7. 중첩 클래스, 내부 클래스1

#1.인강/0.자바/3.자바-중급1편

- /중첩 클래스, 내부 클래스란?
- /정적 중첩 클래스
- /정적 중첩 클래스의 활용
- /내부 클래스
- /내부 클래스의 활용
- /같은 이름의 바깥 변수 접근

중첩 클래스, 내부 클래스란?

다음과 같이 for문 안에 for문을 중첩하는 것을 중첩(Nested) for문이라 한다.

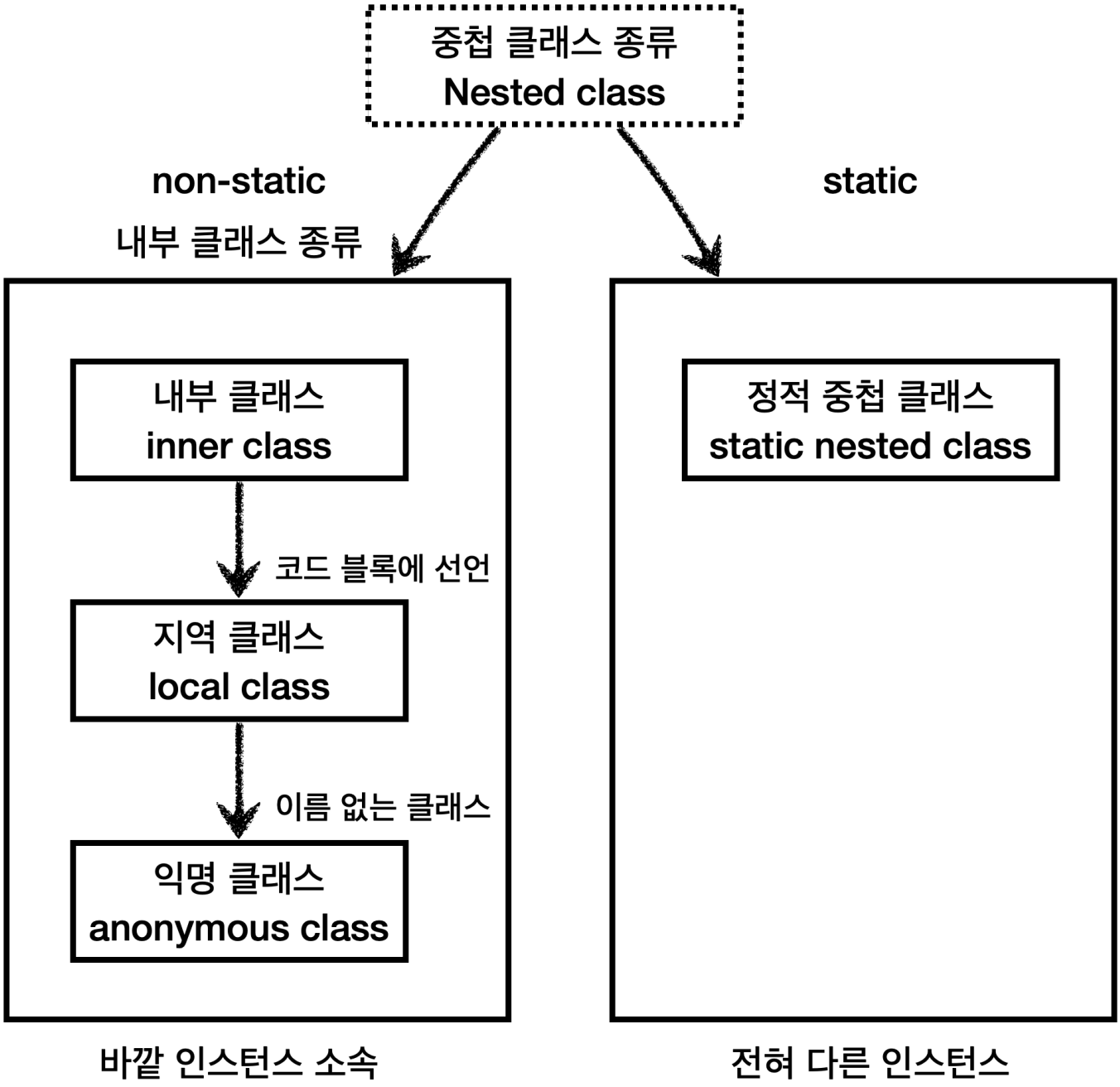
```
for (...) {
    //중첩 for문
    for (...) {
    }
}
```

다음과 같이 클래스 안에 클래스를 중첩해서 정의할 수 있는데, 이것을 중첩 클래스(Nested Class)라 한다.

```
class Outer {
    ...
    //중첩 클래스
    class Nested {
        ...
    }
}
```

중첩 클래스는 클래스를 정의하는 위치에 따라 다음과 같이 분류한다.

중첩 클래스의 분류



중첩 클래스는 총 4가지가 있고, 크게 2가지로 분류할 수 있다.

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

중첩 클래스를 정의하는 위치는 변수의 선언 위치와 같다.

변수의 선언 위치

- 정적 변수(클래스 변수)
- 인스턴스 변수
- 지역 변수

중첩 클래스의 선언 위치

- 정적 중첩 클래스 ➜ 정적 변수와 같은 위치
- 내부 클래스 ➜ 인스턴스 변수와 같은 위치
- 지역 클래스 ➜ 지역 변수와 같은 위치

```
class Outer {
    ...
    //정적 중첩 클래스
    static class StaticNested {
        ...
    }

    //내부 클래스
    class Inner {
        ...
    }
}
```

- 정적 중첩 클래스는 정적 변수와 같이 앞에 `static`이 붙어있다.
- 내부 클래스는 인스턴스 변수와 같이 앞에 `static`이 붙어있지 않다.

```
class Outer {

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

        //지역 클래스
        class Local {...}

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

- 지역 클래스는 지역 변수와 같이 코드 블럭 안에서 클래스를 정의한다.
- 참고로 익명 클래스는 지역 클래스의 특별한 버전이다. 이후에 설명한다.

다시 한번 정리해보자.

중첩 클래스는 총 4가지가 있고, 크게 2가지로 분류할 수 있다.

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

여기서 정적 중첩 클래스와 내부 클래스로 분류하는 것을 확인할 수 있다.

그럼 중첩이라는 단어와 내부라는 단어는 무슨 차이가 있는 것일까?

- **중첩(Nested)**: 어떤 다른 것이 내부에 위치하거나 포함되는 구조적인 관계
- **내부(Inner)**: 나의 내부에 있는 나를 구성하는 요소

쉽게 이야기하면 여기서 의미하는 중첩(`Nested`)은 나의 안에 있지만 내것이 아닌 것을 말한다. 단순히 위치만 안에 있는 것이다. 반면에 여기서 의미하는 내부(`Inner`)는 나의 내부에서 나를 구성하는 요소를 말한다.
예)

- 큰 나무 상자안에 전혀 다른 작은 나무 상자를 넣은 것은 중첩(`Nested`)이라 한다.
- 나의 심장은 나의 내부(`Inner`)에서 나를 구성하는 요소이다.

정리하면 정적 중첩 클래스는 바깥 클래스의 안에 있지만 바깥 클래스와 관계 없는 전혀 다른 클래스를 말한다.
내부 클래스는 바깥 클래스의 내부에 있으면서 바깥 클래스를 구성하는 요소를 말한다.

여기서 의미하는 중첩(`Nested`)과 내부(`Inner`)를 분류하는 핵심은 바로 바깥 클래스 입장에서 볼 때 안에 있는 클래스가 나의 인스턴스에 소속이 되는가 되지 않는가의 차이이다.

- 정적 중첩 클래스는 바깥 클래스와 전혀 다른 클래스이다. 따라서 바깥 클래스의 인스턴스에 소속되지 않는다.
- 내부 클래스는 바깥 클래스를 구성하는 요소이다. 따라서 바깥 클래스의 인스턴스에 소속된다.

정리하면 내부 클래스들은 바깥 클래스의 인스턴스에 소속된다. 정적 중첩 클래스는 그렇지 않다.

정적 중첩 클래스

- `static`이 붙는다.
- 바깥 클래스의 인스턴스에 소속되지 않는다.

내부 클래스

- `static`이 붙지 않는다.
- 바깥 클래스의 인스턴스에 소속된다.

내부 클래스의 종류

- 내부 클래스(inner class): 바깥 클래스의 인스턴스의 멤버에 접근
- 지역 클래스(local class): 내부 클래스의 특징 + 지역 변수에 접근
- 익명 클래스(anonymous class): 지역 클래스의 특징 + 클래스의 이름이 없는 특별한 클래스

용어 정리

- **중첩 클래스**: 정적 중첩 클래스 + 내부 클래스 종류 모두 포함
- **정적 중첩 클래스**: 정적 중첩 클래스를 말함
- **내부 클래스**: 내부 클래스, 지역 클래스, 익명 클래스를 포함해서 말함

> **참고 - 실무 용어**
> 실무에서는 중첩, 내부라는 단어를 명확히 구분하지 않고, 중첩 클래스 또는 내부 클래스라고 이야기한다. 왜냐하면 클래스 안에 클래스가 있는 것을 중첩 클래스라고 하기 때문이다. 그리고 내부 클래스도 중첩 클래스의 한 종류이다. 따라서 둘을 명확히 구분하지는 않는다. 엄밀하게 이야기하면 `static`이 붙어있는 정적 중첩 클래스는 내부 클래스라고 하면 안된다. 하지만 대부분의 개발자들이 둘을 구분해서 말하지 않기 때문에 내부 또는 중첩 클래스라고 하면 상황과 문맥에 따라서 이해하면 된다.

중첩 클래스는 언제 사용해야 하나?

- 내부 클래스를 포함한 모든 중첩 클래스는 특정 클래스가 다른 하나의 클래스 안에서만 사용되거나, 둘이 아주 긴밀하게 연결되어 있는 특별한 경우에만 사용해야 한다. 외부의 여러 클래스가 특정 중첩 클래스를 사용한다면 중첩 클래스로 만들면 안된다.

중첩 클래스를 사용하는 이유

- **논리적 그룹화**: 특정 클래스가 다른 하나의 클래스 안에서만 사용되는 경우 해당 클래스 안에 포함하는 것이 논리적으로 더 그룹화 된다. 패키지를 열었을 때 다른 곳에서 사용될 필요가 없는 중첩 클래스가 외부에 노출되지 않는 장점도 있다.
- **캡슐화**: 중첩 클래스는 바깥 클래스의 `private` 멤버에 접근할 수 있다. 이렇게 해서 둘을 긴밀하게 연결하고 불필요한 `public` 메서드를 제거할 수 있다. 이 부분은 말로 이해하기는 어렵기 때문에 이후에 예제를 통해서 알아보자.

정적 중첩 클래스

예제 코드를 통해 정적 중첩 클래스(static nested class)를 알아보자.

```
package nested.nested;
```

```
public class NestedOuter {

    private static int outClassValue = 3;
    private int outInstanceValue = 2;

    static class Nested {
        private int nestedInstanceValue = 1;

        public void print() {

            // 자신의 멤버에 접근
            System.out.println(nestedInstanceValue);

            // 바깥 클래스의 인스턴스 멤버에는 접근할 수 없다.
            //System.out.println(outInstanceValue);

            // 바깥 클래스의 클래스 멤버에는 접근할 수 있다. private도 접근 가능
            System.out.println(NestedOuter.outClassValue);
        }
    }

}
```

- 정적 중첩 클래스는 앞에 `static`이 붙는다.
- 정적 중첩 클래스는
  - 자신의 멤버에는 당연히 접근할 수 있다.
  - 바깥 클래스의 인스턴스 멤버에는 접근할 수 없다.
  - 바깥 클래스의 클래스 멤버에는 접근할 수 있다.

참고로 `NestedOuter.outClassValue`를 `outClassValue`와 같이 줄여서 사용해도 된다. 이 경우 바깥 클래스에 있는 필드를 찾아서 사용한다.

private 접근 제어자

- `private` 접근 제어자는 같은 클래스 안에 있을 때만 접근할 수 있다.
- 중첩 클래스도 바깥 클래스와 같은 클래스 안에 있다. 따라서 중첩 클래스는 바깥 클래스의 `private` 접근 제어자에 접근할 수 있다.

```
package nested.nested;
```

```java
public class NestedOuterMain {

    public static void main(String[] args) {
        NestedOuter outer = new NestedOuter();
        NestedOuter.Nested nested = new NestedOuter.Nested();
        nested.print();

        System.out.println("nestedClass = " + nested.getClass());
    }
}
```

- 정적 중첩 클래스는 `new 바깥클래스.중첩클래스()` 로 생성할 수 있다.
- 중첩 클래스는 `NestedOuter.Nested` 와 같이 `바깥 클래스.중첩클래스` 로 접근할 수 있다.
- 여기서 `new NestedOuter()` 로 만든 바깥 클래스의 인스턴스와 `new NestedOuter.Nested()` 로 만든 정적 중첩 클래스의 인스턴스는 서로 아무 관계가 없는 인스턴스이다. 단지 클래스 구조상 중첩해 두었을 뿐이다.
    - 참고로 둘이 아무런 관련이 없으므로 정적 중첩 클래스의 인스턴스만 따로 생성해도 된다.

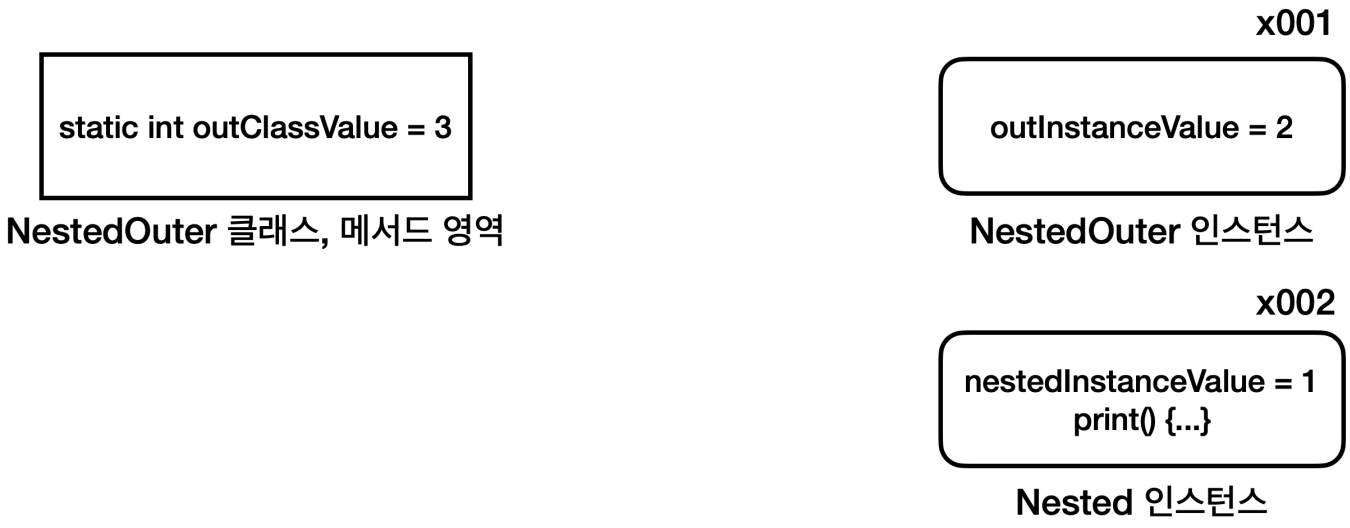
실행 결과

```
1
3
nestedClass = class nested.nested.NestedOuter$Nested
```

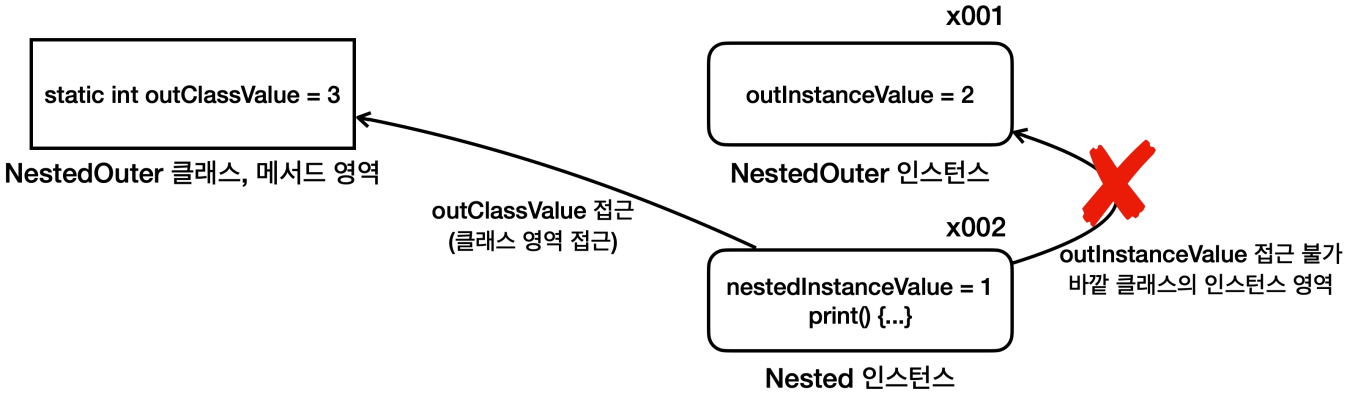
중첩 클래스를 출력해보면 중첩 클래스의 이름은 `NestedOuter$Nested` 와 같이 바깥 클래스, `$` , 중첩 클래스의 조합으로 만들어진다.

그림을 통해 코드를 분석해보자.

인스턴스가 생성된 상태



바깥 클래스의 멤버에 접근



`Nested.print()`를 살펴보자.

정적 중첩 클래스는 바깥 클래스의 정적 필드에는 접근할 수 있다. 하지만 바깥 클래스가 만든 인스턴스 필드에는 바로 접근할 수 없다. 바깥 인스턴스의 참조가 없기 때문이다.

정리

정적 중첩 클래스는 사실 다른 클래스를 그냥 중첩해 둔 것일 뿐이다! 쉽게 이야기해서 둘은 아무런 관계가 없다.

`NestedOuter.outClassValue`와 같은 정적 필드에 접근하는 것은 중첩 클래스가 아니어도 어차피 `클래스명.정적 필드명`으로 접근할 수 있다.

쉽게 이야기해서 다음과 같이 정적 중첩 클래스를 만들지 않고, 그냥 클래스2개를 따로 만든것과 같다.

```
class NestedOuter {
}

class Nested {
}
```

이 코드와 정적 중첩 클래스의 유일한 차이는 같은 클래스에 있으니 `private` 접근 제어자에 접근할 수 있다는 정도이다.

정적 중첩 클래스의 활용

간단한 예제를 리팩토링 하면서 정적 중첩 클래스를 어떻게 활용하는지 알아보자.

정적 중첩 클래스로 리팩토링 전

```
package nested.nested.ex1;
```

```java
// Network 객체 안에서만 사용
public class NetworkMessage {
    private String content;

    public NetworkMessage(String content) {
        this.content = content;
    }

    public void print() {
        System.out.println(content);
    }
}
```

- `NetworkMessage`는 `Network` 객체 안에서만 사용되는 객체이다.

```java
package nested.nested.ex1;

public class Network {

    public void sendMessage(String text) {
        NetworkMessage networkMessage = new NetworkMessage(text);
        networkMessage.print();
    }

}
```

- `text`를 입력 받아서 `NetworkMessage`를 생성하고 출력하는 단순한 기능을 제공한다.

```java
package nested.nested.ex1;

public class NetworkMain {

    public static void main(String[] args) {
        Network network = new Network();
        network.sendMessage("hello java");
    }
}
```

- `Network`를 생성하고 `network.sendMessage()`를 통해 메시지를 전달한다.

- `NetworkMain`은 오직 `Network` 클래스만 사용한다. `NetworkMessage` 클래스는 전혀 사용하지 않는다. `NetworkMessage`는 오직 `Network` 내부에서만 사용된다.

실행 결과

```
hello java
```

`ex1` 패키지를 열어보면 다음 두 클래스가 보일 것이다. (`main`은 제외)

- `Network`
- `NetworkMessage`

`Network` 관련 라이브러리를 사용하기 위해서 `ex1` 패키지를 열어본 개발자는 아마도 두 클래스를 모두 확인해볼 것이다. 그리고 해당 패키지를 처음 확인한 개발자는 `Network`와 `NetworkMessage`를 둘다 사용해야 하나? 라고 생각할 것이다. `NetworkMessage`에 메시지를 담아서 `Network`에 전달해야 하나?와 같은 여러가지 생각을 할 것이다.
아니면 `NetworkMessage`가 다른 여러 클래스에서 사용되겠구나 라고 생각할 것이다.
두 클래스의 코드를 모두 확인하고 나서야 아~ `Network` 클래스만 사용하면 되는구나, `NetworkMessage`는 단순히 `Network` 안에서만 사용되는구나 라고 이해할 수 있다.

정적 중첩 클래스로 리팩토링 후

```
package nested.nested.ex2;

public class Network {

    public void sendMessage(String text) {
        NetworkMessage networkMessage = new NetworkMessage(text);
        networkMessage.print();
    }

    private static class NetworkMessage {
        private String content;

        public NetworkMessage(String content) {
            this.content = content;
        }
```

```
        public void print() {
            System.out.println(content);
        }
    }

}
```

- `NetworkMessage` 클래스를 `Network` 클래스 안에 중첩해서 만들었다.
- `NetworkMessage`의 접근 제어자를 `private` 설정했다. 따라서 외부에서 `NetworkMessage`에 접근할 수 없다.
  - 예) `new Network.NetworkMessage()`처럼 접근할 수 없다.

```
package nested.nested.ex2;

public class NetworkMain {

    public static void main(String[] args) {
        Network network = new Network();
        network.sendMessage("hello java");
    }
}
```

실행 결과

```
hello java
```

`ex1` 패키지를 열어보면 다음 하나의 클래스가 보일 것이다. (`main`은 제외)

- `Network`

`Network` 관련 라이브러리를 사용하기 위해서 `ex2` 패키지를 열어본 개발자는 해당 클래스만 확인할 것이다. 추가로 `NetworkMessage`가 중첩 클래스에 `private` 접근 제어자로 되어 있는 것을 보고, `Network` 내부에서만 단독으로 사용하는 클래스라고 바로 인지할 수 있다.

중첩 클래스의 접근

나의 클래스에 포함된 중첩 클래스가 아니라 다른 곳에 있는 중첩 클래스에 접근할 때는 `바깥클래스.중첩클래스`로 접근

해야 한다.

```
NestedOuter.Nested nested = new NestedOuter.Nested();
```

나의 클래스에 포함된 중첩 클래스에 접근할 때는 바깥 클래스 이름을 적지 않아도 된다.

```
public class Network {
    public void sendMessage(String text) {
        NetworkMessage networkMessage = new NetworkMessage(text);
    }
    private static class NetworkMessage {...}
}
```

중첩 클래스(내부 클래스 포함)는 그 용도가 자신이 소속된 바깥 클래스 안에서 사용되는 것이다. 따라서 자신이 소속된 바깥 클래스가 아닌 외부에서 생성하고 사용하고 있다면, 이미 중첩 클래스의 용도에 맞지 않을 수 있다. 이때는 중첩 클래스를 밖으로 빼는 것이 더 나은 선택이다.

내부 클래스

정적 중첩 클래스는 바깥 클래스와 서로 관계가 없다. 하지만 내부 클래스는 바깥 클래스의 인스턴스를 이루는 요소가 된다. 쉽게 이야기해서 내부 클래스는 바깥 클래스의 인스턴스에 소속된다.

정적 중첩 클래스

- `static`이 붙는다.
- 바깥 클래스의 인스턴스에 소속되지 않는다.

내부 클래스

- `static`이 붙지 않는다.
- 바깥 클래스의 인스턴스에 소속된다.

예제 코드를 통해 내부 클래스를 알아보자.

```
package nested.inner;

public class InnerOuter {
```

```
    private static int outClassValue = 3;
    private int outInstanceValue = 2;

    class Inner {
        private int innerInstanceValue = 1;

        public void print() {
            // 자신의 멤버에 접근
            System.out.println(innerInstanceValue);

            // 외부 클래스의 인스턴스 멤버에 접근 가능, private도 접근 가능
            System.out.println(outInstanceValue);
            // 외부 클래스의 클래스 멤버에는 접근 가능. private도 접근 가능
            System.out.println(InnerOuter.outClassValue);
        }
    }

}
```

- 내부 클래스는 앞에 `static`이 붙지 않는다. 쉽게 이야기해서 인스턴스 멤버가 된다.
- 내부 클래스는
  - 자신의 멤버에는 당연히 접근할 수 있다.
  - 바깥 클래스의 인스턴스 멤버에 접근할 수 있다.
  - 바깥 클래스의 클래스 멤버에 접근할 수 있다.

private 접근 제어자

- `private` 접근 제어자는 같은 클래스 안에 있을 때만 접근할 수 있다.
- 내부 클래스도 바깥 클래스와 같은 클래스 안에 있다. 따라서 내부 클래스는 바깥 클래스의 `private` 접근 제어자에 접근할 수 있다.

```
package nested.inner;

public class InnerOuterMain {

    public static void main(String[] args) {
        InnerOuter outer = new InnerOuter();
        InnerOuter.Inner inner = outer.new Inner();
        inner.print();
```

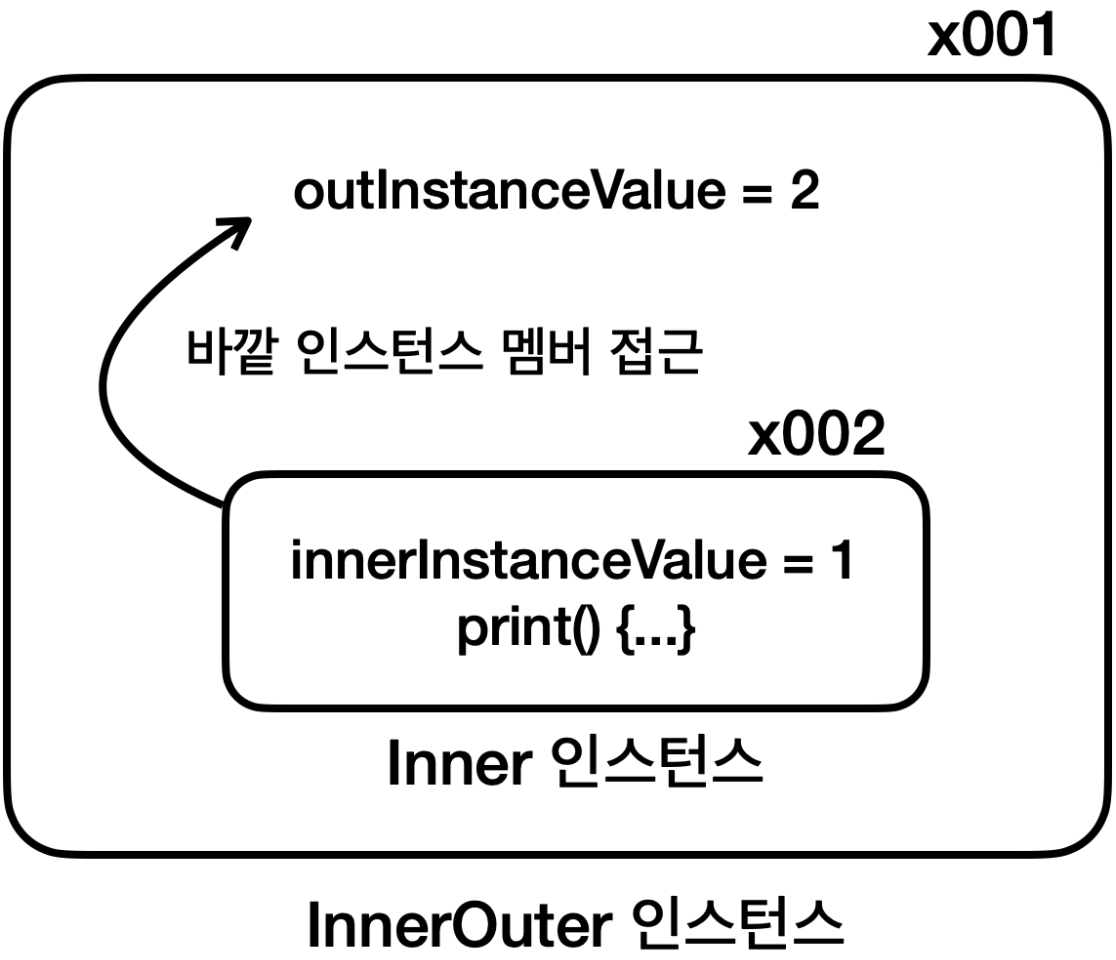
```
        System.out.println("innerClass = " + inner.getClass());
    }
}
```

- 내부 클래스는 바깥 클래스의 인스턴스에 소속된다. 따라서 바깥 클래스의 인스턴스 정보를 알아야 생성할 수 있다.
- 내부 클래스는 `바깥클래스의 인스턴스 참조.new 내부클래스()`로 생성할 수 있다.
  - 내부 클래스는 바깥 클래스의 인스턴스에 소속되어야 한다. 따라서 내부 클래스를 생성할 때, 바깥 클래스의 인스턴스 참조가 필요하다.
  - `outer.new Inner()`에서 `outer`는 바깥 클래스의 인스턴스 참조를 가진다.
- `outer.new Inner()`로 생성한 내부 클래스는 개념상 바깥 클래스의 인스턴스 내부에 생성된다.
- 따라서 바깥 클래스의 인스턴스를 먼저 생성해야 내부 클래스의 인스턴스를 생성할 수 있다.

실행 결과

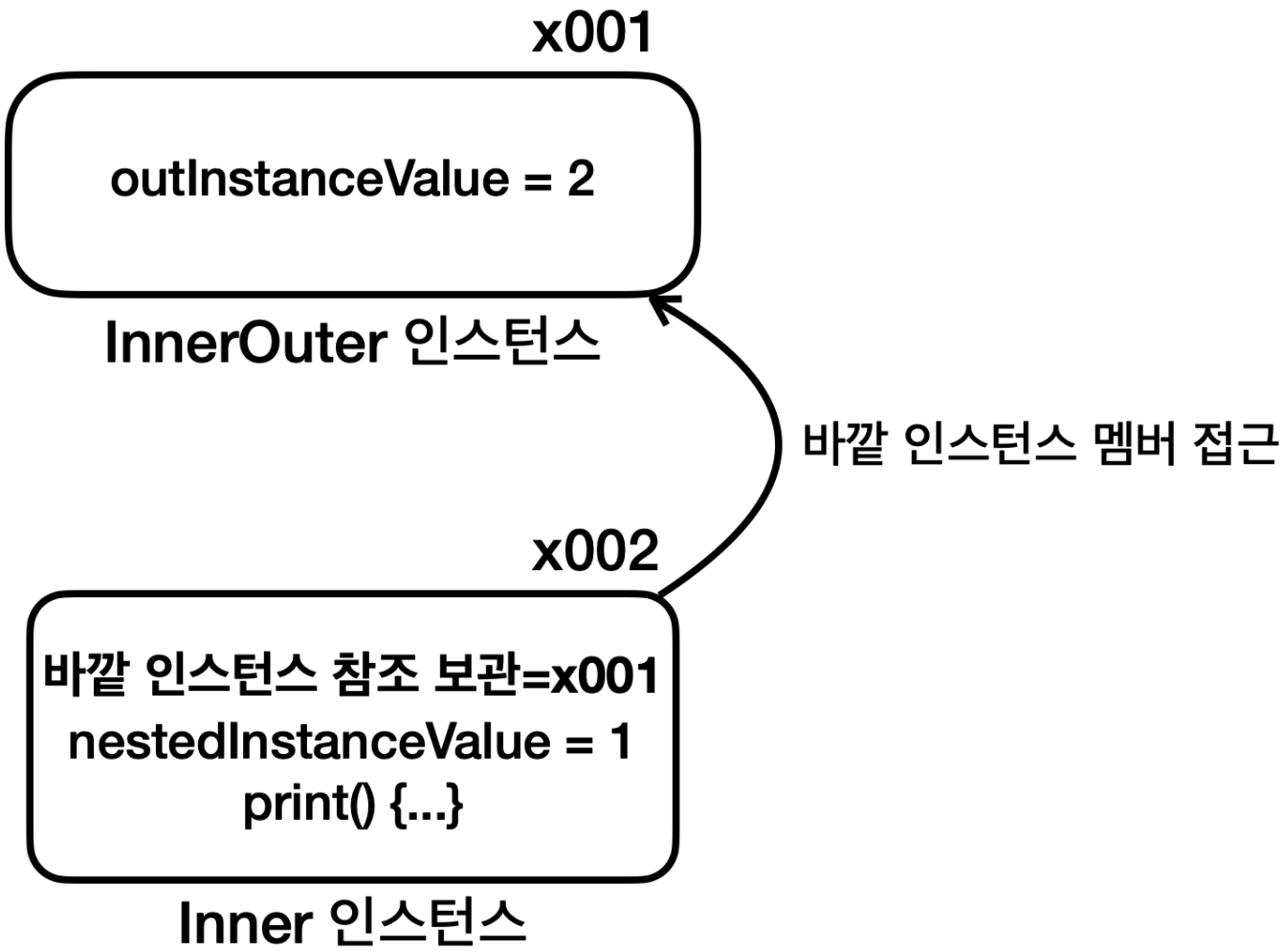
```
1
2
3
innerClass = class nested.inner.InnerOuter$Inner
```

개념 - 내부 클래스의 생성



- 개념상 바깥 클래스의 인스턴스 내부에서 내부 클래스의 인스턴스가 생성된다.
- 따라서 내부 인스턴스는 바깥 인스턴스를 알기 때문에 바깥 인스턴스의 멤버에 접근할 수 있다.

실제 - 내부 클래스의 생성



- 실제로 내부 인스턴스가 바깥 인스턴스 안에 생성되는 것은 아니다. 하지만 개념상 인스턴스 안에 생성된다고 이해하면 충분하다.
- 실제로는 내부 인스턴스는 바깥 인스턴스의 참조를 보관한다. 이 참조를 통해 바깥 인스턴스의 멤버에 접근할 수 있다.

정리

정적 중첩 클래스와 다르게 내부 클래스는 바깥 인스턴스에 소속된다.

중첩이라는 단어와 내부라는 단어의 차이를 다시 한번 정리해보자.

- **중첩(Nested)**: 어떤 다른 것이 내부에 위치하거나 포함되는 구조적인 관계
- **내부(Inner)**: 나의 내부에 있는 나를 구성하는 요소

중첩(`Nested`)은 나의 안에 있지만 내것이 아닌 것을 말한다. 단순히 위치만 안에 있는 것이다. 반면에 여기서 의미하는 내부(`Inner`)는 나의 내부에서 나를 구성하는 요소를 말한다.

정적 중첩 클래스는 다른 클래스를 그냥 중첩해 둔 것일 뿐이다. 쉽게 이야기해서 둘은 아무런 관계가 없다. 반면에 내부 클래스는 바깥 클래스의 인스턴스 내부에서 구성 요소로 사용된다.

내부 클래스의 활용

간단한 예제를 리팩토링 하면서 내부 클래스를 어떻게 활용하는지 알아보자.

내부 클래스로 리팩토링 전

```
package nested.inner.ex1;

//Car에서만 사용
public class Engine {

    private Car car;

    public Engine(Car car) {
        this.car = car;
    }

    public void start() {
        System.out.println("충전 레벨 확인: " + car.getChargeLevel());
        System.out.println(car.getModel() + "의 엔진을 구동합니다.");
    }

}
```

- 엔진은 `Car` 클래스에서만 사용된다.
- 엔진을 시작하기 위해서는 차의 충전 레벨과 차량의 이름이 필요하다.
  - `Car` 인스턴스의 참조를 생성자에서 보관한다.
  - 엔진은 충전 레벨을 확인하기 위해 `Car.getChargeLevel()`이 필요하다.
  - 엔진은 차량의 이름을 확인하기 위해 `Car.getModel()`이 필요하다.

```
package nested.inner.ex1;

public class Car {
    private String model;
    private int chargeLevel;
    private Engine engine;
```

```
    public Car(String model, int chargeLevel) {
        this.model = model;
        this.chargeLevel = chargeLevel;
        this.engine = new Engine(this);
    }

    //Engine에서만 사용하는 메서드
    public String getModel() {
        return model;
    }

    //Engine에서만 사용하는 메서드
    public int getChargeLevel() {
        return chargeLevel;
    }

    public void start() {
        engine.start();
        System.out.println(model + " 시작 완료");
    }

}
```

- `Car` 클래스는 엔진에 필요한 메서드들을 제공해야 한다. 다음 메서드는 엔진에서만 사용하고, 다른 곳에서는 사용하지 않는다.
  - `getModel()`
  - `getChargeLevel()`
- 결과적으로 `Car` 클래스는 엔진에서만 사용하는 기능을 위해 메서드를 추가해서, 모델 이름과 충전 레벨을 외부에 노출해야 한다.

```
package nested.inner.ex1;

public class CarMain {

    public static void main(String[] args) {
        Car myCar = new Car("Model Y", 100);
        myCar.start();
    }
}
```

실행 결과

```
충전 레벨 확인: 100
Model Y의 엔진을 구동합니다.
Model Y 시작 완료
```

내부 클래스로 리팩토링 후

엔진은 차의 내부에서만 사용된다. 엔진을 차의 내부 클래스로 만들어보자. 또한 엔진은 차의 충전 레벨과 모델 명에 접근해야 한다.

```
package nested.inner.ex2;

public class Car {
    private String model;
    private int chargeLevel;
    private Engine engine;

    public Car(String model, int chargeLevel) {
        this.model = model;
        this.chargeLevel = chargeLevel;
        this.engine = new Engine();
    }

    public void start() {
        engine.start();
        System.out.println(model + " 시작 완료");
    }

    private class Engine {
        public void start() {
            System.out.println("충전 레벨 확인: " + chargeLevel);
            System.out.println(model + "의 엔진을 구동합니다.");
        }
    }

}
```

- 엔진을 내부 클래스로 만들었다.

- `Engine.start()`를 기존과 비교해보자.
  - `Car`의 인스턴스 변수인 `chargeLevel`에 직접 접근할 수 있다.
  - `Car`의 인스턴스 변수인 `model`에 직접 접근할 수 있다.

내부 클래스의 생성

- 바깥 클래스에서 내부 클래스의 인스턴스를 생성할 때는 바깥 클래스 이름을 생략할 수 있다.
  - 예) `new Engine()`
- 바깥 클래스에서 내부 클래스의 인스턴스를 생성할 때 내부 클래스의 인스턴스는 자신을 생성한 바깥 클래스의 인스턴스를 자동으로 참조한다. 여기서 `new Engine()`로 생성된 `Engine` 인스턴스는 자신을 생성한 바깥의 `Car` 인스턴스를 자동으로 참조한다.

```
package nested.inner.ex2;

public class CarMain {

    public static void main(String[] args) {
        Car myCar = new Car("Model Y", 100);
        myCar.start();
    }
}
```

실행 결과

```
충전 레벨 확인: 100
Model Y의 엔진을 구동합니다.
Model Y 시작 완료
```

리팩토링 전의 문제

- `Car` 클래스는 엔진에 필요한 메서드들을 제공해야 한다. 다음 메서드는 엔진에서만 사용하고, 다른 곳에서는 사용하지 않는다.
  - `getModel()`
  - `getChargeLevel()`
- 결과적으로 엔진에서만 사용하는 기능을 위해 메서드를 추가해서, 모델 이름과 충전 레벨을 외부에 노출해야 한다.

리팩토링 전에는 결과적으로 모델 이름과 충전 레벨을 외부에 노출했다. 이것은 불필요한 `Car` 클래스의 정보들이 추가

로 외부에 노출되는 것이기 때문에 캡슐화를 떨어뜨린다.
리팩토링 후에는 `getModel()`, `getChargeLevel()`과 같은 메서드를 모두 제거했다. 결과적으로 꼭 필요한 메서드만 외부에 노출함으로써 `Car`의 캡슐화를 더 높일 수 있었다.

이제 처음에 설명한 중첩 클래스를 언제 사용해야 하는지 설명한 내용을 다시 정리해보자.

중첩 클래스는 언제 사용해야 하나?

- 중첩 클래스는 특정 클래스가 다른 하나의 클래스 안에서만 사용되거나, 둘이 아주 긴밀하게 연결되어 있는 특별한 경우에만 사용해야 한다. 외부 여러곳에서 특정 클래스를 사용한다면 중첩 클래스로 사용하면 안된다.

중첩 클래스를 사용하는 이유

- **논리적 그룹화**: 특정 클래스가 다른 하나의 클래스 안에서만 사용되는 경우 해당 클래스 안에 포함하는 것이 논리적으로 더 그룹화가 된다. 패키지를 열었을 때 다른 곳에서 사용될 필요가 없는 중첩 클래스가 외부에 노출되지 않는 장점도 있다.
- **캡슐화**: 중첩 클래스는 바깥 클래스의 `private` 멤버에 접근할 수 있다. 이렇게 해서 둘을 긴밀하게 연결하고 불필요한 `public` 메서드를 제거할 수 있다.

같은 이름의 바깥 변수 접근

바깥 클래스의 인스턴스 변수 이름과 내부 클래스의 인스턴스 변수 이름이 같으면 어떻게 될까?
다음 예제 코드를 보자.

```
package nested;

public class ShadowingMain {

    public int value = 1;

    class Inner {
        public int value = 2;

        void go() {
            int value = 3;
            System.out.println("value = " + value);
            System.out.println("this.value = " + this.value);
            System.out.println("ShadowingMain.value = " +
```

```
ShadowingMain.this.value);
        }
    }

    public static void main(String[] args) {
        ShadowingMain main = new ShadowingMain();
        Inner inner = main.new Inner();
        inner.go();
    }
}
```

실행 결과

```
value = 3
this.value = 2
ShadowingMain.value = 1
```

변수의 이름이 같기 때문에 어떤 변수를 먼저 사용할지 우선순위가 필요하다.
프로그래밍에서 우선순위는 대부분 더 가깝거나, 더 구체적인 것이 우선권을 가진다. 쉽게 이야기해서 사람이 직관적으로 이해하기 쉬운 방향으로 우선순위를 설계한다.

메서드 `go()`의 경우 지역 변수인 `value`가 가장 가깝다. 따라서 우선순위가 가장 높다.
이렇게 다른 변수들을 가려서 보이지 않게 하는 것을 섀도잉(Shadowing)이라 한다.

다른 변수를 가리더라도 인스턴스의 참조를 사용하면 외부 변수에 접근할 수 있다.
`this.value`는 내부 클래스의 인스턴스에 접근하고, `바깥클래스이름.this`는 바깥 클래스의 인스턴스에 접근할 수 있다.

프로그래밍에서 가장 중요한 것은 명확성이다. 이렇게 이름이 같은 경우 처음부터 이름을 서로 다르게 지어서 명확하게 구분하는 것이 더 나은 방법이다.