8. 중첩 클래스, 내부 클래스2

#0.강의/1.자바로드맵/3.자바-중급1편

- /지역 클래스 - 시작
- /지역 클래스 - 지역 변수 캡처1
- /지역 클래스 - 지역 변수 캡처2
- /지역 클래스 - 지역 변수 캡처3
- /익명 클래스 - 시작
- /익명 클래스 활용1
- /익명 클래스 활용2
- /익명 클래스 활용3
- /문제와 풀이1
- /문제와 풀이2
- /정리

지역 클래스 - 시작

- 지역 클래스(Local class)는 내부 클래스의 특별한 종류의 하나이다. 따라서 내부 클래스의 특징을 그대로 가진다. 예를 들어서 지역 클래스도 내부 클래스이므로 바깥 클래스의 인스턴스 멤버에 접근할 수 있다.
- 지역 클래스는 지역 변수와 같이 코드 블럭 안에서 정의된다.

지역 클래스 예

```
class Outer {

    public void process() {
        //지역 변수
        int localVar = 0;

        //지역 클래스
        class Local {...}

        Local local = new Local();
    }
}
```

지역 클래스의 특징

- 지역 클래스는 지역 변수처럼 코드 블럭 안에 클래스를 선언한다.
- 지역 클래스는 지역 변수에 접근할 수 있다.

지역 클래스 예제1

```
package nested.local;

public class LocalOuterV1 {

    private int outInstanceVar = 3;

    public void process(int paramVar) {

        int localVar = 1;

        class LocalPrinter {
            int value = 0;

            public void printData() {
                System.out.println("value=" + value);
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);
                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        }

        LocalPrinter printer = new LocalPrinter();
        printer.printData();
    }

    public static void main(String[] args) {
        LocalOuterV1 localOuter = new LocalOuterV1();
        localOuter.process(2);
    }
}
```

실행 결과

```

```

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
```

지역 클래스의 접근 범위

- 자신의 인스턴스 변수인 `value`에 접근할 수 있다.
- 자신이 속한 코드 블럭의 지역 변수인 `localVar`에 접근할 수 있다.
- 자신이 속한 코드 블럭의 매개변수인 `paramVar`에 접근할 수 있다. 참고로 매개변수도 지역 변수의 한 종류이다.
- 바깥 클래스의 인스턴스 멤버인 `outInstanceVar`에 접근할 수 있다. (지역 클래스도 내부 클래스의 한 종류이다.)

지역 클래스는 지역 변수 처럼 접근 제어자를 사용할 수 없다.

지역 클래스 예제2

당연한 이야기지만 내부 클래스를 포함한 중첩 클래스들도 일반 클래스처럼 인터페이스를 구현하거나, 부모 클래스를 상속할 수 있다.
지역 클래스를 통해 사용 예를 알아보자.

```
package nested.local;

public interface Printer {
    void print();
}
```

```
package nested.local;

public class LocalOuterV2 {

    private int outInstanceVar = 3;

    public void process(int paramVar) {
```

```
        int localVar = 1;

        class LocalPrinter implements Printer {

            int value = 0;

            @Override
            public void print() {
                System.out.println("value=" + value);
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);
                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        }

        Printer printer = new LocalPrinter();
        printer.print();
    }

    public static void main(String[] args) {
        LocalOuterV2 localOuter = new LocalOuterV2();
        localOuter.process(2);
    }
}
```

실행 결과

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
```

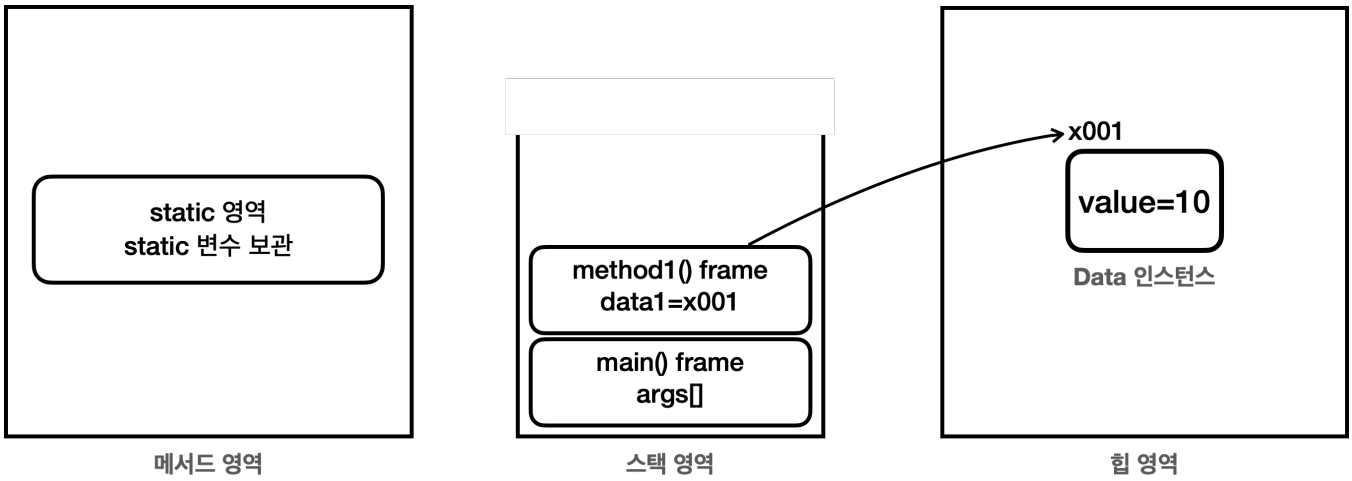
단순히 `Printer` 인터페이스만 추가하고 구현했기 때문에 이해하는데 어려움은 없을 것이다.

지역 클래스 - 지역 변수 캡처1

> **참고**
> 지금부터 설명할 지역 변수 캡처에 관한 내용은 너무 깊이있게 이해하지 않아도 된다. 이해가 어렵다면 단순하게 지역 클래스가 접근하는 지역 변수의 값은 변경하면 안된다 정도로 이해하면 충분하다.

지역 클래스를 더 자세히 알아보기 전에 잠시 변수들의 생명 주기에 대해서 정리해보자.

변수의 생명 주기



- **클래스 변수**: 프로그램 종료 까지, 가장 길다(메서드 영역)
  - 클래스 변수(static 변수)는 메서드 영역에 존재하고, 자바가 클래스 정보를 읽어 들이는 순간부터 프로그램 종료까지 존재한다.
- **인스턴스 변수**: 인스턴스의 생존 기간(힙 영역)
  - 인스턴스 변수는 본인이 소속된 인스턴스가 GC 되기 전까지 존재한다. 생존 주기가 긴 편이다.
- **지역 변수**: 메서드 호출이 끝나면 사라짐(스택 영역)
  - 지역 변수는 스택 영역의 스택 프레임 안에 존재한다. 따라서 메서드가 호출 되면 생성되고, 메서드 호출이 종료되면 스택 프레임이 제거되면서 그 안에 있는 지역 변수도 모두 제거된다. 생존 주기가 아주 짧다. 참고로 매개변수도 지역 변수의 한 종류이다.

지역 클래스 예제3

지금까지 작성한 지역 클래스 예제를 약간 수정해서 새로 만들어보자.

```
package nested.local;

public class LocalOuterV3 {

    private int outInstanceVar = 3;

    public Printer process(int paramVar) {
```

```
        int localVar = 1;  //지역 변수는 스택 프레임이 종료되는 순간 함께 제거된다.

        class LocalPrinter implements Printer {

            int value = 0;

            @Override
            public void print() {
                System.out.println("value=" + value);

                //인스턴스는 지역 변수보다 더 오래 살아남는다.
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);

                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        }

        Printer printer = new LocalPrinter();
        //printer.print()를 여기서 실행하지 않고 Printer 인스턴스만 반환한다.
        return printer;
    }

    public static void main(String[] args) {
        LocalOuterV3 localOuter = new LocalOuterV3();
        Printer printer = localOuter.process(2);
        //printer.print()를 나중에 실행한다. process()의 스택 프레임이 사라진 이후에 실행
        printer.print();
    }
}
```

- `process()`는 `Printer` 타입을 반환한다. 여기서는 `LocalPrinter` 인스턴스를 반환한다.
- 여기서는 `LocalPrinter.print()` 메서드를 `process()` 안에서 실행하는 것이 아니라 `process()` 메서드가 종료된 이후에 `main()` 메서드에서 실행한다.

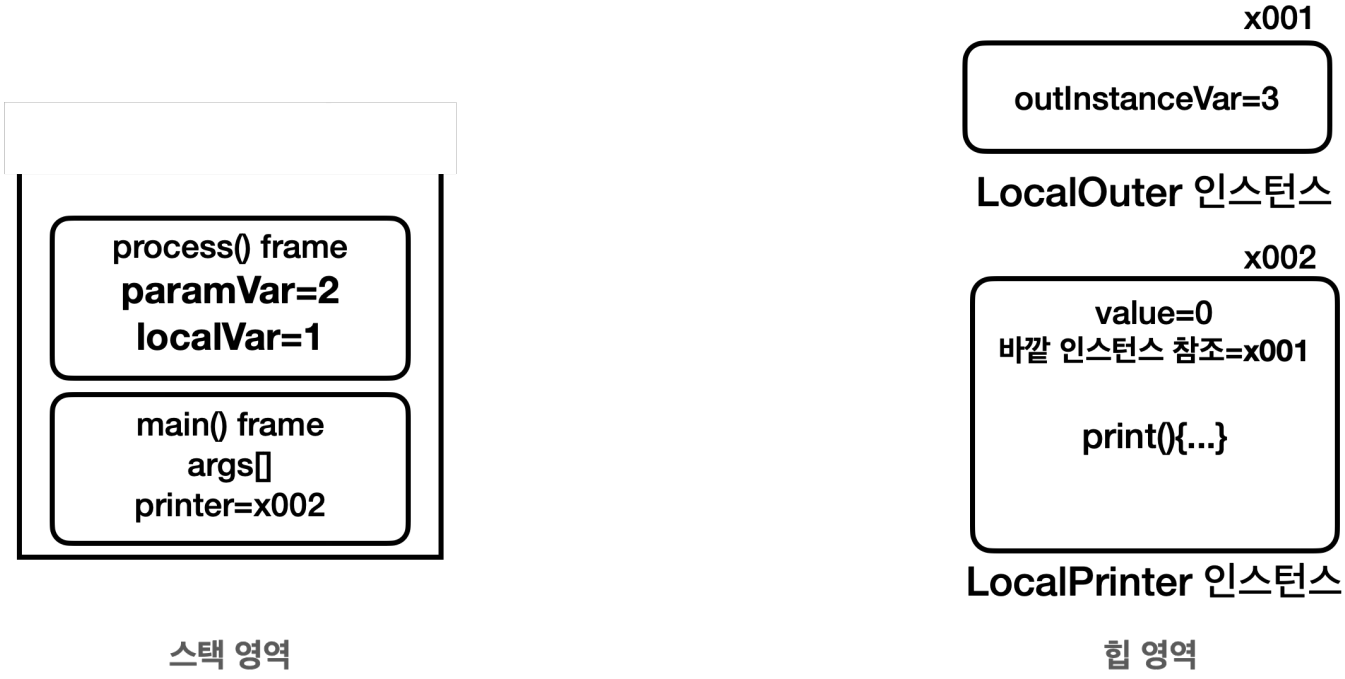
실행 결과

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
```

이 예제를 실행하면서 뭔가 이상한 느낌이 들었다면 제대로 공부를 하고 있는 것이다.

예제 코드를 메모리 그림과 함께 분석해보자.

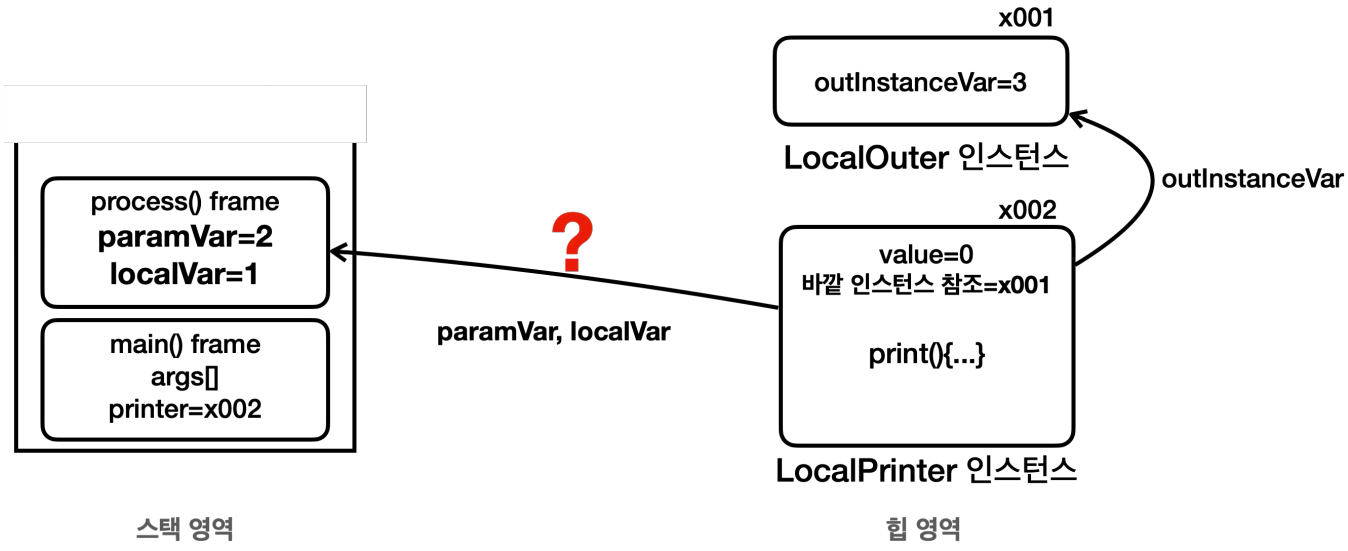
LocalPrinter 인스턴스 생성 직후 메모리 그림



지역 클래스 인스턴스의 생존 범위

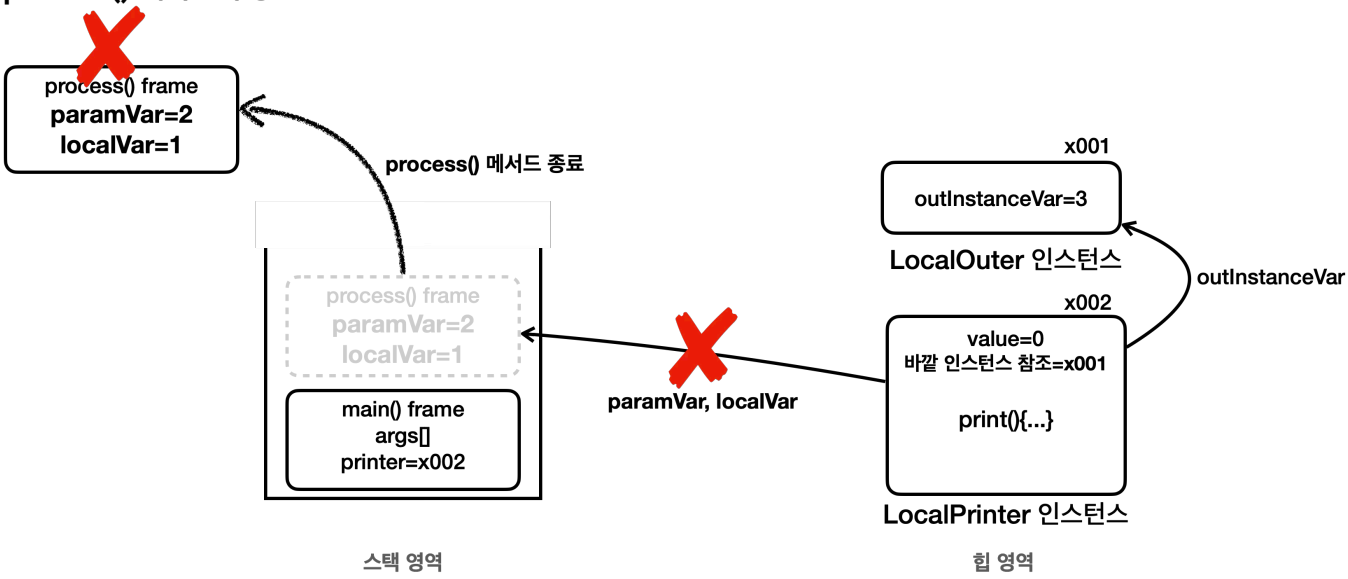
- 지역 클래스로 만든 객체도 인스턴스이기 때문에 힙 영역에 존재한다. 따라서 GC 전까지 생존한다.
  - `LocalPrinter` 인스턴스는 `process()` 메서드 안에서 생성된다. 그리고 `process()`에서 `main()`으로 생성한 `LocalPrinter` 인스턴스를 반환하고 `printer` 변수에 참조를 보관한다. 따라서 `LocalPrinter` 인스턴스는 `main()`이 종료될 때 까지 생존한다.
- `paramVar`, `localVar`와 같은 지역 변수는 `process()` 메서드를 실행하는 동안에만 스택 영역에서 생존한다. `process()` 메서드가 종료되면 `process()` 스택 프레임이 스택 영역에서 제거 되면서 함께 제거된다.

LocalPrinter.print() 접근 메모리 그림



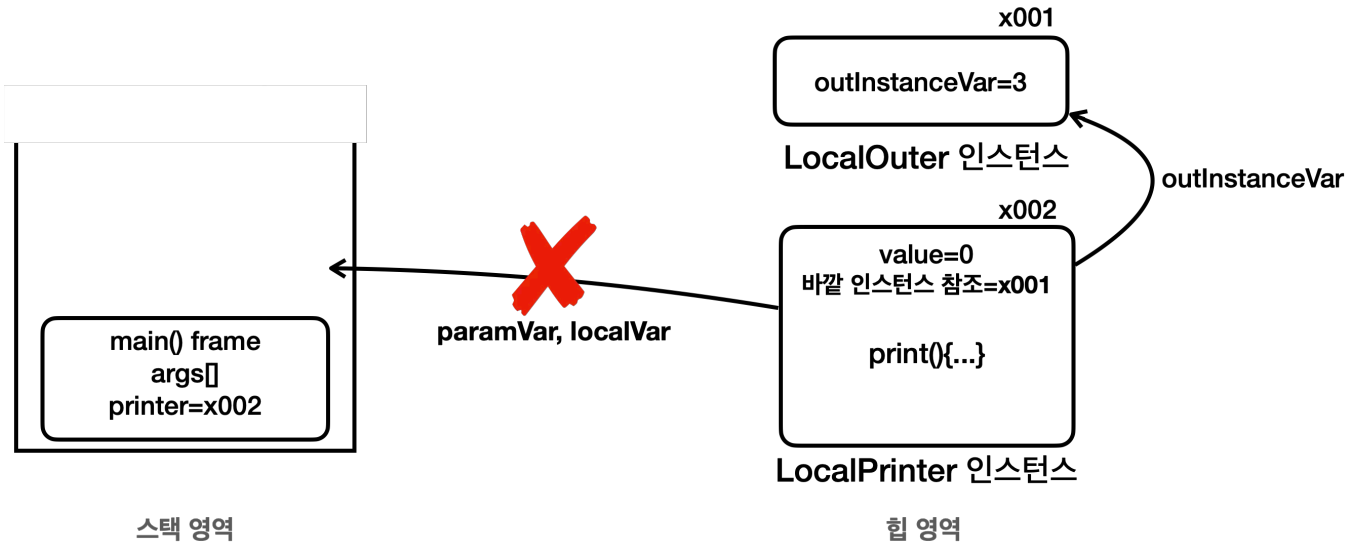
- `LocalPrinter` 인스턴스는 `print()` 메서드를 통해 힙 영역에 존재하는 바깥 인스턴스의 변수인 `outInstanceVar`에 접근한다. 이 부분은 인스턴스의 필드를 참조하는 것이기 때문에 특별한 문제가 없다.
- `LocalPrinter` 인스턴스는 `print()` 메서드를 통해 스택 영역에 존재하는 지역 변수도 접근하는 것 처럼 보인다. 하지만 스택 영역에 존재하는 지역 변수를 힙 영역에 있는 인스턴스가 접근하는 것은 생각처럼 단순하지 않다.

process() 메서드의 종료



- 지역 변수의 생명주기는 매우 짧다. 반면에 인스턴스의 생명주기는 GC 전까지 생존할 수 있다.
- 지역 변수인 `paramVar`, `localVar`는 `process()` 메서드가 실행되는 동안에만 생존할 수 있다. `process()` 메서드가 종료되면 `process()`의 스택 프레임이 제거되면서 두 지역 변수도 함께 제거된다.
- 여기서 문제는 `process()` 메서드가 종료되어도 `LocalPrinter` 인스턴스는 계속 생존할 수 있다는 점이다.

process() 메서드가 종료된 이후에 지역 변수 접근



- 예제를 잘 보자. 여기서는 `process()` 메서드가 종료된 이후에 `main()` 메서드 안에서 `LocalPrinter.print()` 메서드를 호출한다.
- `LocalPrinter` 인스턴스에 있는 `print()` 메서드는 지역 변수인 `paramVar`, `localVar`에 접근해야 한다. 하지만 `process()` 메서드가 이미 종료되었으므로 해당 지역 변수들도 이미 제거된 상태이다.

그런데 실행 결과를 보면 `localVar`, `paramVar`와 같은 지역 변수의 값들이 모두 정상적으로 출력되는 것을 확인할 수 있다.

실행 결과

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
```

어떻게 제거된 지역 변수들에 접근할 수 있는 것일까?

참고

여기서는 이해를 돕기 위해 설명을 단순화 했지만, 더 정확히 이야기 하면 `LocalPrinter.print()` 메서드를 실행하면 이 메서드도 당연히 스택 프레임에 올라가서 실행된다. `main()`에서 `print()`를 실행했으므로 `main()` 스택 프레임 위에 `print()` 스택 프레임이 올라간다. 물론 `process()` 스택 프레임은 이미 제거된 상태이므로 지역 변수인 `localVar`, `paramVar`도 함께 제거되어서 접근할 수 없다.

지역 클래스 - 지역 변수 캡처2

지역 클래스는 지역 변수에 접근할 수 있다.

그런데 앞서 본 것 처럼 지역 변수의 생명주기는 짧고, 지역 클래스를 통해 생성한 인스턴스의 생명 주기는 길다.

지역 클래스를 통해 생성한 인스턴스가 지역 변수에 접근해야 하는데, 둘의 생명 주기가 다르기 때문에 인스턴스는 살아있지만, 지역 변수는 이미 제거된 상태일 수 있다.

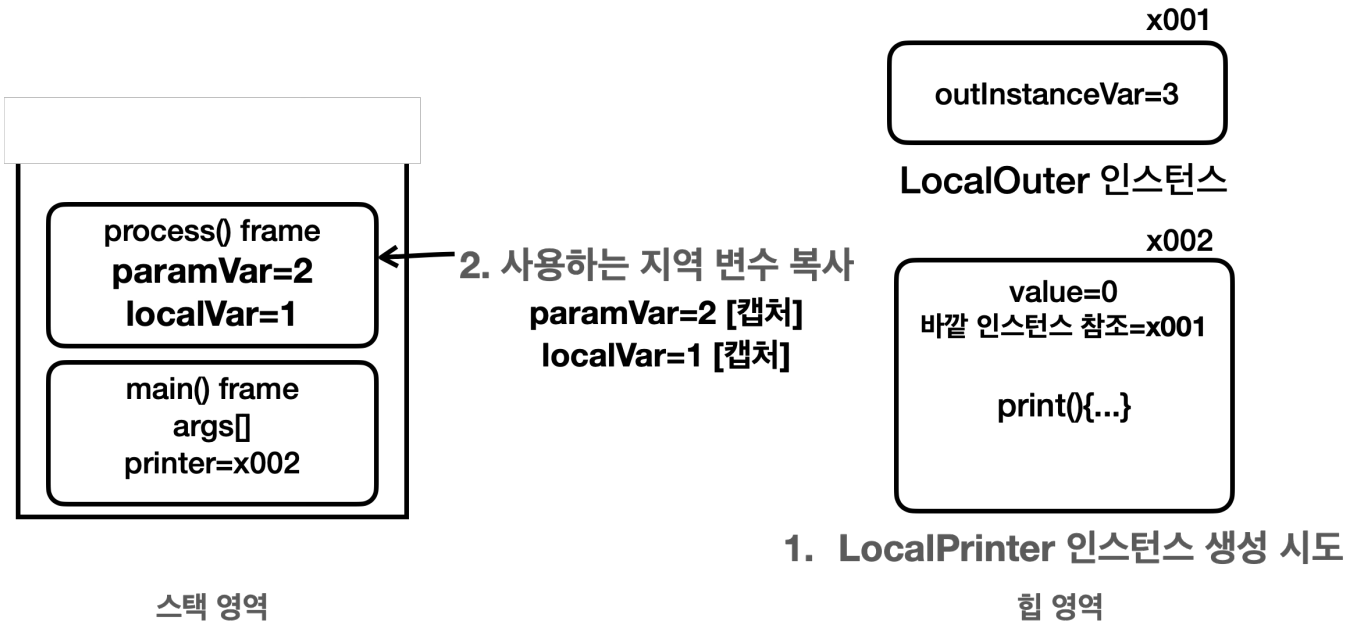
지역 변수 캡처

자바는 이런 문제를 해결하기 위해 지역 클래스의 인스턴스를 생성하는 시점에 필요한 지역 변수를 복사해서 생성한 인스턴스에 함께 넣어둔다. 이런 과정을 변수 캡처(Capture)라 한다.

캡처라는 단어는 스크린 캡처를 떠올려 보면 바로 이해가 될 것이다. 인스턴스를 생성할 때 필요한 지역 변수를 복사해서 보관해 두는 것이다.

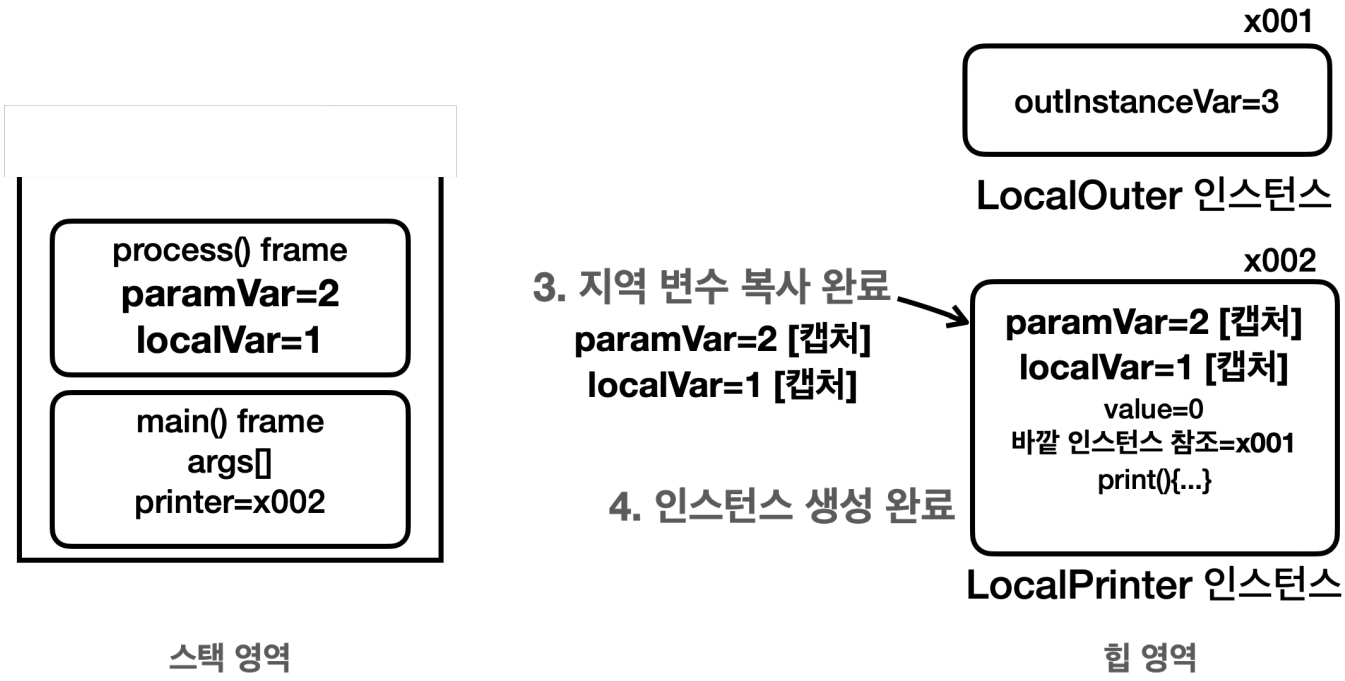
물론 모든 지역 변수를 캡처하는 것이 아니라 접근이 필요한 지역 변수만 캡처한다.

지역 클래스의 인스턴스 생성과 지역 변수 캡처 과정1

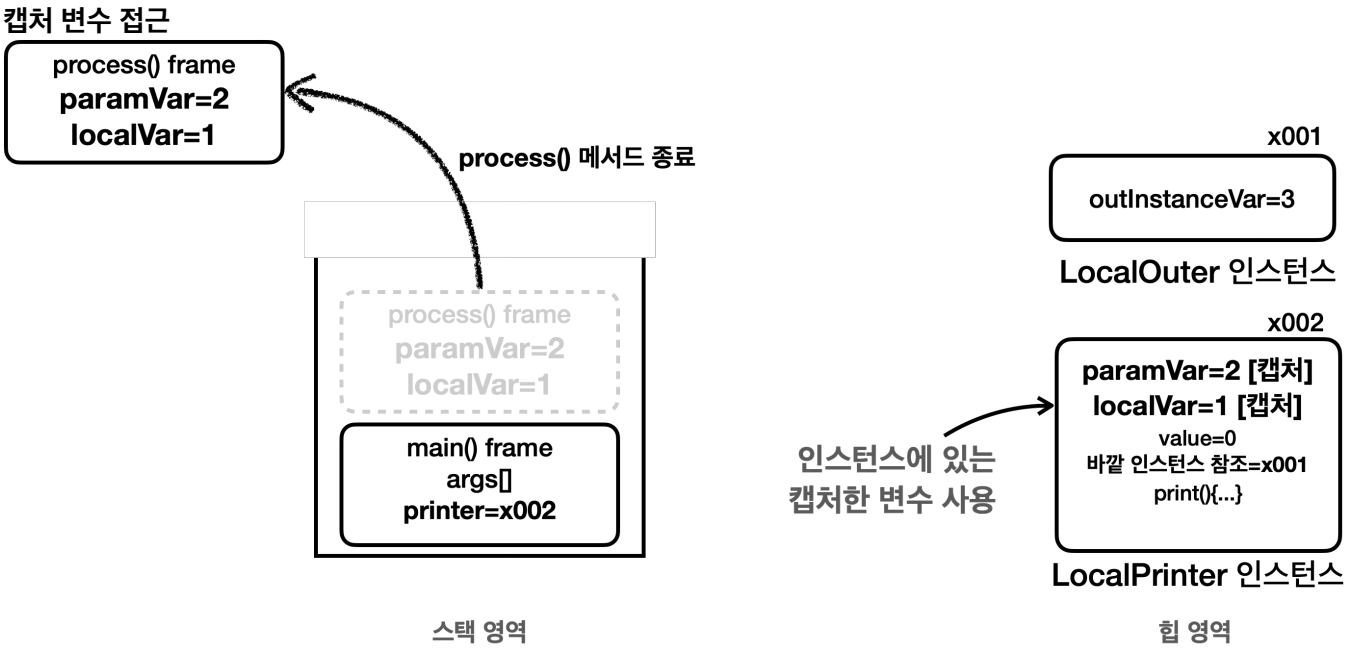


- **1. LocalPrinter 인스턴스 생성 시도**: 지역 클래스의 인스턴스를 생성할 때 지역 클래스가 접근하는 지역 변수를 확인한다.
  - `LocalPrinter` 클래스는 `paramVar`, `localVar` 지역 변수에 접근한다.
- **2. 사용하는 지역 변수 복사**: 지역 클래스가 사용하는 지역 변수를 복사한다. (매개변수도 지역 변수의 한 종류이다)
  - 여기서는 `paramVar`, `localVar` 지역 변수를 복사한다.

지역 클래스의 인스턴스 생성과 지역 변수 캡처 과정2



- **3. 지역 변수 복사 완료**: 복사한 지역 변수를 인스턴스에 포함한다.
- **4. 인스턴스 생성 완료**: 복사한 지역 변수를 포함해서 인스턴스 생성이 완료된다. 이제 복사한 지역 변수를 인스턴스를 통해 접근할 수 있다.



- `LocalPrinter` 인스턴스에서 `print()` 메서드를 통해 `paramVar`, `localVar`에 접근하면 사실은 스택 영역에 있는 지역 변수에 접근하는 것이 아니다. 대신에 인스턴스에 있는 캡처한 변수에 접근한다.
- 캡처한 `paramVar`, `localVar`의 생명주기는 `LocalPrinter` 인스턴스의 생명주기와 같다. 따라서 `LocalPrinter` 인스턴스는 지역 변수의 생명주기와 무관하게 언제든지 `paramVar`, `localVar` 캡처 변수에 접근할 수 있다.
- 이렇게 해서 지역 변수와 지역 클래스를 통해 생성한 인스턴스의 생명주기가 다른 문제를 해결한다.

코드로 캡처 변수 확인

다음 코드를 통해 캡처 변수를 직접 확인해보자.

LocalOuterV3 - 추가

```
public static void main(String[] args) {
    LocalOuterV3 localOuter = new LocalOuterV3();
    Printer printer = localOuter.process(2);
    //printer.print()를 나중에 실행한다. process()의 스택 프레임이 사라진 이후에 실행
    printer.print();

    //추가
    System.out.println("필드 확인");
    Field[] fields = printer.getClass().getDeclaredFields();
    for (Field field : fields) {
        System.out.println("field = " + field);
    }
}
```

실행 결과

```
필드 확인

//인스턴스 변수
field = int nested.local.LocalOuterV3$1LocalPrinter.value

//캡처 변수
field = final int nested.local.LocalOuterV3$1LocalPrinter.val$localVar
field = final int nested.local.LocalOuterV3$1LocalPrinter.val$paramVar

//바깥 클래스 참조
field = final nested.local.LocalOuterV3
nested.local.LocalOuterV3$1LocalPrinter.this$0
```

실행 결과를 통해 `LocalPrinter` 클래스의 캡처 변수를 확인할 수 있다. 추가로 바깥 클래스를 참조하기 위한 필드도 확인할 수 있다. 참고로 이런 필드들은 자바가 내부에서 만들어 사용하는 필드들이다.

정리

지역 클래스는 인스턴스를 생성할 때 필요한 지역 변수를 먼저 캡처해서 인스턴스에 보관한다. 그리고 지역 클래스의 인

스턴스를 통해 지역 변수에 접근하면, 실제로는 지역 변수에 접근하는 것이 아니라 인스턴스에 있는 캡처한 캡처 변수에 접근한다.

지역 클래스 - 지역 변수 캡처3

지역 클래스가 접근하는 지역 변수는 절대로 중간에 값이 변하면 안된다.
따라서 `final`로 선언하거나 또는 사실상 `final`이어야 한다. 이것은 자바 문법이고 규칙이다.

용어 - 사실상 final
영어로 effectively final이라 한다. 사실상 `final` 지역 변수는 지역 변수에 `final` 키워드를 사용하지는 않았지만, 값을 변경하지 않는 지역 변수를 뜻한다. `final` 키워드를 넣지 않았을 뿐이지, 실제로는 `final` 키워드를 넣은 것 처럼 중간에 값을 변경하지 않은 지역 변수이다. 따라서 사실상 `final` 지역 변수는 `final` 키워드를 넣어도 동일하게 작동해야 한다.

지역 클래스가 접근하는 지역 변수는 왜 `final` 또는 사실상 `final` 이어야 할까? 왜 중간에 값이 변하면 안될까?

코드를 통해 확인해보자.

```
package nested.local;

public class LocalOuterV4 {

    private int outInstanceVar = 3;

    public Printer process(int paramVar) {

        int localVar = 1;

        class LocalPrinter implements Printer {

            int value = 0;

            @Override
            public void print() {
                System.out.println("value=" + value);

                //인스턴스는 지역 변수보다 더 오래 살아남는다.
```

```
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);

                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        }

        Printer printer = new LocalPrinter();
        // 만약 localVar의 값을 변경한다면? 다시 캡처해야 하나??
        // localVar = 10; // 컴파일 오류
        // paramVar = 20; // 컴파일 오류

        return printer;
    }

    public static void main(String[] args) {
        LocalOuterV4 localOuter = new LocalOuterV4();
        Printer printer = localOuter.process(2);
        printer.print();
    }
}
```

실행 결과

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
```

Printer printer = new LocalPrinter();

`LocalPrinter`를 생성하는 시점에 지역 변수인 `localVar`, `paramVar`를 캡처한다.

그런데 이후에 캡처한 지역 변수의 값을 다음과 같이 변경하면 어떻게 될까?

```
Printer printer = new LocalPrinter()
// 만약 localVar의 값을 변경한다면? 다시 캡처해야 하나??
localVar = 10; // 컴파일 오류
paramVar = 20; // 컴파일 오류
```

이렇게 되면 스택 영역에 존재하는 지역 변수의 값과 인스턴스에 캡처한 캡처 변수의 값이 서로 달라지는 문제가 발생한다. 이것을 동기화 문제라 한다.

물론 자바 언어를 설계할 때 지역 변수의 값이 변경되면 인스턴스에 캡처한 변수의 값도 함께 변경하도록 설계하면 된다. 하지만 이로 인해 수 많은 문제들이 파생될 수 있다.

캡처 변수의 값을 변경하지 못하는 이유

- 지역 변수의 값을 변경하면 인스턴스에 캡처한 변수의 값도 변경해야 한다.
- 반대로 인스턴스에 있는 캡처 변수의 값을 변경하면 해당 지역 변수의 값도 다시 변경해야 한다.
- 개발자 입장에서 보면 예상하지 못한 곳에서 값이 변경될 수 있다. 이는 디버깅을 어렵게 한다.
- 지역 변수의 값과 인스턴스에 있는 캡처 변수의 값을 서로 동기화 해야 하는데, 멀티쓰레드 상황에서 이런 동기화는 매우 어렵고, 성능에 나쁜 영향을 줄 수 있다. 이 부분은 멀티쓰레드를 학습하면 이해할 수 있다.

이 모든 문제는 캡처한 지역 변수의 값이 변하기 때문에 발생한다. 자바는 캡처한 지역 변수의 값을 변하지 못하게 막아서 이런 복잡한 문제들을 근본적으로 차단한다.

> **참고**
> 변수 캡처에 대한 내용이 이해가 어렵다면 단순하게 지역 클래스가 접근하는 지역 변수의 값은 변경하면 안된다. 정도로 이해하면 충분하다.

익명 클래스 - 시작

익명 클래스(anonymous class)는 지역 클래스의 특별한 종류의 하나이다.
익명 클래스는 지역 클래스인데, 클래스의 이름이 없다는 특징이 있다.

앞서 만들었던 지역 클래스 예제 코드인 `LocalOuterV2` 코드를 다시 한번 살펴보자.

```
package nested.local;

public class LocalOuterV2 {

    private int outInstanceVar = 3;

    public void process(int paramVar) {
```

```
        int localVar = 1;

        class LocalPrinter implements Printer {

            int value = 0;

            @Override
            public void print() {
                System.out.println("value=" + value);
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);
                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        }

        Printer printer = new LocalPrinter();
        printer.print();
    }

    public static void main(String[] args) {
        LocalOuterV2 localOuter = new LocalOuterV2();
        localOuter.process(2);
    }
}
```

여기서는 지역 클래스를 사용하기 위해 선언과 생성이라는 2가지 단계를 거친다.

1. **선언**: 지역 클래스를 `LocalPrinter`라는 이름으로 선언한다. 이때 `Printer` 인터페이스도 함께 구현한다.
2. **생성**: `new LocalPrinter()`를 사용해서 앞서 선언한 지역 클래스의 인스턴스를 생성한다.

지역 클래스의 선언과 생성

```
//선언
class LocalPrinter implements Printer{
    //body
}

//생성
Printer printer = new LocalPrinter();
```

익명 클래스를 사용하면 클래스의 이름을 생략하고, 클래스의 선언과 생성을 한번에 처리할 수 있다.

익명 클래스 - 지역 클래스의 선언과 생성을 한번에

```
Printer printer = new Printer(){
    //body
}
```

예제 코드를 통해 익명 클래스를 만들어보자.

```
package nested.anonymous;

import nested.local.Printer;

public class AnonymousOuter {

    private int outInstanceVar = 3;

    public void process(int paramVar) {

        int localVar = 1;

        Printer printer = new Printer() {
            int value = 0;

            @Override
            public void print() {
                System.out.println("value=" + value);
                System.out.println("localVar=" + localVar);
                System.out.println("paramVar=" + paramVar);
                System.out.println("outInstanceVar=" + outInstanceVar);
            }
        };
        printer.print();
        System.out.println("printer.class=" + printer.getClass());
    }

    public static void main(String[] args) {
        AnonymousOuter main = new AnonymousOuter();
        main.process(2);
    }
```

```
}
```

- 이 코드는 앞서 설명한 `LocalOuterV2`와 완전히 같은 코드이다. 여기서는 익명 클래스를 사용했다.

실행 결과

```
value=0
localVar=1
paramVar=2
outInstanceVar=3
printer.class=class nested.anonymous.AnonymousOuter$1
```

new Printer() {body}

익명 클래스는 클래스의 본문(body)을 정의하면서 동시에 생성한다.

`new` 다음에 바로 상속 받으면서 구현 할 부모 타입을 입력하면 된다.

이 코드는 마치 인터페이스 `Printer`를 생성하는 것 처럼 보인다. 하지만 자바에서 인터페이스를 생성하는 것을 불가능하다. 이 코드는 인터페이스를 생성하는 것이 아니고, `Printer`라는 이름의 인터페이스를 구현한 익명 클래스를 생성하는 것이다. `{body}` 부분에 `Printer` 인터페이스를 구현한 코드를 작성하면 된다. 이 부분이 바로 익명 클래스의 본문이 된다.

쉽게 이야기해서 `Printer`를 상속(구현) 하면서 바로 생성하는 것이다.

익명 클래스 특징

- 익명 클래스는 이름 없는 지역 클래스를 선언하면서 동시에 생성한다.
- **익명 클래스는 부모 클래스를 상속 받거나, 또는 인터페이스를 구현해야 한다.** 익명 클래스를 사용할 때는 상위 클래스나 인터페이스가 필요하다.
- 익명 클래스는 말 그대로 이름이 없다. 이름을 가지지 않으므로, 생성자를 가질 수 없다. (기본 생성자만 사용됨)
- 익명 클래스는 `AnonymousOuter$1`과 같이 자바 내부에서 바깥 클래스 이름 + `$` + 숫자로 정의된다. 익명 클래스가 여러개면 `$1`, `$2`, `$3`으로 숫자가 증가하면서 구분된다.

익명 클래스의 장점

익명 클래스를 사용하면 클래스를 별도로 정의하지 않고도 인터페이스나 추상 클래스를 즉석에서 구현할 수 있어 코드가 더 간결해진다. 하지만, 복잡하거나 재사용이 필요한 경우에는 별도의 클래스를 정의하는 것이 좋다.

익명 클래스를 사용할 수 없을 때

익명 클래스는 단 한 번만 인스턴스를 생성할 수 있다. 다음과 같이 여러 번 생성이 필요하다면 익명 클래스를 사용할 수 없다. 대신에 지역 클래스를 선언하고 사용하면 된다.

```
Printer printer1 = new LocalPrinter();
printer1.print();

Printer printer2 = new LocalPrinter();
printer2.print();
```

정리

- 익명 클래스는 이름이 없는 지역 클래스이다.
- 특정 부모 클래스(인터페이스)를 상속 받고 바로 생성하는 경우 사용한다.
- 지역 클래스가 일회성으로 사용되는 경우나 간단한 구현을 제공할 때 사용한다.

익명 클래스 활용1

익명 클래스가 어떻게 활용되는지 알아보자.

익명 클래스에 들어가기 전에 먼저 간단한 문제를 하나 풀어보자.

리팩토링 전

```
package nested.anonymous.ex;

public class Ex0Main {

    public static void helloJava() {
        System.out.println("프로그램 시작");
        System.out.println("Hello Java");
        System.out.println("프로그램 종료");
    }

    public static void helloSpring() {
        System.out.println("프로그램 시작");
        System.out.println("Hello Spring");
        System.out.println("프로그램 종료");
    }

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

```
        helloSpring();
    }
}
```

실행 결과

```
프로그램 시작
Hello Java
프로그램 종료
프로그램 시작
Hello Spring
프로그램 종료
```

이 코드의 중복이 보일 것이다. 코드를 리팩토링해서 코드의 중복을 제거해보자.
`helloJava()`, `helloSpring()` 메서드를 하나로 통합하면 된다.

리팩토링 후

```
package nested.anonymous.ex;

public class Ex0RefMain {

    public static void hello(String str) {
        System.out.println("프로그램 시작");
        System.out.println(str);
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        hello("hello Java");
        hello("hello Spring");
    }
}
```

실행 결과

```
프로그램 시작
hello Java
프로그램 종료
프로그램 시작
hello Spring
프로그램 종료
```

코드를 분석해보자.

기존 코드에서 변하는 부분과 변하지 않는 부분을 분리해야 한다.

```
public static void helloJava() {
    System.out.println("프로그램 시작"); //변하지 않는 부분
    System.out.println("Hello Java"); //변하는 부분
    System.out.println("프로그램 종료"); //변하지 않는 부분
}

public static void helloSpring() {
    System.out.println("프로그램 시작"); //변하지 않는 부분
    System.out.println("Hello Spring"); //변하는 부분
    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

여기서 핵심은 변하는 부분과 변하지 않는 부분을 분리하는 것이다.

변하는 부분은 그대로 유지하고 변하지 않는 부분을 어떻게 해결할 것 인가에 집중하면 된다.

이해를 돕기 위해 다음과 같은 코드를 먼저 작성해보자.

```
public static void hello() {
    System.out.println("프로그램 시작"); //변하지 않는 부분

    //변하는 부분 시작
    System.out.println("Hello Java");
    System.out.println("Hello Spring");
    //변하는 부분 종료

    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

여기서 `"Hello Java"`, `"Hello Spring"`와 같은 문자열은 상황에 따라서 변한다.

여기서는 상황에 따라 변하는 문자열 데이터를 다음과 같이 외부에서 전달 받아서 출력하면 된다.

```
public static void hello(String str) {
    System.out.println("프로그램 시작"); //변하지 않는 부분
    System.out.println(str);  //str: 변하는 부분
    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

- 변하지 않는 부분은 그대로 유지하고, 변하는 문자열은 외부에서 전달 받아서 처리한다.

단순한 문제였지만 프로그래밍에서 중복을 제거하고, 좋은 코드를 유지하는 핵심은 변하는 부분과 변하지 않는 부분을 분리하는 것이다. 여기서는 변하지 않는 "프로그램 시작", "프로그램 종료"를 출력하는 부분은 그대로 유지하고, 상황에 따라 변화가 필요한 문자열은 외부에서 전달 받아서 처리했다.
이렇게 변하는 부분과 변하지 않는 부분을 분리하고, 변하는 부분을 외부에서 전달 받으면, 메서드(함수)의 재사용성을 높일 수 있다.
리팩토링 전과 후를 비교해보자. `hello(String str)` 함수의 재사용성은 매우 높아졌다.
여기서 핵심은 변하는 부분을 메서드(함수) 내부에서 가지고 있는 것이 아니라, 외부에서 전달 받는다는 점이다.

익명 클래스 활용2

이번에는 비슷한 다른 문제를 한번 풀어보자.

리팩토링 전

```
package nested.anonymous.ex;

import java.util.Random;

public class Ex1Main {

    public static void helloDice() {
        System.out.println("프로그램 시작");

        //코드 조각 시작
        int randomValue = new Random().nextInt(6) + 1;
```

```
        System.out.println("주사위 = " + randomValue);
        //코드 조각 종료

        System.out.println("프로그램 종료");
    }

    public static void helloSum() {
        System.out.println("프로그램 시작");

        //코드 조각 시작
        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
        //코드 조각 종료

        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        helloDice();
        helloSum();
    }

}
```

실행 결과

```
프로그램 시작
주사위 = 5 //랜덤
프로그램 종료
프로그램 시작
i = 1
i = 2
i = 3
프로그램 종료
```

이 코드를 앞에서 리팩토링 한 예와 같이 하나의 메서드에서 실행할 수 있도록 리팩토링 해보자.
참고로 앞의 문제는 변하는 문자열을 외부에서 전달하면 되었다. 이번에는 문자열 같은 데이터가 아니라 코드 조각을 전달해야 한다.

리팩토링 후

```
package nested.anonymous.ex;

public interface Process {
    void run();
}
```

```
package nested.anonymous.ex;

import java.util.Random;

//정적 중첩 클래스 사용
public class Ex1RefMainV1 {

    public static void hello(Process process) {
        System.out.println("프로그램 시작");
        //코드 조각 시작
        process.run();
        //코드 조각 종료
        System.out.println("프로그램 종료");
    }

    static class Dice implements Process {

        @Override
        public void run() {
            int randomValue = new Random().nextInt(6) + 1;
            System.out.println("주사위 = " + randomValue);
        }
    }

    static class Sum implements Process {

        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
```

```
        }
    }

    public static void main(String[] args) {
        Process dice = new Dice();
        Process sum = new Sum();

        System.out.println("Hello 실행");
        hello(dice);
        hello(sum);
    }
}
```

코드를 분석해보자.

여기서는 단순히 데이터를 전달하는 수준을 넘어서, 코드 조각을 전달해야 한다.

리펙토링 전

```
public static void helloDice() {
    System.out.println("프로그램 시작"); //변하지 않는 부분

    //코드 조각 시작
    int randomValue = new Random().nextInt(6) + 1;
    System.out.println("주사위 = " + randomValue);
    //코드 조각 종료

    System.out.println("프로그램 종료"); //변하지 않는 부분
}

public static void helloSum() {
    System.out.println("프로그램 시작"); //변하지 않는 부분

    //코드 조각 시작
    for (int i = 1; i <= 3; i++) {
        System.out.println("i = " + i);
    }
    //코드 조각 종료

    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

리펙토링 진행 단계

```
public static void hello() {
    System.out.println("프로그램 시작"); //변하지 않는 부분

    //코드 조각 시작
    int randomValue = new Random().nextInt(6) + 1;
    System.out.println("주사위 = " + randomValue);
    //코드 조각 종료

    //코드 조각 시작
    for (int i = 1; i <= 3; i++) {
        System.out.println("i = " + i);
    }
    //코드 조각 종료

    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

- 프로그램 시작, 프로그램 종료를 출력하는 부분은 변하지 않는 부분이다.
- 코드 조각을 시작하고 종료하는 부분은 변하는 부분이다.
- 결국 코드 조각을 시작하고 종료하는 부분을 외부에서 전달 받아야 한다. 이것은 단순히 문자열 같은 데이터를 전달 받는 것과는 차원이 다른 문제이다.

어떻게 외부에서 코드 조각을 전달할 수 있을까?

코드 조각은 보통 메서드(함수)에 정의한다. 따라서 코드 조각을 전달하기 위해서는 메서드가 필요하다.

그런데 지금까지 학습한 내용으로는 메서드를 전달할 수 있는 방법이 없다. 대신에 인스턴스를 전달하고, 인스턴스에 있는 메서드를 호출하면 된다.

이 문제를 해결하기 위해 인터페이스를 정의하고 구현 클래스를 만들었다.

```
public interface Process {
    void run();
}
static class Dice implements Process {

    @Override
    public void run() {
        int randomValue = new Random().nextInt(6) + 1;
```

```
            System.out.println("주사위 = " + randomValue);
        }
    }
    static class Sum implements Process {

        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        }
    }
```

- `Dice`, `Sum` 각각의 클래스는 `Process` 인터페이스를 구현하고 `run()` 메서드에 필요한 코드 조각을 구현한다.
- 여기서는 정적 중첩 클래스를 사용했다. 물론 정적 중첩 클래스가 아니라 외부에 클래스를 직접 만들어도 된다.

리팩토링 완료

```
public static void hello(Process process) {
    System.out.println("프로그램 시작"); //변하지 않는 부분
    //코드 조각 시작
    process.run();
    //코드 조각 종료
    System.out.println("프로그램 종료"); //변하지 않는 부분
}
```

- `Process process` 매개변수를 통해 인스턴스를 전달할 수 있다. 이 인스턴스의 `run()` 메서드를 실행하면 필요한 코드 조각을 실행할 수 있다.
- 이때 다형성을 활용해서 외부에서 전달되는 인스턴스에 따라 각각 다른 코드 조각이 실행된다.

```
public static void main(String[] args) {
    Process dice = new Dice();
    Process sum = new Sum();

    System.out.println("Hello 실행");
    hello(dice);
    hello(sum);
}
```

- `hello()`를 호출할 때 어떤 인스턴스를 전달하는 가에 따라서 다른 결과가 실행된다.
- `hello(dice)`를 호출하면 주사위 로직이, `hello(sum)`을 호출하면 계산 로직이 수행된다.

실행 결과

```
Hello 실행
프로그램 시작
주사위 = 5 //랜덤
프로그램 종료
프로그램 시작
i = 1
i = 2
i = 3
프로그램 종료
```

정리

- 문자열 같은 데이터를 메서드에 전달할 때는 `String`, `int`와 같은 각 데이터에 맞는 타입을 전달하면 된다.
- 코드 조각을 메서드에 전달할 때는 인스턴스를 전달하고 해당 인스턴스에 있는 메서드를 호출하면 된다.

익명 클래스 활용3

이번에는 지역 클래스를 사용해서 같은 기능을 구현해보자.

지역 클래스 사용

```
package nested.anonymous.ex;

import java.util.Random;

//지역 클래스 사용
public class Ex1RefMainV2 {

    public static void hello(Process process) {
        System.out.println("프로그램 시작");
        //코드 조각 시작
        process.run();
```

```
        //코드 조각 종료
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {

        class Dice implements Process {

            @Override
            public void run() {
                int randomValue = new Random().nextInt(6) + 1;
                System.out.println("주사위 = " + randomValue);
            }
        }

        class Sum implements Process {

            @Override
            public void run() {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("i = " + i);
                }
            }
        }

        Process dice = new Dice();
        Process sum = new Sum();

        System.out.println("Hello 실행");
        hello(dice);
        hello(sum);
    }
}
```

실행 결과는 기존과 같다. 이해하는데 어려움은 없을 것이다.

익명 클래스 사용1

앞의 지역 클래스는 간단히 한번만 생성해서 사용한다. 이런 경우 익명 클래스로 변경할 수 있다.

```
package nested.anonymous.ex;
```

```java
import java.util.Random;

//익명 클래스 사용
public class Ex1RefMainV3 {

    public static void hello(Process process) {
        System.out.println("프로그램 시작");
        //코드 조각 시작
        process.run();
        //코드 조각 종료
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        Process dice = new Process() {
            @Override
            public void run() {
                int randomValue = new Random().nextInt(6) + 1;
                System.out.println("주사위 = " + randomValue);
            }
        };

        Process sum = new Process() {
            @Override
            public void run() {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("i = " + i);
                }
            }
        };

        System.out.println("Hello 실행");
        hello(dice);
        hello(sum);
    }
}
```

실행 결과는 기존과 같다.

익명 클래스 사용2 - 참조값 직접 전달

이 경우 익명 클래스의 참조값을 변수에 담아둘 필요 없이, 인수로 바로 전달할 수 있다.

```java
package nested.anonymous.ex;

import java.util.Random;

//익명 클래스 참조 바로 전달
public class Ex1RefMainV4 {

    public static void hello(Process process) {
        System.out.println("프로그램 시작");
        //코드 조각 시작
        process.run();
        //코드 조각 종료
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        hello(new Process() {
            @Override
            public void run() {
                int randomValue = new Random().nextInt(6) + 1;
                System.out.println("주사위 = " + randomValue);
            }
        });

        hello(new Process() {
            @Override
            public void run() {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("i = " + i);
                }
            }
        });
    }
}
```

실행 결과는 기존과 같다.

람다(lambda)

자바8 이전까지 메서드에 인수로 전달할 수 있는 것은 크게 2가지였다.

- `int`, `double`과 같은 기본형 타입

- `Process` `Member`와 같은 참조형 타입(인스턴스)

결국 메서드에 인수로 전달할 수 있는 것은 간단한 데이터나, 인스턴스의 참조이다.

지금처럼 코드 조각을 전달하기 위해 클래스를 정의하고 메서드를 만들고 또 인스턴스를 꼭 생성해서 전달해야 할까? 생각해보면 클래스나 인스턴스와 관계 없이 다음과 같이 메서드만 전달할 수 있다면 더 간단하지 않을까?

```
public void runDice() {
    int randomValue = new Random().nextInt(6) + 1;
    System.out.println("주사위 = " + randomValue);
}

public void runSum() {
    for (int i = 1; i <= 3; i++) {
        System.out.println("i = " + i);
    }
}
```

```
hello(메서드 전달: runDice())
hello(메서드 전달: runRun())
```

자바8에 들어서면서 큰 변화가 있었는데 바로 메서드(더 정확히는 함수)를 인수로 전달할 수 있게 되었다. 이것을 간단히 람다(Lambda)라 한다.

리팩토링 - 람다

```
package nested.anonymous.ex;

import java.util.Random;

//람다 사용
public class Ex1RefMainV5 {

    public static void hello(Process process) {
        System.out.println("프로그램 시작");
        //코드 조각 시작
        process.run();
        //코드 조각 종료
```

```
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        hello(() -> {
            int randomValue = new Random().nextInt(6) + 1;
            System.out.println("주사위 = " + randomValue);
        });

        hello(() -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        });
    }
}
```

- 코드를 보면 클래스나 인스턴스를 정의하지 않고, 메서드(더 정확히는 함수)의 코드 블럭을 직접 전달하는 것을 확인할 수 있다.

람다에 대한 자세한 내용은 이후에 별도로 다룬다. 지금은 이런 것이 있구나 정도만 알아두자.

문제와 풀이1

다음 클래스를 간단히 만들어보자.

- 정적 중첩 클래스
- 내부 클래스
- 지역 클래스
- 익명 클래스

문제1 - 정적 중첩 클래스를 완성해라

```
package nested.test;

public class OuterClass1 {
    // 여기에 NestedClass를 구현해라. 그리고 hello() 메서드를 만들어라.
}
```

```
package nested.test;

public class OuterClass1Main {

    public static void main(String[] args) {
        // 여기에서 NestedClass의 hello() 메서드를 호출하라.
    }
}
```

실행 결과

```
NestedClass.hello
```

정답

```
package nested.test;

public class OuterClass1 {
    // 여기에 NestedClass를 구현해라. 그리고 hello() 메서드를 만들어라.
    static class NestedClass {
        public void hello() {
            System.out.println("NestedClass.hello");
        }
    }
}
```

```
package nested.test;

public class OuterClass1Main {

    public static void main(String[] args) {
        // 여기에서 NestedClass의 hello() 메서드를 호출하라.
        OuterClass1.NestedClass nested = new OuterClass1.NestedClass();
```

```
        nested.hello();
    }
}
```

문제2 - 내부 클래스를 완성해라

```
package nested.test;

public class OuterClass2 {
    // 여기에 InnerClass를 구현해라. 그리고 hello() 메서드를 만들어라.
}
```

```
package nested.test;

public class OuterClass2Main {
    public static void main(String[] args) {
        // 여기에서 InnerClass의 hello() 메서드를 호출해라.
    }
}
```

실행 결과

```
InnerClass.hello
```

정답

```
package nested.test;

public class OuterClass2 {

    // 여기에 InnerClass를 구현해라. 그리고 hello() 메서드를 만들어라.
    class InnerClass {
        public void hello() {
```

```
            System.out.println("InnerClass.hello");
        }
    }
}
```

```
package nested.test;

public class OuterClass2Main {
    public static void main(String[] args) {
        // 여기에서 InnerClass의 hello() 메서드를 호출해라.
        OuterClass2 outer = new OuterClass2();
        OuterClass2.InnerClass inner = outer.new InnerClass();
        inner.hello();
    }
}
```

문제3 - 지역 클래스를 완성해라

```
package nested.test;

class OuterClass3 {
    public void myMethod() {
        // 여기에 지역 클래스 LocalClass를 구현하고 hello() 메서드를 호출해라.
    }
}
```

```
package nested.test;

class OuterClass3Main {

    public static void main(String[] args) {
        OuterClass3 outerClass3 = new OuterClass3();
        outerClass3.myMethod();
    }
}
```

실행 결과

```
LocalClass.hello
```

정답

```
package nested.test;

class OuterClass3 {
    public void myMethod() {
        // 여기에 지역 클래스 LocalClass를 구현하고 hello() 메서드를 호출해라.
        class LocalClass {
            public void hello() {
                System.out.println("LocalClass.hello");
            }
        }

        LocalClass local = new LocalClass();
        local.hello();
    }
}
```

문제4 - 익명 클래스를 완성해라

```
package nested.test;

public interface Hello {
    void hello();
}
```

```
package nested.test;

class AnonymousMain {
```

```
    public static void main(String[] args) {
        // 여기에서 Hello의 익명 클래스를 생성하고 hello()를 호출해라.
    }
}
```

실행 결과

```
Hello.hello
```

정답

```
package nested.test;

class AnonymousMain {

    public static void main(String[] args) {
        // 여기에서 Hello의 익명 클래스를 생성하고 hello()를 호출해라.
        Hello hello = new Hello() {
            @Override
            public void hello() {
                System.out.println("Hello.hello");
            }
        };

        hello.hello();
    }
}
```

문제와 풀이2

문제: 도서 관리 시스템

도서관에서 사용할 수 있는 간단한 도서 관리 시스템을 만들어 보자. 이 시스템은 여러 권의 도서 정보를 관리할 수 있어야 한다. 각 도서는 도서 제목(`title`)과 저자명(`author`)을 가지고 있다. 시스템은 도서를 추가하고, 모든 도서의 정보를 출력하는 기능을 제공해야 한다.

- `Library` 클래스를 완성해라.
  - `LibraryMain`과 실행 결과를 참고하자.
- `Book` 클래스는 `Library` 내부에서만 사용된다. `Library` 클래스 외부로 노출하면 안된다.
- `Library` 클래스는 `Book` 객체 배열을 사용해서 도서 목록을 관리한다.

```
package nested.test.ex1;

public class Library {
    //코드 작성
}
```

```
package nested.test.ex1;

public class LibraryMain {

    public static void main(String[] args) {
        Library library = new Library(4); // 최대 4권의 도서를 저장할 수 있는 도서관 생
성
        library.addBook("책1", "저자1");
        library.addBook("책2", "저자2");
        library.addBook("책3", "저자3");
        library.addBook("자바 ORM 표준 JPA 프로그래밍", "김영한");
        library.addBook("OneMoreThing", "잡스");
        library.showBooks(); // 도서관의 모든 도서 정보 출력
    }
}
```

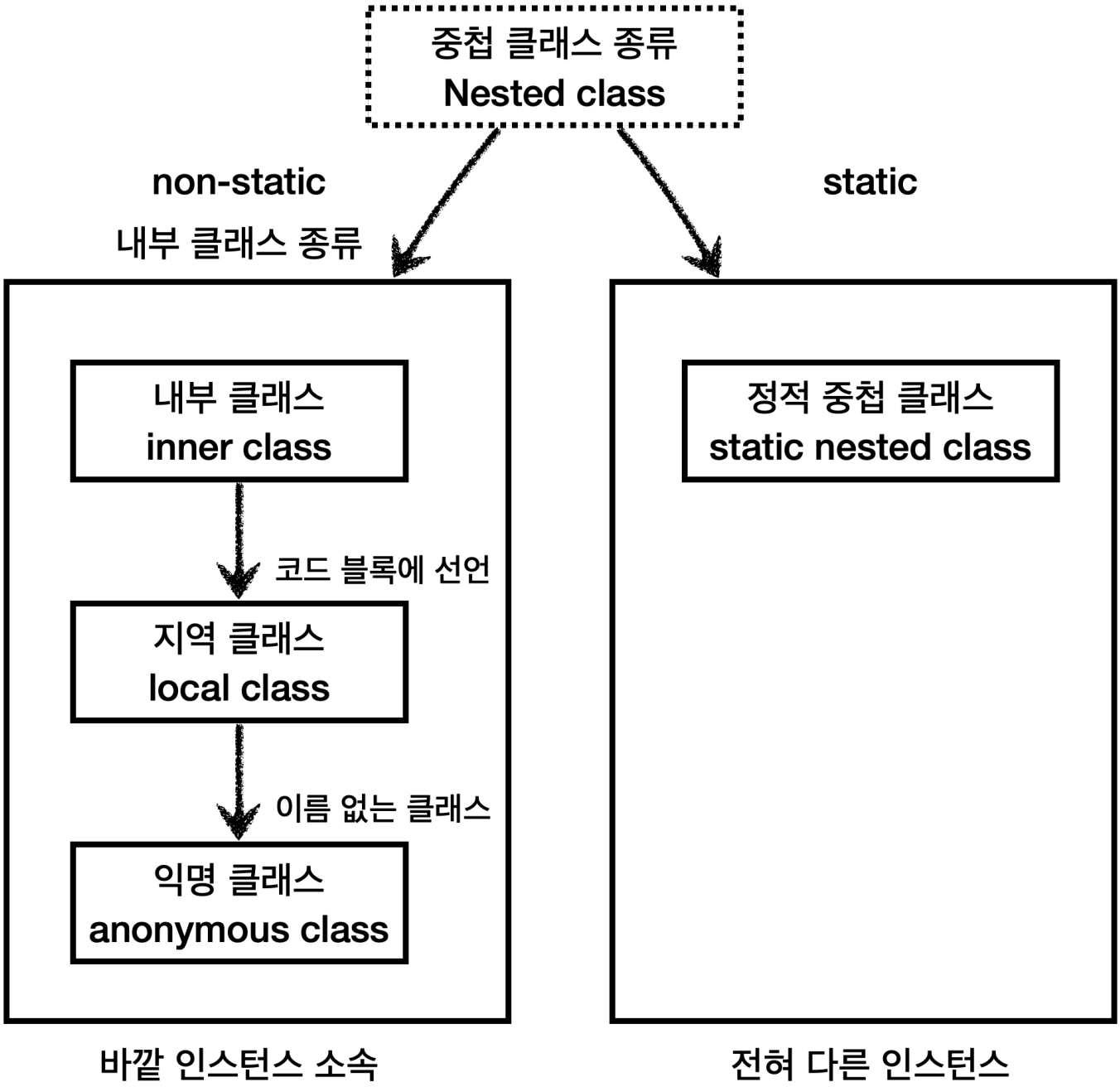
실행 결과

```
도서관 저장 공간이 부족합니다.
== 책 목록 출력 ==
도서 제목: 책1, 저자: 저자1
도서 제목: 책2, 저자: 저자2
도서 제목: 책3, 저자: 저자3
도서 제목: 자바 ORM 표준 JPA 프로그래밍, 저자: 김영한
```

정답

```
package nested.test.ex1;

public class Library {
    private Book[] books;
    private int bookCount;

    public Library(int size) {
        books = new Book[size];
        bookCount = 0;
    }

    public void addBook(String title, String author) {
        if (bookCount < books.length) {
            books[bookCount++] = new Book(title, author);
        } else {
            System.out.println("도서관 저장 공간이 부족합니다.");
        }
    }

    public void showBooks() {
        System.out.println("== 책 목록 출력 ==");
        for (int i = 0; i < bookCount; i++) {
            System.out.println("도서 제목: " + books[i].title + ", 저자: " +
books[i].author);
        }
    }

    private static class Book {
        private String title;
        private String author;

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

}
```

정리



- **정적 중첩 클래스**: 바깥 클래스와 밀접한 관련이 있지만, 인스턴스 간에 데이터 공유가 필요 없을 때 사용한다.
- **내부 클래스**: 바깥 클래스의 인스턴스와 연결되어 있고, 바깥 클래스의 인스턴스 상태에 의존하거나 강하게 연관된 작업을 수행할 때 사용한다.
- **지역 클래스**:
  - 내부 클래스의 특징을 가진다.
  - 지역 변수에 접근할 수 있다. 접근하는 지역 변수는 `final`이거나 사실상 `final`이어야 한다.

    - 주로 특정 메서드 내에서만 간단히 사용할 목적으로 사용한다.
- **익명 클래스**:
    - 지역 클래스인데, 이름이 없다.
    - 상위 타입을 상속 또는 구현하면서 바로 생성된다.
    - 주로 특정 상위 타입을 간단히 구현해서 일회성으로 사용할 때 유용하다.