



It's Your Life

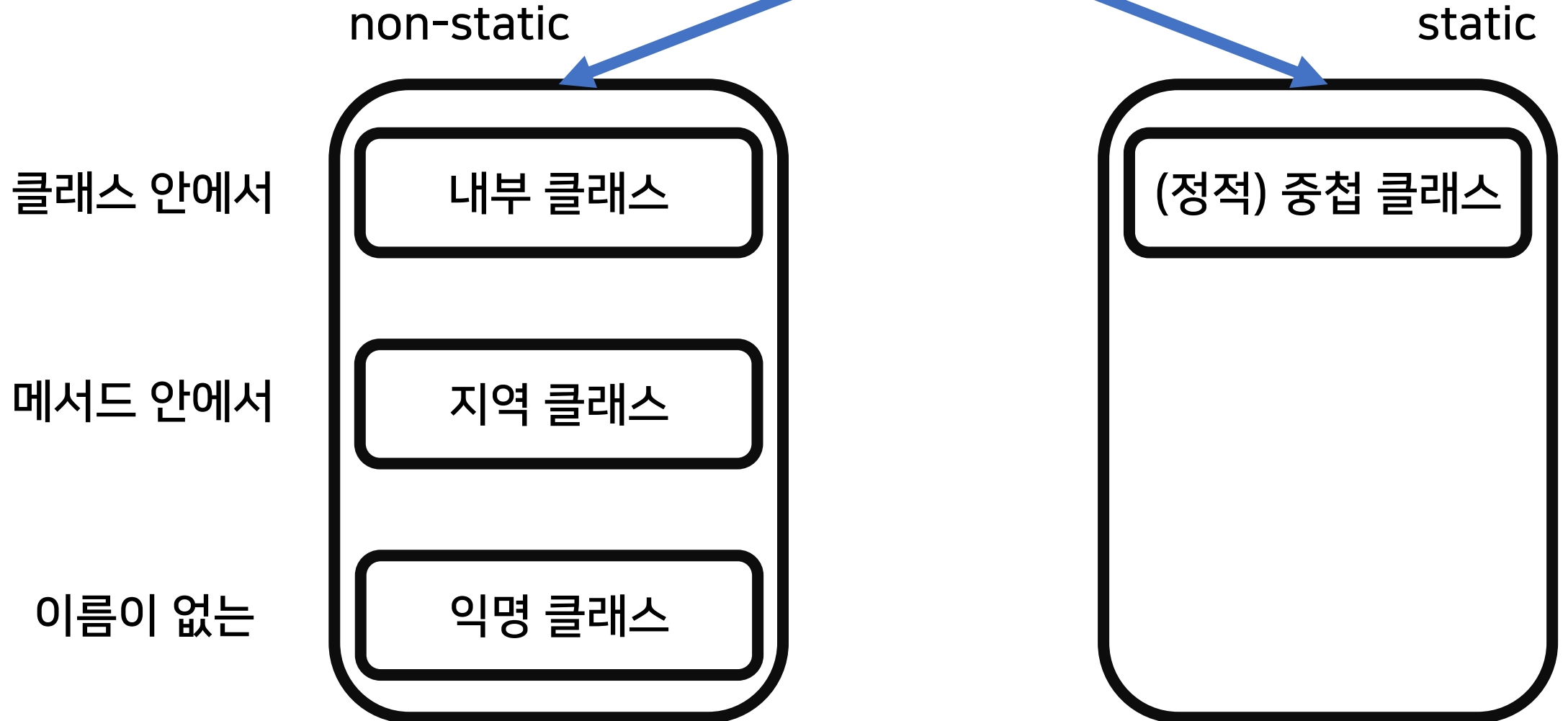
with





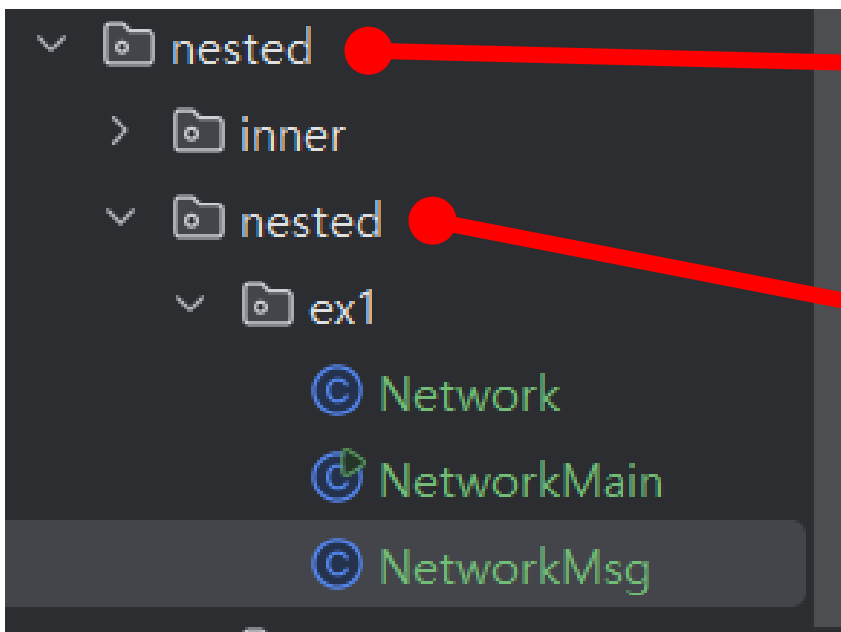
# 중첩 클래스

# 중첩 클래스의 종류









중첩 클래스를 위한  
nested 패키지 생성

(정적) 중첩 클래스를 위한  
nested 패키지 생성





# 그럼 집어 넣읍시다!





```
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!



# 중첩 클래스와 내부 클래스의 차이



# 중첩 클래스의

# 특징



```
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?

# 메서드 영역 (공용 영역)

Outer



```
static outerStatic;
```

Nested

```
static nestedStatic;
```

# 힙 영역 (공용 X)



Outer@x001

```
outerInstance;
```

Nested@x002

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



## 메서드 영역 (공용 영역)

Outer

```
static outerStatic;
```

Nested

```
static nestedStatic;
```

## 힙 영역 (공용 X)



Outer@x001

```
outerInstance;
```

Nested@x002

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





# 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);  
}
```



# 내부 클래스의

# 특징



```
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!!!!



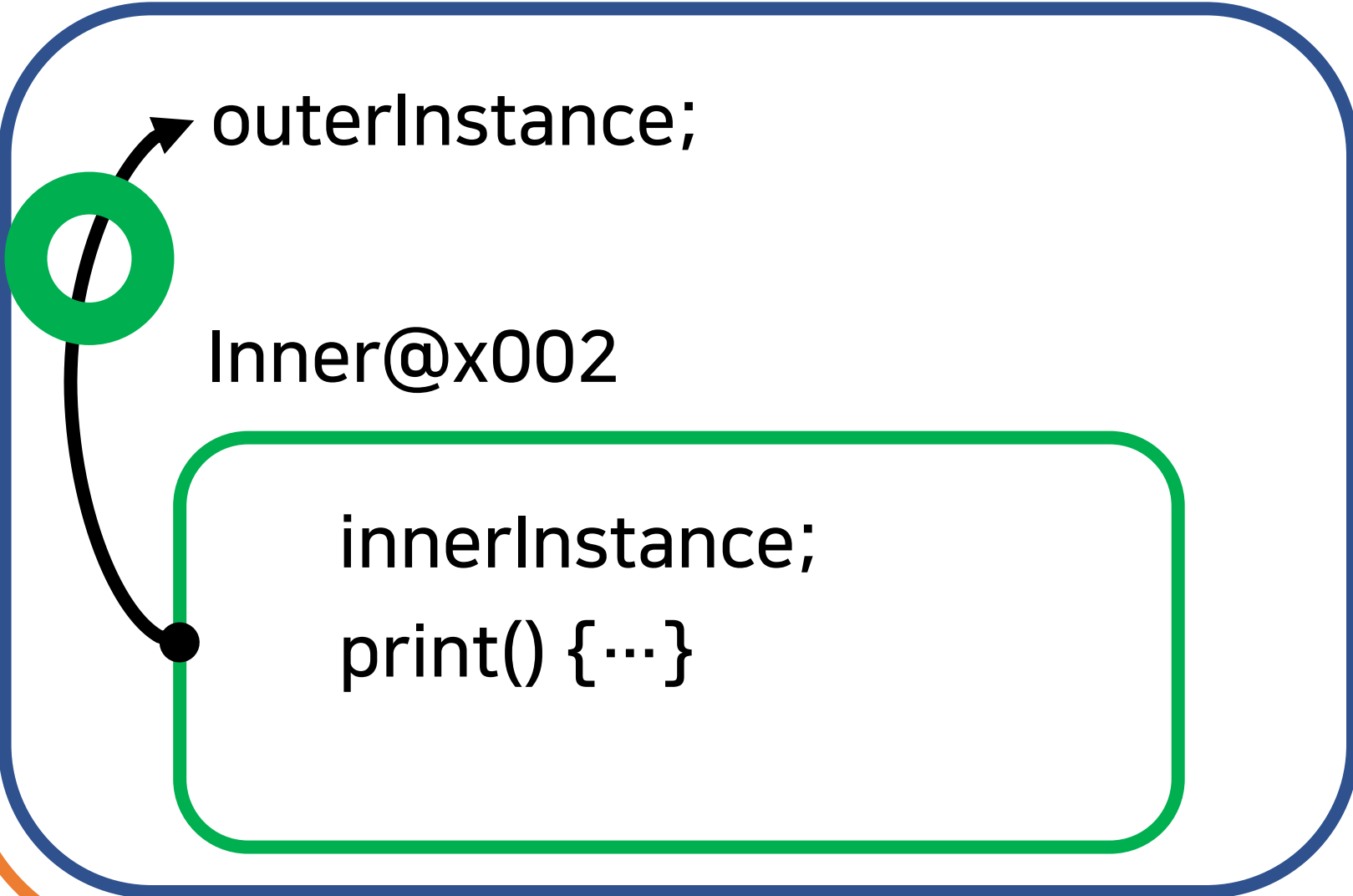


Outer@x001

outerInstance;

Inner@x002

innerInstance;  
print() {...}





Outer@x001

outerInstance;

Inner@x002

- 외부 클래스의 참조값을 보관(x001)

innerInstance;

print() {...}





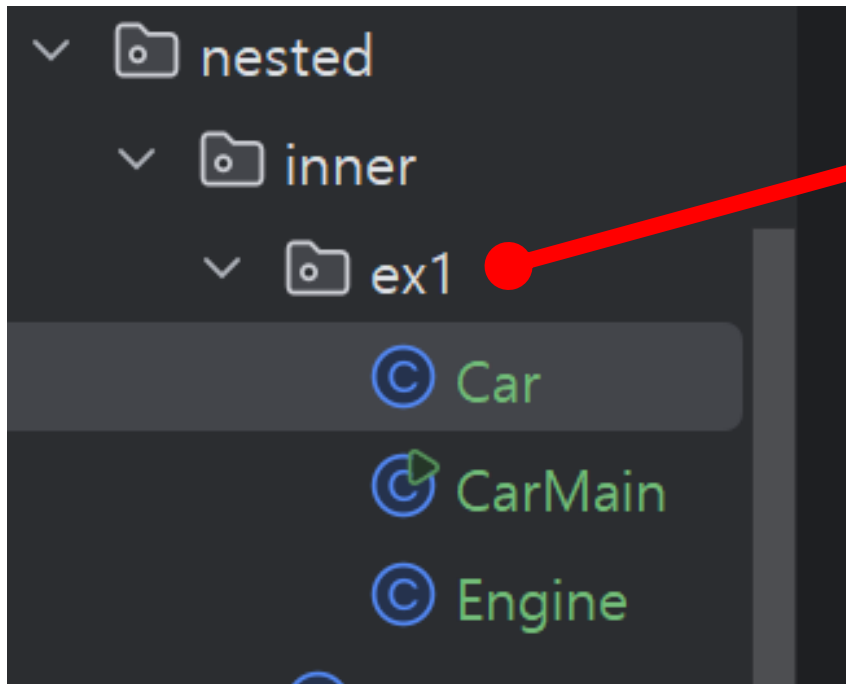
아 갈매기짤 겨우찾았네 조현우랑 너무  
닮아서 오기로찾음





# 내부 클래스

# 사용 예제



예제 작성을 위한 ex1 패키지 생성

```
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");  
    }  
}
```

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

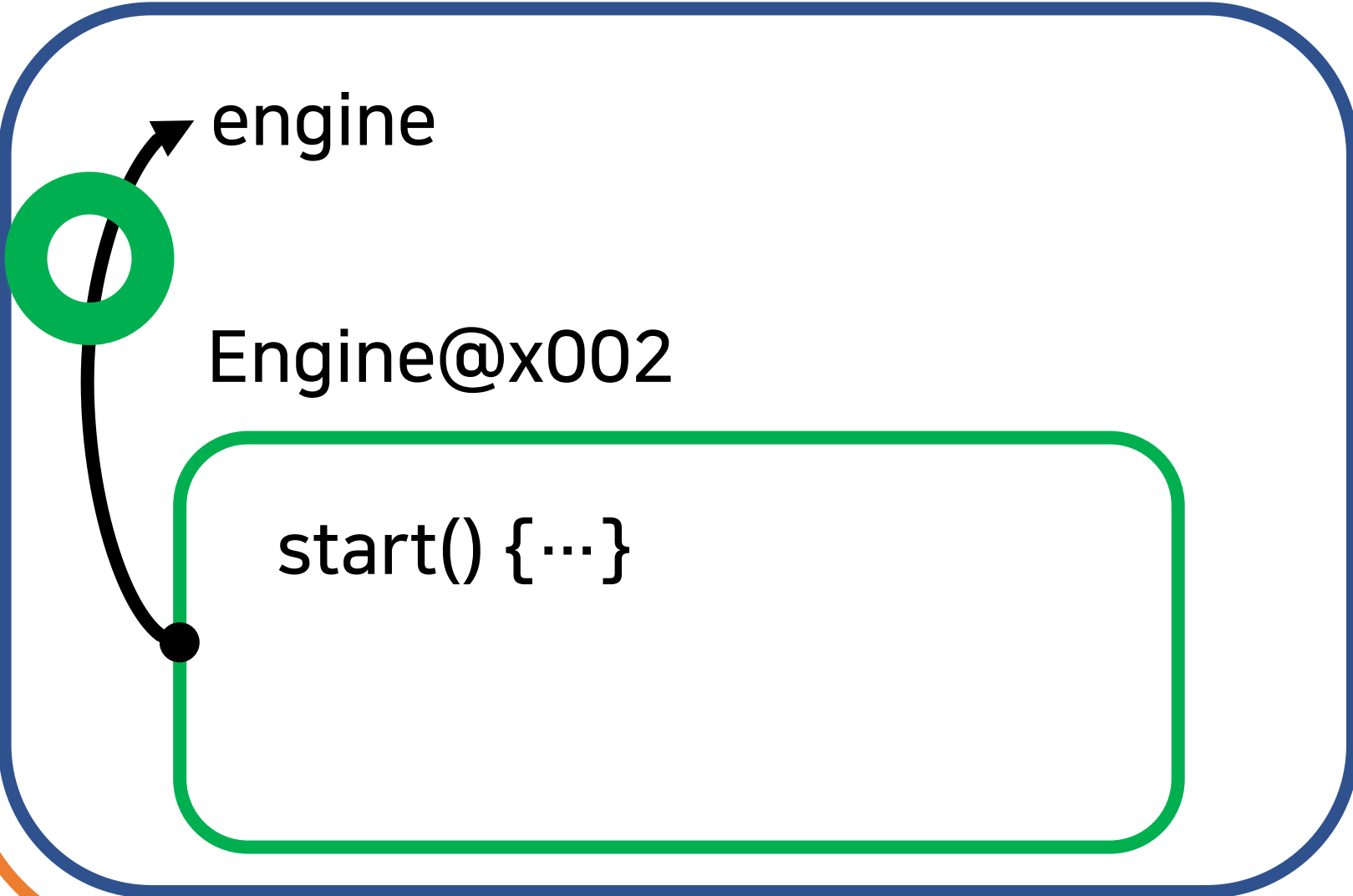


Car@x001

engine

Engine@x002

start() {...}





# 지역 클래스와

# 익명 클래스



# 중첩 클래스의 종류



중첩 클래스

non-static

static

클래스 안에서

내부 클래스

메서드 안에서

지역 클래스

이름이 없는

익명 클래스

(정적) 중첩 클래스



```
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



LocalOuter@x001

outerInstance

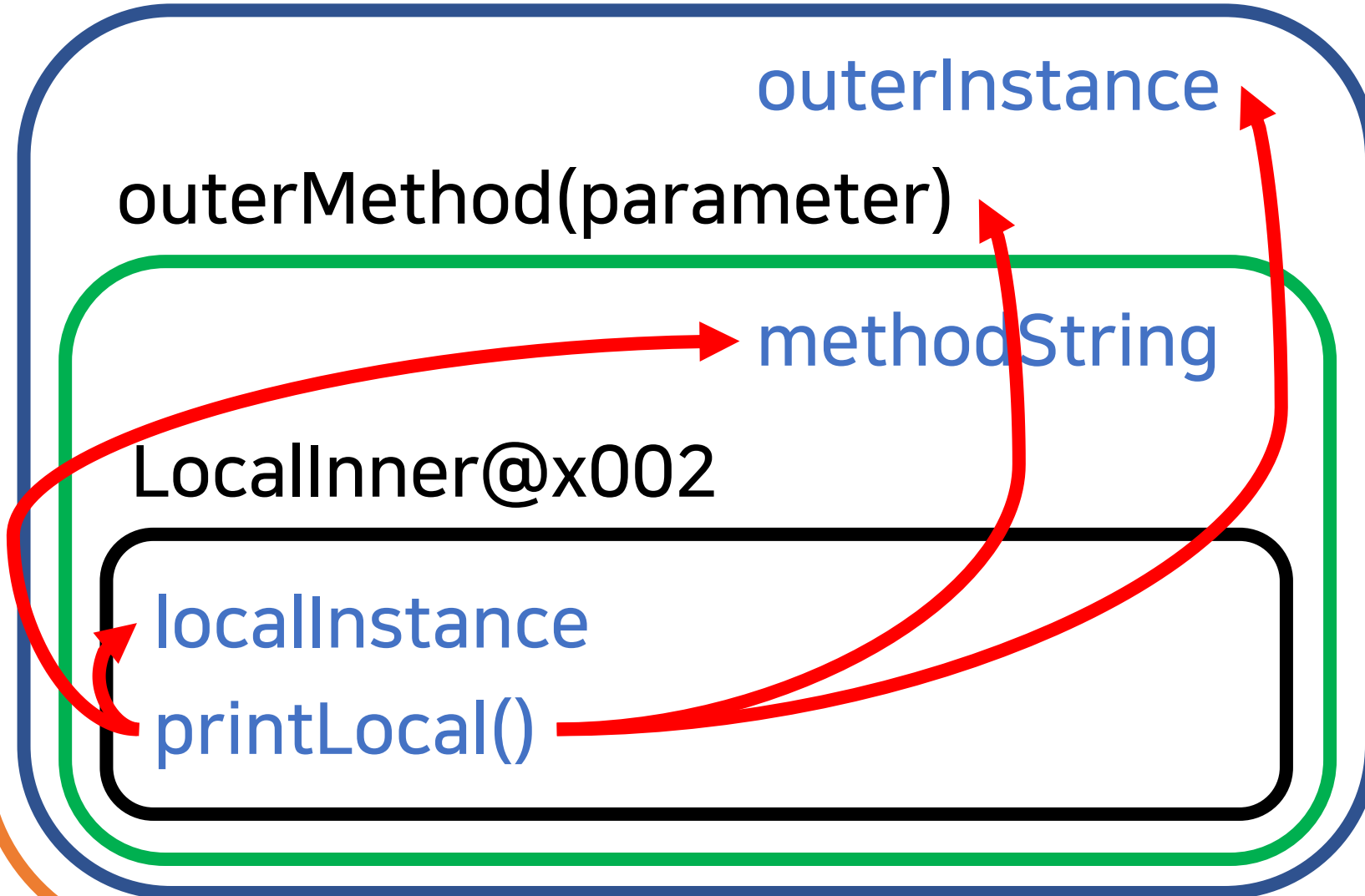
outerMethod(parameter)

methodString

LocalInner@x002

localInstance

printLocal()



# 힙 영역



LocalOuter@x001

outerInstance

outerMethod(parameter)

methodString

LocalInner@x002

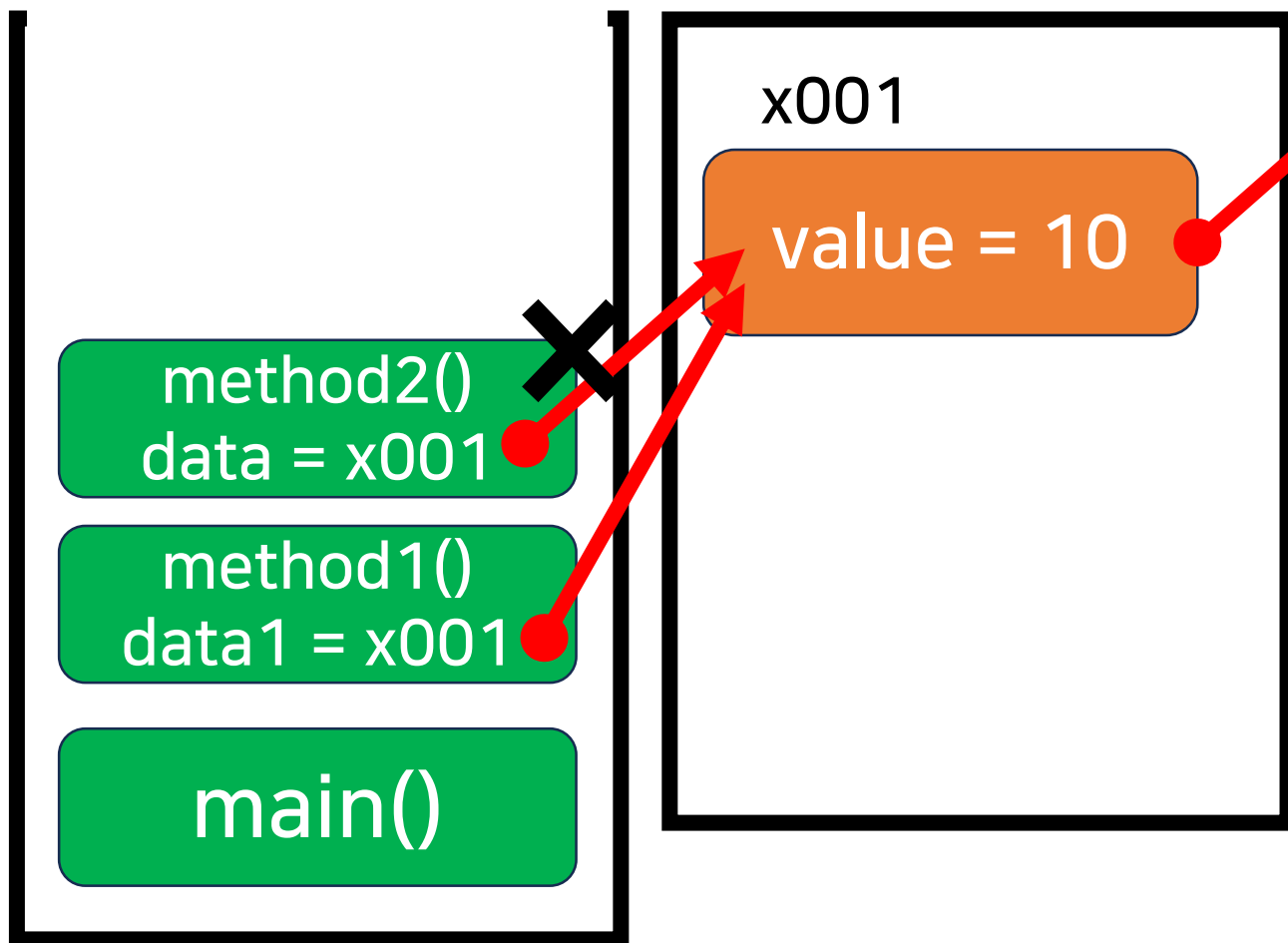
localInstance

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!



# 익명 클래스

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

Print 인터페이스는  
간단하게 printLocal() 이라는  
추상 메서드만을 가집니다





```
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();
```

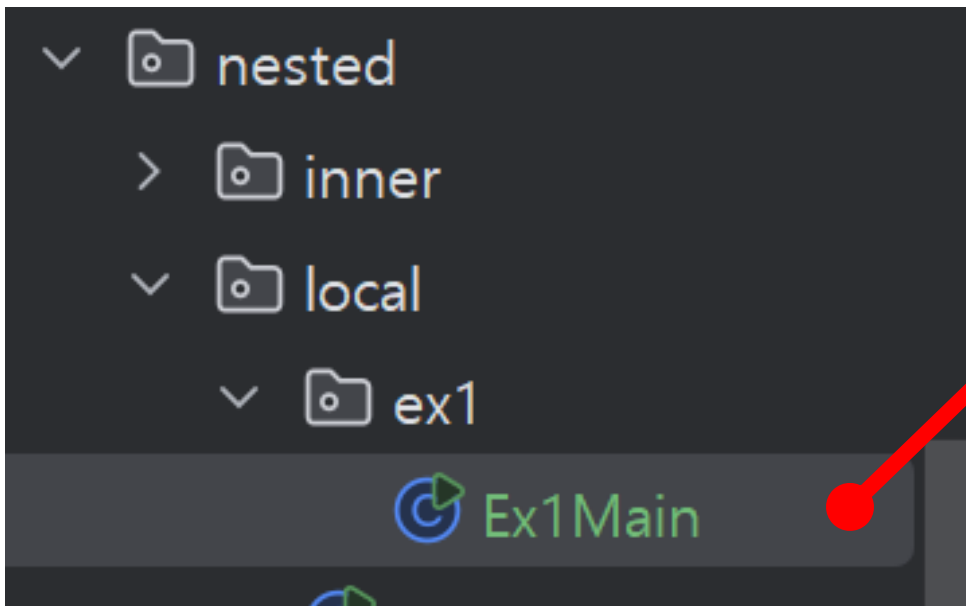
이름 없는 클래스의 인스턴스가  
print 변수에 저장 되었으므로  
바로 사용!



오케이 이제 부터 시작이다 렛츠고



**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(bound: 6) + 1;  
    int rand2 = new Random().nextInt(bound: 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

주사위 프로그램을 실행하기  
전 후에 복잡한 과정을  
수행해야만 합니다!!

그란데 말입니다





```
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**



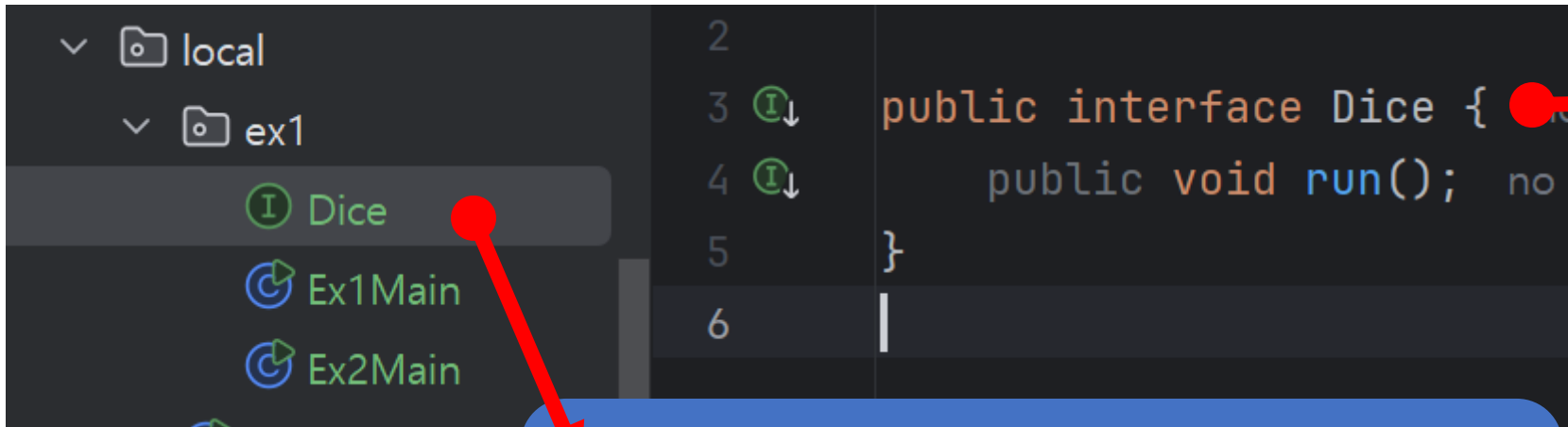
# 자! 해봅시다!





# 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);  
    }  
}
```

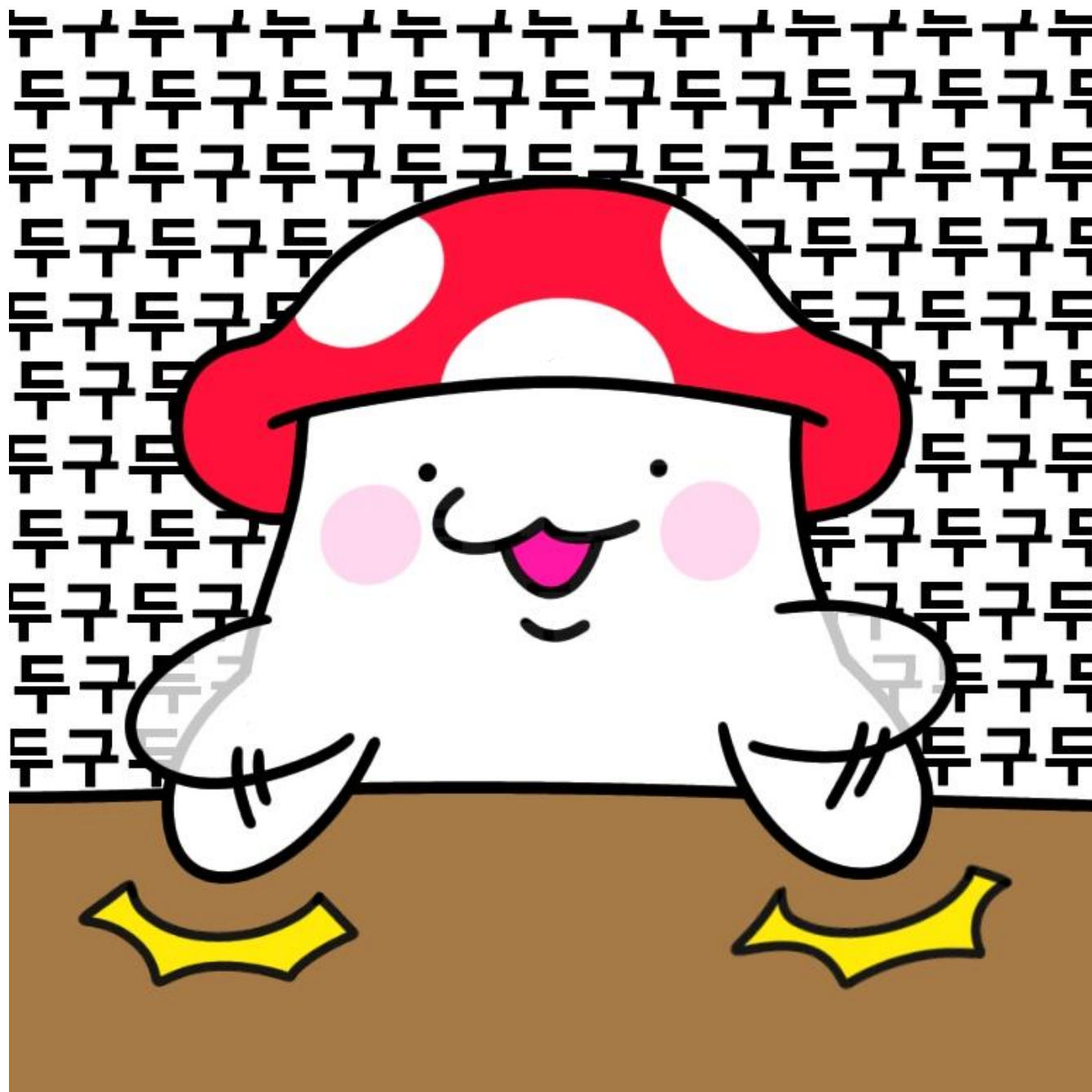
중첩 클래스를 활용 +  
Dice 인터페이스를 구현하여

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



# 실습, 코드 전달 실습



- Fortune 이라는 인터페이스를 만들고, Fortune 인터페이스는 추상 메서드로 run 을 가집니다.
- Ex4Main 클래스를 복사해서 Ex5Main 이라는 클래스를 만들어 주세요.
- Ex5Main 의 complicatedProgram 메서드는 아래와 같이 이제 매개변수로 Fortune 타입의 fortune 를 받고, 메서드 내부에서 fortune.run() 을 실행 시킵니다

# 실습, 코드 전달 실습



```
public class Ex5Main { new *
    R Rename usages
    public static void complicatedProgram(Fortune fortune) { 10
        System.out.println("복잡한 과정 시작");
        System.out.println("복잡한 과정 종료 후, 원하는 기능 실행");

        fortune.run();

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

# 실습, 코드 전달 실습



- 코드 전달 방법을 사용하여 아래의 조건을 만족하는 코드를 main 메서드에서 완성해 주세요.
- `fortune.run()` 실행 시, "\*\*\* 오늘은 행운의 날입니다. 행복하세요!" 출력
- `fortune.run()` 실행 시, "\*\*\* 오늘의 행운의 번호는 ~~ 입니다" 를 출력
  - 행운의 번호는 1 ~ 99 사이의 숫자 중에서 랜덤 숫자를 구해서 출력



복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

\*\*\* 오늘은 행운의 날입니다. 행복하세요

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

\*\*\* 오늘 행운의 번호는 86 입니다

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

# 도전, 코드 전달 실습



- fortune.run() 실행 시, “오늘의 행운 번호를 입력하세요 : ” 가 출력되고 행운의 번호 1 ~ 99 사이의 숫자를 입력 받기
- 1 ~ 99 사이의 숫자를 3회 랜덤하게 발생 시킨 후, 입력 받은 번호와 같은 번호가 있으면 “와우!! 완전한 행운의 날입니다. 맞춘 행운의 번호는 ~~ 입니다” 출력
- 번호가 없으면, “입력 하신 번호는 불운의 번호이니 오늘은 피하세요! 😊” 출력





복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

오늘의 행운 번호를 입력하세요 : 7

와우! 완전한 행운의 날입니다. 맞춘 행운의 번호는 7 입니다

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료

복잡한 과정 시작

복잡한 과정 종료 후, 원하는 기능 실행

오늘의 행운 번호를 입력하세요 : 7

입력하신 번호는 불운의 번호이니 오늘은 피하세요 :)

다시 복잡한 과정 시작

복잡한 과정 종료 후 프로그램 종료



**중요한 건  
꺾이지 않는 마음**

WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022 WORLDS 2022

# 실습, 도서관 프로그램 작성 하기



- 도서를 저장하고 도서 목록을 출력하는 프로그램을 작성해 봅시다
- 먼저 Library 클래스를 만들어 주세요
- Library 클래스는 아래와 같은 멤버 변수와 생성자, 메서드를 가집니다
- 멤버 변수

```
public class Library { 2 usages    new *  
    Book[] books; 7 usages  
    int bookCount; 9 usages  
    static final int LIBRARY_SIZE = 4; 2 usages
```

# 실습, 도서관 프로그램 작성 하기



- Library 의 생성자

```
public Library() {  
    this.books = new Book[LIBRARY_SIZE];  
    bookCount = 0;  
}
```

- Library 의 메서드 목록을 보고 코드를 완성해 주세요

```
public void addBook() {}  
public void showBooks() {}
```

# 실습, 도서관 프로그램 작성 하기



- 멤버 변수를 보셔서 아시겠지만 도서관은 최대 책 4권을 보관할 수 있으며, 새로운 책을 보관 하려고 했을 때 이미 책이 4권이라면 “더 이상 책을 보관할 수 없습니다” 를 출력해 주시면 됩니다.
- 책이 정상적으로 보관 되면 “보관 된 책의 수는 :  $\${\text{보관된 책의 수}}$ ” 를 출력해 줍니다.
- 책 목록 출력은 “제목 : ~~ / 저자 : ~~” 형태로 출력 하시면 됩니다

# 실습, 도서관 프로그램 작성 하기



- 멤버 변수를 보면 도서는 Book 이라는 클래스의 형태로 관리되고 있습니다.  
Book 클래스는 Library 의 중첩 클래스로 선언하여 사용하시면 됩니다

```
public void addBook() {} 1 usage new *  
public void showBooks() {} 1 usage new *  
  
// Book 중첩 클래스 선언 및 사용
```

- Book 클래스는 책 제목과 저자를 멤버 변수로 가지며, 외부로 부터 제목과 저자를 받아서 책 인스턴스를 생성하는 생성자를 가지고 있습니다

# 실습, 도서관 프로그램 작성 하기



- 아래의 운영 클래스 코드를 보고 프로그램을 완성해 주세요

```
public class LibraryMain2 { new *
    public static void main(String[] args) { new *
        Library lib = new Library();
        Scanner scanner = new Scanner(System.in);

        System.out.println("=== 도서 관리 프로그램에 오신 것을 환영 합니다 ===");
        while (true) {
            System.out.println("=== 원하는 기능을 선택 하세요 ===");
            System.out.print("1. 도서 추가 / 2. 도서 목록 출력 / 3. 프로그램 종료) : ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // 엔터키로 인한 버퍼 삭제

            if (choice == 1) {
                lib.addBook();
            } else if (choice == 2) {
                lib.showBooks();
            } else {
                System.out.println("=== 프로그램을 종료 합니다 ===");
                return;
            }
        }
    }
}
```

# 실습, 실제 실행 화면 캡처



=== 원하는 기능을 선택 하세요 ===

1. 도서 추가 / 2. 도서 목록 출력 / 3. 프로그램 종료 : 1

책 제목을 입력 하세요 : 자바

책 저자를 입력 하세요 : 자바

보관 된 책의 수는 : 1

=== 책 보관 완료 ===

1. 도서 추가 / 2. 도서 목록 출력 / 3. 프로그램 종료 : 2

=== 책 목록 출력 ===

\*\*\* 총 보관 책의 수는 : 4 입니다 \*\*\*

1. 제목 : 자바 / 저자 : 자바

2. 제목 : 자바2 / 저자 : 자바2

3. 제목 : 자바3 / 저자 : 자바3

4. 제목 : 자바4 / 저자 : 자바4

1. 도서 추가 / 2. 도서 목록 출력 / 3. 프로그램 종료 : 3

=== 프로그래밍을 종료 합니다 ===



# 도전, 가장 오래 된 책 삭제 기능 추가



- 가장 오래된 책을 삭제하는 removeBook 메서드를 추가해 주세요

```
public void addBook() {} 2 usages new *  
public void showBooks() {} 2 usages new *  
public void removeBook() {} // 가장 오래 된 책을 삭제하는 코드를 추가해 주세요!
```

- 보관 된 책이 없을 경우, "삭제할 책이 없습니다" 를 출력해 주세요.
- 운영 클래스에서는 1. 도서 추가 / 2. 가장 오래된 도서 삭제 / 3. 도서 목록 출력 / 4. 프로그램 종료로 만들어 주시면 됩니다!

# 도전, 실제 실행 화면 캡처



1. 도서 추가 / 2. 가장 오래된 도서 삭제 / 3. 도서 목록 출력 / 4. 프로그램 종료 : 1

책 제목을 입력 하세요 : 자바

책 저자를 입력 하세요 : 자바

보관 된 책의 수는 : 1

=== 책 보관 완료 ===

=== 원하는 기능을 선택 하세요 ===

1. 도서 추가 / 2. 가장 오래된 도서 삭제 / 3. 도서 목록 출력 / 4. 프로그램 종료 : 2

\*\*\* 가장 오래된 책이 삭제되었습니다. \*\*\*

=== 원하는 기능을 선택 하세요 ===

1. 도서 추가 / 2. 가장 오래된 도서 삭제 / 3. 도서 목록 출력 / 4. 프로그램 종료 : 2

\*\*\* 보관 된 책이 존재하지 않습니다 \*\*\*