12. 내부 클래스

인스턴스 내부 클래스

🟦 1. 인스턴스 내부 클래스란?

> 클래스 안에 선언된 **인스턴스 멤버 클래스**로,
> 바깥 클래스(Outer Class)의 **인스턴스에 종속**되는 구조.

```
class Outer {
    class Inner {
        // 인스턴스 내부 클래스
    }
}
```

📌 외부 클래스의 객체가 **먼저 생성되어야** 내부 클래스 객체를 생성할 수 있음

🧪 2. 기본 사용 예시

```
public class Outer {
    private String outerField = "Outer Field";

    public class Inner {
        public void show() {
            System.out.println("Accessing from Inner: " + outerField);
        }
    }

    public void createInner() {
        Inner in = new Inner(); // 내부 클래스 생성
        in.show();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); // 반드시 outer 객체가 필요
        inner.show();
    }
}
```

🟧 3. 주요 특징 요약

항목	설명
선언 위치	클래스 내부, 메서드 외부
접근 가능 범위	외부 클래스의 모든 멤버 (private 포함)
외부 객체 필요 여부	✅ 필요 (`outer.new Inner()`)
정적 멤버 정의 가능 여부	❌ 불가 (static 변수/메서드 정의 X)

🟩 4. 외부 클래스 참조법 (`OuterClass.this`)

내부 클래스에서 외부 클래스 객체를 명시적으로 참조하고 싶다면:

```
public class Outer {
    private int number = 42;

    public class Inner {
        public void printOuter() {
            System.out.println(Outer.this.number); // 외부 클래스 명시적 참조
        }
    }
}
```

🟥 5. 외부 클래스와 멤버 이름 충돌 시

```
public class Outer {
    int value = 10;

    class Inner {
        int value = 20;

        public void print() {
            int value = 30;
            System.out.println(value);            // 30 (지역)
            System.out.println(this.value);       // 20 (내부 클래스 멤버)
            System.out.println(Outer.this.value); // 10 (외부 클래스 멤버)
        }
    }
}
```

🚧 6. 주의점

- 내부 클래스는 **정적(static)으로 선언할 수 없음**
  → 정적 멤버 클래스(`static class`)는 별도로 있음
- 내부 클래스 객체를 생성하려면 반드시 외부 클래스 인스턴스가 **먼저** 있어야 함

```
Outer.Inner inner = new Outer().new Inner(); // O
Outer.Inner inner = new Outer.Inner();       // ❌ 컴파일 에러
```

🔌 7. 어디에 쓰일까?

- 외부 객체의 **컨텍스트에 종속적인 기능**을 정의할 때
- GUI 이벤트 핸들러 (Swing 등)
- 캡슐화를 유지하면서 **구조적으로 논리적 결합**이 필요한 경우

🔚 요약

항목	설명
클래스 위치	외부 클래스 내부 (메서드 외부)
생성 방법	`Outer outer = new Outer(); Outer.Inner in = outer.new Inner();`
외부 클래스 접근	`Outer.this` 로 접근 가능
정적 멤버 정의	❌ 불가
일반 용도	외부 클래스 기능과 밀접한 하위 개념 구현

정적 내부 클래스 (`static class`)

🟦 1. 정적 내부 클래스란?

> 외부 클래스 내부에 `static`**으로 선언된 내부 클래스**.
> 인스턴스 내부 클래스와 달리 **외부 클래스의 인스턴스에 종속되지 않음**.

```
class Outer {
    static class StaticInner {
        // 정적 내부 클래스
    }
}
```

✅ 외부 클래스의 인스턴스를 생성하지 않고도
✅ 정적 내부 클래스의 인스턴스를 만들 수 있음

🧪 2. 기본 사용 예시

```
public class Outer {
    private static String staticData = "Static Data";
    private String instanceData = "Instance Data";

    static class StaticInner {
        public void display() {
            System.out.println("Accessing: " + staticData); // ✅ OK
            // System.out.println(instanceData);            // ❌ Error
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Outer.StaticInner inner = new Outer.StaticInner(); // 외부 인스턴스 없이 생성 가능
        inner.display();
    }
}
```

🟧 3. 주요 특징 요약

항목	설명
선언 위치	클래스 내부, `static` 키워드 사용
외부 클래스 인스턴스 필요?	❌ 불필요
외부 멤버 접근	🔵 `static` 멤버만 접근 가능
정적 멤버 정의 가능 여부	✅ 가능
용도	외부 클래스와 논리적 관련이 있으나 인스턴스와는 무관할 때

🟩 4. 일반 클래스와 비교

```
class Outer {
    private static int staticValue = 100;
    private int instanceValue = 200;

    static class StaticInner {
        void print() {
            System.out.println(staticValue);  // OK
            // System.out.println(instanceValue); // 컴파일 에러
        }
    }
}
```

> 내부 클래스가 static이므로, `instanceValue`에는 접근 불가
> → static 멤버만 접근 가능

🟧 5. 외부 클래스와의 연결이 약한 이유

- 일반 내부 클래스는 외부 인스턴스의 **스코프에 존재**
- 정적 내부 클래스는 **외부 클래스의 정적 멤버로 취급됨**
- 따라서 외부 인스턴스와 **관계없이 독립적으로 존재 가능**

```
Outer outer = new Outer();
Outer.StaticInner inner = new Outer.StaticInner();  // outer 필요 없음!
```

🛠️ 6. 정적 내부 클래스 내 정적 멤버 정의

```
static class StaticInner {
    static int counter = 0;

    static void showCounter() {
        System.out.println("Count: " + counter);
    }
}
```

✅ 일반 내부 클래스에서는 static 멤버를 선언할 수 없지만,
✅ 정적 내부 클래스에서는 static 필드와 메서드가 허용됨.

🧩 7. 실제 사용 사례

- **Builder 패턴**에서 매우 자주 사용됨

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

    public static class Builder {
        private String name;
        private int age;

        public Builder name(String name) {
            this.name = name; return this;
        }

        public Builder age(int age) {
            this.age = age; return this;
        }

        public Person build() {
            Person p = new Person();
```

```
            p.name = this.name;
            p.age = this.age;
            return p;
        }
    }
}
```

⬅ END 요약 정리

항목	정적 내부 클래스 (`static`)
외부 인스턴스 필요	❌ 필요 없음
외부 멤버 접근	🔵 static 멤버만 접근 가능
static 멤버 선언 가능	✅ 가능
생성 방식	`Outer.StaticInner obj = new Outer.StaticInner();`
대표 용도	Builder, 복잡한 유틸리티 클래스 구조 설계 시

지역 내부 클래스

🟦 1. 지역 내부 클래스란?

> 메서드나 생성자 내에서 **지역 변수처럼 선언**되는 클래스.
> 선언된 **블록(scope)** 내에서만 사용 가능.

```
void someMethod() {
    class LocalClass {
        void doSomething() {
            System.out.println("Hello");
        }
    }

    LocalClass lc = new LocalClass();
    lc.doSomething();
}
```

✅ 바깥 클래스의 멤버뿐만 아니라,
✅ 메서드의 **지역 변수**도 접근 가능(단, final or effectively final인 경우)

🟧 2. 사용 예시

```
public class Outer {
    private int outerValue = 100;

    public void methodWithLocalClass() {
        int localValue = 42; // 암묵적 final
```

```
6
7           class LocalInner {
8               void print() {
9                   System.out.println("outerValue = " + outerValue);   // OK
10                  System.out.println("localValue = " + localValue);   // OK
11              }
12          }
13
14          LocalInner li = new LocalInner();
15          li.print();
16      }
17  }
```

🔍 3. 접근 범위와 제한 사항

요소	접근 가능 여부	설명
외부 클래스의 멤버	✅ 가능	`outerValue` 처럼 접근 가능
메서드의 지역 변수	⚠️ 조건부 가능	`final` 또는 **effectively final**일 때만
static 멤버 선언	❌ 불가	지역 클래스는 static 선언 불가

⚠️ 'effectively final' 이란?

Java 8 이후, **명시적으로 `final`이 아니어도**
한 번만 할당되고 변경되지 않으면 → `effectively final`

```
1  void foo() {
2      int count = 10; // effectively final
3
4      class L {
5          void print() {
6              System.out.println(count); // OK
7          }
8      }
9  }
```

하지만 아래는 ❌ 불가

```java
void foo() {
    int count = 10;

    count = 20; // 변경되므로 effectively final ❌

    class L {
        void print() {
            System.out.println(count); // 컴파일 에러
        }
    }
}
```

🧩 4. 용도 및 쓰임새

- 임시적인 기능 구현
- 메서드의 컨텍스트에 강하게 결합된 클래스를 표현
- GUI 이벤트 핸들링이나 콜백용 클래스 정의에 유리

🛠️ 5. 비교 정리

항목	지역 내부 클래스 (Local)
선언 위치	메서드 or 생성자 내부
접근 가능한 외부 멤버	외부 클래스의 멤버 모두
지역 변수 접근	`final` 또는 effectively final 변수만
static 선언 가능 여부	❌ static 멤버 정의 불가
생성 방식	메서드 블록 내에서만 생성 가능

🧠 6. 메모리와 생명주기

- 지역 클래스는 **정의된 메서드가 실행될 때 생성**됨
- 메서드 호출이 끝나면 지역 클래스 타입 자체도 **접근 불가**
- 하지만 내부적으로는 익명 클래스와 같이 **JVM에서 바이트코드 클래스 생성됨**

✅ 예시 정리

```
public void process(String msg) {
    class Processor {
        void print() {
            System.out.println("Message: " + msg);
        }
    }

    Processor p = new Processor();
    p.print();
}
```

🔚 요약

특징	설명
클래스 위치	메서드 또는 생성자 내부
외부 클래스 접근	✅ 가능
지역 변수 접근 조건	`final` or effectively final
static 멤버 선언 가능 여부	❌ 불가
일반 용도	메서드에 특화된 클래스 필요 시

익명 클래스 (Anonymous class)

🟦 1. 익명 클래스란?

> 이름이 없는 **일회용 내부 클래스**
> 클래스를 정의하면서 **동시에 인스턴스를 생성**
> 인터페이스, 추상 클래스, 일반 클래스의 **하위 클래스**를 **즉석에서 구현**

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Hello from anonymous class!");
    }
};
```

✅ 이 코드는 `Runnable` 인터페이스를 구현하는
✅ 이름 없는 클래스를 **정의 + 인스턴스화** 한 것

🟧 2. 기본 구조

```
new 수퍼타입(생성자 인자) {
    // 클래스 정의 및 구현
};
```

- `수퍼타입`: 인터페이스 or 클래스 (보통 추상 클래스)
- 생성자 호출과 동시에 **{} 블록으로 구현**
- 컴파일러는 내부적으로 `Outer$1.class` 같은 이름으로 생성

🧪 3. 예제: 인터페이스 구현

```
public class Test {
    public static void main(String[] args) {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                System.out.println("Running via anonymous class");
            }
        };

        task.run();
    }
}
```

- `Runnable`은 인터페이스
- `run()` 메서드를 익명 클래스에서 즉시 구현

🧩 4. 예제: 추상 클래스 확장

```
abstract class Animal {
    abstract void makeSound();
}

public class Test {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            @Override
            void makeSound() {
                System.out.println("Bark!");
            }
        };

        dog.makeSound();
    }
}
```

- `Animal`은 추상 클래스
- 익명 클래스로 바로 **구현 + 인스턴스화**

🚫 5. 제약사항

제한 요소	설명
생성자 선언 ❌	생성자를 정의할 수 없음 (익명이므로)
여러 메서드 정의 가능 ✅	내부에 메서드 여러 개 정의 가능
static 멤버 선언 ❌	static 필드나 메서드는 선언 불가
오직 한 번만 사용할 수 있음 🔂	재사용 어려움, 유지보수 불리

🧠 6. 람다와의 관계

람다식과 익명 클래스는 매우 유사하지만 아래와 같은 차이점이 있어:

항목	익명 클래스	람다식
문법	복잡	간결 (`() -> {}` 형태)
this 키워드	익명 클래스 자신	외부 클래스(this)
생성된 클래스 이름	내부적으로 `$1`, `$2` 등	내부적으로 `invoke dynamic`
가능 대상	모든 인터페이스/클래스	함수형 인터페이스만

✅ Java 8 이후에는 **익명 클래스보다 람다식**이 대부분의 경우 더 선호됨

🔨 7. 실전 예시: GUI 이벤트 처리

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

> `ActionListener` 인터페이스를 익명 클래스로 구현해서 바로 `addActionListener`에 전달

🛠️ 8. 사용 위치

위치	사용 가능 여부	예
메서드 내부	✅	지역 변수처럼

위치	사용 가능 여부	예
인스턴스 초기화 블록 내부	✅	복잡한 초기화
필드 초기화	✅	단일 필드 익명 구현체

⬅ END 요약

항목	설명
이름	❌ 없음
선언 위치	메서드 내부 등
상속 가능	인터페이스 or 클래스 (단 1개만)
생성자 정의	❌ 불가
static 멤버	❌ 불가
사용 이유	빠르게 일회성 클래스 구현

람다식과 함수형 인터페이스

🟦 1. 함수형 인터페이스 (Functional Interface)

> 단 하나의 **추상 메서드**만 가지는 인터페이스
> 람다식을 사용할 수 있는 **타입(타겟)** 역할

```
@FunctionalInterface
interface MyFunc {
    int calculate(int x, int y);
}
```

✅ `@FunctionalInterface` 어노테이션은 선택사항이지만,
✅ 붙이면 컴파일러가 "**추상 메서드가 1개인지**" 확인해줌

✅ 자바 기본 제공 함수형 인터페이스 (java.util.function 패키지)

인터페이스	매개변수	반환값	용도 예시
`Function<T,R>`	1개	O	변환: `x -> x.length()`
`Consumer<T>`	1개	X	소비: `x -> System.out.println(x)`
`Supplier<T>`	0개	O	공급: `() -> "Hello"`
`Predicate<T>`	1개	boolean	조건: `x -> x > 10`
`BiFunction<T,U,R>`	2개	O	`add(x, y)`

인터페이스	매개변수	반환값	용도 예시
`UnaryOperator<T>`	1개	T	단항 연산
`BinaryOperator<T>`	2개	T	이항 연산

🟧 2. 람다식이란?

> “익명 메서드” 표현.
> 함수형 인터페이스를 간결하게 구현하는 문법

```
(x, y) -> x + y
```

✅ 인터페이스의 단일 추상 메서드와 자동 매핑됨

🔧 형식

```
(매개변수) -> { 실행문; 반환문; }
```

예시:

```
(int x, int y) -> { return x + y; }
(x, y) -> x + y
() -> System.out.println("Hi")
```

🧪 3. 실전 예제

```
@FunctionalInterface
interface MyOperation {
    int operate(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        MyOperation add = (a, b) -> a + b;
        System.out.println(add.operate(3, 4)); // 7
    }
}
```

📦 4. 표준 함수형 인터페이스 사용

```
Function<String, Integer> strLen = s -> s.length();
System.out.println(strLen.apply("Hello")); // 5

Consumer<String> printer = s -> System.out.println(s);
printer.accept("Hi");

Supplier<Double> rand = () -> Math.random();
System.out.println(rand.get());

Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true
```

🔁 5. 메서드 참조 (`::`)

람다식을 메서드 이름만으로 표현

```
Consumer<String> print = System.out::println;

Function<String, Integer> strLen = String::length;

BiFunction<Integer, Integer, Integer> max = Math::max;
```

🔁 6. 람다식과 익명 클래스 차이

항목	익명 클래스	람다식
문법	복잡	간단
`this` 의미	익명 클래스 자체	외부 클래스
목적	복잡한 다중 메서드 구현 가능	단일 메서드 함수형 인터페이스만
생성 클래스	컴파일 시 클래스 파일 생성	invoke dynamic으로 최적화

🧠 7. 람다식 내부 동작 (고급)

- 컴파일 시 `invokeDynamic` 바이트코드로 표현
- **메서드 인스턴스화가 아닌 참조 최적화**
- `Serializable` 처리, 캡처링된 값들은 **final 또는 effectively final**

📌 8. 함수형 프로그래밍 활용

- `Stream API`와 함께 필수로 사용됨
- `Optional`, `Map`, `filter`, `collect` 등과 조합하여 강력한 컬렉션 처리 가능

```
List<String> list = List.of("a", "bb", "ccc");
list.stream()
    .filter(s -> s.length() >= 2)
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

🔚 요약

항목	설명
람다식	함수형 인터페이스 구현용 익명 메서드
함수형 인터페이스	추상 메서드 1개만 있는 인터페이스
주요 패키지	`java.util.function`
메서드 참조 지원	`::` 연산자로 축약
Stream 조합	`filter`, `map`, `collect`, `reduce`