



It's Your Life

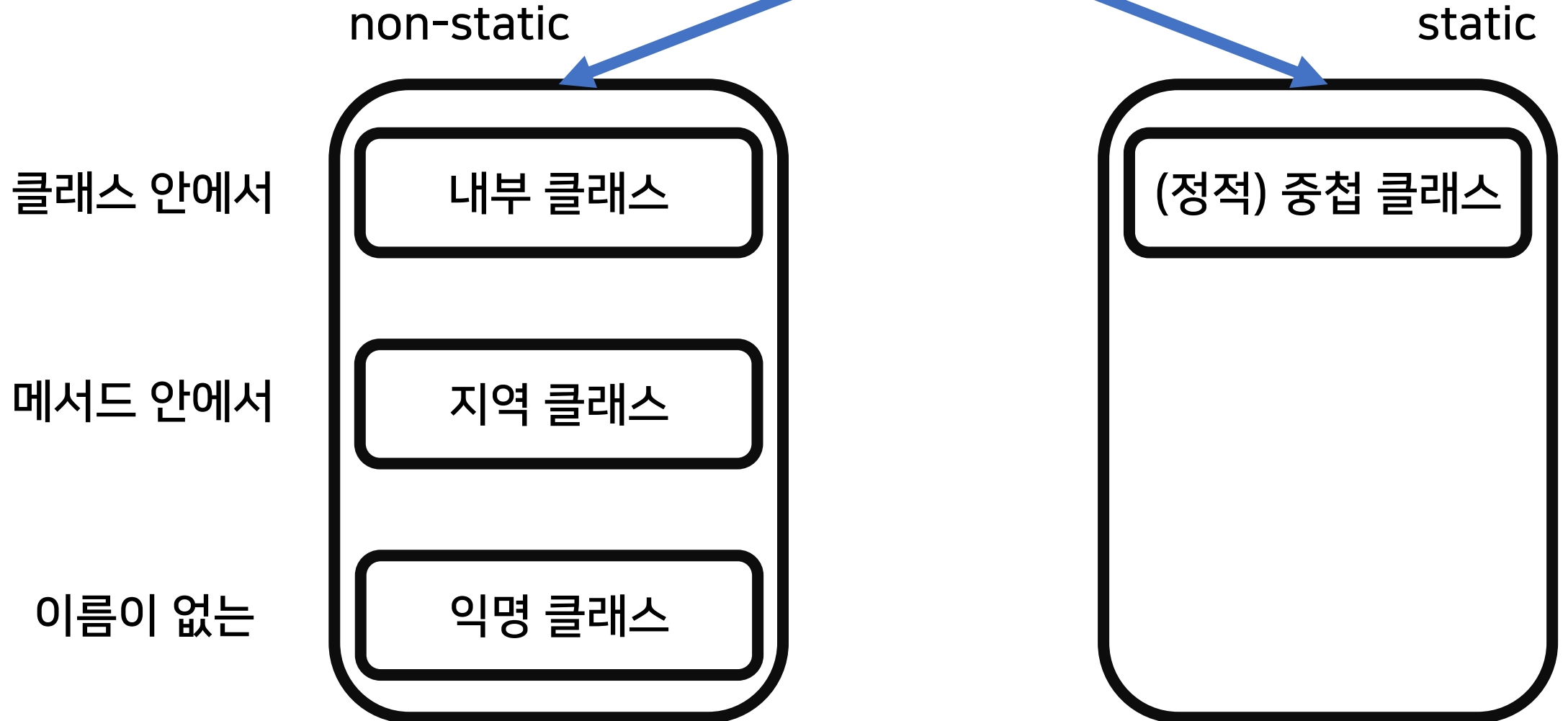
with





# 중첩 클래스

# 중첩 클래스의 종류





또 술마실 이유를  
찾아내셨군요 정말  
말 끝이 없  
습니다



1일차



2일차





그래서 그거

왜 쓰나요?



**클래스를 외부에 꺼낼 필요가 없을 때 씁니다!**

**그럼 그럴 때는 언제 일까요?**











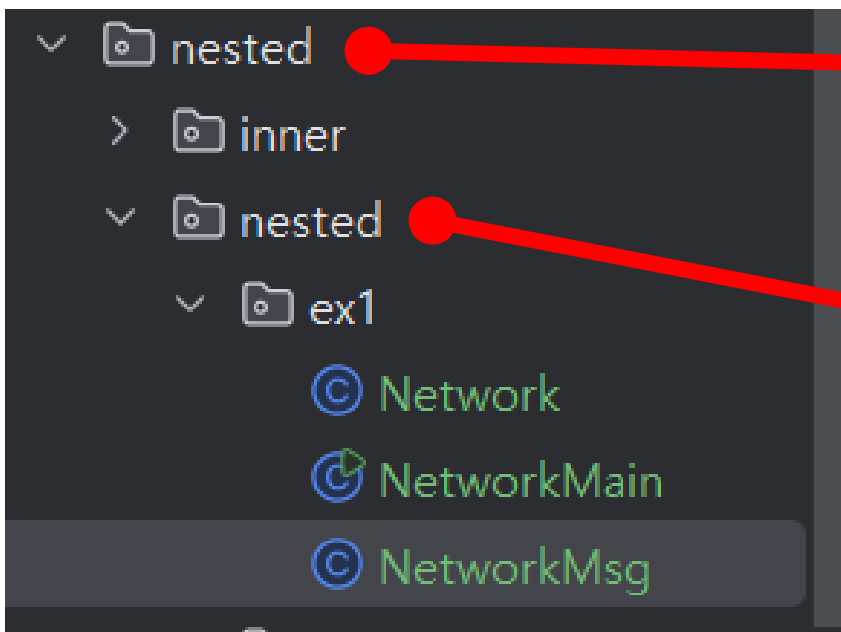


**즉 특정 클래스가 특정 클래스 내부에서만  
의미를 가질 때 쓰면 좋습니다!**



# 간단한(?) 예제





중첩 클래스를 위한  
nested 패키지 생성

(정적) 중첩 클래스를 위한  
nested 패키지 생성

```
public class NetworkMsg { 2 usages new *  
    private String msg; 2 usages  
  
    public NetworkMsg(String msg) { 1 usage new *  
        this.msg = msg;  
    }  
  
    public void send() { 1 usage new *  
        System.out.println("네트워크 메시지를 전송합니다.");  
        System.out.println(msg);  
        System.out.println("네트워크 메시지를 전송 종료.");  
    }  
}
```

생성 시, msg 를 받아서  
전송하는 간단한 클래스

단, 네트워크를 통해서만  
메시지가 전달이 가능한  
상황이라고 가정해 봅시다!





```
public class Network {  
    private boolean networkCondition; 3 usages  
  
    public boolean checkNetwork() { 1 usage new *  
        System.out.println("네트워크 상태를 점검합니다");  
        Random rand = new Random();  
        networkCondition = rand.nextBoolean();  
        if(networkCondition) {  
            System.out.println("네트워크 상태 정상");  
        } else {  
            System.out.println("네트워크 상태 이상");  
            System.out.println("네트워크 종료");  
        }  
        return networkCondition;  
    }  
}
```

실제 네트워크 역할을 하는  
클래스!

네트워크 답게(?)  
다양한 기능이 있습니다!?

랜덤 클래스를 이용해서  
네트워크 상태를 점검(?)하고  
점검 결과를 리턴하는 메서드!

```
public void sendMsg(String msg) { usage new *  
    NetworkMsg networkMsg = new NetworkMsg(msg);  
    networkMsg.send();  
}
```

실제로 메시지를 전달하는 메서드  
대신, 전달 역할은 NetworkMsg  
클래스가 전담 합니다!

like a 다형성(?) ㅎㅎㅎ  
like a 실제 서비스(?)



```
public class NetworkMain { new *  
    public static void main(String[] args) { new *  
        Network network = new Network();  
        if(network.checkNetwork()) network.sendMsg("내일도 비오려나!?");  
    }  
}
```

네트워크 상태를 점검합니다  
네트워크 상태 정상  
네트워크 메시지를 전송합니다.  
내일도 비오려나!?  
네트워크 메시지를 전송 종료.

네트워크 상태를 점검합니다  
네트워크 상태 이상  
네트워크 종료

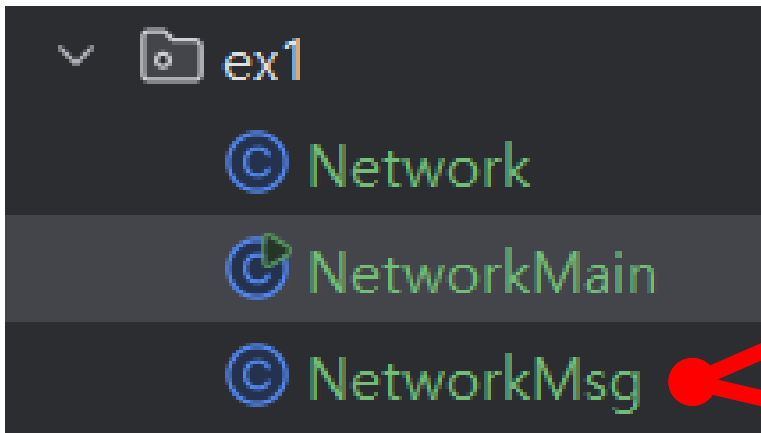
네트워크 상태를 체크하고  
상태가 정상이면 메시지를  
전송하는 운영 클라(?)쓰!

그것이 알고싶다

MB

아리아나 그란데말입니다





요, NetworkMsg 클래스는  
Network 클래스가 없으면  
사용이 가능한 클래스 인가요?

굳이 개발자 헛갈리게  
밖에 나둘 필요가 있을까요?

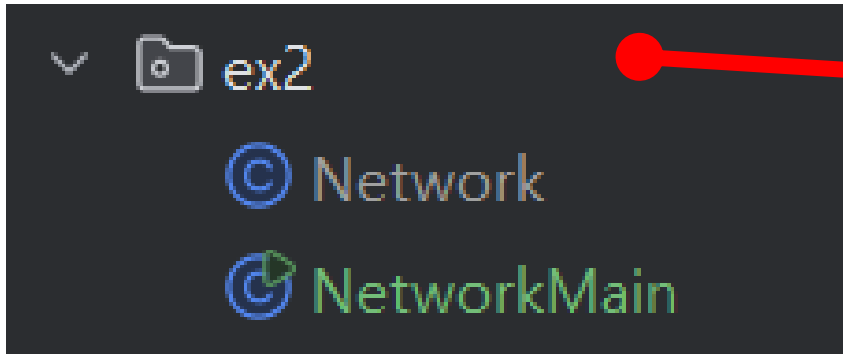
그리고 괜히 누가 저거  
잘 못 사용해서 문제 생기면  
누가 책임 지나요?





# 그럼 집어 넣읍시다!





ex1 패키지를 만들고  
저 2개의 클래스만 복붙 합시다!

```
public void sendMsg(String msg) { 1 usage new *  
    NetworkMsg networkMsg = new NetworkMsg(msg);  
    networkMsg.send();  
}
```

그리고 NetworkMsg 클래스는  
Network 클래스 내부에  
private, static 으로 넣어 버립시다!

```
private static class NetworkMsg { 2 usages new *  
    private String msg; 2 usages  
  
    public NetworkMsg(String msg) { 1 usage new *  
        this.msg = msg;  
    }  
  
    public void send() { 1 usage new *  
        System.out.println("네트워크 메시지를 전송합니다.");  
        System.out.println(msg);  
        System.out.println("네트워크 메시지를 전송 종료.");  
    }  
}
```

그럼 어떤 좋은 점이 있을까요?

```
public void sendMsg(String msg) { 1 usage new *
    NetworkMsg networkMsg = new NetworkMsg(msg);
    networkMsg.send();
}

private static class NetworkMsg { 2 usages new *
    private String msg; 2 usages

    public NetworkMsg(String msg) { 1 usage new *
        this.msg = msg;
    }

    public void send() { 1 usage new *
        System.out.println("네트워크 메시지를 전송합니다.");
        System.out.println(msg);
        System.out.println("네트워크 메시지를 전송 종료.");
    }
}
```

그럼 어떤 좋은 점이 있을까요?

일단 private 이기 때문에  
외부에서 접근이 불가능하여  
더 좋은 캡슐화를 유지할 수 있다!

클래스가 외부에 노출이 안되므로  
협업하는 개발자들이  
덜 헛갈릴 수 있다!



```
public void sendMsg(String msg) { 1 usage new *  
    NetworkMsg networkMsg = new NetworkMsg(msg);  
    networkMsg.send();  
}
```

```
private class NetworkMsg { 1 usages new *  
    private String msg;  
  
    public NetworkMsg(String msg) {  
        this.msg = msg;  
    }  
}
```

Inner class 'NetworkMsg' may be 'static'

Make 'static' Alt+Shift+Enter More actions... Alt+Enter

📁 nested.nested.ex2

private class Network.NetworkMsg

📁 kb-java-lecture



내부 클래스는 반드시  
static 일 필요는 없습니다

그런데 인텔리제이가 static 을 추천하네요!  
왜 일까요?

```
public void send() { 1 usage new *  
    System.out.println("네트워크 메시지를 전송합니다.");  
    System.out.println(msg);  
    System.out.println("네트워크 메시지를 전송 종료.");  
}
```





**이 녀석...  
천재다!!**



지금까지 그 이유를 알아보았습니다! ㅋㅋㅋㅋㅋㅋㅋㅋ





# 중첩 클래스와 내부 클래스의 차이



# 중첩 클래스의

# 특징



왜?  
static 과 non-static 으로 나눌까요!?



## 메서드 영역

클래스 정보

Static 영역

상수 영역

## 스택 영역

method1 ()

method2 ()

## 힙 영역

x001

인스턴스  
name = "이효석"  
method1()





## 메서드 영역

클래스 정보

Static 영역

상수 영역





```
▼ nested
  > ex1
  > ex2
    © Outer
    © OuterMain
```

중첩 클래스 특징 확인을 위한  
Outer, OuterMain 클래스 작성

```
public class Outer { 4 usages  Tetz *
    private static String outerStatic = "outerStatic"; 1 usage
    private String outerInstance = "outerInstance"; 1 usage

    static class Nested { 1 usage  Tetz *
        private static String nestedStatic = "innerStatic"; 1 usage
        private String nestedInstance = "innerInstance"; 1 usage

        public void print() { new *
            // 클래스 내부의 static 값에 접근
            System.out.println("innerStatic = " + nestedStatic);
            System.out.println("outerStatic = " + outerStatic);

            // 클래스 내부의 non-static 값에 접근
            System.out.println("innerInstance = " + nestedInstance);
            System.out.println("outerInstance = " + outerInstance);
        }
    }
}
```

외부 클래스에 static 과  
instance 멤버를 선언

중첩 클래스 선언을 위해  
static 클래스로 Nested 선언



```
public class Outer { 4 usages  Tetz *
    private static String outerStatic = "outerStatic"; 1 usage
    private String outerInstance = "outerInstance"; 1 usage

    static class Nested { 2 usages  Tetz *
        private static String nestedStatic = "innerStatic"; 1 usage
        private String nestedInstance = "innerInstance"; 1 usage

        public void print() { new
            // 클래스 내부의 static 값에 접근
            System.out.println("innerStatic = " + nestedStatic);
            System.out.println("outerStatic = " + outerStatic);

            // 클래스 내부의 non-static 값에 접근
            System.out.println("innerInstance = " + nestedInstance);
            System.out.println("outerInstance = " + outerInstance);
        }
    }
}
```

중첩 클래스에 static 과  
instance 멤버 선언

중첩 클래스 내부에  
static print() 메서드로  
각각의 값에 접근이  
가능한지 확인



```
public class Outer { 4 usages  🧑 Tetz *
    private static String outerStatic = "outerStatic"; 1 usage
    private String outerInstance = "outerInstance"; 1 usage

    static class Nested { 2 usages  🧑 Tetz *
        private static String nestedStatic = "innerStatic"; 1 usage
        private String nestedInstance = "innerInstance"; 1 usage

        public void print() { new *
            // 클래스 내부의 static 값에 접근
            System.out.println("innerStatic = " + nestedStatic);
            System.out.println("outerStatic = " + outerStatic);

            // 클래스 내부의 non-static 값에 접근
            System.out.println("innerInstance = " + nestedInstance);
            System.out.println("outerInstance = " + outerInstance);
        }
    }
}
```

일단 private 멤버에  
접근이 가능합니다!

따라서,  
특정 용도로 사용이 가능



```
public class Outer { 4 usages  🧑 Tetz *
    private static String outerStatic = "outerStatic"; 1 usage
    private String outerInstance = "outerInstance"; 1 usage

    static class Nested { 2 usages  🧑 Tetz *
        private static String nestedStatic = "innerStatic"; 1 usage
        private String nestedInstance = "innerInstance"; 1 usage

        public void print() { new *
            // 클래스 내부의 static 값에 접근
            System.out.println("innerStatic = " + nestedStatic);
            System.out.println("outerStatic = " + outerStatic);

            // 클래스 내부의 non-static 값에 접근
            System.out.println("innerInstance = " + nestedInstance);
            System.out.println("outerInstance = " + outerInstance);
        }
    }
}
```

그럼 왜 Outer 클래스의  
인스턴스 멤버에는  
접근이 불가능 할까요?




일단 클래스를 전부  
인스턴스화 시켜놓고  
생각 합시다

잠

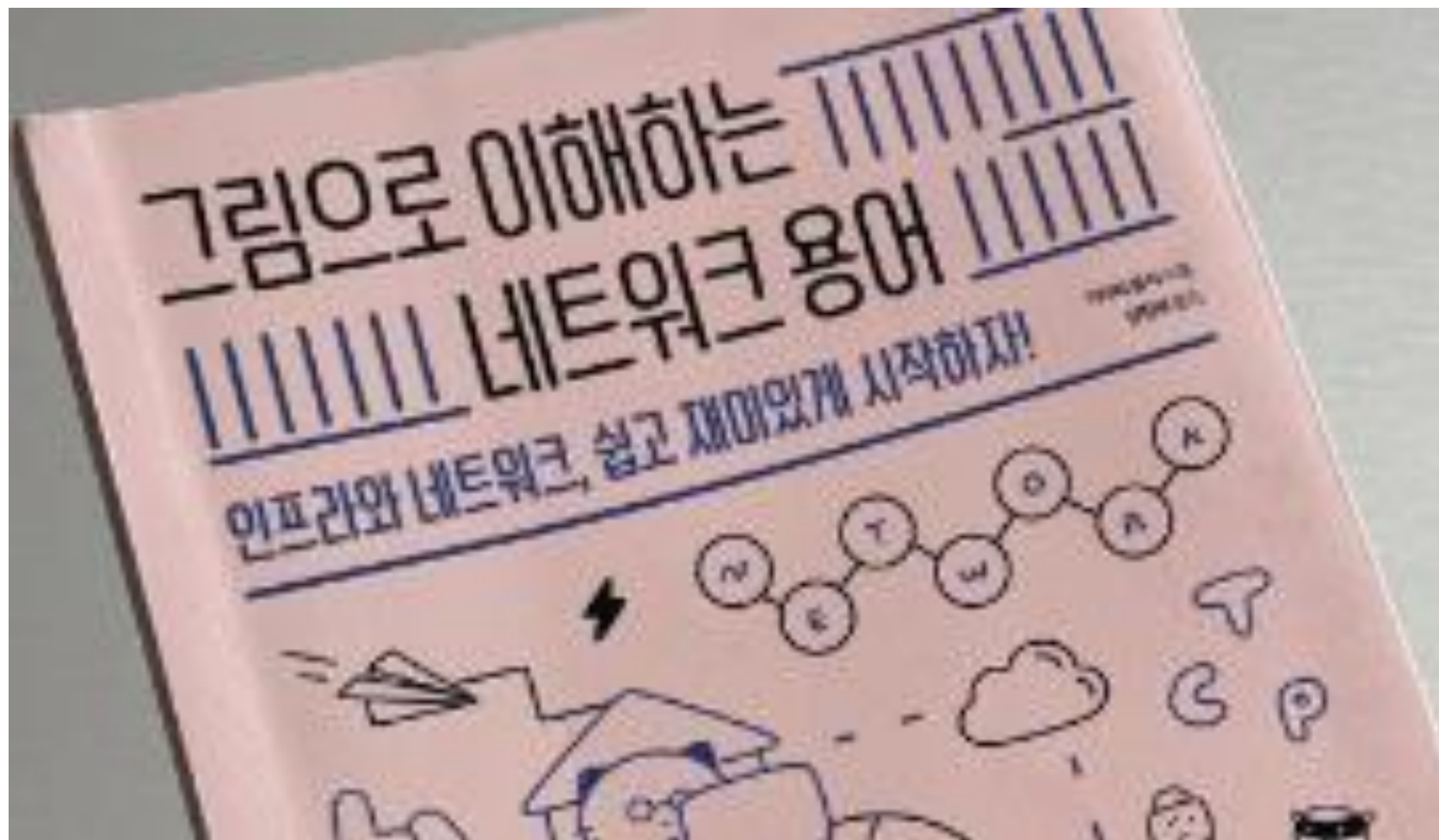
관



```
public class OuterMain {  👤 Tetz *  
    public static void main(String[] args) {  👤 Tetz *  
        Outer outer = new Outer();  
        Outer.Nested nested = new Outer.Nested();  
    }  
}
```



일단 중첩 클래스를  
인스턴스화 하는 방법은  
요렇게 하시면 됩니다!



# 메서드 영역 (공용 영역)

Outer



```
static outerStatic;
```

Nested

```
static nestedStatic;
```

# 힙 영역 (공용 X)



Outer@x001

```
outerInstance;
```

Nested@x002

```
print();  
nestedInstance;
```





```
public void print() { new *  
    // 클래스 내부의 static 값에 접근  
    System.out.println("innerStatic = " + nestedStatic);  
    System.out.println("outerStatic = " + outerStatic);  
  
    // 클래스 내부의 non-static 값에 접근  
    System.out.println("innerInstance = " + nestedInstance);  
    System.out.println("outerInstance = " + outerInstance);  
}
```

자기 인스턴스 내부의 멤버는  
스스로 알기 때문에  
문제가 없습니다!

## 메서드 영역 (공용 영역)

Outer

```
static outerStatic;
```

Nested

```
static nestedStatic;
```

## 힙 영역 (공용 X)



Outer@x001

```
outerInstance;
```

Nested@x002

```
print();  
nestedInstance;
```





```
public void print() { new *  
    // 클래스 내부의 static 값에 접근  
    System.out.println("innerStatic = " + nestedStatic);  
    System.out.println("outerStatic = " + outerStatic);  
  
    // 클래스 내부의 non-static 값에 접근  
    System.out.println("innerInstance = " + nestedInstance);  
    System.out.println("outerInstance = " + outerInstance);  
}
```

outer 인스턴스는  
nested 인스턴스와는  
별개의 인스턴스이므로  
알 수 없습니다!



# 중첩 클래스의

# 외부에선?

```
public class OuterMain {  👤 Tetz *  
    public static void main(String[] args) {  👤 Tetz *  
        Outer outer = new Outer();  
        Outer.Nested nested = new Outer.Nested();  
  
        System.out.println(outer.);  
    }  
}
```

```
④ equals(Object obj)  
④ toString() String  
④ hashCode() int  
④ getClass() Class<? extends Outer>  
📎 arg functionCall(expr)  
④ notify() void
```

OuterMain 은  
외부 클래스이므로  
Outer 클래스 내부의  
private 멤버에  
접근이 불가능 합니다!







하지만 중첩 클래스인 Nested 안에 존재하는 print() 메서드는요?

하지만 드라군이  
출동하면 어떨까?



```
nested.print();
```

```
public void print() { new *  
    // 클래스 내부의 static 값에 접근  
    System.out.println("innerStatic = " + nestedStatic);  
    System.out.println("outerStatic = " + outerStatic);  
  
    // 클래스 내부의 non-static 값에 접근  
    System.out.println("innerInstance = " + nestedInstance);  
    // System.out.println("outerInstance = " + outerInstance);  
}
```

```
public class OuterMain {  🧑 Tetz *  
|   public static void main(String[] args) {  🧑 Tetz *  
    Outer outer = new Outer();  
    Outer.Nested nested = new Outer.Nested();  
  
    nested.print();  
  }  
}
```

같은 클래스 내부에 존재하므로  
private 멤버에 접근이 가능!

```
innerStatic = innerStatic  
outerStatic = outerStatic  
innerInstance = innerInstance
```



그래서

중첩 클래스는!?



**특정 클래스 내부에서만 사용이 되어  
외부 노출이 굳이 필요 없을 때!**

**특정 클래스 내부의 private 멤버에 대한  
접근이 필요할 때!**

# 실습, 정적 중첩 클래스 만들기 및 사용하기



- nested 패키지에 ex3 패키지를 만들어 주세요
- ex3 패키지에 OuterClass1 를 만들어 주시고, 해당 클래스 내부에 NestedClass 를 정적 중첩 클래스로 만들어 주세요
- 해당 NestedClass 중첩 클래스는 public void 타입으로 hello 메서드를 가지고 있으며, 실행 시 "안녕하세요. 중첩 클래스의 hello 입니다" 를 출력합니다.
- OuterClass1Main 운영 클래스를 만들어서 정적 중첩 클래스만 인스턴스화 시킨 뒤, hello 메서드를 실행하여 주세요.



# 내부 클래스의

# 특징



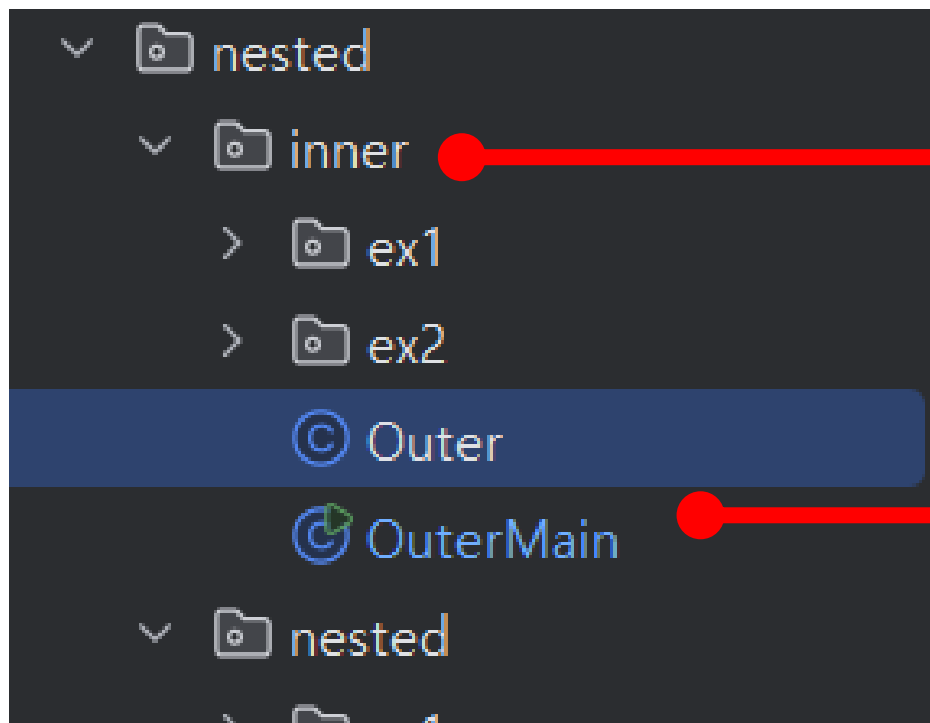


**내부 클래스는 중첩 클래스에서  
static 만 빼면 됩니다!**



그럼 저거 하나 뺀다고 어떤 차이점이 생길까요?





내부 클래스의 특징을 알아보기 위해  
inner 패키지 만들기!

nested 패키지의  
클래스 그대로 복붙하기!



```
public class Outer { 4 usages  Tetz *  
    private static String outerStatic = "outerStatic"; 1 usage  
    private String outerInstance = "outerInstance"; no usages  
      
    class Inner { 3 usages  Tetz *  
        private static String nestedStatic = "innerStatic"; 1 usage  
        private String nestedInstance = "innerInstance"; 1 usage
```

명칭상 내부 클래스 이므로  
Inner 로 이름 변경

내부 클래스로 만들기 위해  
static 삭제



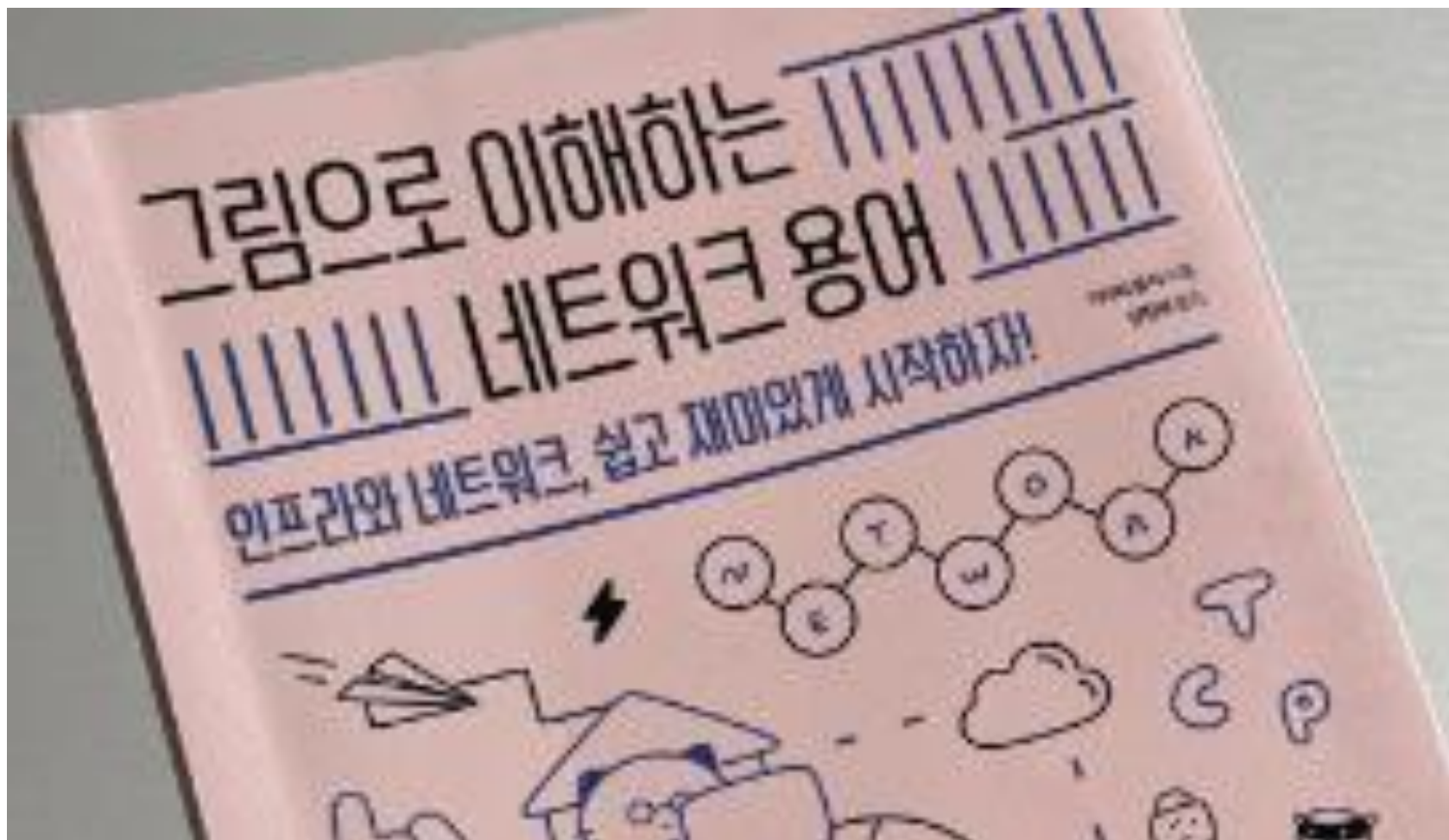


```
class Inner { no usages  Tetz *  
    private static String innerStatic = "innerStatic"; 1 usage  
    private String innerInstance = "innerInstance"; 1 usage  
  
    public void print() { Tetz *  
        // 클래스 내부의 static 값에 접근  
        System.out.println("innerStatic = " + innerStatic);  
        System.out.println("outerStatic = " + outerStatic);  
  
        // 클래스 내부의 non-static 값에 접근  
        System.out.println("innerInstance = " + innerInstance);  
        System.out.println("outerInstance = " + outerInstance);  
    }  
}
```

아까는 접근이 불가능했던  
outerInstance 에  
접근이 가능합니다!!!!







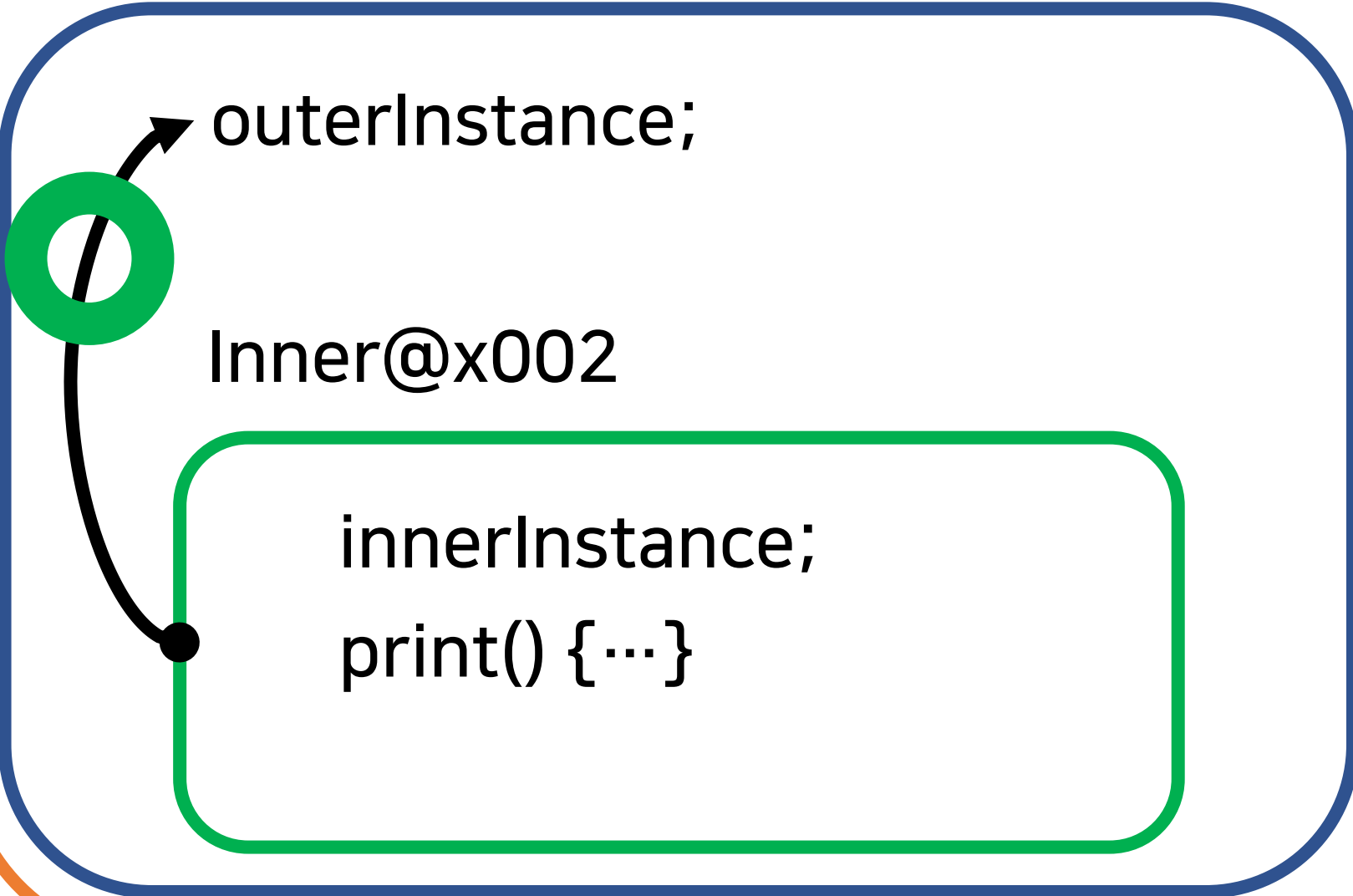


Outer@x001

outerInstance;

Inner@x002

innerInstance;  
print() {...}






# Instagram

 **shu.ga**  
samui island



# Instagram

 **shu.ga**  
samui island







Outer@x001

outerInstance;

Inner@x002

- 외부 클래스의 참조값을 보관(x001)

innerInstance;

print() {...}



아 갈매기짤 겨우찾았네 조현우랑 너무  
닮아서 오기로찾음



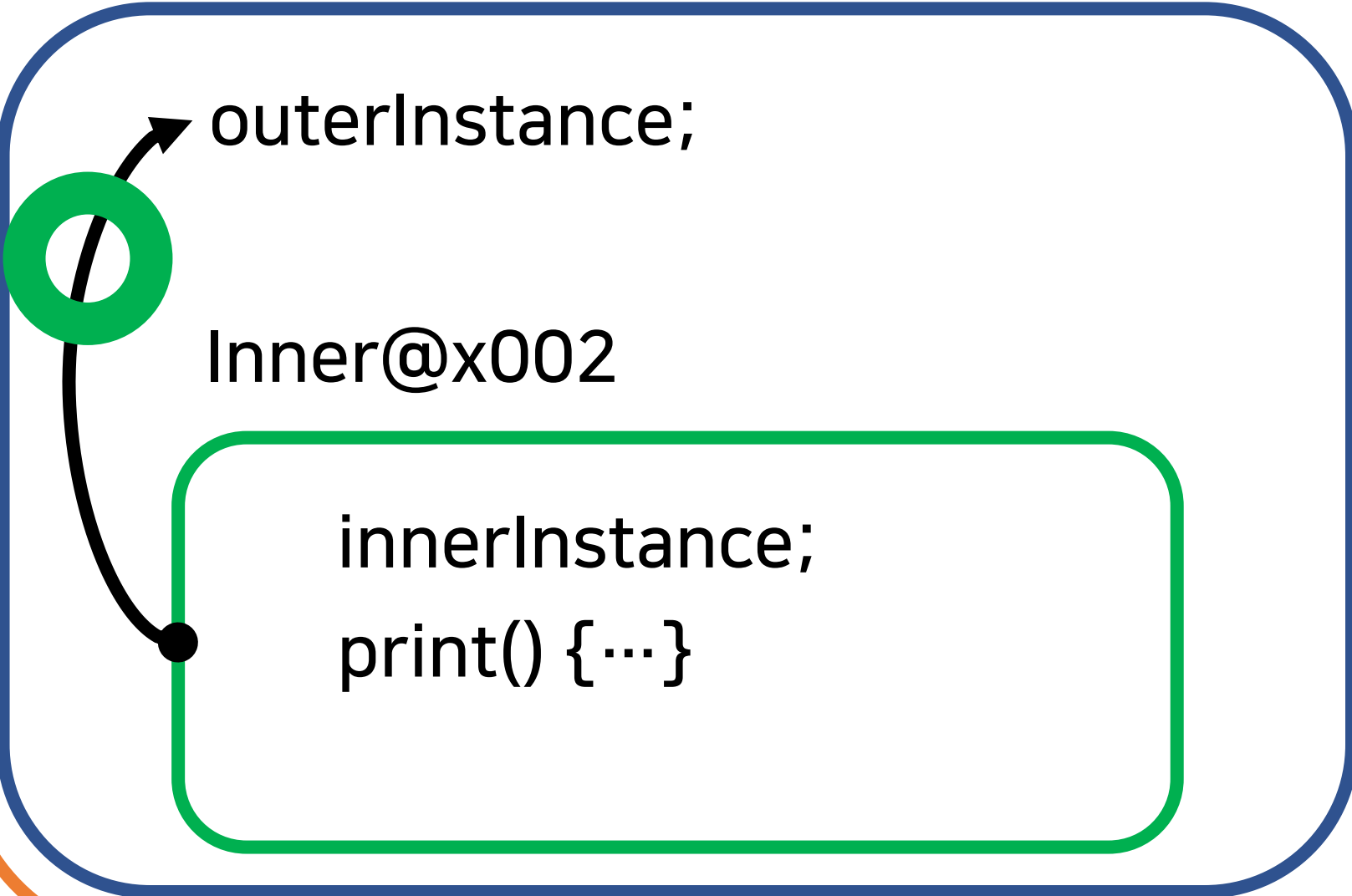


Outer@x001

outerInstance;

Inner@x002

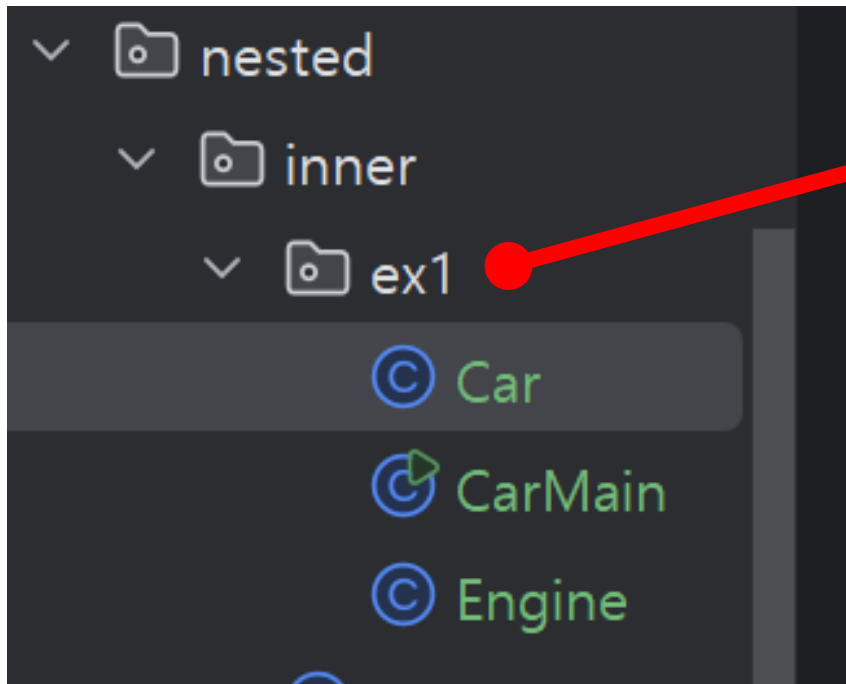
innerInstance;  
print() {...}





# 내부 클래스

# 사용 예제



예제 작성을 위한 ex1 패키지 생성

```
public class Car { new *  
    private String model; 3 usages  
    private int oilAmount; 2 usages  
    private Engine engine; 2 usages
```

```
    public Car(String model, int oilAmount) { new *  
        this.model = model;  
        this.oilAmount = oilAmount;  
        this.engine = new Engine(car: this);  
    }
```





```
public String getModel() { 1 usage new *  
    return model;  
}
```


```
public int getOilAmount() { 1 usage new *  
    return oilAmount;  
}
```

```
public void start() { 1 usage new *  
    engine.start();  
    System.out.println(model + "의 주행을 시작합니다!");  
}
```

```
public class Engine { 2 usages    new *
    private Car car; 1 usages

    public Engine(Car car) { 1 usage    new *
        this.car = car;
    }

    public void start() { no usages    new *
        System.out.println("자동차 주유 상태 확인 : " + car.getOilAmount());
        System.out.println(car.getModel() + "의 엔진을 구동합니다");
    }
}
```



원래는 Car 와 Engine 은 서로  
다른 클래스이기 때문에  
인스턴스화를 해도 서로  
다른 인스턴스로 만들어 집니다!

따라서, Engine 은 Car 라는  
인스턴스가 어디 있는지  
모르기 때문에  
Car 의 인스턴스를 제공 받아서  
직접 연결 해야만 했습니다!



# 힙 영역



Car@x001 ●

model;  
oilAmount;  
engine;

Engine@x002

Car; ←  
start();



```
public class CarMain {  
    public static void main(String[] args) {  
        Car myCar = new Car(model: "미니쿠퍼", oilAmount: 10);  
        myCar.start();  
    }  
}
```

자동차 주유 상태 확인 : 10  
미니쿠퍼의 엔진을 구동합니다  
미니쿠퍼의 주행을 시작합니다!

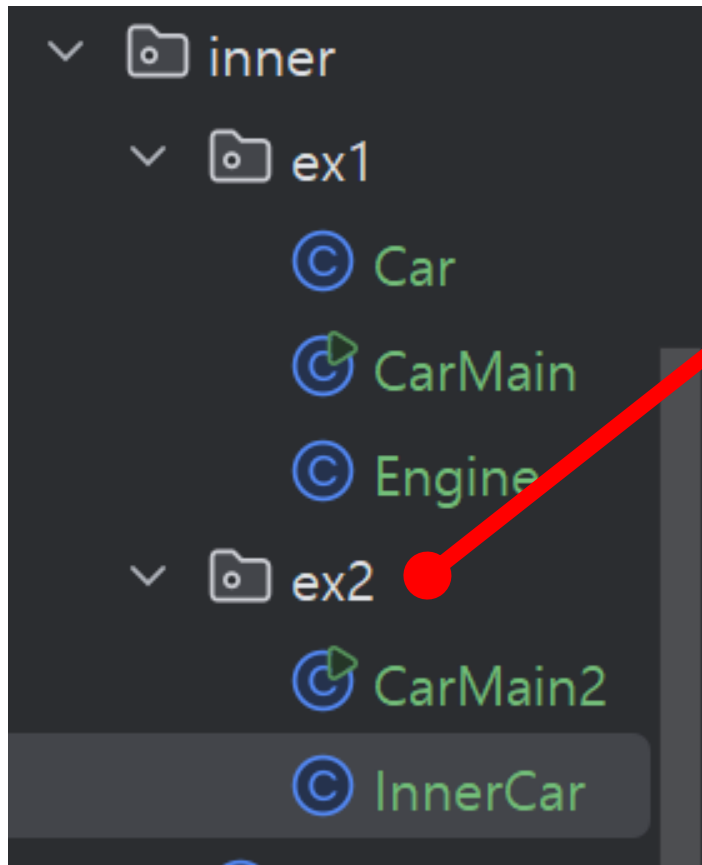


# 내부 클래스로

# refactoring



내부 클래스를 사용하는 코드로  
refactoring 하기 위해 ex2 패키지 생성



```
public class InnerCar { 3 usages    new *
    private String model; 4 usages
    private int oilAmount; 3 usages
    private Engine engine; 2 usages

    public InnerCar(String model, int oilAmount) { 1 use
        this.model = model;
        this.oilAmount = oilAmount;
        this.engine = new Engine();
    }

    public String getModel() { return model; }

    public int getOilAmount() { return oilAmount; }

    public void start() { 1 usage    new *
        engine.start();
        System.out.println(model + "의 주행을 시작합니다!");
    }
```

기존 Car 클래스의 내용을 그대로 가져오기

단, 구분을 위해 클래스명만 변경!



```
public void start() { 1 usage new *  
    engine.start();  
    System.out.println(model + "의 주행을 시작합니다!");  
}
```

```
private class Engine { 2 usages new *  
    public void start() { 1 usage new *  
        System.out.println("자동차 주유 상태 확인 : " + oilAmount);  
        System.out.println(model + "의 엔진을 구동합니다");  
    }  
}
```

Car 클래스에서만 사용되는  
Engine 클래스이므로  
내부 클래스 + private 로  
포함 시키기

```
public class Engine { 2 usages n
    private Car car; 3 usages

    public Engine(Car car) { 1 us
        this.car = car;
    }
}
```

Car@x001

model;  
oilAmount;  
engine;

Engine@x002

Car;  
start();

A diagram within a black rectangular frame. At the top, the text 'Car@x001' is followed by a black dot. A thick black curved arrow originates from this dot and points to the 'Car;' line in a lower orange box. The lower orange box is preceded by the text 'Engine@x002'. Both orange boxes contain text representing object state or code snippets.

힙 영역



```
public void start() { 1 usage new *  
    engine.start();  
    System.out.println(model + "의 주행을 시작합니다!");  
}
```

```
private class Engine { 2 usages new *  
    public void start() { 1 usage new *  
        System.out.println("자동차 주유 상태 확인 : " + oilAmount);  
        System.out.println(model + "의 엔진을 구동합니다");  
    }  
}
```

이제 자동차 정보는  
외부 클래스에 존재 하므로  
굳이 알려줄 필요가 없습니다!

따라서 Car 멤버와 생성자 제거



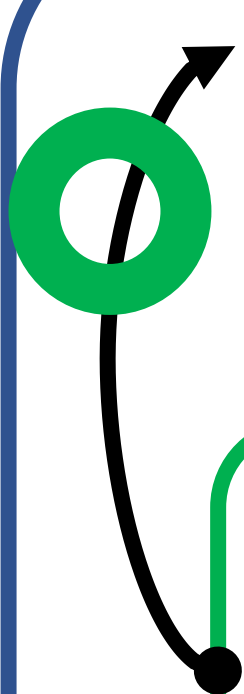


Car@x001

engine

Engine@x002

start() {...}





```
public void start() { 1 usage    new *  
    engine.start();  
    System.out.println(model + "의 주행을 시작합니다!");  
}  
  
private class Engine { 2 usages    new *  
    public void start() { 1 usage    new *  
        System.out.println("자동차 주유 상태 확인 : " + oilAmount);  
        System.out.println(model + "의 엔진을 구동합니다");  
    }  
}
```

oilAmount 와 model 도  
별도의 메서드로 받을 필요 없이  
외부 클래스에 바로 접근이 가능!

→ 따라서 바로 사용!



```
public class CarMain2 { new *  
    public static void main(String[] args) { new *  
        InnerCar myCar = new InnerCar(model: "미니쿠퍼", oilAmount: 10);  
        myCar.start();  
    }  
}
```

클래스만 내부 클래스가 적용된  
InnerCar 클래스로 변경!

자동차 주유 상태 확인 : 10  
미니쿠퍼의 엔진을 구동합니다  
미니쿠퍼의 주행을 시작합니다!



# 실습, 내부 클래스 만들기 및 사용하기



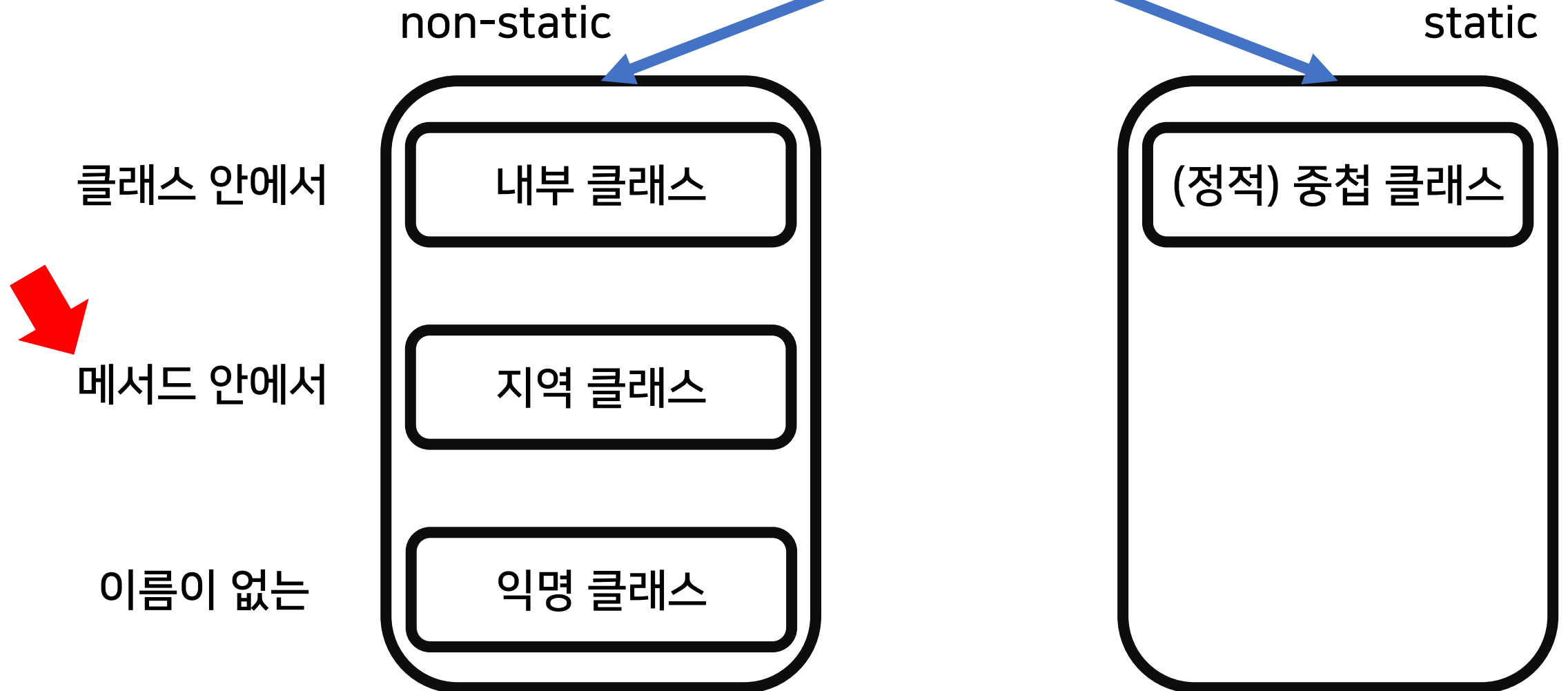
- inner 패키지에 ex3 패키지를 만들어 주세요
- ex3 패키지에 OuterClass2 를 만들어 주시고, 해당 클래스 내부에 InnerClass 를 내부 클래스로 만들어 주세요
- 해당 InnerClass 내부에는 public void 타입으로 hello 메서드를 가지고 있으며, 실행 시 “안녕하세요. 내부 클래스의 hello 입니다” 를 출력 합니다.
- OuterClass2Main 운영 클래스를 만들어서 InnerClass 를 인스턴스화 시킨 뒤, hello 메서드를 실행하여 주세요(여기가 어렵습니다!)



# 지역 클래스와

# 익명 클래스

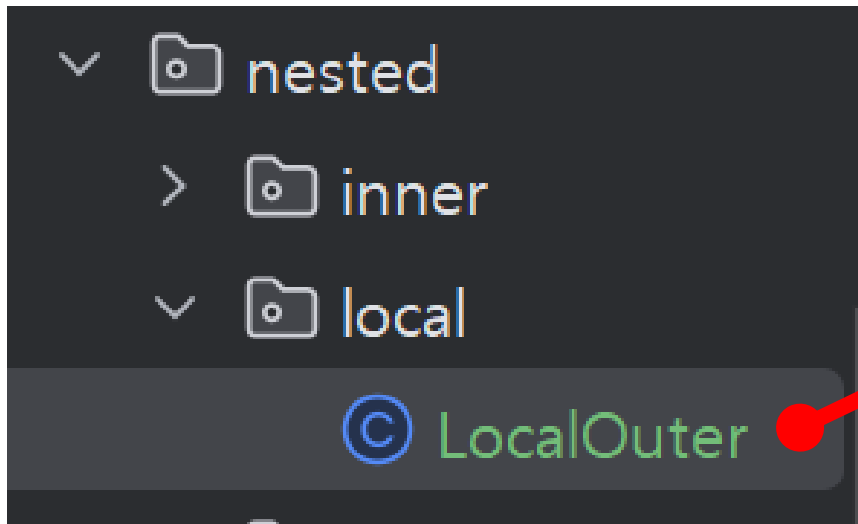
# 중첩 클래스의 종류





# 지역 클래스





지역 클래스 확인을 위한  
local 패키지와  
LocalOuter 클래스 생성

```
public class LocalOuter { new *
    private String outerInstance = "outerInstance"; 1 usage

    public void outerMethod(String methodParameter) { usage
        String methodString = "methodString"; // 지역 변수

        class LocalInner { usages new *
            String localInstance = "localInstance"; 1 usage

            public void printLocal() { 1 usage new *
                System.out.println("outerInstance = " + outerInstance);
                System.out.println("methodString = " + methodString);
                System.out.println("localInstance = " + localInstance);
                System.out.println("parameter = " + methodParameter);
            }
        }

        LocalInner localInner = new LocalInner();
        localInner.printLocal();
    }
}
```

지역 클래스는  
메소드 내부에 존재해야 하므로  
메소드 작성

메소드 내부에  
LocalInner 클래스  
작성

```
public class LocalOuter { new *
    private String outerInstance = "outerInstance"; 1 usage

    public void outerMethod(String methodParameter) { 1 usage
        String methodString = "methodString"; // 지역 변수

        class LocalInner { 2 usages new *
            String localInstance = "localInstance"; 1 usage

            public void printLocal() { usage new *
                System.out.println("outerInstance = " + outerInstance);
                System.out.println("methodString = " + methodString);
                System.out.println("localInstance = " + localInstance);
                System.out.println("parameter = " + methodParameter);
            }
        }

        LocalInner localInner = new LocalInner();
        localInner.printLocal();
    }
}
```

LocalInner 클래스는  
메소드 내부에 존재하는 만큼

메소드 내부에 존재하는  
지역 변수와 파라미터에도  
접근이 가능!

이를 통해서 특별한 상황에  
이용이 가능

메소드 내부에서 인스턴스화  
작업 후 printLocal 메서드 실행



```
public static void main(String[] args) { new *  
    LocalOuter localOuter = new LocalOuter();  
    localOuter.outerMethod(methodParameter: "parameter");  
}
```

LocalOuter 내부에  
바로 psvm 을 만들어서 테스트!

```
outerInstance = outerInstance  
methodString = methodString  
localInstance = localInstance  
parameter = parameter
```

기본적으로 내부 클래스의  
특성을 가지기 때문에  
외부와 내부의 인스턴스에 접근 가능



LocalOuter@x001

outerInstance

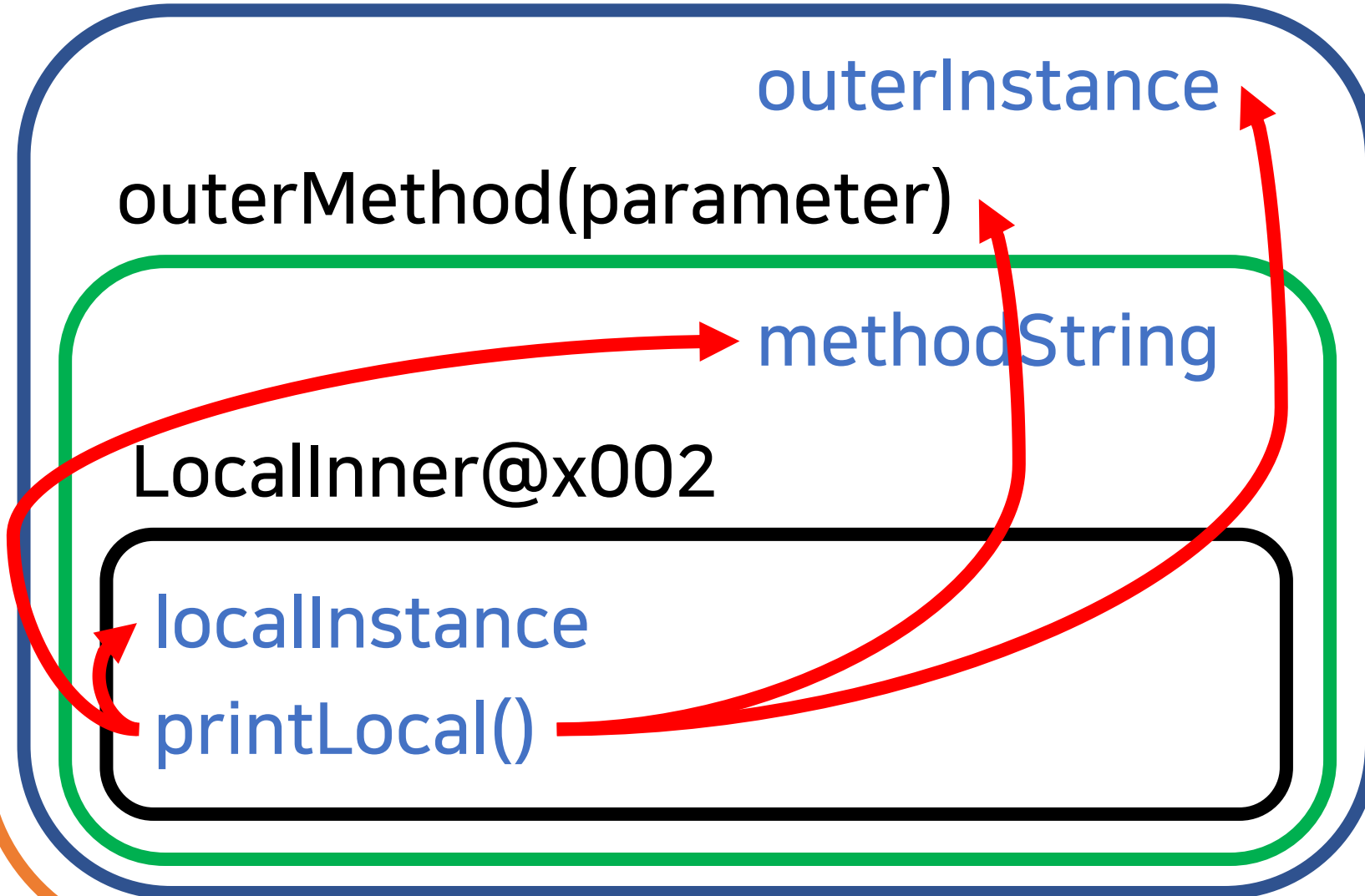
outerMethod(parameter)

methodString

LocalInner@x002

localInstance

printLocal()





근데 이제 뭐함 ?





# 지역 변수 캡처



# 힙 영역



LocalOuter@x001

outerInstance

outerMethod(parameter)

methodString

LocalInner@x002

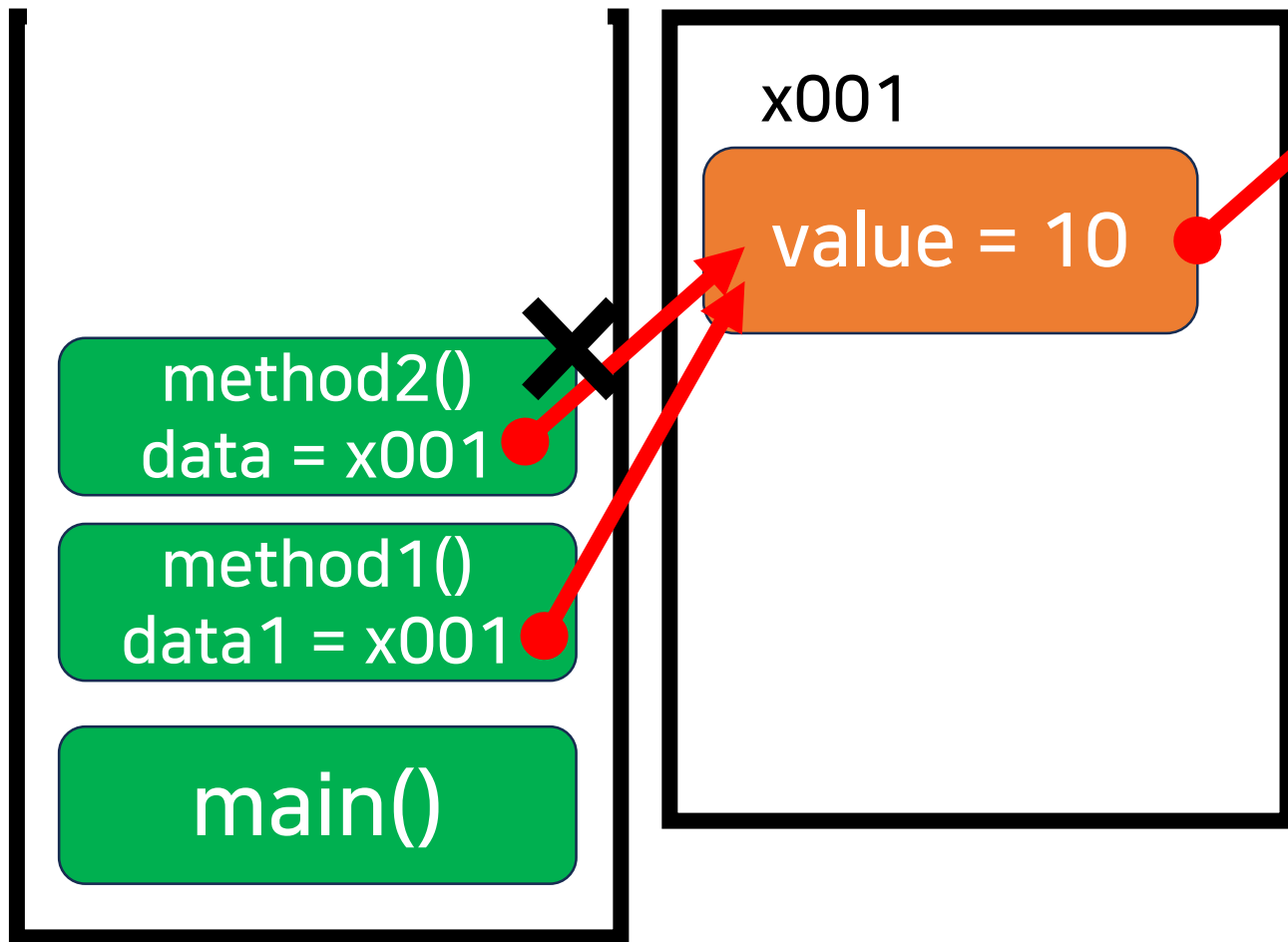
localInstance

printLocal()

```
outerInstance = outerInstance  
methodString = methodString  
localInstance = localInstance  
parameter = parameter
```

## 스택 영역

## 힙 영역



참조만 유지가 된다면  
스택 영역의 메서드보다  
힙의 인스턴스가  
더 오래 살아 남습니다

그래서 이것 이용해서  
메서드의 지역 변수와 매개 변수를  
저장(캡처)하는 형태로 사용이  
가능합니다!



안

스킵하면 네 피부는 변사체가 된다

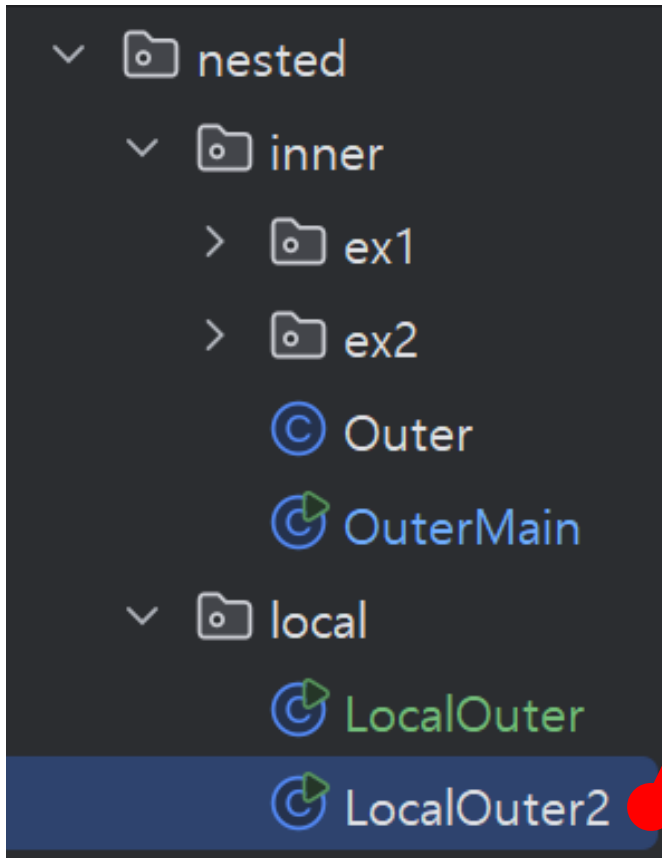
# 실습, 지역 클래스 만들기 및 사용하기



- local 패키지에 ex2 패키지를 만들어 주세요
- ex2 패키지에 OuterClass3 를 만들어 주시고, 해당 클래스 내부에 myMethod() 메서드를 만들어 주세요
- myMethod() 내부에 LocalClass 라는 지역 클래스를 만들어 주세요
- 해당 LocalClass 내부에는 public void 타입으로 hello 메서드를 가지고 있으며, 실행 시 “안녕하세요. 지역 클래스의 hello 입니다” 를 출력 합니다.
- OuterClass3Main 운영 클래스를 만들어서 OuterClass3 클래스를 인스턴스화 시킨 뒤, myMethod() 를 실행하면 LocalClass 의 hello 메서드가 실행 되도록 운영 클래스 코드를 완성해 주세요



# 익명 클래스



익명 클래스 확인을 위해서  
LocalOuter 클래스를  
복붙하여 LocalOuter2 를  
만들어 봅시다!



Outer

OuterMain

local

LocalOuter

LocalOuter2

nested

pack

poly

poly3

polyfinal

6

7

8

9

10

11

12

13

14

15

16

public void outerMethod(String methodParameter

New Java Class

I Print

C Class


I Interface

R Record

E Enum

@ Annotation

오랜만에 Print 라는 인터페이스를 만들어 봅시다!

```
public interface Print { 1  
     Rename usages  
    void printLocal(); new  
}
```

Print 인터페이스는  
간단하게 printLocal() 이라는  
추상 메서드만을 가집니다







그런데 말입니다

```
public void outerMethod(String  
    String methodString = "meth  
  
class LocalInner {  
    String localInstance =  
  
    public void printLocal()  
        System.out.println()  
        System.out.println()  
        System.out.println()  
        System.out.println()  
    }  
}
```

요 LocalInner 클래스가  
Print 인터페이스를  
구현 할 수 있나요!?



```
class LocalInner implements Print {  
    String localInstance = "localInstance";
```

Print 인터페이스를  
구현 시키기!

```
@Override  
public void printLocal() {
```

printLocal() 메서드는  
자동으로 오버라이딩  
처리

```
    System.out.println("outerInstance = " + outerInstance);  
    System.out.println("methodString = " + methodString);  
    System.out.println("localInstance = " + localInstance);  
    System.out.println("parameter = " + methodParameter);
```

```
}
```

```
}
```



```
public static void main(String[] args) { new *  
    LocalOuter2 localOuter = new LocalOuter2();  
    localOuter.outerMethod(methodParameter: "parameter");  
}  
}
```

```
outerInstance = outerInstance  
methodString = methodString  
localInstance = localInstance  
parameter = parameter
```





익명?????

클래스

이게 왜 익명 클래스죠!?



그란데 말입니다





익명 클래스를 위해  
LocalOuter2 를 복붙해서  
AnonymousOuter 로 리팩터

AnonymousOuter

LocalOuter

LocalOuter2

Print

```
public void outerMethod(String methodParameter) { 1 usage  
    String methodString = "methodString"; // 지역 변수
```

```
class LocalInner implements Print { 2 usages new *  
    String localInstance = "localInstance"; 1 usage
```

```
@Override 2 usages new *
```

```
public void printLocal() {
```

```
    System.out.println("outerInstance = " + outerInstance);
```

```
    System.out.println("methodString = " + methodString);
```

```
    System.out.println("localInstance = " + localInstance);
```

```
    System.out.println("parameter = " + methodParameter);
```

```
}
```

```
}
```

이 친구를 익명으로  
변경해야 하는데  
어떻게 하면 될까요?



```
Print print = new Print() {  
    String localInstance = "localInstance"; 1 usage
```

인터페이스를 인스턴스화 하고  
그 다음에 필요 코드를  
바로 구현하여 전달!

```
@Override 2 usages new *
```

```
public void printLocal() {
```

```
    System.out.println("outerInstance = " + outerInstance);
```

```
    System.out.println("methodString = " + methodString);
```

```
    System.out.println("localInstance = " + localInstance);
```

```
    System.out.println("parameter = " + methodParameter);
```

```
}
```

```
};
```

```
print.printLocal();
```

이름 없는 클래스의 인스턴스가  
print 변수에 저장 되었으므로  
바로 사용!

```
Print print = new Print() {  
    String localInstance = "localInstance"; 1 usage
```

이 코드 덩어리(클래스)는  
이름이 있나요!?

```
@Override 2 usages new *  
public void printLocal() {  
    System.out.println("outerInstance = " + outerInstance);  
    System.out.println("methodString = " + methodString);  
    System.out.println("localInstance = " + localInstance);  
    System.out.println("parameter = " + methodParameter);  
}  
};  
  
print.printLocal();
```

```
LocalInner localInner = new LocalInner();  
localInner.printLocal();
```

이전 코드는 Print 를 상속 받은  
LocalInner 라는 클래스 명이 존재

```
Print print = new Print()
```

변경 코드는 Print 인터페이스의  
이름과 참조값이 저장된 변수만 존재  
구현 된 코드 덩어리를 지칭하는  
클래스 명이 존재하지 않음

→ 익명 클래스라 부릅니다!

```
outerInstance = outerInstance  
methodString = methodString  
localInstance = localInstance  
parameter = parameter
```



# 익명 클래스의 특징



1. 익명 클래스는 이름이 없는 지역 클래스로 사용 된다
2. 이름 없이 생성이 되어야 하므로 반드시 부모 클래스(주로 인터페이스)를 상속 받아서 구현해야하는 강제성을 가진다
3. 일회성으로 이용되는 경우는 사용이 권장되지만, 여러 번 사용되면 기명 클래스로 선언하여 사용하는 방법이 좋다 (이유를 아시는 분?)

이제 뭐함?



그래서 이거 어따 쓰져!?

# 실습, 지역 클래스 만들기 및 사용하기

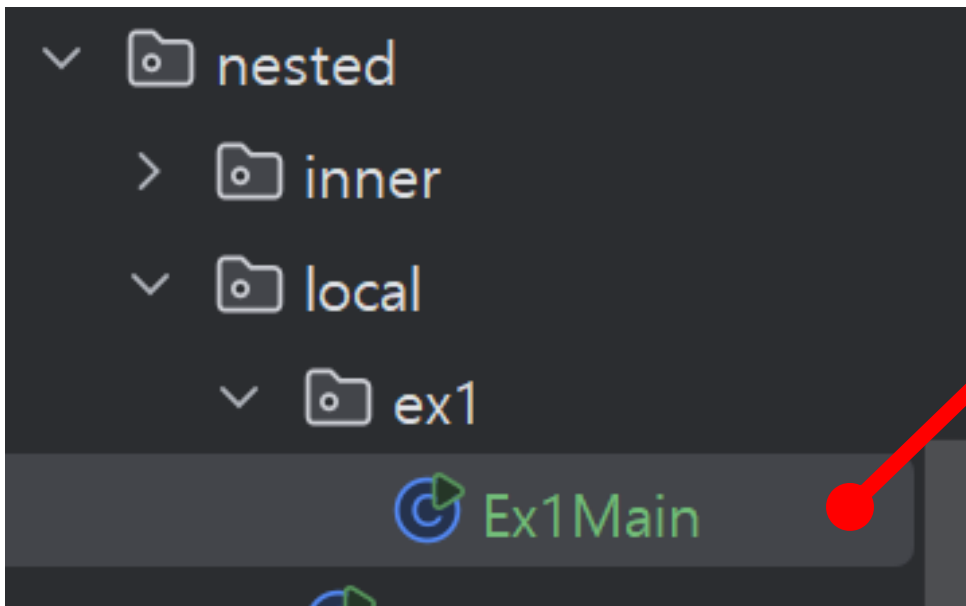


- local 패키지에 ex3 패키지를 만들어 주세요
- ex3 패키지에 Hello 인터페이스를 만들고, `public void hello()` 추상 메서드를 만들어 주세요
- ex3 패키지에 `AnonymousMain` 운영 클래스를 만들어 주세요
- 운영 클래스 내부에서 Hello 익명 클래스를 구현하고, `hello` 메서드를 실행시켜 주세요



**코드 덩어리를  
전달해 봅시다!**





익명 클래스 활용을 위한  
ex1 패키지와 Ex1Main 클래스 만들기



```
public class Ex1Main {  
    public static void helloDice() {  
        System.out.println("프로그램 시작");  
  
        // 코드 조각 시작  
        int rand = new Random().nextInt(6) + 1;  
        System.out.println("주사위의 값은 : " + rand);  
        // 코드 조각 종료  
  
        System.out.println("프로그램 종료");  
    }  
}
```

랜덤 주사위의 값을  
출력하는 간단한 프로그램

```
public static void helloDiceSum() {  
    System.out.println("프로그램 시작");  
  
    // 코드 조각 시작  
    int rand1 = new Random().nextInt(6) + 1;  
    int rand2 = new Random().nextInt(6) + 1;  
    int sum = rand1 + rand2;  
    System.out.println("주사위를 두 번 굴린 값의 합은 : " + sum);  
    // 코드 조각 종료  
  
    System.out.println("프로그램 종료");  
}
```

주사위를 2번 던져서 두 주사위 값의 합을  
출력하는 간단한 프로그램



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

프로그램 시작

주사위의 값은 : 1

프로그램 종료

프로그램 시작

주사위를 두 번 굴린 값의 합은 : 8

프로그램 종료



그란데 말입니다

그란데 말입니다

주사위 프로그램을 실행하기  
전 후에 복잡한 과정을  
수행해야만 합니다!!





```
public class Ex2Main { new *  
    public static void complicatedProgram() { no usages new *  
        System.out.println("복잡한 과정 시작");  
        System.out.println("복잡한 과정 종료 후, 원하는 기능 실행");  
  
        // 코드 조각 시작  
        // 여기에 아까 만든 주사위 프로그램을 실행 시켜 봅시다!  
        // 코드 조각 종료  
  
        System.out.println("다시 복잡한 과정 시작");  
        System.out.println("복잡한 과정 종료 후 프로그램 종료");  
    }  
  
    public static void main(String[] args) { new *  
  
    }  
}
```

우리가 원하는 건  
코드 덩어리를 전달 하는 것!



```
public class Ex2Main { new *
    public static void complicatedProgram(/* 매개 변수 전달 필요 */) {
        System.out.println("복잡한 과정 시작");
        System.out.println("복잡한 과정 종료 후, 원하는 기능 실행");

        // 코드 조각 시작
        // 전달 받은 코드 조각 실행하기
        // 코드 조각 종료

        System.out.println("다시 복잡한 과정 시작");
        System.out.println("복잡한 과정 종료 후 프로그램 종료");
    }

    public static void main(String[] args) { new *

}
```

지금까지 배운 것들(중첩 클래스, 다형성 등)과 매개 변수를 잘 사용해서 원하는 결과를 어떻게 만들지 고민해 봅시다!





오늘 점심 뭐먹지



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**



**2000 YEARS  
LATER**





**2000 YEARS  
LATER**



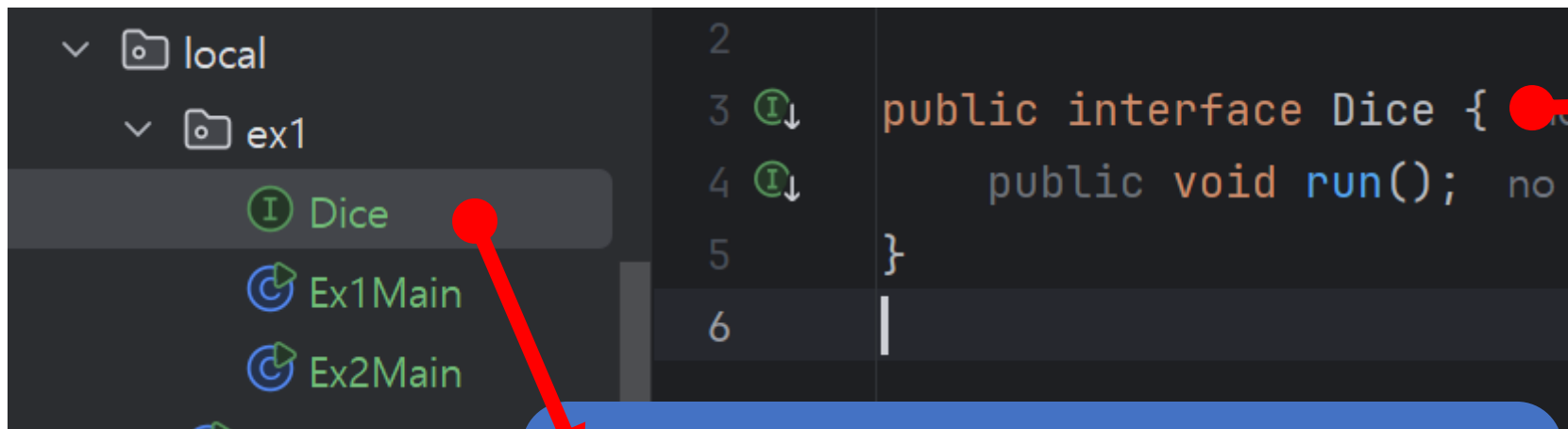
# 자! 해봅시다!





# 중첩 클래스를

# 활용



우리는 아직 코드 덩어리는  
클래스의 다형성으로 전달하는  
방법 밖에 모릅니다 (Feat. 람다)

다형적 부모 역할을 할  
Dice 인터페이스 선언

Dice 타입으로 받아서  
run() 을 구동하면  
다형적으로 구현 된  
코드가 실행되도록 구성

```
public class Ex2Main { new *
    public static void complicatedProgram(Dice dice) { 2 usages
        System.out.println("복잡한 과정 시작");
        System.out.println("복잡한 과정 종료 후, 원하는 기능 실행");
        |
        // 코드 조각 시작
        dice.run();
        // 코드 조각 종료

        System.out.println("다시 복잡한 과정 시작");
        System.out.println("복잡한 과정 종료 후 프로그램 종료");
    }
}
```

Dice 인터페이스를  
구현한 인스턴스를  
전달할 예정이므로  
Dice 타입으로 받습니다

(Feat. 다형성)

Dice 인터페이스를  
구현한 인스턴스는

반드시 run() 메서드를  
오버라이딩 해야하므로  
해당 메서드를 실행

```
static class DiceOnce implements Dice {  
    @Override  
    public void run() {  
        int rand = new Random().nextInt(6) + 1;  
        System.out.println("주사위의 값은 : " + rand);  
    }  
}
```

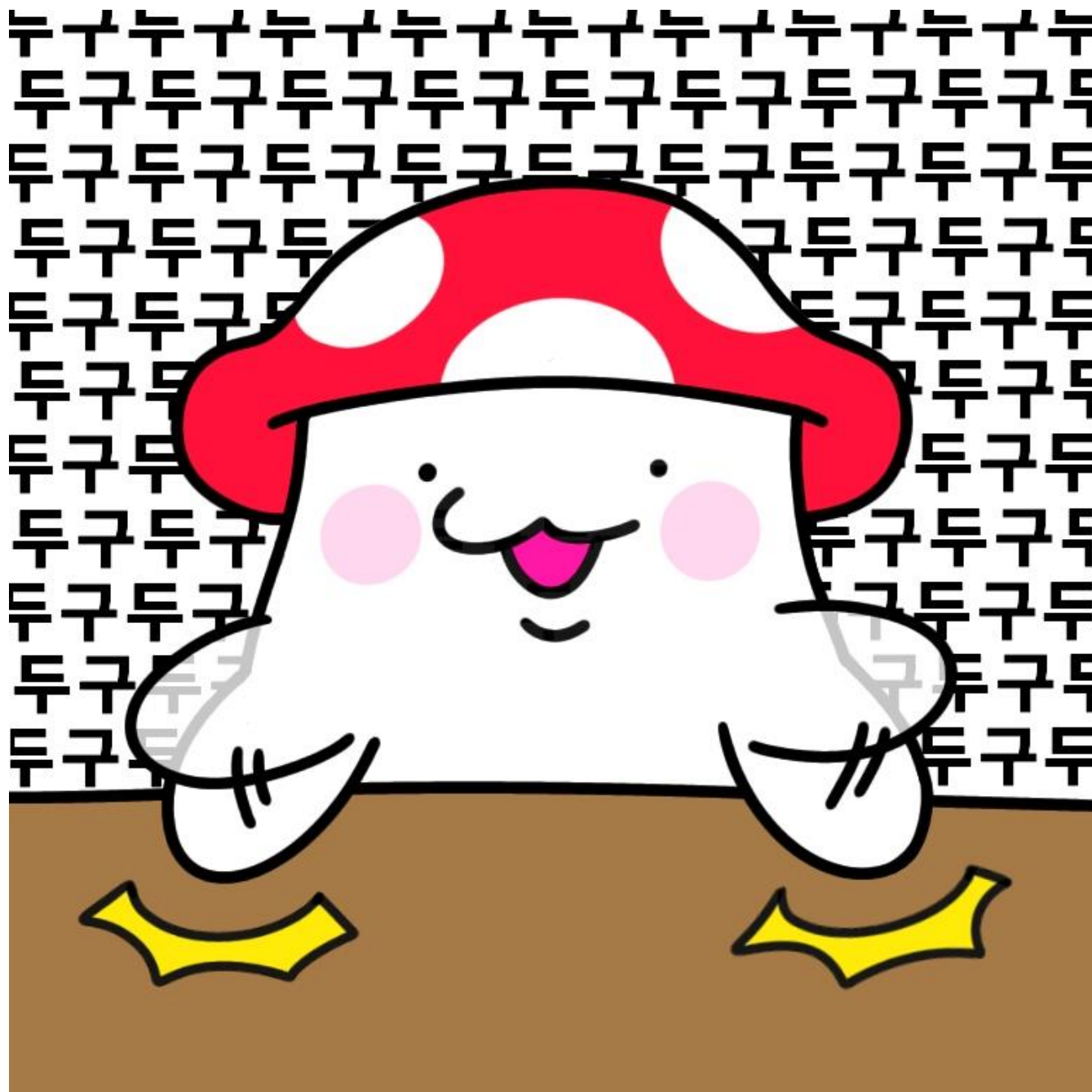
중첩 클래스를 활용 +  
Dice 인터페이스를 구현하여

원하는 코드 덩어리를  
run() 메서드에 구현!

```
static class DiceSum implements Dice {  
    @Override  
    public void run() {  
        int rand1 = new Random().nextInt(6) + 1;  
        int rand2 = new Random().nextInt(6) + 1;  
        int sum = rand1 + rand2;  
        System.out.println("주사위를 두 번 굴린 값의 합은 : " + sum);  
    }  
}
```

중첩 클래스를 활용 +  
Dice 인터페이스를 구현하여

원하는 코드 덩어리를  
run() 메서드에 구현!







```
public static void main(String[] args) { new *  
    complicatedProgram(new DiceOnce());  
    complicatedProgram(new DiceSum());  
}
```

역할을 하는  
complicatedProgram 에  
익명 클래스를 인스턴스화  
하여 바로 전달!

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

주사위의 값은 : 5

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

주사위를 두 번 굴린 값의 합은 : 4

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료





그런데 말입니다

그런데 말입니다

DiceOnce 와 DiceSum 은  
complicatedProgram 에  
종속 된 형태를 띄고 있는데  
굳이 중요한 영역인  
Static 에 보관할 필요가  
있을까요!?



# 지역 클래스

## 활용

```
public static void main(String[] args) {  
    class DiceOnce implements Dice {  
        @Override  
        public void run() {  
            int rand = new Random().nextInt(6) + 1;  
            System.out.println("주사위의 값은 : " + rand);  
        }  
    }  
  
    class DiceSum implements Dice {  
        @Override  
        public void run() {  
            int rand1 = new Random().nextInt(6) + 1;  
            int rand2 = new Random().nextInt(6) + 1;  
            int sum = rand1 + rand2;  
            System.out.println("주사위를 두 번 굴린 값의 합은 : " + sum);  
        }  
    }  
}
```

기존의 중첩 클래스(static)를  
main 메서드 내부의  
지역 클래스로 변경!

→ static 삭제 필요



```
complicatedProgram(new DiceOnce());  
complicatedProgram(new DiceSum());
```

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

주사위의 값은 : 5

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

주사위를 두 번 굴린 값의 합은 : 4

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료





그런데 말입니다

그런데 말입니다

DiceOnce 와 DiceSum 은  
complicatedProgram  
실행 시에 한 번만 사용되고  
사라지는데 굳이 이름까지  
붙여서 만들어 줄  
필요가 있을까요?



# 익명 클래스

## 활용



```

public static void main(String[] args) {
    Dice diceOnce = new Dice() {
        @Override
        public void run() {
            int rand = new Random().nextInt(6) + 1;
            System.out.println("주사위의 값은 : " + rand);
        }
    };

    Dice diceSum = new Dice() {
        @Override
        public void run() {
            int rand1 = new Random().nextInt(6) + 1;
            int rand2 = new Random().nextInt(6) + 1;
            int sum = rand1 + rand2;
            System.out.println("주사위를 두 번 굴린 값의 합은 : " + sum);
        }
    };
}

```

Dice 인터페이스를  
 바로 구현하여  
 변수에 인스턴스를 저장하는  
 익명 클래스 형태로 변경

```
complicatedProgram(diceOnce);  
complicatedProgram(diceSum);
```

실행 하기 위한 코드 덩어리는  
인스턴스에 저장 되어있으므로  
인스턴스 참조 값을 저장한 변수를 전달

복잡한 과정 시작  
복잡한 과정 종료 후, 원하는 기능 실행  
주사위의 값은 : 5  
다시 복잡한 과정 시작  
복잡한 과정 종료 후 프로그램 종료  
복잡한 과정 시작  
복잡한 과정 종료 후, 원하는 기능 실행  
주사위를 두 번 굴린 값의 합은 : 4  
다시 복잡한 과정 시작  
복잡한 과정 종료 후 프로그램 종료





# 익명 클래스

## 활용2



```

public static void main(String[] args) {
    complicatedProgram(new Dice() {
        @Override
        public void run() {
            int rand = new Random().nextInt(6) + 1;
            System.out.println("주사위의 값은 : " + rand);
        }
    });

    complicatedProgram(new Dice() {
        @Override
        public void run() {
            int rand1 = new Random().nextInt(6) + 1;
            int rand2 = new Random().nextInt(6) + 1;
            int sum = rand1 + rand2;
            System.out.println("주사위를 두 번 굴린 값의 합은 : " + sum);
        }
    });
}

```

변수에 저장하는 것조차 사치다  
리얼 한번 쓰고 말 것이라면  
그냥 바로 구현해서 쓰고  
바로 GC로 처리한다!!



복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행  
주사위의 값은 : 5

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행  
주사위를 두 번 굴린 값의 합은 : 4

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

인사이드7

마지막 용불 대방출



# 끝