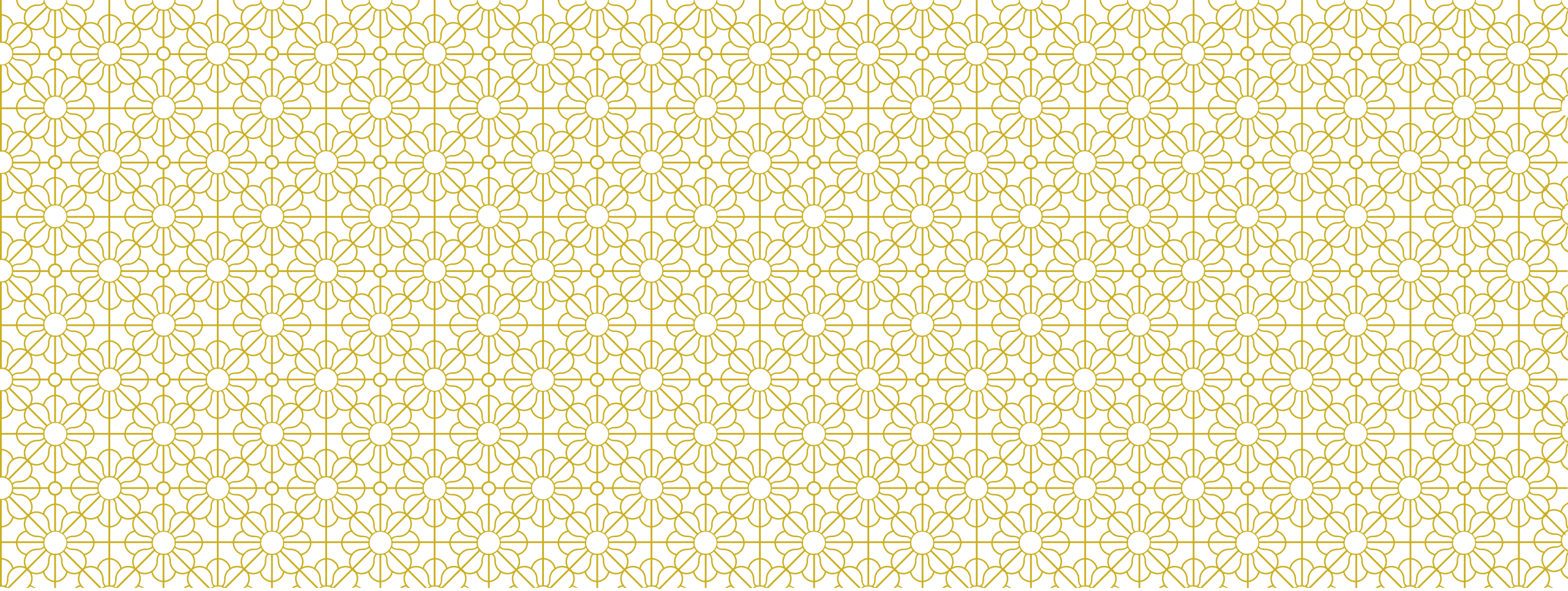


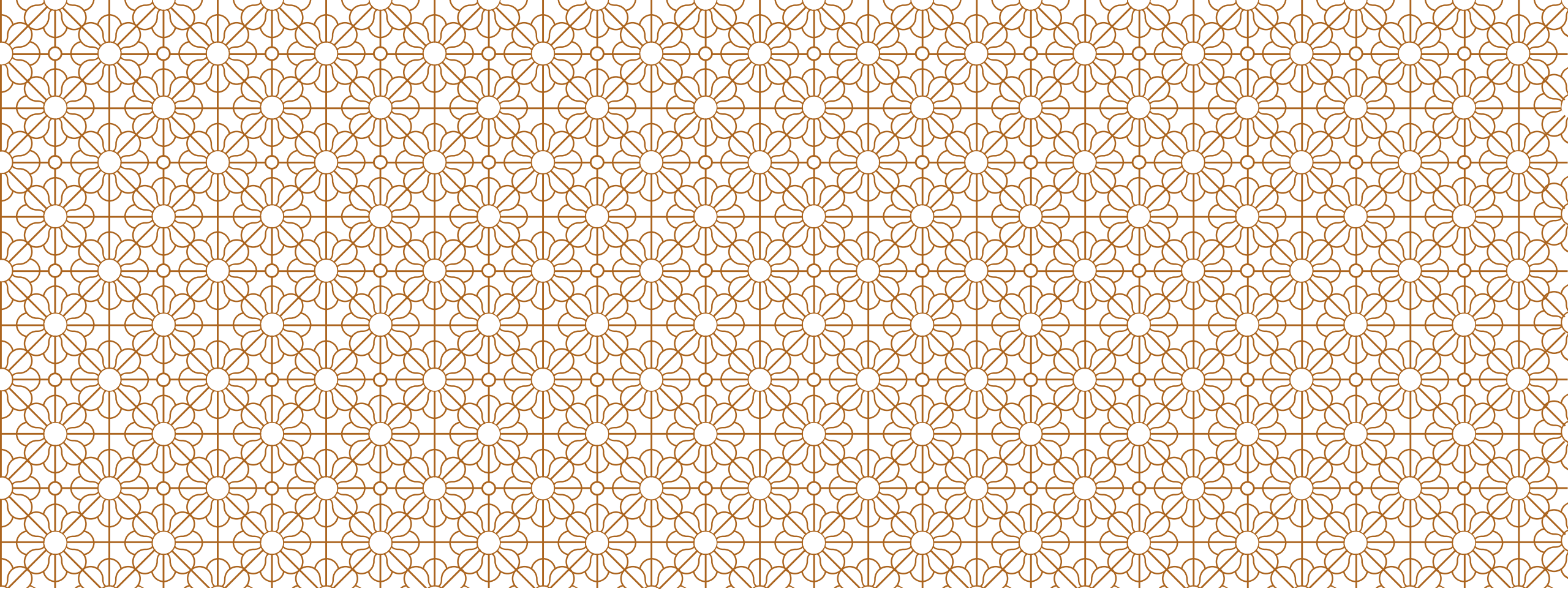
# Exception Handling & Parameter Passing

Java Tutoral #3

신용호
컴퓨터과학과 12

# contents

1. Exception Handling
2. Parameter Passing
3. Q & A



# Exception Handling

Exception
Exception Throwing
Exception Catching

# Java의 Exception Handling

Java는 Exception에 매우 민감합니다.

Exception을 처리하지 않으면 Compile Error를 때려버리죠.

이는 Exception, Error 등에 너그러운 C/C++과 비교되는 Java의 특징 중 하나입니다.

그래서 코딩을 할 때 많이 번거로울 수 있답니다.

하지만 적절한 Exception Handling은 프로그램의 안정성을 높이는 데 도움이 됩니다.

# Exception Handling의 순서?


- 코드의 진행 중에 무엇인가가 잘못된 경우 예외 객체를 생성하고 throw를 사용하여 알린다.
- 발생한 예외가 해당 메소드에서 해결할 수 없을 때, throws를 사용하여 이를 상위의 메소드로 투척한다.
- try-catch 구문을 활용하여 try block 내부의 예외를 처리한다.

# 이번엔 삼각형입니다.

```
public class Triangle {

	private Set<Point> pointSet;

	public Triangle(Set<Point> pointSet) {
		this.pointSet = pointSet;
	}

	public Set<Point> getPointSet() {
		return pointSet;
	}

	public void setPointSet(Set<Point> pointSet) {
		this.pointSet = pointSet;
	}

	public double getAreaSize() {

		Iterator<Point> iter = pointSet.iterator();
		Point p1 = iter.next();
		Point p2 = iter.next();
		Point p3 = iter.next();

		double x1y2 = p1.getX() * p2.getY();
		double x2y3 = p2.getX() * p3.getY();
		double x3y1 = p3.getX() * p1.getY();
		double x1y3 = p1.getX() * p3.getY();
		double x3y2 = p3.getX() * p2.getY();
		double x2y1 = p2.getX() * p1.getY();

		double result = Math.abs(((x1y2 + x2y3 + x3y1) - (x1y3 + x3y2 + x2y1)) / 2.f);

		return result;
	}
}
```

Point는 저번에 구현한 것입니다.

이 삼각형은 Point를 Set으로 가지고 있으며 Set의 정점으로 이루어진 삼각형의 넓이를 구하는 메소드를 갖고 있습니다.

얼핏 보아도 뭔가 문제가 많아 보이지요?

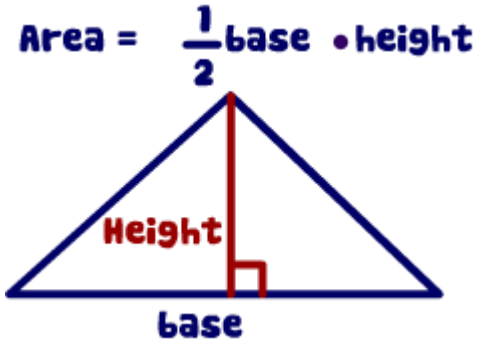
일단은 잘 돌아가는 지 확인해 봅시다.

# 직각삼각형의 넓이를 구하자~

(0, 0), (4, 0), (4, 3)으로 이루어진 삼각형의 넓이는 어떻게 될까요?

당연히 6이 되겠지요?

우리의 코드도 당연히 답을 맞추는 지 볼까요?

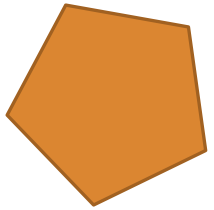



```
Point a1 = new Point(0f, 0f);
Point a2 = new Point(4f, 0f);
Point a3 = new Point(4f, 3f);

Set<Point> setA = new HashSet<Point>();
setA.add(a1);
setA.add(a2);
setA.add(a3);

Triangle triA = new Triangle(setA);
System.out.println(triA.getAreaSize());
```

# 잠깐! 만약에요...

만약 Set의 정점의 수가 3개가 아니라면 어떻게 될까요?

만약 4개 이상이라면 코드는 돌아가겠지만 믿음직스럽지 않겠군요.
만약 1개나 2개라면? 아마 런타임 에러가 날 것만 같군요.



하지만 우리의 코드는 이러한 돌발 상황에 대처하지 못합니다.

이런 문제를 대표하는 예외를 만들고 이런 상황을 잡아봅시다!

# 3개가 아니라니까!

```
public class NotThreePointsException extends Exception {
	public void printStackTrace() {
		System.out.println("Points are more or fewer than 3!");
	}
}
```

이러한 객체를 만들어 보겠습니다.

이 객체는 정점이 3개가 아닌 상황을 대표하는 예외 객체입니다.

Exception 객체를 상속하면 자신이 정의한 예외 상황을 표현하는 객체를 만들 수 있습니다.

참고로 printStackTrace 함수는 Throwable이라는 인터페이스에 정의되어 있습니다.

# 그렇다면 삼각형에는...

삼각형을 생성할 때 Set에 담겨있는 Point의 개수가 3개인지 아닌지를 파악합시다.

만약 3개가 아니다?

그럼 과감히 예외 상황을 발생시키는 것입니다.

이 부분은 throw를 사용하여 할 수 있습니다!

throws를 활용하여 이것을 호출한 메소드에게 이 예외를 투척할 수 있게 됩니다.

```
public Triangle(Set<Point> pointSet) throws NotThreePointsException {
        if (pointSet.size() != 3) {
                throw new NotThreePointsException();
        }

        this.pointSet = pointSet;
}
```

# Eclipse는 참 편리하죠?

```
Triangle triA;
try {
        triA = new Triangle(setA);
} catch (NotThreePointsException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
}
```

main 함수에서 빨간 줄이 표시된 것을 볼 수 있을 겁니다.

그 빨간 줄에 대고 이번에는
Surround with try/catch를 눌러주세요!

자동완성 기능으로 손쉽게 try-catch문을 만들 수 있답니다.

자 이제 우리의 예외가 잘 작동하는 지 확인해 볼까요?

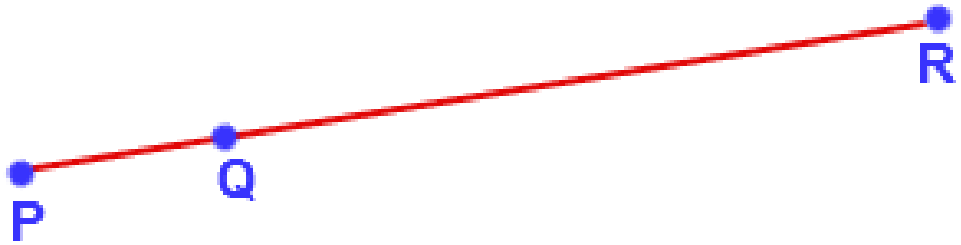
# 점이 이렇게 들어온다면?

```
Set<Point> setB = new HashSet<Point>();
setB.add(new Point(0f, 0f));
setB.add(new Point(0f, 0f));
setB.add(new Point(9f, 9f));
```

# 음... 이런 경우는요?

자 이제 서로 다른 세 개의 정점을 제외한 경우는 모두 예외 상황으로 처리할 수 있게 되었습니다.

그럼 이제 문제가 없을까요?




서로 다른 세 개의 점으로는 분명 삼각형을 만들 수 있을 겁니다.

단 한 가지 경우만 빼면요. 바로 모든 정점이 한 직선 상에 위치할 때이죠.

# 일직선이라니...

```
public class PointsInLineException extends Exception {
        public void printStackTrace() {
                System.out.println("Points are in a line!");
        }
}
```

# 그리고 삼각형에도!

```
Iterator<Point> iter = pointSet.iterator();
Point p1 = iter.next();
Point p2 = iter.next();
Point p3 = iter.next();

if ((p1.getX() - p2.getX()) * (p1.getY() - p3.getY()) == (p1.getX() - p3.getX()) * (p1.getY() - p2.getY())) {
        throw new PointsInLineException();
}
```

삼각형의 생성자에 이렇게 코드를 작성해 봅시다.

Iterator가 뽑아내는 Point의 개수를 통해서 어디에 작성해야 할지는 눈에 딱 보이시죠?

참 생성자의 throws 옆에도 무엇인가 추가가 되어야 한다는 점을 있지 마세요~

# 이런 코드가 되겠지요?

```
Triangle triB = null;
try {
        triB = new Triangle(setB);
} catch (NotThreePointsException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
} catch (PointsInLineException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
System.out.println(triB.getAreaSize());
```

최종적으로는 좌측의 코드와 비슷하겠지요?

우리가 임의로 정의한 예외 객체를 발생시키고 이를 상위의 메소드에게 떠 넘기고, try-catch 문이 발생하면 이를 처리하였어요.

예외 처리는 지금은 별로 중요해 보이지 않지만 실제로는 매우 중요한 기술이니까 반드시 숙지해 주세요!

# 끝내기 전에!

만약에 setB가 삼각형을 이루지 못하는 정점들의 Set으로 이루어진 상태에서
위의 코드가 실행되면 어떻게 될까요?

triB는 제대로 된 객체를 생성하지 못하게 되고 그대로 null이 됩니다.

그러다가 getAreaSize 함수를 만났네요!

이런 이런… 어떻게 될까요?

# 우리의 친구 NullPointerException!

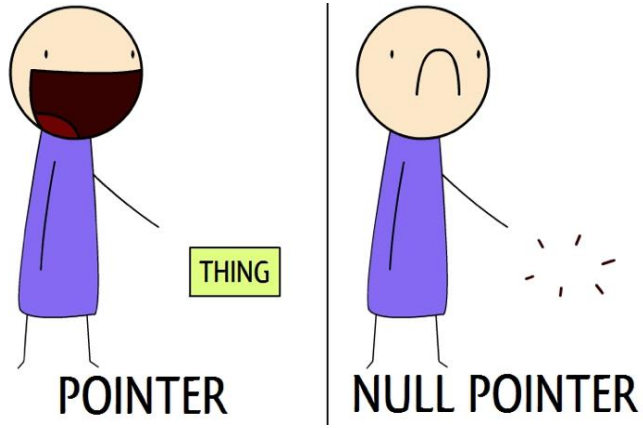
만약 객체가 제대로 생성되지 않은 상태에서 해당 객체의 함수를 호출하게 된다면
Java의 입장에서는 곤욕이 따로 없습니다.

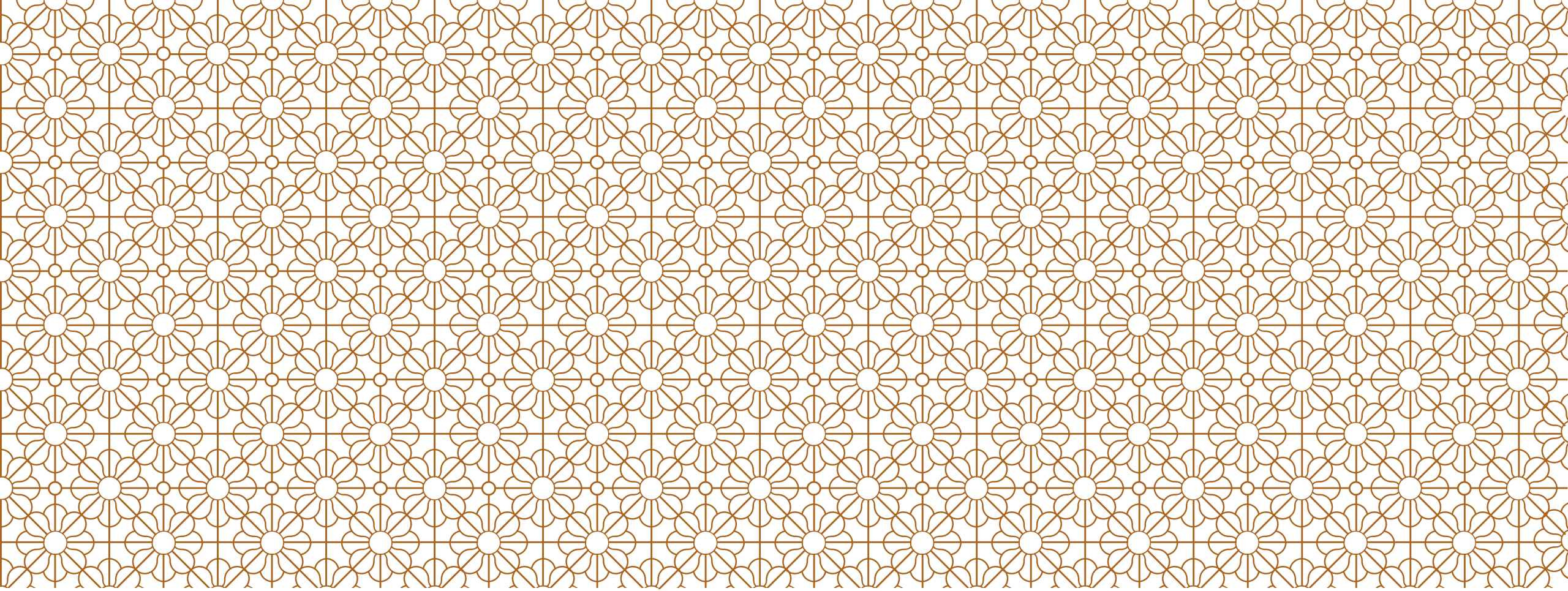
지시한 대로 실행시키려고 하는데 아무것도 없는 겁니다. 즉 null인 것이죠.

이 때 발생하는 Exception이 NullPointerException입니다.

Java 코딩하면서 이 예외를 무수히 많이 보게 될 것입니다.

그 때는 반드시 무엇인가가 제대로 생성이 되지 않은 상태를 의미하니까,
거꾸로 잘 파고들어가 보세요!





# Parameter Passing

Pass-by-Value
Pass-by-Reference
Java Passing Mechanism

# 이번 장은 여러분이 설명해야 할 것 같아요.

C++에서는 함수에 parameter를 넘길 때 다양한 mechanism을 지원합니다.

혹시…

그것들에 대해서 정확히 설명해 주실 수 있나요?

PL을 들었다면 이해하기도 편할 테고 설명도 할 수 있을 거라고 생각합니다.

안 들으셔도 컴프나 객지프때 하지 않나요…?

아닌가요?

# Java의 Parameter Passing

일단 우리는 C/C++이 아니라 Java니까

Java의 parameter passing에 대해서 논해봅시다.


이런 글을 보았습니다.

"Java의 passing mechanism은 primitive type은 pass-by-value, object type은 pass-by-reference이다."

여러분은 동의하나요?

# 이제부터 문제가 나갑니다.

이번 장에는 코딩을 하실 필요가 없습니다.

나오는 문제를 보고 결과가 어떻게 될지 맞춰보세요.

# 문제 1

```
public static void main(String[] args) {
        int i1 = 3;


        reset(i1);
        System.out.println(i1);

}


public static void reset(int i) {
        i = 0;
}
```

콘솔 창의 결과는 무엇인가요?

1. 0
2. 3

# 문제 2

```
public class IntegerHolder {
	public int i;

	public IntegerHolder(int i) {
		this.i = i;
	}

	public IntegerHolder() {
		this.i = 0;
	}
}


public static void main(String[] args) {
	IntegerHolder h1 = new IntegerHolder(3);


	reset(h1);
	System.out.println(h1.i);
}


public static void reset(IntegerHolder holder) {
	holder.i = 0;
}
```

콘솔 창의 결과는 무엇인가요?

1. 0

2. 3

# 문제 3

```
public static void main(String[] args) {
        int i1 = 3;
        int i2 = 5;

        System.out.print("(" + i1 + ", " + i2 + ") -> ");
        swap(i1, i2);
        System.out.println("(" + i1  + ", " + i2 + ")");
}


public static void swap(int i1, int i2) {
        int temp = i1;
        i1 = i2;
        i2 = temp;
}
```

콘솔 창의 결과는 무엇인가요?

1. (3, 5) -> (3, 5)
2. (3, 5) -> (5, 3)
3. (3, 5) -> (3, 3)
3. (3, 5) -> (5, 5)

# 문제 4

```
public static void main(String[] args) {
	IntegerHolder h1 = new IntegerHolder(3);
	IntegerHolder h2 = new IntegerHolder(5);

	System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
	swap(h1, h2);
	System.out.println("(" + h1.i  + ", " + h2.i + ")");
}


public static void swap(IntegerHolder h1, IntegerHolder h2) {
	IntegerHolder temp = h1;
	h1 = h2;
	h2 = temp;
}
```

콘솔 창의 결과는 무엇인가요?

(IntegerHolder는 문제 2와 동일합니다.)

1. (3, 5) -> (3, 5)
2. (3, 5) -> (5, 3)
3. (3, 5) -> (3, 3)
3. (3, 5) -> (5, 5)

# 아니 왜?

Primitive Type은 Call-by-Value고,
Object Type은 Call-by-Reference라매!
근데 왜 안 되는 건데?

C++에서는 어떠냐고요?
다음 C++ 코드를 실행시켜 보세요.

# C++의 경우?

```cpp
#include <cstdio>

void swap(int i1, int i2) {
    int temp = i1;
    i1 = i2;
    i2 = temp;
}

void deepSwap(int& i1, int& i2){
    int temp = i1;
    i1 = i2;
    i2 = temp;
}

int main()
{
    int i1 = 3;
    int i2 = 5;

    printf("(%d, %d) -> ", i1, i2);
    swap(i1, i2);
    printf("(%d, %d)\n", i1, i2);

    printf("(%d, %d) -> ", i1, i2);
    deepSwap(i1, i2);
    printf("(%d, %d)\n", i1, i2);

    return 0;
}
```

결과는 다음과 같습니다.

```
sh-4.3$ gcc -o main main.cpp
sh-4.3$ main
(3, 5) -> (3, 5)
(3, 5) -> (5, 3)
```

C++은 & operator를 활용하여
pass-by-reference로 parameter passing이 가능합니다.

그런데 왜 Java는 되지 않는 걸까요?

그 질문에 답하기 전에 pass-by-value가 무엇인지 알아봅시다.

# Pass-by-value

Parameter passing 중에서 가장 보편적인 친구입니다.

Subroutine을 실행할 때 각각의 parameter에 값을 넣습니다.

여기서 '값'을 넣는다는 의미는 말 그대로
받은 변수의 값을 그대로 복사한다는 의미입니다.

이해가 잘 되지 않으신다고요?

문제 1번을 가지고 한 번 더 설명해 드릴게요.

메모리 구조와 완전히 동일하지는 않습니다만,
이해하는 데 도움이 될 것이라고 생각합니다.

# 문제 1 다시 보기

청록색 공간이 할당된 메모리 공간입니다.

노란색 공간은 지역 변수가 할당된 것을 의미합니다.

주황색 부분이 현재까지 읽은 부분입니다.

그러면 메모리 공간이 오른쪽과 같을 것입니다.

```java
public static void main(String[] args) {
    int i1 = 3;

        reset(i1);
        System.out.println(i1);

}

public static void reset(int i) {
        i = 0;
}
```

# 문제 1 다시 보기

이제 reset 함수로 들어가게 되면 reset을 위한 공간을 할당받게 됩니다.

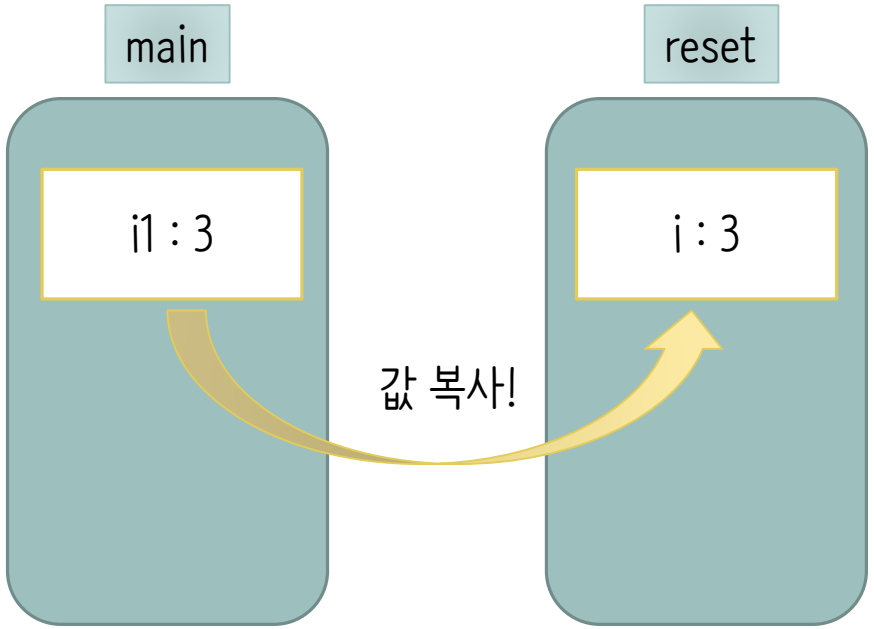
이 때 reset의 메모리 공간 안에는 parameter i에 해당하는 공간이 있을 것입니다.

또한 call-by-value이므로 i3의 값이 곧장 i로 복사됩니다.

이는 노란 화살표로 표시하겠습니다.

좀 더 정확히는 0과 1로 이루어진 4byte겠지요?

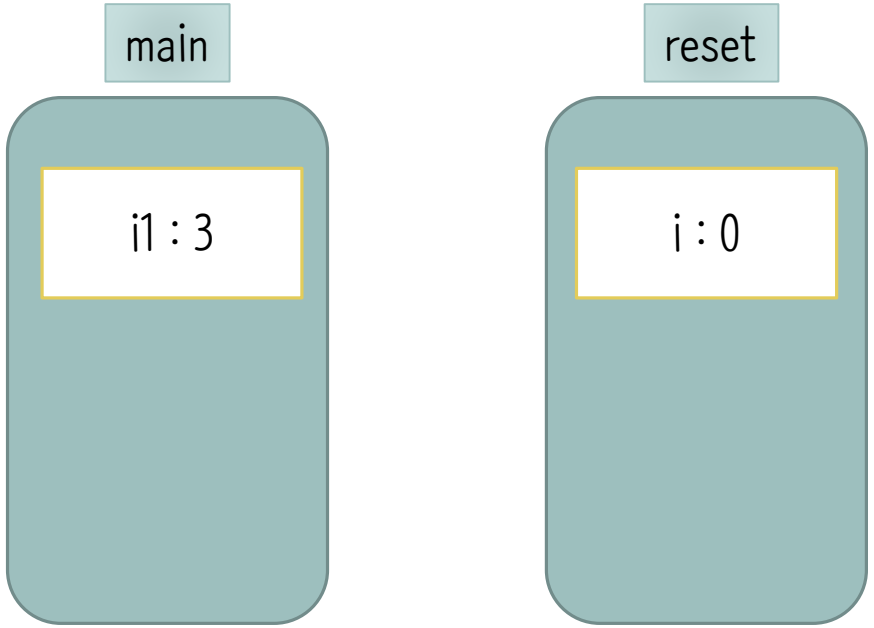
```
public static void main(String[] args) {
        int i1 = 3;


        reset(i1);
        System.out.println(i1);

}


public static void reset(int i) {
        i = 0;
}
```



# 문제 1 다시 보기

이러니 reset함수에서 아무리 i를 0으로 바꿔봐도 변하지 않겠지요.

```java
public static void main(String[] args) {
        int i1 = 3;


        reset(i1);
        System.out.println(i1);

}


public static void reset(int i) {
        i = 0;
}
```



# 문제 1 다시 보기

reset 함수가 끝나면 메모리에 할당되었던 모든 관련된 부분이 소멸됩니다.

i1은 바뀌지 않은 채로 말이죠.

```java
public static void main(String[] args) {
        int i1 = 3;

        reset(i1);
        System.out.println(i1);
}

public static void reset(int i) {
        i = 0;
}
```

# 그럼 문제 2는요?

충분히 궁금하신 부분이 아닌가요?

왜 문제 2에서는 값이 바뀐 것일까요?

그 이유는 바로 Java에서 모든 객체는 해당 객체가 할당된 주소값을 가지기 때문입니다.

그러니까… 쉽게 말해서… 포인터 같달까요?

문제 2를 같이 풀어보도록 하겠습니다.

사실 이 부분의 메모리 구조는 개인적인 견해입니다.

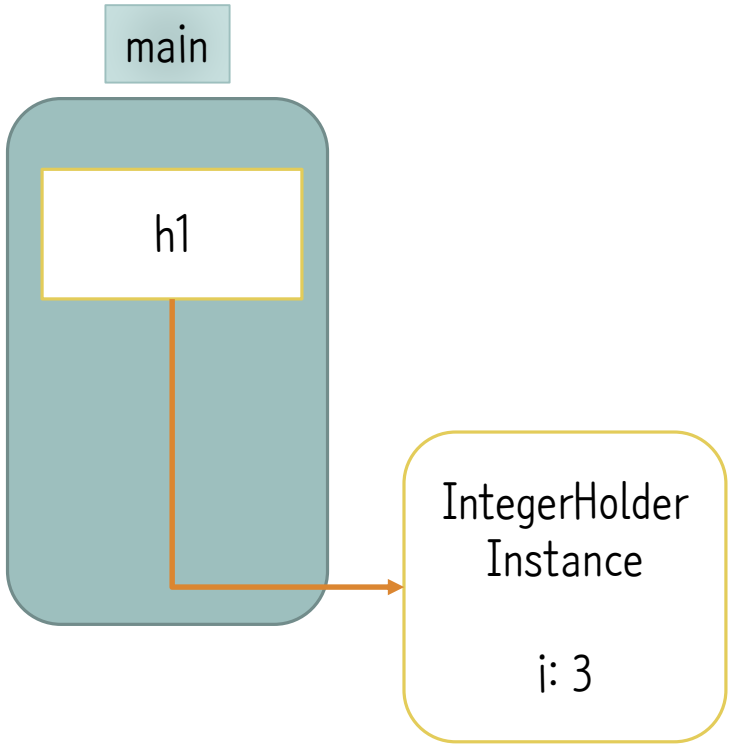
Java는 C/C++과 다르게 내부 메모리 구조를 파악하는 것이 거의 불가능하기 때문입니다.

# 문제 2 다시 보기

```java
public static void main(String[] args) {
    IntegerHolder h1 = new IntegerHolder(3);

    reset(h1);
    System.out.println(h1.i);
}

public static void reset(IntegerHolder holder) {
    holder.i = 0;
}
```

저는 메모리 구조가 이런 형식일 것이라고 생각하고 있습니다.

즉 main에서 h1이 할당받은 공간에는 사실은 IntegerHolder의 주소값이 들어가 있고, 실제 인스턴스는 다른 곳에 구현되어 있는 것이죠.



# 문제 2 다시 보기

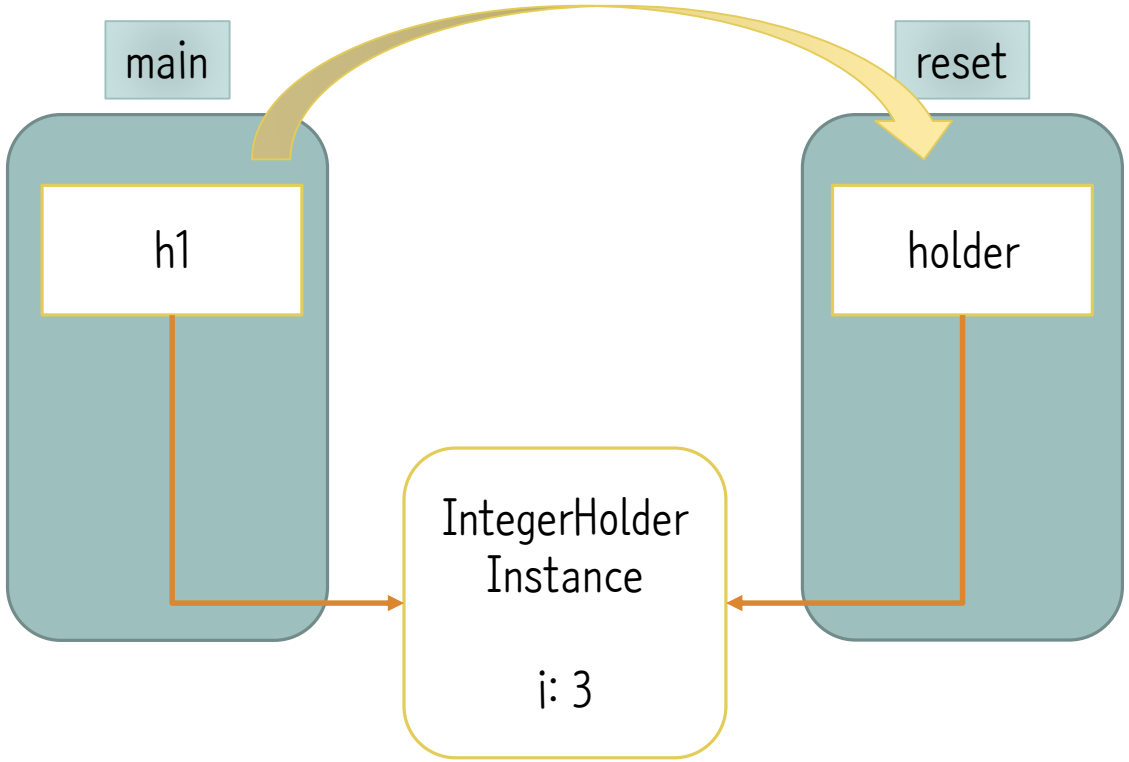
```
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);

        reset(h1);
        System.out.println(h1.i);
}

public static void reset(IntegerHolder holder) {
        holder.i = 0;
}
```

자 reset함수를 호출하게 되면,
holder라는 변수에 값이 들어가야 하겠지요?

이 값은 바로 main 안의 h1의 값!
즉, Instance의 주소값이 되는 것입니다.

즉 main의 h1과 reset의 holder가 같은 주소 공간을 보고 있는 것이지요.

이 상태를 alias(별명)이라고 합니다.

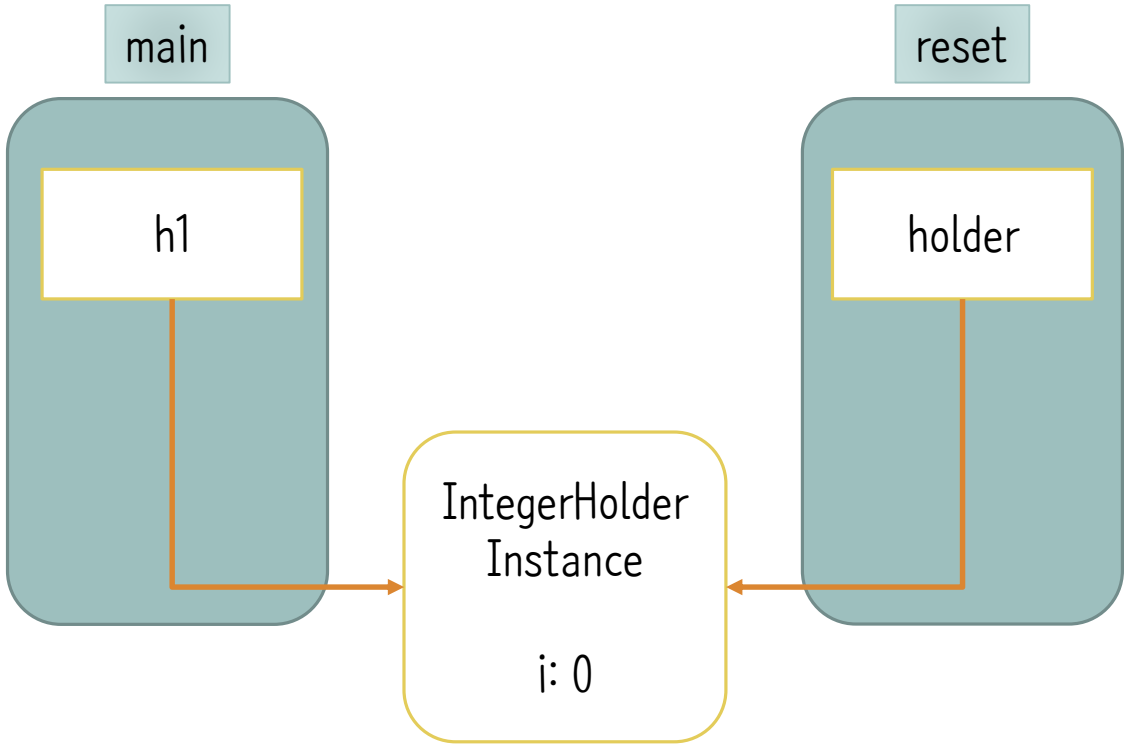


# 문제 2 다시 보기

이제 holder의 i를 바꿔 봅시다.

그러면 main의 h1도 똑같이 바뀌겠지요?

```
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);


        reset(h1);
        System.out.println(h1.i);
}

public static void reset(IntegerHolder holder) {
    holder.i = 0;
}
```

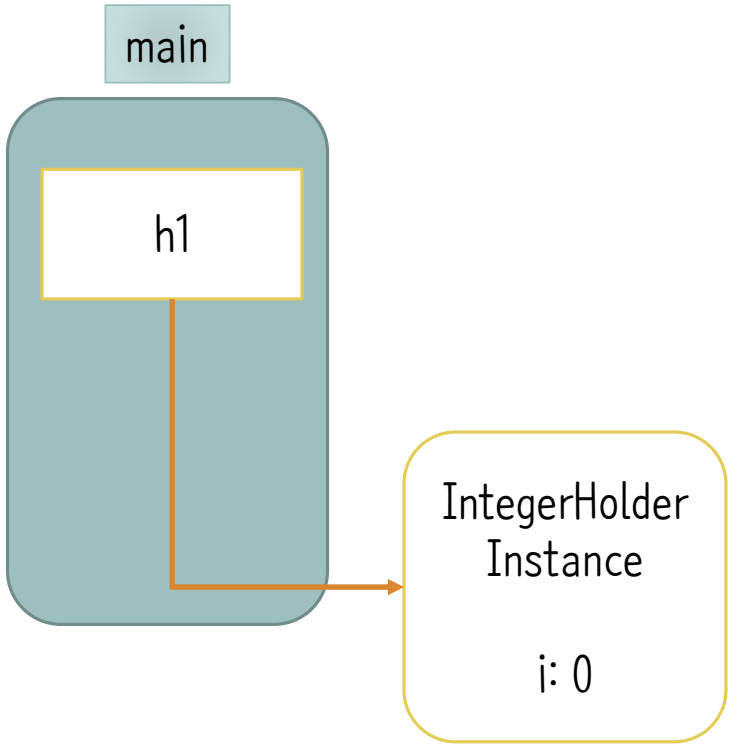


# 문제 2 다시 보기

```java
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);


        reset(h1);
        System.out.println(h1.i);
}

public static void reset(IntegerHolder holder) {
    holder.i = 0;
}
```

이 상태로 reset이 끝나면 할당되었던 모든 주소 공간이 해제됩니다.

하지만 우리는 alias를 통해서 이미 h1의 i를 0으로 바꿨습니다.



# 문제 2 다시 보기

좀 이해가 되시나요?
객체는 덩어리가 크기 때문에 그 전체를 복사해서 다시 할당하는 것은 매우 비효율적입니다.
대신 객체를 주소값으로 관리한다면 parameter를 주고 받을 때 훨씬 간단하겠지요?
때문에 이런 상황이 가능한 것입니다.

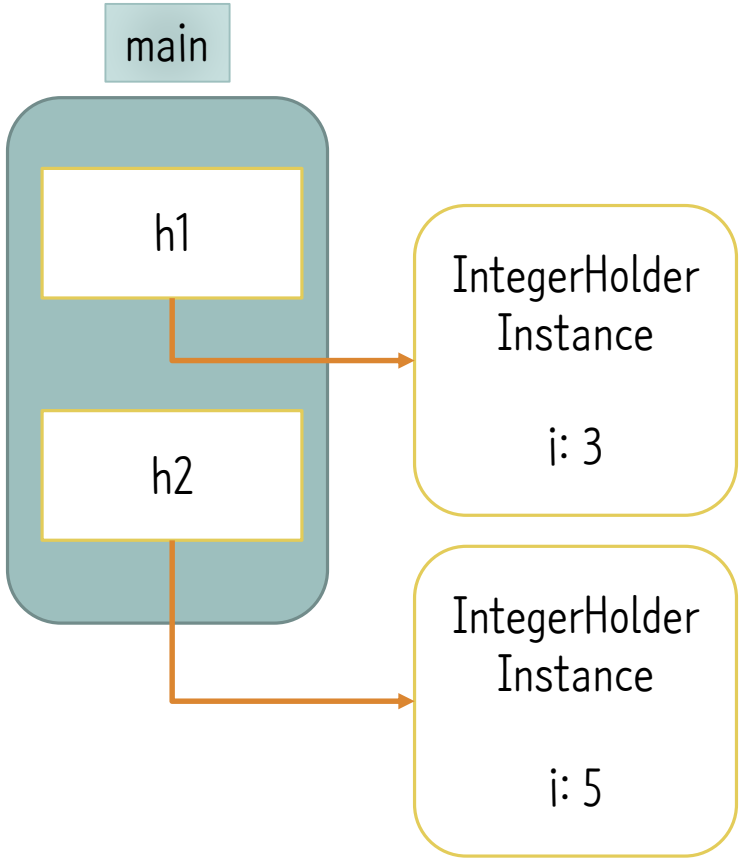
여기서 중요한 것은 object type도 분명히 pass-by-value로 넘겼다는 점입니다.
문제 3은 쉬우니까 이대로 문제 4를 한 번 같이 풀어보죠.

# 문제 4 다시 보기

h2까지 읽었을 때의 메모리 상황입니다.

두 개의 서로 다른 객체 h1과 h2가 있고 main이 할당받은 공간 안에서는 각각의 instance의 주소값을 가지고 있습니다.

```
public static void main(String[] args) {
    IntegerHolder h1 = new IntegerHolder(3);
    IntegerHolder h2 = new IntegerHolder(5);

    System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
    swap(h1, h2);
    System.out.println("(" + h1.i  + ", " + h2.i + ")");
}

public static void swap(IntegerHolder h1, IntegerHolder h2) {
    IntegerHolder temp = h1;
    h1 = h2;
    h2 = temp;
}
```



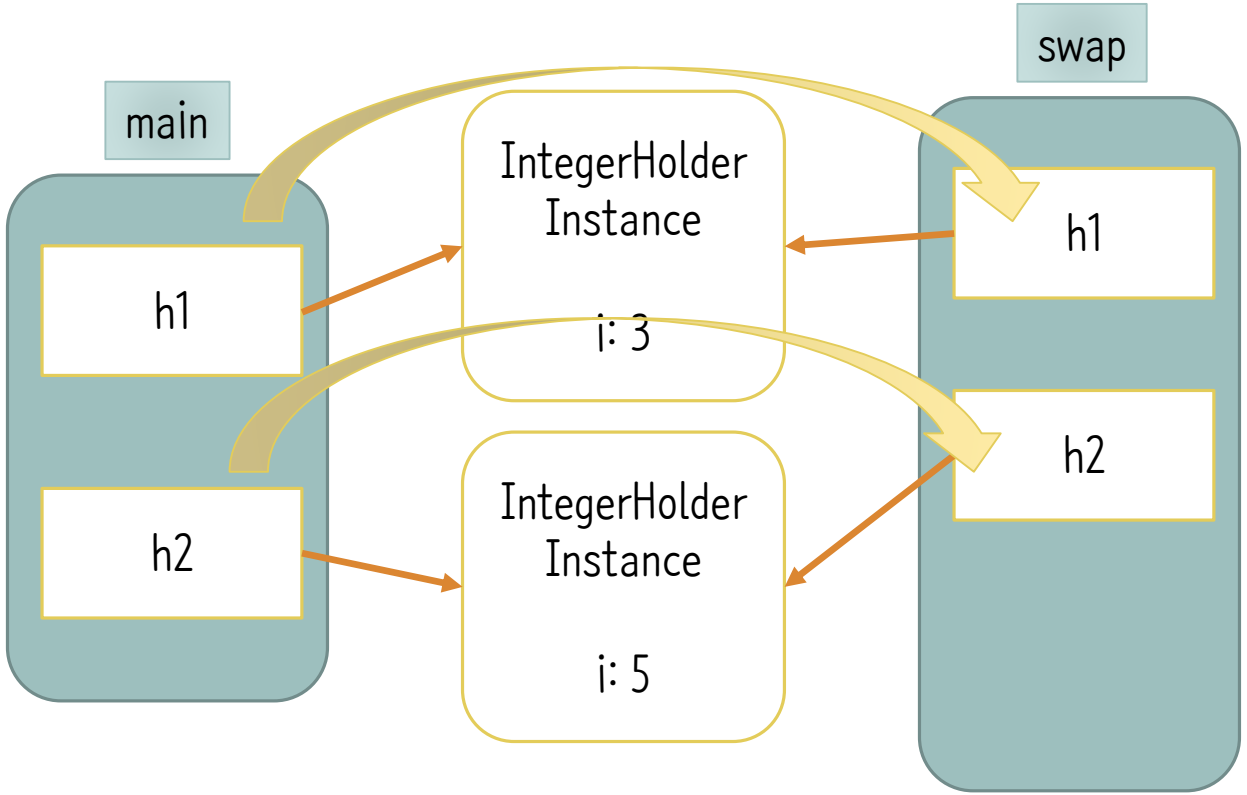
# 문제 4 다시 보기

이제 swap까지 들어왔습니다.

swap의 h1에는 main h1의 값인 위에 있는 instance의 주소값이 swap h2는 main h2의 값인 아래 instance의 주소값이 들어갑니다.

여기서 중요한 것은 main h1과 swap h1, 그리고 main h2와 swap h2가 변수 이름은 같지만 메모리 공간이 엄연히 다른 것임을 인지하세요!

```java
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);
        IntegerHolder h2 = new IntegerHolder(5);

        System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
        swap(h1, h2);
        System.out.println("(" + h1.i  + ", " + h2.i + ")");
}

public static void swap(IntegerHolder h1, IntegerHolder h2) {
        IntegerHolder temp = h1;
        h1 = h2;
        h2 = temp;
}
```



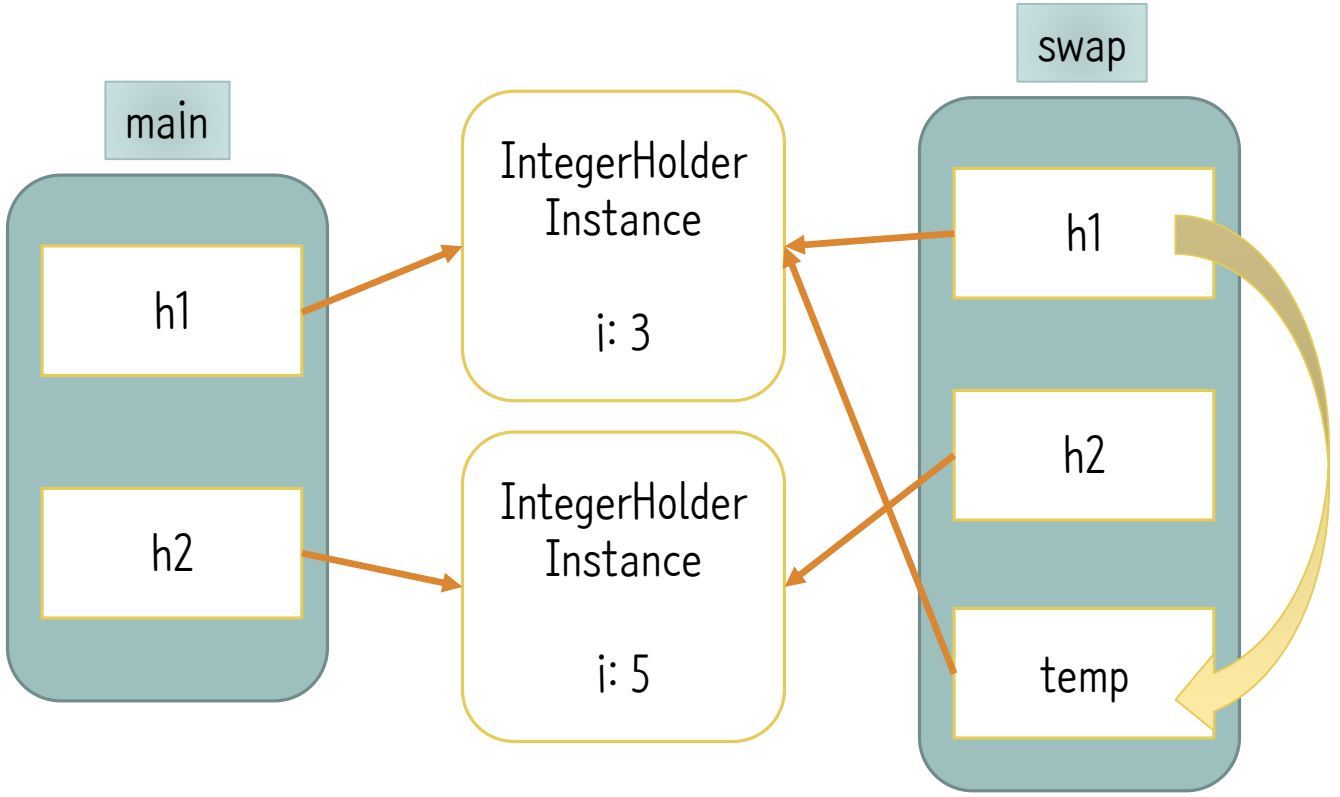
# 문제 4 다시 보기

temp에 h1의 값을 넣습니다.

그러면 temp도 h1이 보는 것을 같이 바라보겠지요?

현재 위의 instance를 바라보는 변수가 총 3개나 됩니다.

```
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);
        IntegerHolder h2 = new IntegerHolder(5);

        System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
        swap(h1, h2);
        System.out.println("(" + h1.i  + ", " + h2.i + ")");
}

public static void swap(IntegerHolder h1, IntegerHolder h2) {
        IntegerHolder temp = h1;
        h1 = h2;
        h2 = temp;
}
```

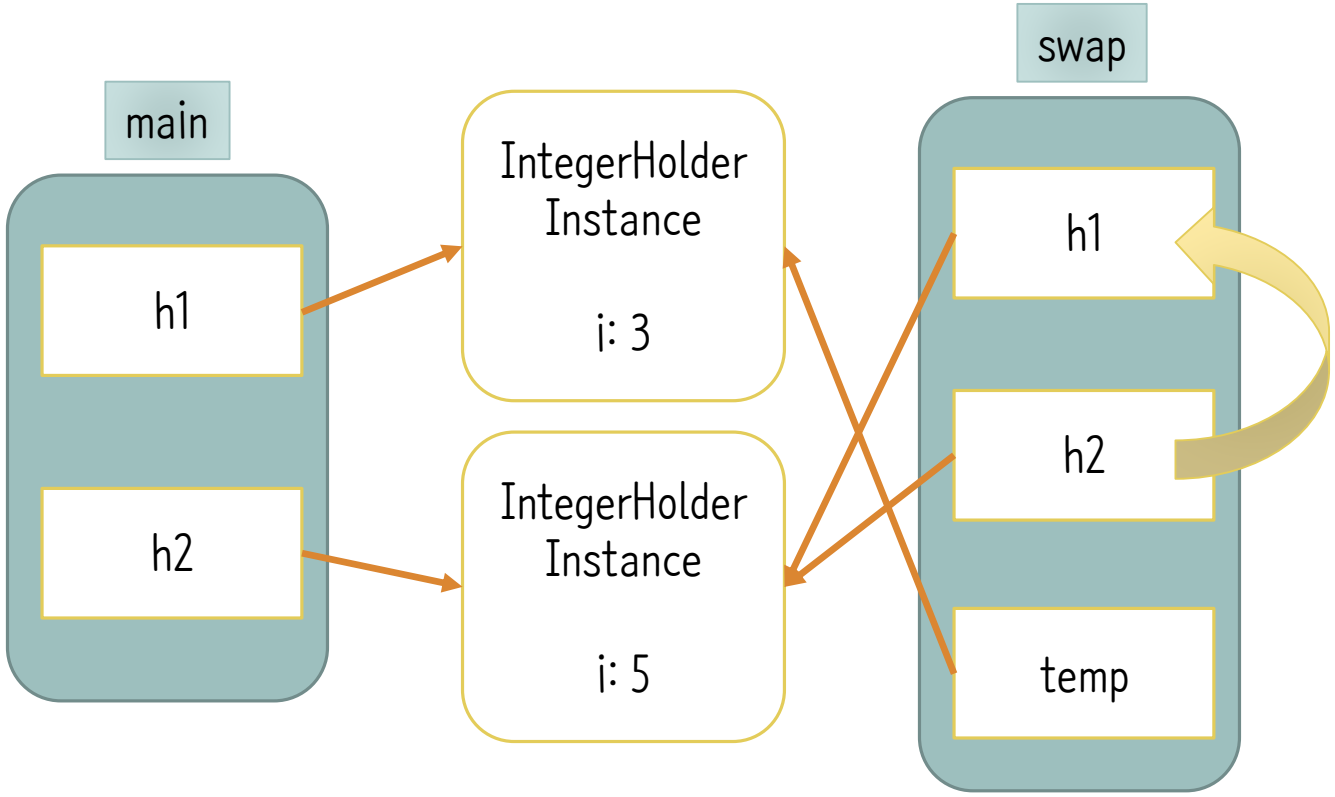


# 문제 4 다시 보기

h1에 h2의 값을 넣습니다.
그러면 어떻게 될까요?

그러면 원래 swap의 h2가 바라보던 것을
swap의 h1도 함께 바라보겠네요?

```
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);
        IntegerHolder h2 = new IntegerHolder(5);

        System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
        swap(h1, h2);
        System.out.println("(" + h1.i  + ", " + h2.i + ")");
}

public static void swap(IntegerHolder h1, IntegerHolder h2) {
        IntegerHolder temp = h1;
        h1 = h2;
        h2 = temp;
}
```



# 문제 4 다시 보기

이제 마지막입니다.
h2에 temp의 값을 넣겠습니다.

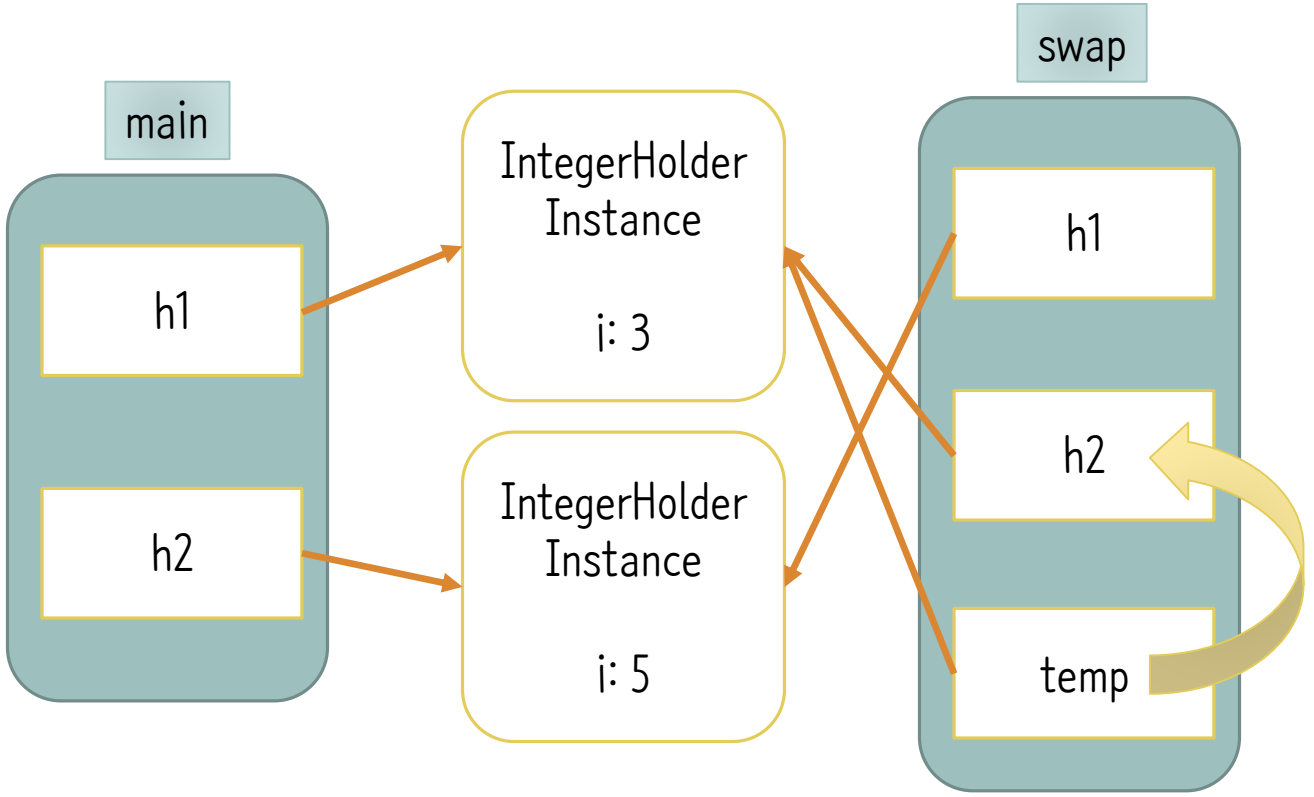
그러면 temp가 보는 것을 h2가 같이 보게 되겠지요?

swap의 내부에서는 h1과 h2가 보는 것이 서로 바뀐 것을 확인할 수 있습니다.

하지만… main은 어떻지요?

이 상태로 swap이 종료되어도
전혀 바뀌지 않았습니다.

```java
public static void main(String[] args) {
        IntegerHolder h1 = new IntegerHolder(3);
        IntegerHolder h2 = new IntegerHolder(5);

        System.out.print("(" + h1.i + ", " + h2.i + ") -> ");
        swap(h1, h2);
        System.out.println("(" + h1.i  + ", " + h2.i + ")");
}

public static void swap(IntegerHolder h1, IntegerHolder h2) {
        IntegerHolder temp = h1;
        h1 = h2;
        h2 = temp;
}
```



# call-By-Reference는 뭐예요?

지금까지의 설명 중에서 제일 중요한 것은 Java가 pass-by-value를 사용한다는 점입니다.
그러니까 object type은 pass-by-reference한다는 거 다 뻥이에요.
이 부분이 실제 코딩할 때 헷갈리기 쉬운 부분이니 조심하시기 바라요.

그러면 call-by-reference는 무엇이냐?
그 질문에 대한 답은 이 슬라이드의 난이도를 너무 높일 것 같아요.
간략하게 설명드리면 callee의 parameter의 값이 변하면 caller의 변수 값도 같이 변하는 것입니다.
만약 더 깊게 알고 싶으신 용자가 있으시면 끝나고 물어보세요.

수고하셨습니다