회고록

❓ enoum타입

내부클래스 외부클래스의 사용방법,개념,실사례 and 각 접근제한자 별로의 의미

함수형 인터페이스

객체지향

람다표현식

중첩클래스

new로 안 만들면 String만 저장하는 메모리영역에 저장된다?(

heap,stack,metaspace(메서드영역) 중에서 metaspace에 저장된대요)→ 리터럴 문자열: **클래스 로딩 시 상수 풀(Constant Pool)** → JVM의 메서드 영역/Metaspace에서 관리

1) enum 타입

개념

- **서로 관련된 상수들의 타입 안전한 집합**. 실수로 잘못된 값을 넣을 위험을 없앱니다.
- 각 상수는 사실상 **싱글톤 객체**이며, 필드/메서드/생성자도 가질 수 있습니다.

언제/왜

- 상태/코드 값을 **정형화**하고 **스위치/분기**를 안전하게 할 때.
- 문자열/정수 상수보다 **컴파일러 체크**와 **자동 완성**이 뛰어남.

예시

```
public enum OrderStatus {
    CREATED("접수"), PAID("결제"), SHIPPED("출고"), CANCELLED("취소");
```

```
    private final String label;
    OrderStatus(String label) { this.label = label; }
    public String label() { return label; }
}

// 사용
OrderStatus s = OrderStatus.PAID;
switch (s) {
    case PAID → System.out.println("결제 완료: " + s.label());
}
```

- 유틸: `values()`, `valueOf("PAID")`, `name()`, `ordinal()`
- **실사례**: 주문 상태, 권한(Role), 에러 코드, 머신 상태 등

---

2) 내부/외부(중첩) 클래스 + 접근 제한자

개념

- **외부 클래스(Outer)**: 일반 클래스.
- **중첩 클래스(Nested)**: 클래스 안의 클래스(내부 클래스).
  - **인스턴스 내부 클래스**: 외부 인스턴스와 연결
  - **정적 중첩 클래스(static nested)**: 외부 인스턴스 없이 독립
  - **지역 내부 클래스**: 메서드 내부에 선언
  - **익명 내부 클래스**: 이름 없이 1회성 구현

언제/왜

- 외부와 **강결합된 보조 타입**을 외부에 노출하지 않고 **응집**/캡슐화.
- 자료구조 내부 노드, 빌더 패턴, 이벤트 핸들러에 적합.

간단 예시

```
class LinkedList<E> {
    private static class Node<E> {  // 정적 중첩: 외부 인스턴스 필요 X
```

```
        E val; Node<E> next;
        Node(E v) { this.val = v; }
    }
    private Node<E> head;
}
```

실사례

- `Map.Entry`(인터페이스)
- `LinkedList.Node`(JDK 내부)
- 빌더 패턴의 `public static class Builder { ... }`

접근 제한자 요약

대상	public	protected	(default)	private
최상위 클래스	O	X	O(패키지 내)	X
멤버(필드/메서드/중첩클래스)	O	O(패키지+하위)	O(패키지)	O(클래스 내부만)

- **최상위 클래스**는 `public` 또는 **패키지 전용**(기본)만 가능.
- **중첩 클래스**는 네 가지 모두 가능(캡슐화에 유리).

---

3) 함수형 인터페이스 & 람다 표현식

함수형 인터페이스 (SAM)

- **추상 메서드가 1개**인 인터페이스. `@FunctionalInterface` 권장.
- 전략/콜백을 **함수처럼 전달**하기 위한 타입.

```
@FunctionalInterface
interface Discount { int apply(int price); }
```

람다 표현식

- 함수형 인터페이스의 인스턴스를 **간단히 작성하는 문법**.

```
Discount rate10 = p → (int)(p * 0.9);
System.out.println(rate10.apply(10000)); // 9000
```

팁

- **메서드 참조**: `String::length`, `Integer::parseInt`
- **effectively final**: 람다에서 캡처하는 지역변수는 사실상 불변이어야 함
- **this**: 람다의 `this`는 **바깥 인스턴스**를 가리킴(익명 클래스와 다름)

실사례: `Comparator.comparing(...)`, 스트림 API, 이벤트 핸들러, 전략 주입

---

4) 객체지향 핵심

개념

- **추상화**: 본질만 모델링
- **캡슐화**: 상태 보호(접근제어, 불변성, 유효성)
- **상속**: 공통 뼈대 재사용(단, 남용 금지)
- **다형성**: 상위 타입으로 코딩, 구현은 갈아끼움

응용 포인트

- **인터페이스 = 계약 / 추상 클래스 = 공통 구현 뼈대**
- **합성(조합) 우선, 상속은 신중히**
- 전략은 **인터페이스+람다**로 주입, 알고리즘 골격은 **추상 클래스**로 템플릿화

```
@FunctionalInterface interface Formatter { String fmt(String s); }

abstract class Exporter {
    private final Formatter f;
    protected Exporter(Formatter f){ this.f = f; }
    public final void export(String raw){ write(f.fmt(pre(raw))); }
    protected String pre(String s){ return s.trim(); } // 공통
```

```
    protected abstract void write(String out);        // 훅(오버라이드)
}
class FileExporter extends Exporter {
    public FileExporter(Formatter f){ super(f); }
    protected void write(String out){ System.out.println("FILE: "+out); }
}
new FileExporter(str -> "[JSON]"+str).export(" data ");
```

---

5) 문자열과 메모리(상수 풀, Heap/Stack/Metaspace)

핵심 정리

- **지역/매개/참조변수 → Stack**
- `new`**로 만든 객체 → Heap**
- **클래스 메타데이터 & “클래스의 상수 풀(constant pool)” → Metaspace(메서드 영역)**

> 중요 구분
>
> - “클래스의 **상수 풀 테이블**(메타데이터)”은 **Metaspace**에 있음.
> - 하지만 **String 인턴 풀(실제 String 객체를 공유)**은 **Heap**에 있음(현대 HotSpot).
> - 즉, 리터럴 `"Hello"`는 상수 풀에 **기호 정보**로 존재하고, 필요 시 **Heap의 String 풀**(interned) 객체로 연결됩니다.

케이스별

```
String a = "Hello";          // 같은 리터럴은 같은 interned 객체를 참조
String b = "Hello";
System.out.println(a == b);   // true

String c = new String("Hello"); // Heap에 새 객체
```

```
System.out.println(a == c);     // false

String d = c.intern();          // Heap 객체를 풀 참조로 교체
System.out.println(a == d);     // true
```

컴파일 타임 상수 접합

```
String s1 = "He" + "llo";   // 컴파일 타임에 "Hello"로 상수 접합 → 풀 공유
String part = "He";
String s2 = part + "llo";   // 런타임 접합 → 새 객체(보통 StringBuilder 경유)
System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

활용 포인트

- 중복 리터럴 공유로 **메모리 절약/빠른 비교**
- 외부 입력 등 반복 문자열은 `intern()` 으로 풀 공유 최적화(과도 사용은 주의)

내부/중첩 클래스 — 한눈에 실무 팁

- **정적 중첩 클래스**: 빌더/노드/DTO 그룹핑(외부 인스턴스 불필요)
- **인스턴스 내부 클래스**: 외부 인스턴스 상태를 자연스럽게 쓰는 보조 객체
- **지역/익명 클래스**: 1회성 콜백/리스너(요즘은 **람다**가 더 간결)

접근 제한자 — 선택 가이드

- **public**: 외부 노출 API
- **(default)**: 같은 패키지에서만(모듈 내부 구현 숨길 때)
- **protected**: 상속 계층 + 같은 패키지
- **private**: 클래스 내부 구현만

"언패킹"과 getter의 차이

- **언패킹(unpacking)**: 보통 리스트, 튜플 등의 묶음을 풀어 여러 변수에 나누는 개념(파이썬에서 주로 사용).
- **getter**: 객체 내부 필드를 외부에 노출하지 않고 접근하게 하는 메서드.

> ✅ getter 메서드는
>
> **객체 내부의 private 필드 값에 안전하게 접근하기 위한 캡슐화 (encapsulation) 수단**
>
> 입니다.
>
> 즉,
>
> **외부에서 필드 값에 직접 접근할 수 없게 막고, 대신 메서드를 통해 읽게 만드는 것**
>
> 입니다.

→ 목적은 *정보은닉 + 캡슐화*, 내부 구현을 바꾸더라도 외부 코드에 영향을 주지 않기 위함.