일반 메서드를 포함할 수 있다는 차이가 있음

✅ 9. 주의 사항

상황	설명
일반 클래스가 추상 메서드 포함 ❌	반드시 추상 클래스여야 함
추상 메서드는 구현부 없음	`{}` 쓰면 컴파일 에러
추상 메서드 오버라이딩 필수	자식 클래스에서 미구현 시, 자식도 `abstract`로 선언해야 함

✅ 10. 요약 정리

항목	설명
정의	하나 이상의 추상 메서드를 가진 클래스
목적	템플릿 제공, 구현 강제화
특징	인스턴스 생성 불가, 일반 메서드/필드 포함 가능
상속	자식 클래스에서 구현 강제
인터페이스와 차이	상속은 단일, 공통 로직 구현 가능

인터페이스의 정의와 구현 (`implements`)

✅ 1. 인터페이스란?

> 인터페이스는 클래스가 **반드시 구현해야 할 메서드들의 집합**을 정의하는 **계약서**다.

- 모든 메서드는 **기본적으로** `public abstract`
- 모든 필드는 **기본적으로** `public static final` **(상수)**
- **다중 구현 가능**
- Java 8 이후부터는 `default`, `static`, `private` **메서드**도 정의 가능

✅ 2. 인터페이스 선언 문법

```
public interface 인터페이스이름 {
    반환형 메서드이름(매개변수);
}
```

```
public interface Animal {
    void sound();  // public abstract가 생략된 형태
}
```

✅ 3. 인터페이스 구현: `implements`

```
public class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

> 클래스는 반드시 **인터페이스의 모든 메서드를 구현**해야 함

✅ 4. 인터페이스 다중 구현

```
interface Walkable {
    void walk();
}

interface Runnable {
    void run();
}

class Human implements Walkable, Runnable {
    public void walk() { System.out.println("걷는다"); }
    public void run()  { System.out.println("뛴다"); }
}
```

> Java는 **다중 상속은 안 되지만, 인터페이스는 다중 구현 가능**

✅ 5. 인터페이스 상속

> 인터페이스 간에도 **상속 가능** (`extends` 사용)

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

interface B extends A {
    void b();
}
```

> A → B로 확장된 계약을 형성

✅ 6. Java 8 이후: `default`, `static` 메서드

🔹 default 메서드

```
interface Printable {
    default void print() {
        System.out.println("기본 출력");
    }
}
```

> 인터페이스에 **기본 구현을 제공**, 자식 클래스에서 오버라이딩 가능

🔹 static 메서드

```
interface MathUtil {
    static int square(int x) {
        return x * x;
    }
}
```

> 클래스처럼 **인터페이스명으로 호출** → `MathUtil.square(5)`

🔹 Java 9 이후: private 메서드도 가능

```
interface Helper {
    private void log() {
        System.out.println("로그 출력");
    }
}
```

> 인터페이스 내부에서만 쓰는 **헬퍼 메서드 구현 가능**

✅ 7. 인터페이스와 다형성

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("원을 그린다");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("사각형을 그린다");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();

        s1.draw(); // 원을 그린다
        s2.draw(); // 사각형을 그린다
    }
}
```

> 인터페이스 타입으로 다양한 구현을 참조 → **다형성(polymorphism)** 실현

✅ 8. 추상 클래스 vs 인터페이스

항목	추상 클래스	인터페이스
선언 키워드	`abstract class`	`interface`
다중 상속	❌ 단일 클래스만 상속 가능	✅ 다중 인터페이스 구현 가능
필드	변수/상수 모두 가능	상수만 (`public static final`)
메서드	일반 + 추상 메서드	추상 + default/static 메서드
생성자	있음	없음
목적	**공통 로직 제공 + 설계 템플릿**	**행동 규약 정의 (계약)**
사용 예	동일 기능 구현 + 기본 구현 제공	행동을 정의 (걷기, 출력 등)

✅ 9. 인터페이스 + 익명 클래스 구현

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("익명 객체 실행");
    }
};
```

> 람다식이나 간단한 구현에 많이 쓰임

✅ 10. 요약 정리

항목	설명
목적	다형성, 계약 기반 설계, 느슨한 결합
구현 방식	`implements`
다중 구현	O
상속 가능	인터페이스끼리는 `extends` 가능
메서드 유형	`abstract`, `default`, `static`, `private`
필드 유형	`public static final` 상수

다중 구현 및 다형성

✅ 1. 다중 구현(Multiple Implementation)이란?

> 하나의 클래스가 **둘 이상의 인터페이스를 구현하는 것**

문법

```
class 클래스명 implements 인터페이스1, 인터페이스2 {
    // 모든 메서드 구현 필요
}
```

> Java는 **다중 클래스 상속은 금지**지만
> **다중 인터페이스 구현은 허용** → 유연한 설계 가능

✅ 2. 예제: 다중 인터페이스 구현

```
interface Flyable {
    void fly();
}

interface Swimmable {
```

```
    void swim();
}

class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("날다");
    }
    public void swim() {
        System.out.println("헤엄치다");
    }
}
```

> `Duck` 클래스는 `Flyable`, `Swimmable` **모두의 계약을 이행**
> → 객체 다형성 활용 가능

✅ 3. 다형성(Polymorphism)이란?

> **같은 타입으로 다양한 객체를 참조하고 호출**할 수 있는 개념
> → **동일한 메시지로 다른 동작을 유도**

✅ 4. 다형성의 전제: 상속 or 인터페이스 구현

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("원을 그린다");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("사각형을 그린다");
    }
}
```

✅ 5. 다형성 활용 예

```
Shape s1 = new Circle();
Shape s2 = new Rectangle();

s1.draw();  // 원을 그린다
s2.draw();  // 사각형을 그린다
```

- `Shape` 타입으로 `Circle`, `Rectangle` 객체를 사용할 수 있음

- → 코드의 **유연성, 재사용성, 확장성** 향상

✅ 6. 다형성과 배열 + 반복문

```
Shape[] shapes = { new Circle(), new Rectangle() };

for (Shape s : shapes) {
    s.draw();  // 다형성에 의해 각자의 draw() 호출
}
```

> 동질적 배열 + 동작 다양화 → **전략 패턴의 기초**

✅ 7. 인터페이스 기반 설계의 장점

항목	설명
느슨한 결합	구현체와 사용자를 분리 가능
테스트 용이성	모의 객체(mock) 생성 가능
유지보수	구현 클래스 교체가 쉬움
OCP 원칙	새로운 기능 추가 시 기존 코드 수정 불필요

✅ 8. 익명 클래스와 다형성 결합

```
Runnable task = new Runnable() {
    public void run() {
        System.out.println("익명 객체 실행");
    }
};

task.run(); // 다형성 + 무명 클래스
```

> 런타임에 새로운 행동을 지정 가능

✅ 9. 다형성과 instanceof 활용

```
for (Shape s : shapes) {
    if (s instanceof Circle) {
        System.out.println("원이네요!");
    }
}
```

> 타입 검사 후 안전한 다운캐스팅 가능

✅ 10. 요약 정리

항목	설명
다중 구현	한 클래스가 여러 인터페이스를 구현
다형성	하나의 타입으로 여러 객체 참조 및 호출
전제 조건	인터페이스 구현 or 클래스 상속
효과	유연한 구조, 코드 재사용성 향상, 테스트 용이성
핵심 문법	`implements`, `interface`, `override` 등

Java 8 이후 인터페이스의 `default`, `static` 메서드

✅ 1. Java 8 이전의 인터페이스

> 인터페이스는 오직 **추상 메서드와 상수만** 가질 수 있었음.

```
interface Animal {
    void sound(); // public abstract
}
```

- 구현 없이 선언만 가능
- Java는 **다중 상속을 허용하지 않기 때문에** 공통 로직을 정의할 수 없어 불편함 존재

✅ 2. Java 8 이후의 변화

> 인터페이스에 `default` **메서드와** `static` **메서드**를 도입함으로써,
> **구현 코드를 포함할 수 있게 됨**

주요 목적:

- **기존 인터페이스를 변경 없이 확장 가능**
- **코드 중복 최소화**
- **유틸리티 또는 기본 구현 제공 가능**

✅ 3. `default` 메서드란?

> 인터페이스에서 **기본 구현을 제공**하는 메서드
> 자식 클래스는 **선택적으로 오버라이딩** 가능

🔹 문법

```
interface MyInterface {
    default void greet() {
        System.out.println("Hello from interface!");
    }
}
```

🔹 사용 예시

```
interface Printer {
    default void print() {
        System.out.println("기본 프린트 출력");
    }
}

class CustomPrinter implements Printer {
    // 오버라이딩 하지 않으면 default 그대로 사용
}
```

```
Printer p = new CustomPrinter();
p.print(); // 출력: 기본 프린트 출력
```

✅ 4. `static` 메서드란?

> **인터페이스 소속의 정적 유틸리티 메서드**
> 반드시 **인터페이스명으로 호출**

🔹 문법

```
interface MathUtil {
    static int square(int x) {
        return x * x;
    }
}
```

```
int result = MathUtil.square(4);  // ✅ 16
```

✅ 5. 두 메서드 차이 정리

항목	`default` 메서드	`static` 메서드
호출 방식	인스턴스에서 호출	인터페이스명으로 호출
목적	기본 동작 구현	유틸리티 제공

항목	`default` 메서드	`static` 메서드
오버라이딩	가능	불가능
예시	`p.print()`	`MathUtil.square(3)`

✅ 6. default 메서드 충돌 해결 (다중 인터페이스 구현 시)

```
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

interface B {
    default void hello() {
        System.out.println("Hello from B");
    }
}

class C implements A, B {
    @Override
    public void hello() {
        // 충돌 해결
        A.super.hello();
    }
}
```

> `A.super.hello()` 또는 `B.super.hello()`로 명시적 선택 가능
> **반드시 충돌 해결 코드를 작성해야 컴파일 됨**

✅ 7. 인터페이스 변경의 유연성 확보

기존 코드:

```
interface OldInterface {
    void run();  // Java 7 이하
}
```

Java 8 이상 확장:

```
interface OldInterface {
    void run();  // 여전히 추상

    default void log() {
        System.out.println("로그 출력");
    }
}
```

> 기존 구현체에는 영향 없음 → **레거시와의 호환성 유지**

✅ 8. 응용: Iterable 예제

```
default void forEach(Consumer<? super T> action)
default Spliterator<T> spliterator()
```

> `Iterable`, `Collection`, `List`, `Map` 등 **기본 인터페이스들도 default 메서드로 강화됨**

✅ 9. 주의사항

- `default` 메서드가 있다고 해서 상태(state)를 가질 수는 없음 (필드는 `public static final` 만 가능)
- 다중 구현 시 충돌 조심
- `default` 메서드는 **인터페이스 간 계약 확장의 수단**일 뿐, 모든 공통 로직을 담는 용도는 아님

✅ 10. 요약 정리

항목	설명
`default`	인스턴스 기반 기본 구현 제공, 오버라이딩 가능
`static`	유틸리티 메서드 제공, 오버라이딩 불가
목적	인터페이스 기능 확장, 공통 로직 제공
활용	레거시 호환, 공통 로직 템플릿, 다형성 강화
주의점	충돌 시 명시적 해결 필요 (`인터페이스.super.메서드()`)