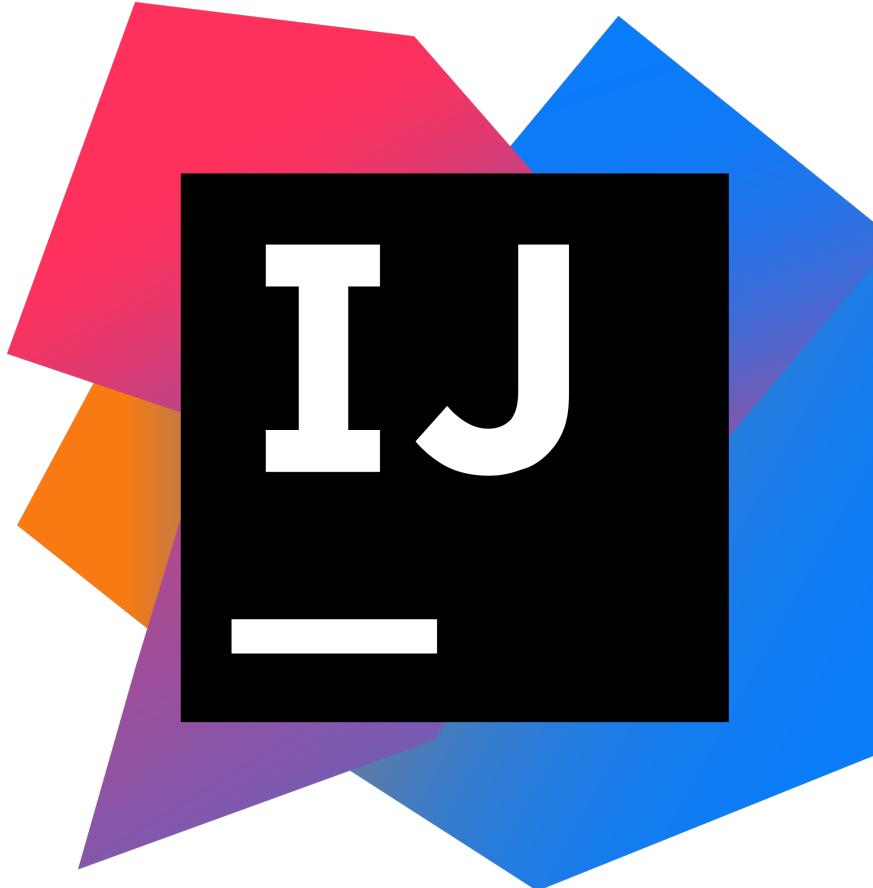


Lec 00. 코틀린에 관한 34가지 사실

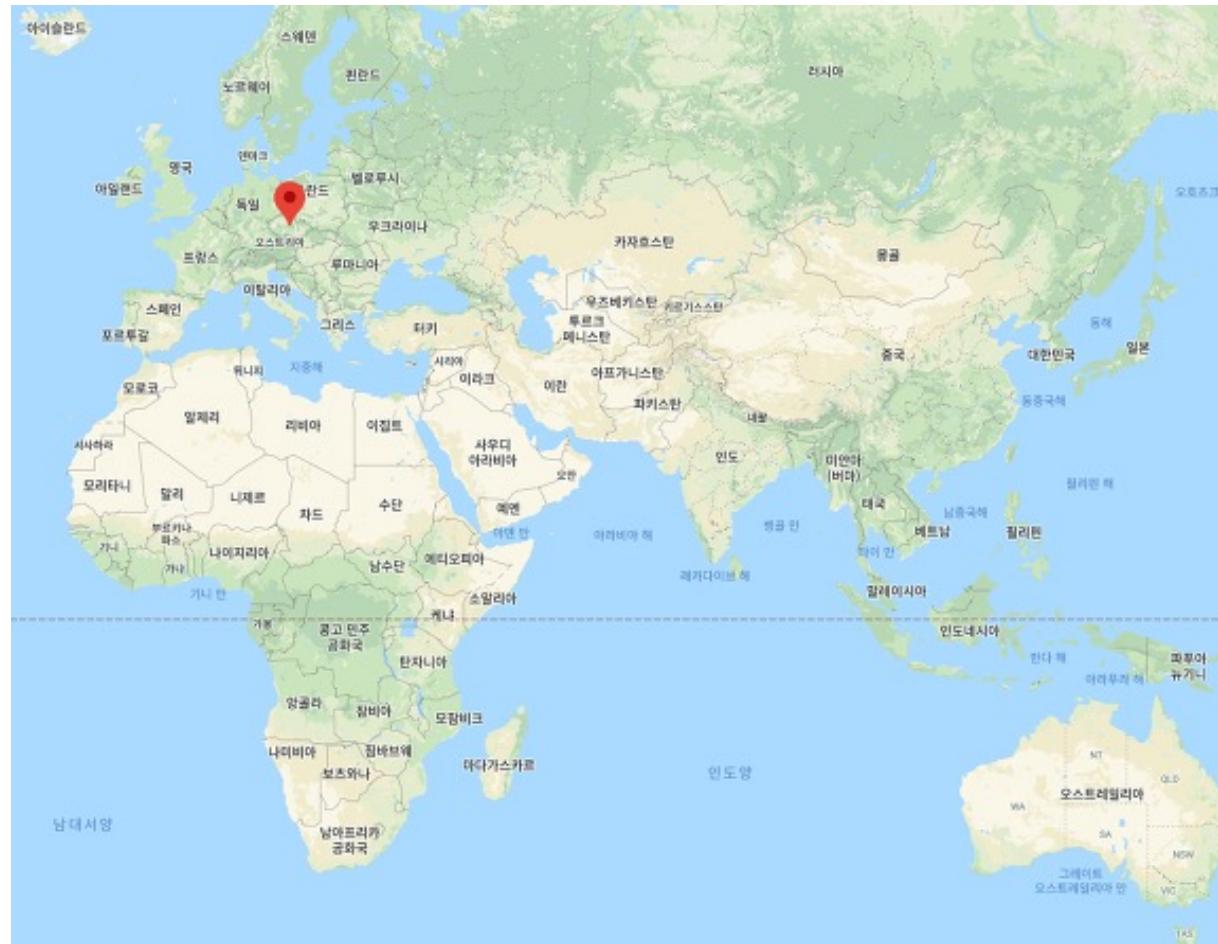
#1. 코틀린이라는 프로그래밍 언어는 IntelliJ를 만든 JetBrains라는 회사에서 만들었다.



#2. IntelliJ는 Java 및 Kotlin을 이용한 프로젝트에서 널리 사용되는 통합개발환경(IDE)이다.



#3. JetBrains는 체코의 회사이다.



#4. JetBrains는 IntelliJ 외에도 PyCharm, WebStorm 등
python, JS를 위한 IDE도 만들고 있으며 많이 사용되는 추세라고...

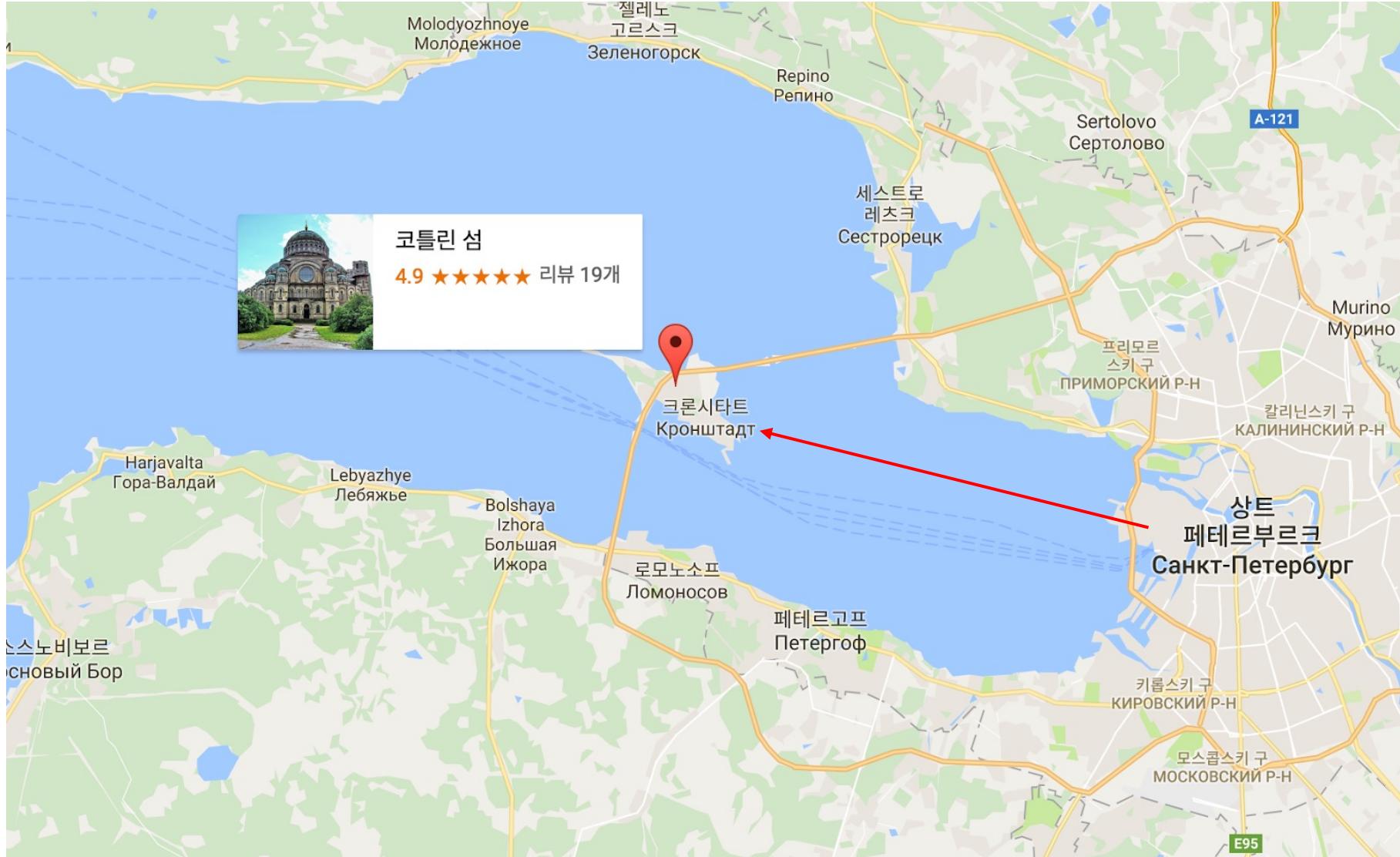


PyCharm



WebStorm

#5. Kotlin이라는 이름은 코틀린(Котлин) 섬에서 따왔는데 이는 JetBrains의 R&D 센터가 상트페테르부르크에 있기 때문이다.



#6. Kotlin이 구동되는 JVM의 대표언어 Java가 인도네시아 Java(Jawa) 섬에서 이름을 따왔음을 감안해보면 타겟으로 하는 언어가 분명한 셈.



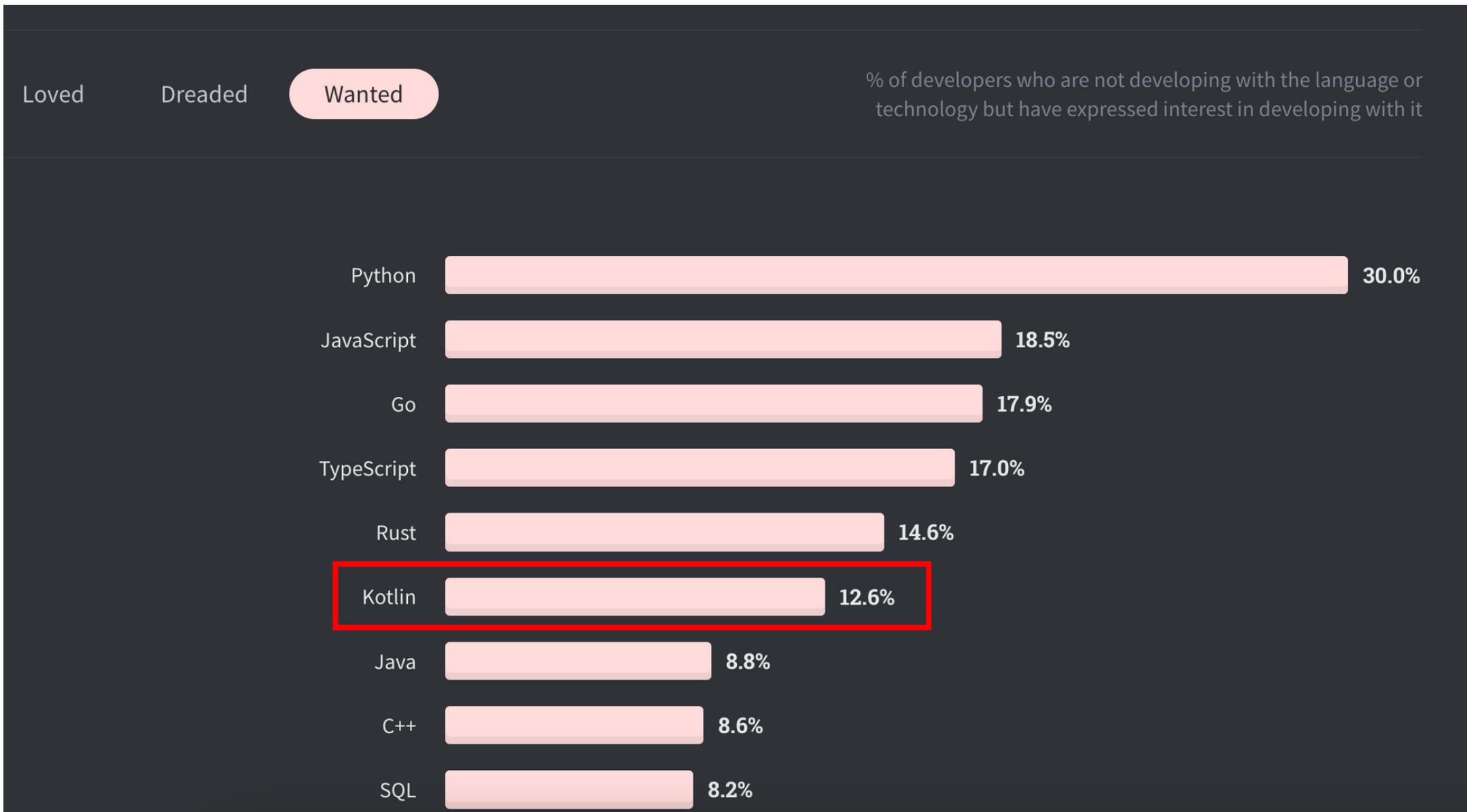
#7. 코틀린 섬은 원래 스웨덴령이었지만, 1703년 포트르 대제가 빼앗아 현재까지 러시아의 섬이 되었다.



#8. 코틀린은 2020년 stack over flow에서 조사한 '개발자들이 가장 사랑하는 언어' 4위에 랭크된 적이 있다.



#9. 같은 해 조사된, 배우고 싶은 언어에는 6위에 랭크되었다.



#10. 2021년 12월 프로그래머스에서 조사한 설문결과에서 코틀린은 가장 배우고 싶은 언어 1위를 차지하였다.

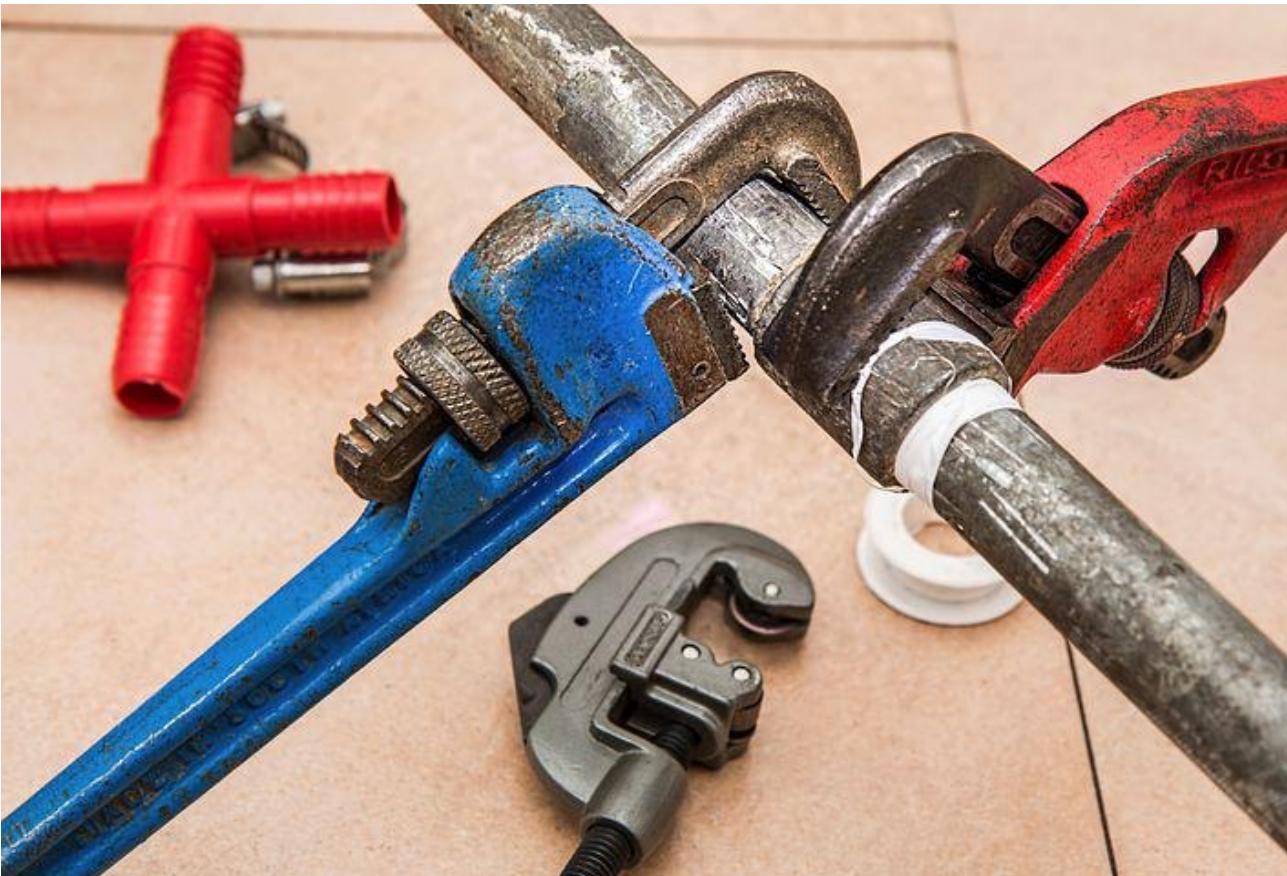
5

가장 배우고 싶은 언어 **Kotlin**

- Kotlin은 Python과 Go를 제치고 가장 배우고 싶은 언어 1위에 올랐습니다.

#11. 코틀린은 Java와 100% 호환 가능하면서도 현대적이고, 간결하며 안전한 언어를 사용하기 위해 탄생되었다.

JetBrains에서 만드는 IntelliJ가 Java로 작성되어 있는데, 유지보수 하다가 화가 났다고...



#12. 코틀린 언어 공식 홈페이지에서 코틀린의 철학을 엿볼 수 있다.

Why Kotlin

현대적인

Modern,

간결한

concise and safe **안전한**
programming language

#13. 코틀린은 Java와 100% 호환 가능하기 때문에 JVM 위에서 동작한다.



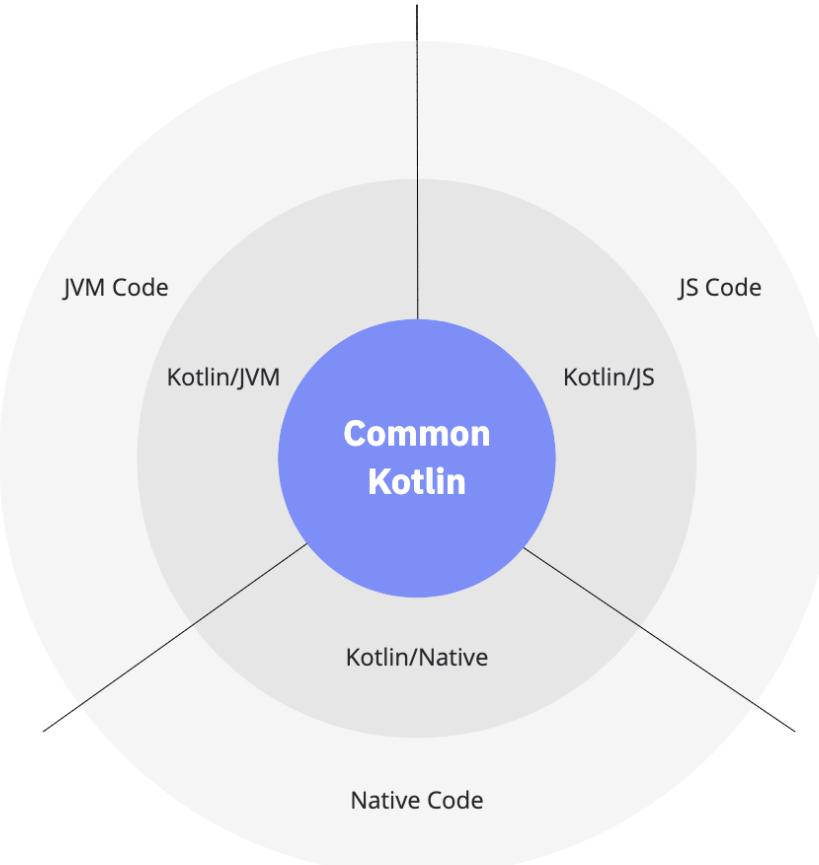
JVM

(Java Virtual Machine)

#14. JVM 위에서 동작하는 언어에는 Java와 Kotlin 말고도
Scala, Groovy 등이 있다.



#15. 코틀린은 멀티 플랫폼 언어로 Android 앱개발 / IOS 앱개발 / 서버 개발 / 웹 개발 / 임베디드와 IoT / 데스크톱까지 다양한 플랫폼과 Data Science 까지 사용되는 것을 목표로 하고 있다.



#16. 하지만 현재까지 가장 많이 사용되는 곳은 Android 앱 개발과
서버 개발이다.



#17. Android의 개발사인 구글은 2017년 안드로이드 공식 언어로 코틀린을 추가했으며, 2019년부터는 Kotlin First를 외치며 공식 문서 샘플 코드를 Java에서 Kotlin 우선으로 변경했다.



#18. 또한 이때부터 구글에서 작성하는 Android 프로젝트는 코틀린으로 작성되었다고 한다.



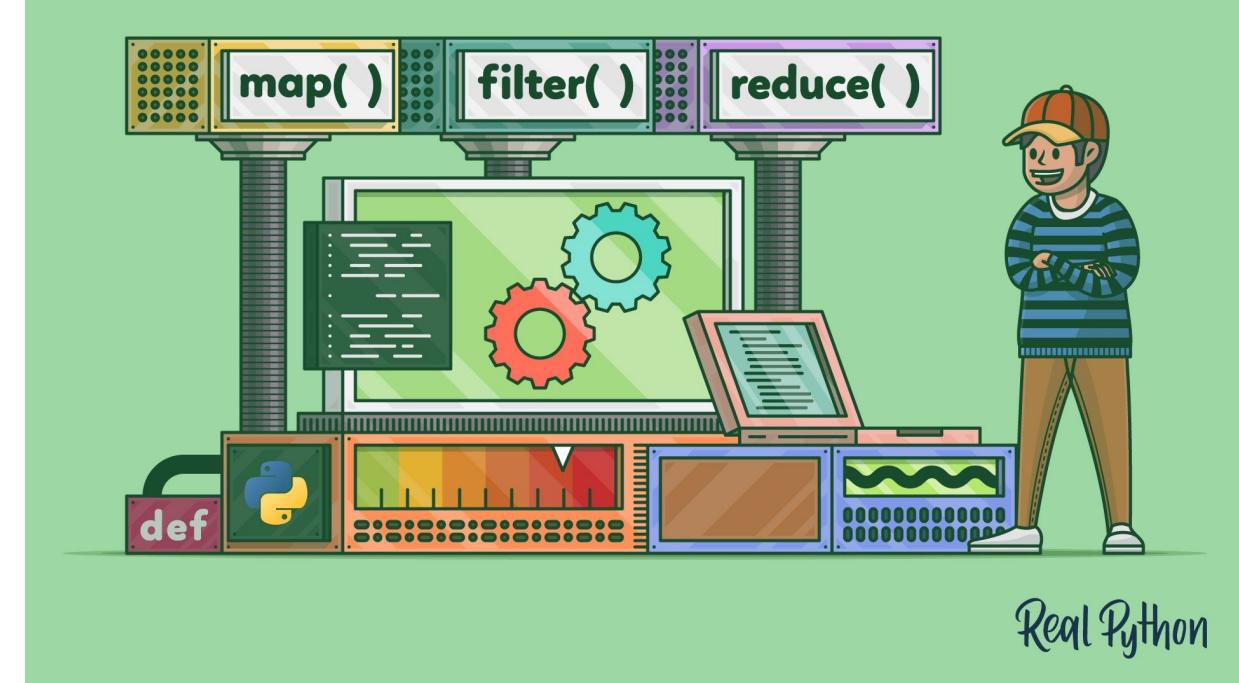
#19. 코틀린은 정적 타입 언어이다. 프로그램 구성 요소의 타입을 컴파일 시점에 알 수 있고, 프로그램 안에서 필드나 메소드를 사용할 때 컴파일러가 타입을 검증해준다는 뜻이다.

```
var str: String = "ABC"  
str = 3
```

#20. 코틀린은 객체지향형 프로그래밍(OOP)과 함수형 프로그래밍(FP)를 조화롭게 지원하고 있다.



Real Python



Real Python

#21. 코틀린은 무료 오픈소스로 아파치2.0 라이센스로 가지고 있다.



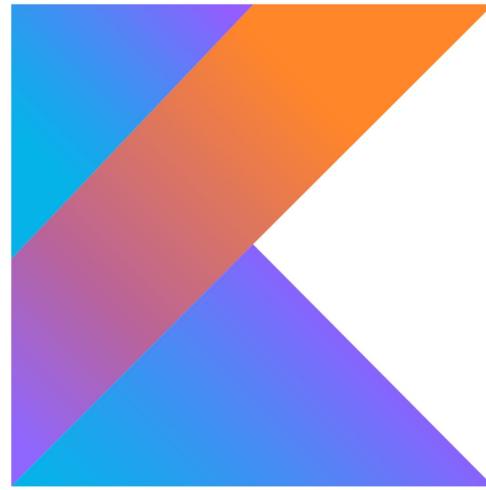
#22. 아파치 2.0 라이센스는 소스코드 공개 의무가 존재하지 않으며, 상업적 이용에 제한을 두고 있지도 않다. 다시 말해, 코틀린을 사용해 만들어진 프로그램은 소스 코드를 공개하지 않고 상업적 이용을 해도 된다는 의미이다.



#23. 코틀린 언어 개발자들은, 코틀린 언어의 간결함을 살리기 위해
프로그래머가 작성하는 코드에서 의미 없는 부분은 줄이고,
언어가 요구하는 구조를 만족시키기 위해 별 뜻은 없지만 프로그램에 꼭
넣어야 하는 부수적인 요소를 줄이기 위해 많은 노력을 하였다.



#24. Kotlin의 파일 확장자는 .kt 이다.



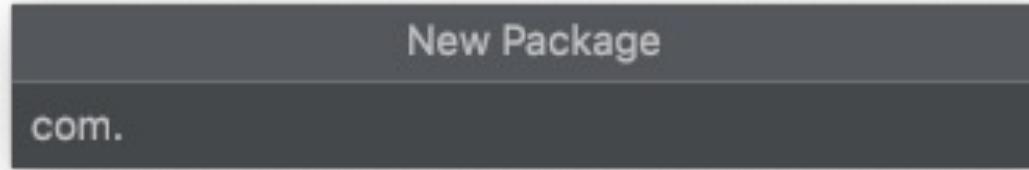
#25. Kotlin에서는 Java와 달리, 세미콜론을 붙이지 않아도 된다.

```
println("A");  
println("A")
```

#26. Kotlin에서 주석을 처리하는 방법은 Java와 동일하다.

```
// 한 줄 주석  
/**  
 * 여러 줄 주석  
 */
```

#27. Kotlin에는 Java와 동일하게 패키지라는 개념이 있다.



#28. Kotlin에서는 별도 지시어를 붙이지 않으면 모두 public이다

```
public fun function1() {  
}  
  
fun function2() {  
}
```

#29. Kotlin에서는 출력을 할 때에 System.out.println() 대신 println()만 작성하면 된다.

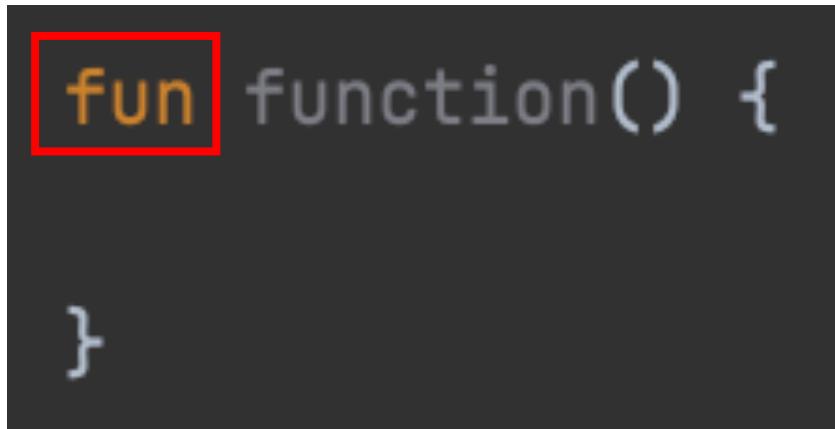


```
System.out.println("Hello World");
```



```
println("Hello World")
```

#30. Kotlin에서는 함수를 작성할 때 fun이라는 키워드를 사용한다.
fun이라는 단어는 재미있는 이라는 뜻도 가지고 있어 가끔
fun fun (재미있는 함수) 라는 외국 드립이 보일 때도 있다.



```
fun function() {  
}
```

#31. Java에서는 '타입 변수명'을 사용했지만, 코틀린에서는 TS와 유사하게 '변수명: 타입'을 사용한다.



```
int number = 100;
```



```
val number: Int = 100
```

#32. Kotlin의 Hello World는 다음과 같다.

```
fun main() {  
    println("Hello World")  
}
```

#33. Kotlin에서는 변수나 함수 클래스 모두 파일 최상단에 선언할 수 있다.

```
val a = 3

fun function() {

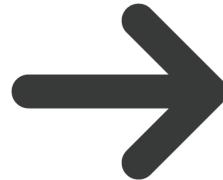
}

class Cat()
```

#34. Java를 알고 있는데 Kotlin을 쉽고 빠르게 배우고 싶으신 분들을 위한 강의가 인프런에 존재한다.



Java



Kotlin

Lec 01. 코틀린에서 변수를 다루는 방법

1. 변수 선언 키워드 - var과 val의 차이점
2. Kotlin에서의 Primitive Type
3. Kotlin에서의 nullable 변수
4. Kotlin에서의 객체 인스턴스화

1. 변수 선언 키워드 - var과 val의 차이점



1. 변수 선언 키워드 - var과 val의 차이점

```
long number1 = 10L; // (1)  
final long number2 = 10L; // (2)
```

```
Long number3 = 1_000L; // (3)
```

```
Person person = new Person(name: "최태현"); // (4)
```

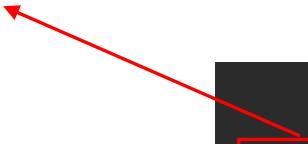
1. 변수 선언 키워드 - var과 val의 차이점

Java에서 long과 final long의 차이..!

이 변수는 가변인가, 불변인가(read-only)

1. 변수 선언 키워드 - var과 val의 차이점

Variable 의 약자 / 발



```
var number1 = 10L
val number2 = 10L
```

1. 변수 선언 키워드 - var과 val의 차이점

```
var number1 = 10L  
val number2 = 10L
```



Value 의 약자 / 밸

1. 변수 선언 키워드 - var과 val의 차이점

```
var number1 = 10L  
val number2 = 10L
```

코틀린에서는 모든 변수에
수정 가능 여부(var / val)를 명시해주어야 한다.

1. 변수 선언 키워드 - var과 val의 차이점

```
var number1: Long = 10L  
val number2: Long = 10L
```

타입을 명시적으로 작성해줄 수도 있다.

1. 변수 선언 키워드 - var과 val의 차이점

초기값을 지정해주지 않는 경우는?!

```
var number: Int  
println(a) // 컴파일 에러 발생, Variable 'a' must be initialize
```

1. 변수 선언 키워드 - var과 val의 차이점

val 컬렉션에는 element를 추가할 수 있다

1. 변수 선언 키워드 - var과 val의 차이점

간단한 TIP

모든 변수는 우선 val로 만들고 꼭 필요한 경우 var로 변경한다

2. Kotlin에서의 Primitive Type

```
long number1 = 10L; // (1)  
final long number2 = 10L; // (2)  
  
Long number3 = 1_000L; // (3)  
Person person = new Person(name: "최태현"); // (4)
```

long은 primitive type / Long은 reference type

2. Kotlin에서의 Primitive Type

연산을 사용할 때는 Reference Type 대신 Primitive Type을 사용해야 한다

```
var number1: Long = 10L  
val number2: Long = 10L
```

2. Kotlin에서의 Primitive Type

```
var number1: Long = 10L  
val number2: Long = 10L
```

오잉 Kotlin에서는 모두 Long 인데?! 성능상 문제 없는것인가?

2. Kotlin에서의 Primitive Type

코틀린 공식 문서

Some types can have a special internal representation - for example, numbers, characters and booleans - can be represented as primitive values at runtime - but to the user they look like ordinary classes.

2. Kotlin에서의 Primitive Type

숫자, 문자, 불리언과 같은 몇몇 타입은
내부적으로 특별한 표현을 갖는다.

이 타입들은 실행시에 Primitive Value로 표현되지만,
코드에서는 평범한 클래스처럼 보인다.

2. Kotlin에서의 Primitive Type

즉, 프로그래머가 boxing / unboxing을 고려하지 않아도 되도록
Kotlin이 알아서 처리 해준다.

3. Kotlin에서의 nullable 변수

```
long number1 = 10L; // (1)
```

```
final long number2 = 10L; // (2)
```

```
Long number3 = 1_000L; // (3)
```

```
Person person = new Person(name: "최태현"); // (4)
```

3. Kotlin에서의 nullable 변수

```
var number3: Long? = 1_000L  
number3 = null
```

Kotlin에서 null이 변수에 들어갈 수 있다면 “타입?”를 사용해야 한다.

4. Kotlin에서의 객체 인스턴스화

```
long number1 = 10L; // (1)
final long number2 = 10L; // (2)

Long number3 = 1_000L; // (3)
Person person = new Person(name: "최태현"); // (4)
```

4. Kotlin에서의 객체 인스턴스화

```
val person = Person(name: "최태현")
```

Kotlin에서 객체 인스턴스화를 할 때에는 new를 붙이지 않아야 한다.

Lec 01. 코틀린에서 변수를 다루는 방법

- 모든 변수는 var / val을 붙여 주어야 한다.
 - var : 변경 가능하다 / val : 변경 불가능하다 (read-only)
- 타입을 명시적으로 작성하지 않아도, 타입이 추론된다.
- Primitive Type과 Reference Type을 구분하지 않아도 된다.
- Null이 들어갈 수 있는 변수는 타입 뒤에 ? 를 붙여주어야 한다.
 - 아예 다른 타입으로 간주된다.
- 객체를 인스턴스화 할 때 new를 붙이지 않아야 한다.

Lec 02. 코틀린에서 null을 다루는 방법

1. Kotlin에서의 null 체크
2. Safe Call과 Elvis 연산자
3. 널 아님 단언!!
4. 플랫폼 타입

1. Kotlin에서의 null 체크



1. Kotlin에서의 null 체크

```
public boolean startsWithA(String str) {  
    return str.startsWith("A");  
}
```

이 코드는 안전한 코드일까요?!

1. Kotlin에서의 null 체크

```
public boolean startsWithA(String str) {  
    return str.startsWith("A");  
}
```

NO

1. Kotlin에서의 null 체크

```
public boolean startsWithA(String str) {  
    return str.startsWith("A");  
}
```

str에 null이 들어오면 NPE가 나게 된다.

1. Kotlin에서의 null 체크

```
public boolean startsWithA1(String str) {  
    if (str == null) {  
        throw new IllegalArgumentException("null이 들어왔습니다");  
    }  
    return str.startsWith("A");  
}
```

1. str이 null일 경우 Exception을 낸다.

1. Kotlin에서의 null 체크

```
public Boolean startsWithA2(String str) {  
    if (str == null) {  
        return null;  
    }  
    return str.startsWith("A");  
}
```

2. str이 null일 경우 null을 반환한다.

1. Kotlin에서의 null 체크

```
public boolean startsWithA3(String str) {  
    if (str == null) {  
        return false;  
    }  
    return str.startsWith("A");  
}
```

3. str이 null일 경우 false를 반환한다.

1. Kotlin에서의 null 체크

```
public boolean startsWithA1(String str) {  
    if (str == null) {  
        throw new IllegalArgumentException("null이 들어왔습니다");  
    }  
    return str.startsWith("A");  
}
```

Java

```
fun startsWithA1(str: String?): Boolean {  
    if (str == null) {  
        throw IllegalArgumentException("null이 들어왔습니다")  
    }  
    return str.startsWith(prefix: "A")  
}
```

Kotlin

1. Kotlin에서의 null 체크

```
public Boolean startsWithA2(String str) {  
    if (str == null) {  
        return null;  
    }  
    return str.startsWith("A");  
}
```

Java

```
fun startsWithA2(str: String?): Boolean? {  
    if (str == null) {  
        return null  
    }  
    return str.startsWith(prefix: "A")  
}
```

Kotlin

1. Kotlin에서의 null 체크

```
public boolean startsWithA3(String str) {  
    if (str == null) {  
        return false;  
    }  
    return str.startsWith("A");  
}
```

Java

```
fun startsWithA3(str: String?): Boolean {  
    if (str == null) {  
        return false  
    }  
    return str.startsWith(prefix: "A")  
}
```

Kotlin

1. Kotlin에서의 null 체크

```
fun startsWithA(str: String): Boolean {  
    return str.startsWith("A")  
}
```

Kotlin에서는 null이 가능한 타입을 완전히 다르게 취급한다!

2. Safe Call과 Elvis 연산자

Kotlin에서는 null이 가능한 타입을 완전히 다르게 취급한다!

null이 가능한 타입만을 위한 기능은 없나?!

2. Safe Call과 Elvis 연산자

```
val str: String? = "ABC"  
str.length // 불가능  
str?.length // 가능!!
```

Safe Call

null이 아니면 실행하고,
null이면 실행하지 않는다 (그대로 null)

2. Safe Call과 Elvis 연산자

```
val str: String? = "ABC"  
str?.length ?: 0
```

Elvis 연산자

앞의 연산 결과가 null이면 뒤의 값을 사용

2. Safe Call과 Elvis 연산자

?:

90도 회전하면...



2. Safe Call과 Elvis 연산자

?:

90도 회전하면...

:~

2. Safe Call과 Elvis 연산자

?:

90도 회전하면...

:~



2. Safe Call과 Elvis 연산자

아까 작성했던 3가지 함수에 Safe Call과 Elvis 연산을 사용하면?!

2. Safe Call과 Elvis 연산자

```
fun startsWithA1(str: String?): Boolean {  
    return str?.startsWith(prefix: "A")  
    ?: throw IllegalArgumentException("null이 들어왔습니다")  
}
```

2. Safe Call과 Elvis 연산자

```
fun startsWithA2(str: String?): Boolean? {  
    return str?.startsWith( prefix: "A")  
}
```

2. Safe Call과 Elvis 연산자

```
fun startsWithA3(str: String?): Boolean {  
    return str?.startsWith( prefix: "A") ?: false  
}
```

2. Safe Call과 Elvis 연산자

```
public long calculate(Long number) {  
    if (number == null) {  
        return 0;  
    }  
  
    // 다음 로직  
}
```



```
fun calculate(number: Long?): Long {  
    number ?: return 0  
    // 다음 로직  
}
```

Elvis연산은 early return에도 사용할 수 있다!

3. 널 아님 단언!!

nullable type이지만, 아무리 생각해도 null이 될 수 없는 경우

```
fun startsWithA1(str: String?): Boolean {  
    return str!!.startsWith( prefix: "A")  
}
```

3. 널 아님 단언!!

```
fun startsWithA1(str: String?): Boolean {  
    return str!!.startsWith( prefix: "A")  
}
```

혹시나 null이 들어오면 NPE가 나오기 때문에
정말 null이 아닌게 확실한 경우에만 널 아님 단언!! 을 사용해야 한다.

4. 플랫폼 타입

Kotlin에서 Java 코드를 가져다 사용할 때 어떻게 처리될까?!

```
import org.jetbrains.annotations.Nullable;

public class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Nullable
    public String getName() {
        return name;
    }
}
```

4. 플랫폼 타입

Kotlin에서 Java 코드를 가져다 사용할 때 어떻게 처리될까?!

```
import org.jetbrains.annotations.Nullable;

public class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Nullable
    public String getName() {
        return name;
    }
}
```

4. 플랫폼 타입

Kotlin에서 Java 코드를 가져다 사용할 때 어떻게 처리될까?!

```
import org.jetbrains.annotations.Nullable;

public class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Nullable
    public String getName() {
        return name;
    }
}
```

- javax.annotation 패키지
- android.support.annotation 패키지
- org.jetbrains.annotation 패키지

4. 플랫폼 타입

```
import org.jetbrains.annotations.Nullable;

public class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Nullable
    public String getName() {
        return name;
    }
}
```

@Nullable이 없다면??

Kotlin에서는 이 값이 nullable인지 non-nullable인지 알 수가 없다~

플랫폼 타입

코틀린이 null 관련 정보를 알 수 없는 타입
Runtime 시 Exception이 날 수 있다

Lec 02. 코틀린에서 null을 다루는 방법

- 코틀린에서 null이 들어갈 수 있는 타입은 완전히 다르게 간주된다
 - 한번 null 검사를 하면 non-null임을 컴파일러가 알 수 있다

Lec 02. 코틀린에서 null을 다루는 방법

- 코틀린에서 null이 들어갈 수 있는 타입은 완전히 다르게 간주된다
 - 한번 null 검사를 하면 non-null임을 컴파일러가 알 수 있다
 - null이 아닌 경우에만 호출되는 Safe Call (`?.`) 이 있다

```
str?.length
```

Lec 02. 코틀린에서 null을 다루는 방법

- 코틀린에서 null이 들어갈 수 있는 타입은 완전히 다르게 간주된다
 - 한 번 null 검사를 하면 non-null임을 컴파일러가 알 수 있다
- null이 아닌 경우에만 호출되는 Safe Call (?.) 이 있다
- null인 경우에만 호출되는 Elvis 연산자 (?:) 가 있다

```
str?.length ?: 0
```

Lec 02. 코틀린에서 null을 다루는 방법

- 코틀린에서 null이 들어갈 수 있는 타입은 완전히 다르게 간주된다
 - 한 번 null 검사를 하면 non-null임을 컴파일러가 알 수 있다
 - null이 아닌 경우에만 호출되는 Safe Call (?.) 이 있다
 - null인 경우에만 호출되는 Elvis 연산자 (?:) 가 있다
 - null이 절대 아닐때 사용할 수 있는 널 아님 단언 (!!) 이 있다

```
fun startsWithA1(str: String?): Boolean {  
    return str!!.startsWith(prefix: "A")  
}
```

Lec 02. 코틀린에서 null을 다루는 방법

- 코틀린에서 null이 들어갈 수 있는 타입은 완전히 다르게 간주된다
 - 한 번 null 검사를 하면 non-null임을 컴파일러가 알 수 있다
 - null이 아닌 경우에만 호출되는 Safe Call (?) 이 있다
 - null인 경우에만 호출되는 Elvis 연산자 (?:) 가 있다
 - null이 절대 아닐 때 사용할 수 있는 널 아님 단언 (!!) 이 있다
- Kotlin에서 Java 코드를 사용할 때 **플랫폼 타입** 사용에 유의해야 한다
 - Java 코드를 읽으며 널 가능성 확인 / Kotlin으로 wrapping

Lec 02. 코틀린에서 null을 다루는 방법

(Tony Hoare - ALGOL 60 개발자)

1965년에 탄생한 NULL은 수조원의 실수이다.

이때 나는 객체지향적 언어에서 주소값에 대한 종합적인 타입
시스템을 디자인하고 있었다. 나의 목표는 참조값 사용의 안정성을
컴파일러에 의해 자동 수행되어, 완전하게 보장하는 것이었다.

하지만 나는 NULL을 사용하는 방식의 구현이 쉬웠기에 NULL 참조를
넣을 수 밖에 없다. 이 선택은 지난 40년동안 수조원에 가까운 셀수 없는
에러, 침해, 시스템 크래시 등을 만들었다.

Lec 03. 코틀린에서 Type을 다루는 방법

1. 기본 타입
2. 타입 캐스팅
3. Kotlin의 3가지 특이한 타입
4. String Interpolation, String indexing

1. 기본 타입

Byte

Short

Int

Long

Float

Double

부호 없는 정수들

1. 기본 타입

Byte

Short

Int

Long

Float

Double

부호 없는 정수들

1. 기본 타입

코틀린에서는 선언된 기본값을 보고 타입을 추론한다

```
val number1 = 3 // Int  
val number2 = 3L // Long
```

1. 기본 타입

코틀린에서는 선언된 기본값을 보고 타입을 추론한다

```
val number1 = 3.0f // Float  
val number2 = 3.0 // Double
```

1. 기본 타입

Java와 다른 내용



기본 타입간의 변환은 **암시적으로** 이루어질 수 있다



Kotlin 기본 타입간의 변환은 **명시적으로** 이루어져야 한다

1. 기본 타입



```
int number1 = 4;  
long number2 = number1;  
  
System.out.println(number1 + number2);
```

int 타입의 값이 long 타입으로 암시적으로 변경되었습니다.

Java에서 더 큰 타입으로는 암시적 변경이 되었죠!

1. 기본 타입



```
val number1 = 4
val number2: Long = number1 // Type mismatch

println(number1 + number2)
```

Kotlin에서는 암시적 타입 변경이 불가능합니다!

어떻게 해야 할까요?!

1. 기본 타입



```
val number1: Int = 4  
val number2: Long = number1.toLong()  
  
println(number1 + number2)
```

to변환타입()을 사용해야 합니다.

1. 기본 타입



```
val number1 = 3
val number2 = 5
val result = number1 / number2.toDouble()

println(result)
```

1. 기본 타입



```
val number1: Int = 4
val number2: Long = number1.toLong()
println(number1 + number2)
```

to변환타입()을 사용해야 합니다.

1. 기본 타입

변수가 nullable이라면 적절한 처리가 필요합니다!

2. 타입 캐스팅

기본 타입이 아닌 일반 타입은 어떨까요?!

2. 타입 캐스팅



```
public static void printAgeIfPerson(Object obj) {  
    if (obj instanceof Person) {  
        Person person = (Person) obj;  
        System.out.println(person.getAge());  
    }  
}
```

2. 타입 캐스팅

instanceof : 변수가 주어진 타입이면 true, 그렇지 않으면 false

```
public static void printAgeIfPerson(Object obj) {  
    if (obj instanceof Person) {  
        Person person = (Person) obj;  
        System.out.println(person.getAge());  
    }  
}
```

2. 타입 캐스팅

```
public static void printAgeIfPerson(Object obj) {  
    if (obj instanceof Person) {  
        Person person = (Person) obj;  
        System.out.println(person.getAge());  
    }  
}
```

(타입) : 주어진 변수를 해당 타입으로 변경한다

2. 타입 캐스팅



```
fun printAgeIfPerson(obj: Any) {  
    if (obj is Person) {  
        val person = obj as Person  
        println(person.age)  
    }  
}
```

Java의 instanceof

2. 타입 캐스팅



Java의 (Person) obj

```
fun printAgeIfPerson(obj: Any) {  
    if (obj is Person) {  
        val person = obj as Person  
        println(person.age)  
    }  
}
```

2. 타입 캐스팅



```
fun printAgeIfPerson(obj: Any) {  
    if (obj is Person) {  
        println(obj.age)  
    }  
}
```

스마트 캐스트

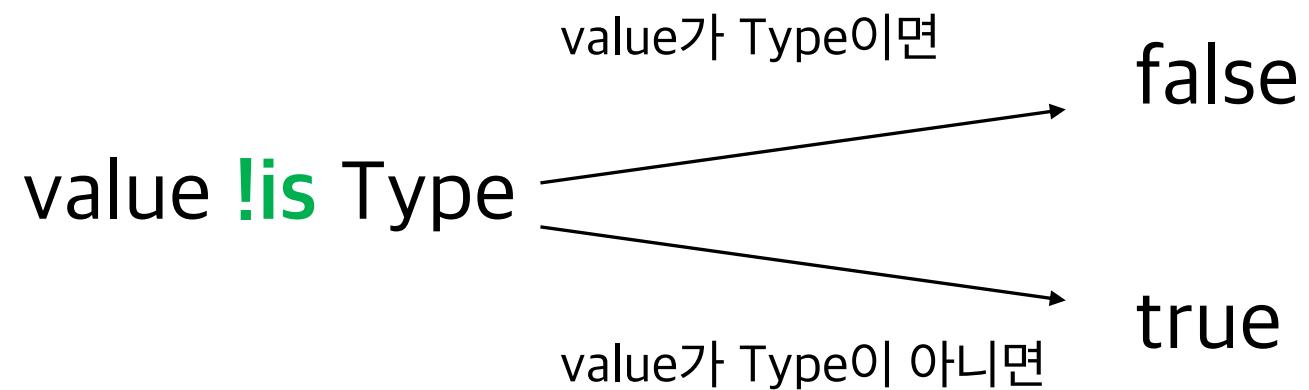
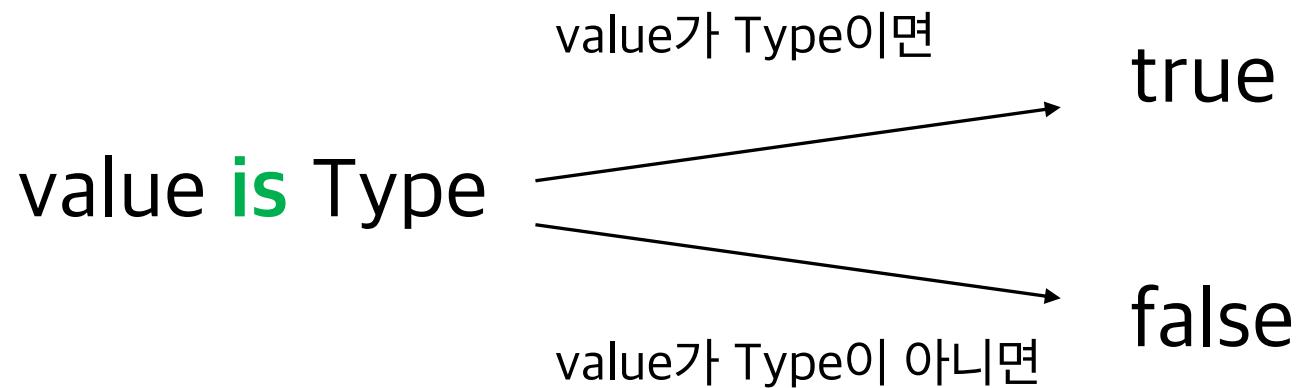
2. 타입 캐스팅

Instanceof의 반대도 존재할까?!

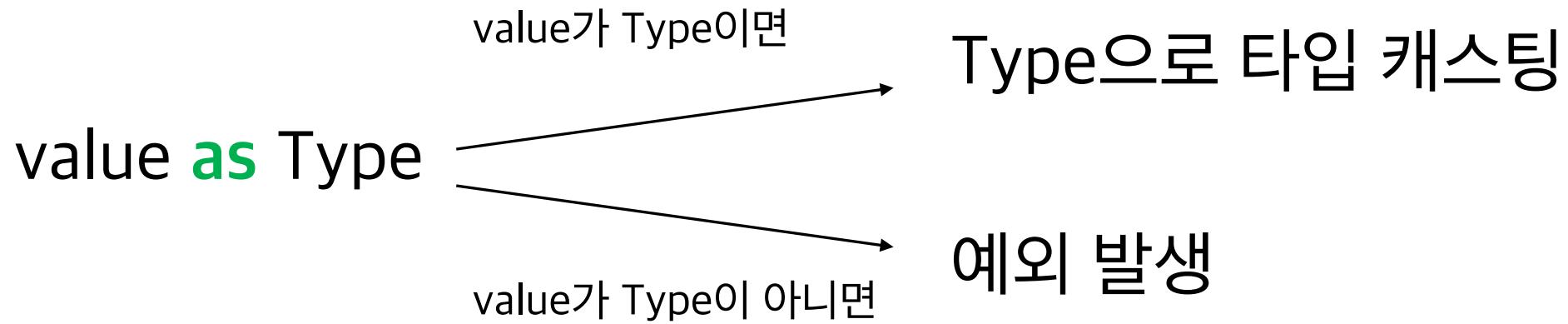
2. 타입 캐스팅

만약 obj에 null이 들어올 수 있다면?!

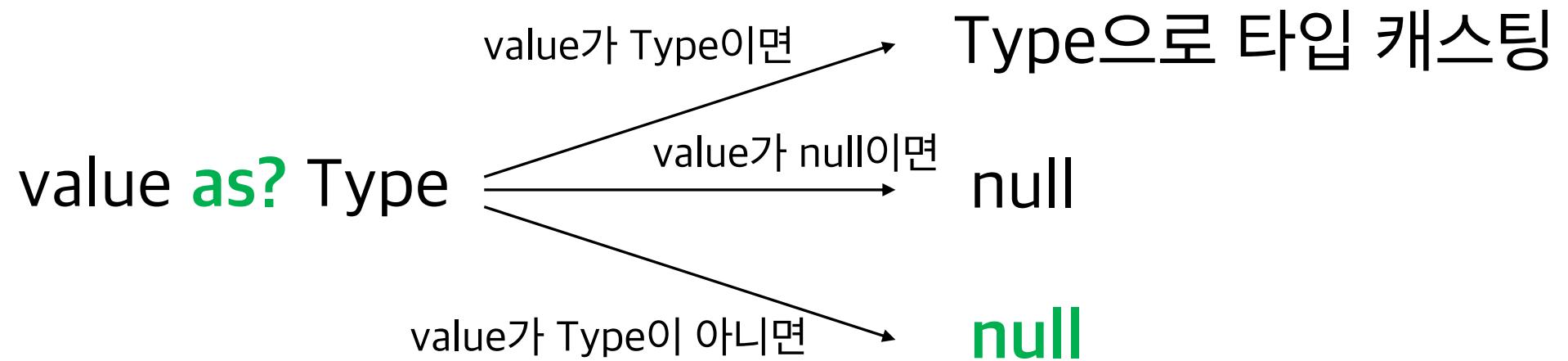
2. 타입 캐스팅



2. 타입 캐스팅



2. 타입 캐스팅



3. Kotlin의 특이한 타입 3가지

Any

Unit

Nothing

3. Kotlin의 특이한 타입 3가지

Any

Unit

Nothing

3. Kotlin의 특이한 타입 3가지 - Any

- Java의 Object 역할. (모든 객체의 최상위 타입)
- 모든 Primitive Type의 최상의 타입도 Any이다.
- Any 자체로는 null을 포함할 수 없어 null을 포함하고 싶다면, Any?로 표현.
- Any에 equals / hashCode / toString 존재.

3. Kotlin의 특이한 타입 3가지

Any

Unit

Nothing

3. Kotlin의 특이한 타입 3가지 - Unit

- Unit은 Java의 void와 동일한 역할.
- (살짝 어려운 내용) void와 다르게 Unit은 그 자체로 타입 인자로 사용 가능하다.
- 함수형 프로그래밍에서 Unit은 단 하나의 인스턴스만 갖는 타입을 의미. 즉, 코틀린의 Unit은 실제 존재하는 타입이라는 것을 표현

3. Kotlin의 특이한 타입 3가지

Any

Unit

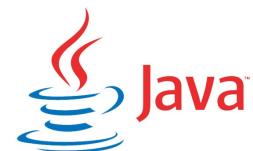
Nothing

3. Kotlin의 특이한 타입 3가지 - Nothing

- Nothing은 함수가 정상적으로 끝나지 않았다는 사실을 표현하는 역할
- 무조건 예외를 반환하는 함수 / 무한 루프 함수 등

```
fun fail(message: String): Nothing {  
    throw IllegalArgumentException(message)  
}
```

4. String interpolation / String indexing



```
Person person = new Person(name: "최태현", age: 100);
String log = String.format("사람의 이름은 %s이고 나이는 %s세 입니다", person.getName(), person.getAge());
```

```
StringBuilder builder = new StringBuilder();
builder.append("사람의 이름은");
builder.append(person.getName());
builder.append("이고 나이는");
builder.append(person.getAge());
builder.append("세 입니다");
```

4. String interpolation / String indexing



```
val person = Person(name: "최태현", age: 100)
val log = "사람의 이름은 ${person.name}이고 나이는 ${person.age}세 입니다"
```

`\${변수}` 를 사용하면 값이 들어간다

4. String interpolation / String indexing



```
val name = "최태현"  
val age = 100  
val log = "사람의 이름: $name 나이: $age"
```

\$변수를 사용할 수도 있다

4. String interpolation / String indexing

TIP

변수 이름만 사용하더라도 \${변수}를 사용하는 것이

- 1) 가독성
- 2) 일괄 변환
- 3) 정규식 활용

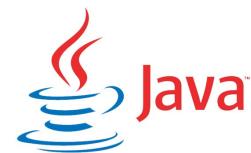
측면에서 좋았습니다.

4. String interpolation / String indexing



```
val withoutIndent =  
    """  
        ABC  
        123  
        456  
    """.trimIndent()  
    println(withoutIndent)
```

4. String interpolation / String indexing



```
String str = "ABCDE";
char ch = str.charAt(1);
```

Java에서 문자열의 특정 문자 가져오기

4. String interpolation / String indexing



```
val str = "ABCDE"  
val ch = str[1]
```

Kotlin에서 문자열의 특정 문자 가져오기

Lec 03. 코틀린에서 Type을 다루는 방법

- 코틀린의 변수는 초기값을 보고 타입을 추론하며, 기본 타입들 간의 변환은 명시적으로 이루어진다.
- 코틀린에서는 `is`, `!is`, `as`, `as?` 를 이용해 타입을 확인하고 캐스팅한다.
- 코틀린의 `Any`는 Java의 `Object`와 같은 최상위 타입이다.
- 코틀린의 `Unit`은 Java의 `void`과 동일하다.
- 코틀린에 있는 **Nothing**은 정상적으로 끝나지 않는 함수의 반환을 의미한다.

Lec 03. 코틀린에서 Type을 다루는 방법

- 문자열을 가공할때 \${변수}와 """ """" 를 사용하면 깔끔한 코딩이 가능하다.
- 문자열에서 문자를 가져올때의 Java의 배열처럼 []를 사용한다

Lec 04. 코틀린에서 연산자를 다루는 방법

1. 단항 연산자 / 산술 연산자
2. 비교 연산자와 동등성, 동일성
3. 논리 연산자 / 코틀린에 있는 특이한 연산자
4. 연산자 오버로딩

1. 단항 연산자 / 산술 연산자

	+	+ =
	-	- =
	*	* =
++	/	/ =
--	%	% =

단항 연산자

산술 연산자

산술대입 연산자

Java, Kotlin 완전 동일합니다.

2. 비교 연산자와 동등성, 동일성

>

<

>=

<=

비교 연산자

Java, Kotlin 사용법은 동일합니다! 단!

2. 비교 연산자와 동등성, 동일성



>
<
>=
<=

비교 연산자

```
public class JavaMoney implements Comparable<JavaMoney> {  
  
    private final long amount;  
  
    public JavaMoney(long amount) {  
        this.amount = amount;  
    }  
  
    @Override  
    public int compareTo(@NotNull JavaMoney o) {  
        return Long.compare(this.amount, o.amount);  
    }  
}
```

Java와 다르게 객체를 비교할 때

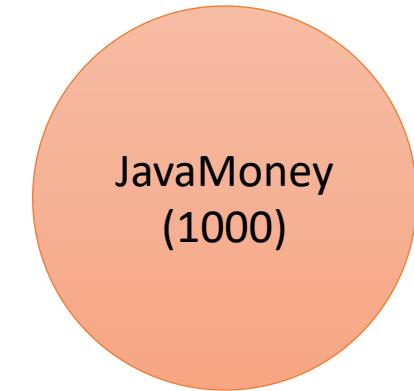
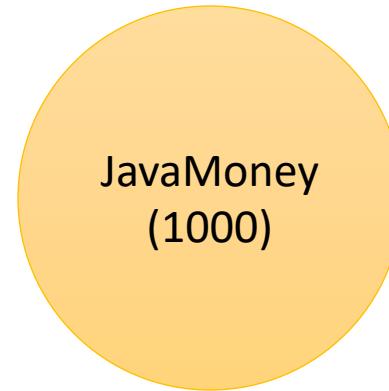
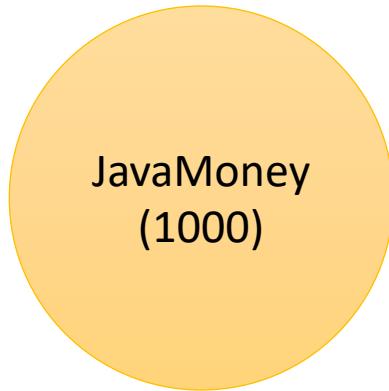
비교 연산자를 사용하면 자동으로 compareTo를 호출해줍니다

2. 비교 연산자와 동등성, 동일성

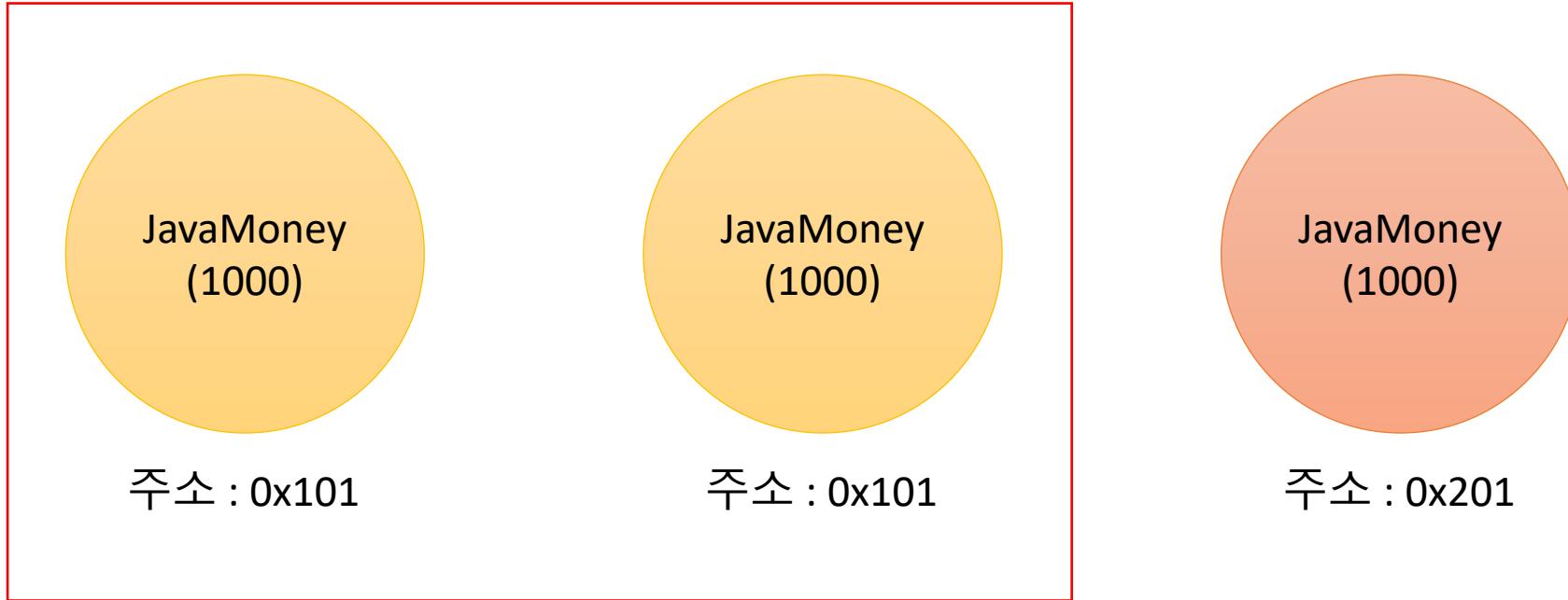
동등성(Equality) : 두 객체의 값이 같은가?!

동일성(Identity) : 완전히 동일한 객체인가?! 즉 주소가 같은가?!

2. 비교 연산자와 동등성, 동일성

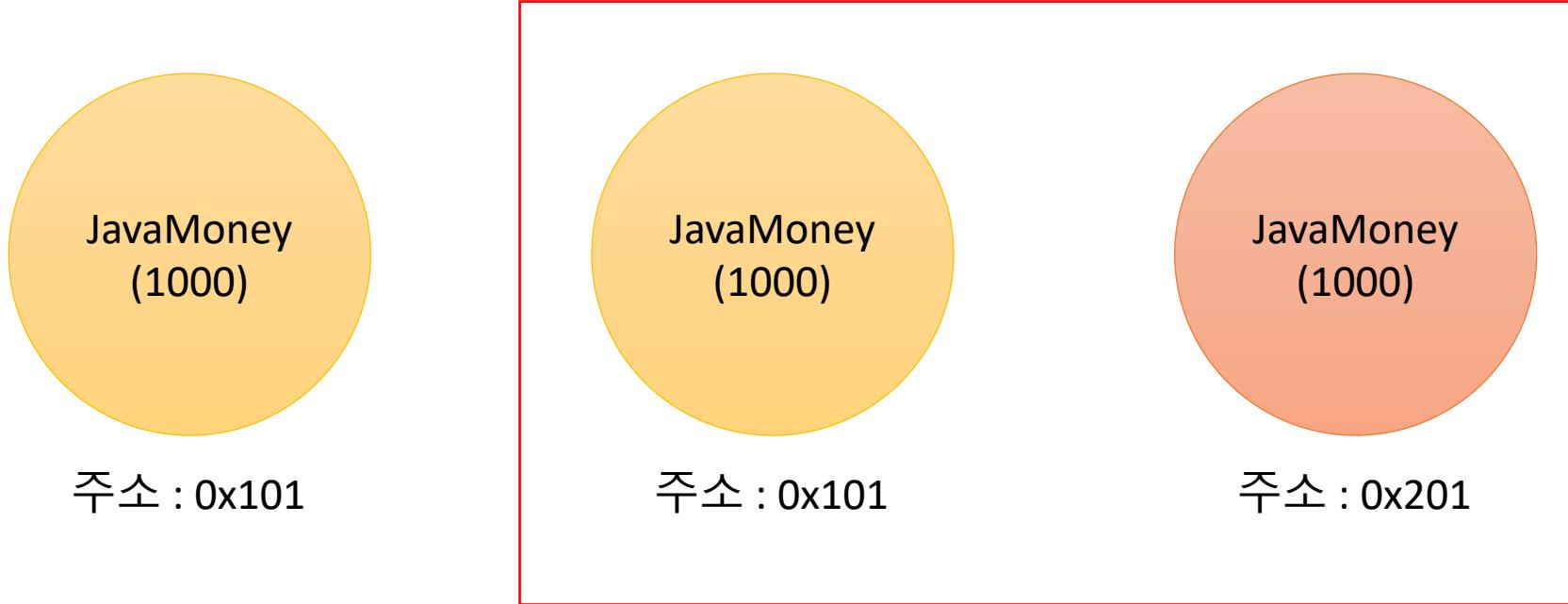


2. 비교 연산자와 동등성, 동일성



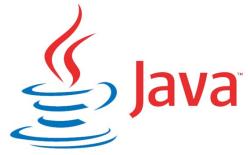
두 인스턴스는 정체성이 동일하다
Java에서 **==** 를 사용

2. 비교 연산자와 동등성, 동일성



두 인스턴스는 값이 동등하다
Java에서 **equals** 를 사용

2. 비교 연산자와 동등성, 동일성



Java에서는 동일성에 `==`를 사용, 동등성에 `equals`를 직접 호출



Kotlin에서는 동일성에 `==`을 사용, 동등성에 `==`을 호출
`==`을 사용하면 간접적으로 `equals`를 호출해준다

3. 논리 연산자와 코틀린에 있는 특이한 연산자

&&
||
!

논리연산자

Java와 완전히 동일합니다.
Java 처럼 Lazy 연산을 수행합니다.

3. 논리 연산자와 코틀린에 있는 특이한 연산자

- in / !in
 - 컬렉션이나 범위에 포함되어 있다, 포함되어 있지 않다

```
println(1 in numbers)
```

3. 논리 연산자와 코틀린에 있는 특이한 연산자

- in / !in
 - 컬렉션이나 범위에 포함되어 있다, 포함되어 있지 않다
- a..b
 - a부터 b 까지의 범위 객체를 생성한다
 - <코틀린에서 반복문을 다루는 방법> 강의에서 다룹니다!

3. 논리 연산자와 코틀린에 있는 특이한 연산자

- in / !in
 - 컬렉션이나 범위에 포함되어 있다, 포함되어 있지 않다
- a..b
 - a부터 b 까지의 범위 객체를 생성한다
 - <코틀린에서 반복문을 다루는 방법> 강의에서 다룹니다!
- a[i]
 - a에서 특정 Index i로 값을 가져온다

```
val str = "ABC"  
println(str[2]) // C
```

3. 논리 연산자와 코틀린에 있는 특이한 연산자

- in / !in
 - 컬렉션이나 범위에 포함되어 있다, 포함되어 있지 않다
- a..b
 - a부터 b 까지의 범위 객체를 생성한다
 - <코틀린에서 반복문을 다루는 방법> 강의에서 다룹니다!
- a[i]
 - a에서 특정 Index i로 값을 가져온다
- a[i] = b
 - a의 특정 index i에 b를 넣는다

4. 연산자 오버로딩

Kotlin에서는 객체마다 연산자를 직접 정의할 수 있다

```
val money1 = Money(amount: 1_000L)
val money2 = Money(amount: 2_000L)
println(money1 + money2) // Money(amount=3000)
```

Lec 04. 코틀린에서 연산자를 다루는 방법

- 단항연산자, 산술연산자, 산술대입연산자 Java와 똑같다~
- 비교 연산자 사용법도 Java와 똑같다~
 - 단, 객체끼리도 자동 호출되는 compareTo를 이용해 비교 연산자를 사용할 수 있다.
- in, !in / a..b / a[i] / a[i] = b 와 같이 코틀린에서 새로 생긴 연산자도 있다.
- 객체끼리의 연산자를 직접 정의할 수 있다

Lec 05. 코틀린에서 조건문을 다루는 방법

1. if문
2. Expression과 Statement
3. switch와 when

1. if문



```
private void validateScoreIsNotNegative(int score) {  
    if (score < 0) {  
        throw new IllegalArgumentException(String.format("%s는 0보다 작을 수 없습니다.", score));  
    }  
}
```

1. if문



```
fun validateScoreIsNotNegative(score: Int) {  
    if (score < 0) {  
        throw IllegalArgumentException("${score}는 0보다 작을 수 없습니다")  
    }  
}
```

1. if문

함수에서 Unit(void)이 생략됨

```
fun validateScoreIsNotNegative(score: Int) {  
    if (score < 0) {  
        throw IllegalArgumentException("${score}는 0보다 작을 수 없습니다")  
    }  
}
```

1. if문

함수를 만들 때 fun을 사용

```
fun validateScoreIsNotNegative(score: Int) {  
    if (score < 0) {  
        throw IllegalArgumentException("${score}는 0보다 작을 수 없습니다")  
    }  
}
```

1. if문

```
fun validateScoreIsNotNegative(score: Int) {  
    if (score < 0) {  
        throw IllegalArgumentException("${score}는 0보다 작을 수 없습니다")  
    }  
}
```



new 를 사용하지 않고 예외를 throw

1. if문

```
fun validateScoreIsNotNegative(score: Int) {  
    if (score < 0) {  
        throw IllegalArgumentException("${score}는 0보다 작을 수 없습니다")  
    }  
}
```

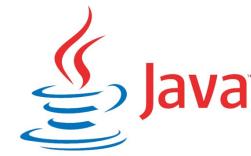
네, Java와 차이가 없죠!

1. if문

if (조건) {

}

1. if문



```
private String getPassOrFail(int score) {  
    if (score >= 50) {  
        return "P";  
    } else {  
        return "F";  
    }  
}
```

1. if문



```
fun getPassOrFail(score: Int): String {  
    if (score >= 50) {  
        return "P"  
    } else {  
        return "F"  
    }  
}
```

1. if문



```
fun getPassOrFail(score: Int): String {  
    if (score >= 50) {  
        return "P"  
    } else {  
        return "F"  
    }  
}
```

이번에도 Java와 똑같이 생겼죠?
하지만, 한 가지 다른 점이 있습니다.

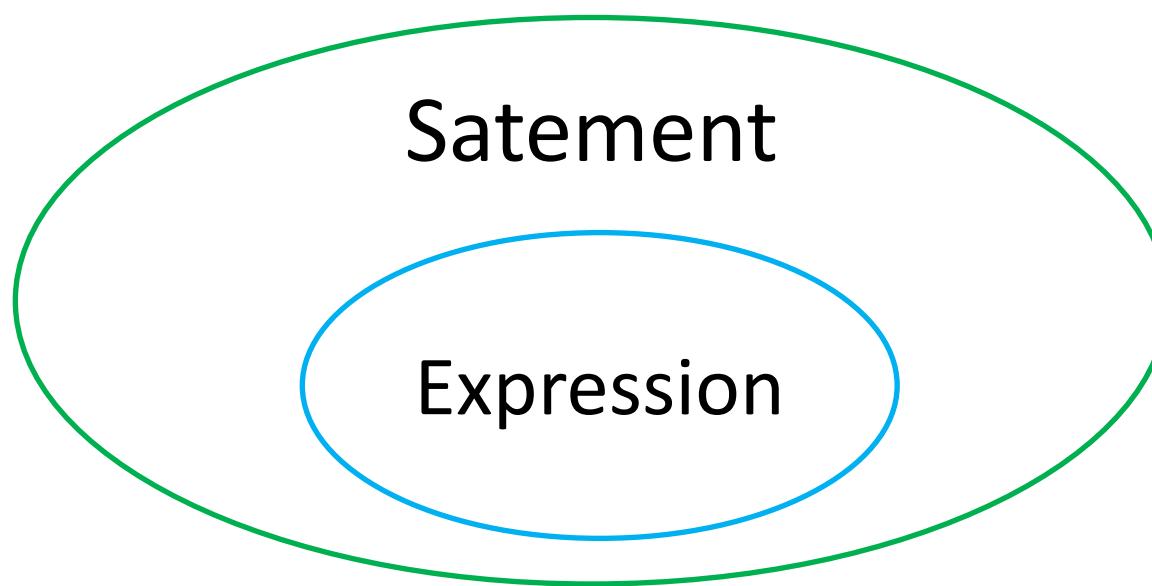
2. Expression & Statement

Java에서 if-else는 **Statement**이지만,
Kotlin에서는 **Expression**입니다.

2. Expression & Statement

Statement : 프로그램의 문장, 하나의 값으로 도출되지 않는다

Expression : 하나의 값으로 도출되는 문장



2. Expression & Statement

```
int score = 30 + 40;
```

30 + 40 은 70이라는 하나의 결과가 나옵니다

Expression이면서 Statement

2. Expression & Statement

```
String grade = if (score >= 50) {  
    "P";  
} else {  
    "F";  
}
```

if 문을 하나의 값으로 취급하지 않으니 에러가 나죠!

Statement

2. Expression & Statement

```
String grade = score >= 50 ? "P" : "F";
```

3항 연산자는 하나의 값으로 취급하므로 에러가 없다!

Expression이면서 Statement

2. Expression & Statement

Java에서 if-else는 **Statement**이지만,
Kotlin에서는 **Expression**입니다.

2. Expression & Statement

```
fun getPassOrFail(score: Int): String {  
    if (score >= 50) {  
        return "P"  
    } else {  
        return "F"  
    }  
}
```

```
fun getPassOrFail(score: Int): String {  
    return if (score >= 50) {  
        "P"  
    } else {  
        "F"  
    }  
}
```

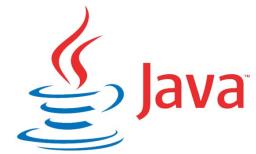
2. Expression & Statement

Kotlin에서는 if-else를 expression으로 사용할 수 있기 때문에
3항 연산자가 없습니다.

2. Expression & Statement

if - else if - else 문도 문법이 동일합니다.

2. Expression & Statement



```
private String getGrade(int score) {  
    if (score >= 90) {  
        return "A";  
    } else if (score >= 80) {  
        return "B";  
    } else if (score >= 70) {  
        return "C";  
    } else {  
        return "D";  
    }  
}
```

2. Expression & Statement



```
fun getGrade(score: Int): String {  
    return if (score >= 90) {  
        "A"  
    } else if (score >= 80) {  
        "B"  
    } else if (score >= 70) {  
        "C"  
    } else {  
        "D"  
    }  
}
```

2. Expression & Statement

간단한 TIP

어떠한 값이 특정 범위에 포함되어 있는지, 포함되어 있지 않은지

```
if (0 <= score && score <= 100) {
```

2. Expression & Statement

간단한 TIP

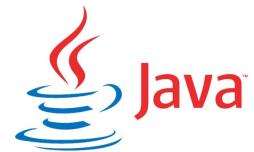
어떠한 값이 특정 범위에 포함되어 있는지, 포함되어 있지 않은지

```
if (0 <= score && score <= 100) {
```



```
if (score in 0..100) {
```

3. switch 와 when



```
private String getGradeWithSwitch(int score) {  
    switch (score / 10) {  
        case 9:  
            return "A";  
        case 8:  
            return "B";  
        case 7:  
            return "C";  
        default:  
            return "D";  
    }  
}
```

3. switch 와 when



```
fun getGradeWithSwitch(  
    score: Int  
): String {  
    return when (score / 10) {  
        9 -> "A"  
        8 -> "B"  
        7 -> "C"  
        else -> "D"  
    }  
}
```

```
fun getGradeWithSwitchV2(  
    score: Int  
): String {  
    return when (score) {  
        in 90..99 -> "A"  
        in 80..89 -> "B"  
        in 70..79 -> "C"  
        else -> "D"  
    }  
}
```

3. switch 와 when

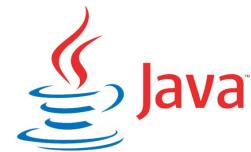
```
when (값) {  
    조건부 -> 어떠한 구문  
    조건부 -> 어떠한 구문  
    else -> 어떠한 구문  
}
```

3. switch 와 when

```
when (값) {  
    조건부 -> 어떠한 구문  
    조건부 -> 어떠한 구문  
    else -> 어떠한 구문  
}
```

어떠한 expression이라도 들어갈 수 있다 (ex. **is** Type)

3. switch 와 when



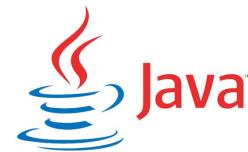
```
private boolean startsWithA(Object obj) {  
    if (obj instanceof String) {  
        return ((String) obj).startsWith("A");  
    } else {  
        return false;  
    }  
}
```

3. switch 와 when

```
when (값) {  
    조건부 -> 어떠한 구문  
    조건부 -> 어떠한 구문  
    else -> 어떠한 구문  
}
```

여러개의 조건을 동시에 검사할 수 있다 (로 구분)

3. switch 와 when



```
private void judgeNumber(int number) {
    if (number == 1 || number == 0 || number == -1) {
        System.out.println("어디서 많이 본 숫자입니다");
    } else {
        System.out.println("1, 0, -1이 아닙니다");
    }
}
```

3. switch 와 when

```
when (값) {
```

조건부 -> 어떠한 구문

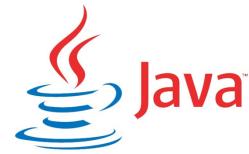
조건부 -> 어떠한 구문

else -> 어떠한 구문

```
}
```

(값)이 없을 수도 있다 - early return 처럼 동작

3. switch 와 when



```
private void judgeNumber2(int number) {  
    if (number == 0) {  
        System.out.println("주어진 숫자는 0입니다");  
        return;  
    }  
  
    if (number % 2 == 0) {  
        System.out.println("주어진 숫자는 짝수입니다");  
        return;  
    }  
  
    System.out.println("주어지는 숫자는 홀수입니다");  
}
```

3. switch 와 when

when은 **Enum Class** 혹은 **Sealed Class**와 함께 사용할 경우, 더욱더 진가를 발휘한다.

Lec 05. 코틀린에서 조건문을 다루는 방법

- if / if - else / if - else if - else 모두 Java와 문법이 동일하다.
- 단 Kotlin에서는 Expression으로 취급된다.
 - 때문에 Kotlin에서는 삼항 연산자가 없다

```
fun getPassOrFail(score: Int): String {  
    return if (score >= 50) {  
        "P"  
    } else {  
        "F"  
    }  
}
```

Lec 05. 코틀린에서 조건문을 다루는 방법

- if / if - else / if - else if - else 모두 Java와 문법이 동일하다.
- 단 Kotlin에서는 Expression으로 취급된다.
 - 때문에 Kotlin에서는 삼항 연산자가 없다
- Java의 switch는 Kotlin에서 when으로 대체되었고, when은 더 강력한 기능을 갖는다.

Lec 06. 코틀린에서 반복문을 다루는 방법

1. for-each문
2. 전통적인 for문
3. Progression과 Range
4. while 문

1. for each 문

숫자가 들어 있는 리스트를 하나씩 출력하는 예제

```
List<Long> numbers = Arrays.asList(1L, 2L, 3L);
for (Long number : numbers) {
    System.out.println(number);
}
```



1. for each は

```
val numbers = listOf(1L, 2L, 3L)
for (number in numbers) {
    println(number)
}
```



1. for each 문

```
val numbers = listOf(1L, 2L, 3L)
for (number in numbers) {
    println(number)
}
```

컬렉션을 만드는 방법

1. for each 문

```
val numbers = listOf(1L, 2L, 3L)
for (number in numbers) {
    println(number)
}
```

: 대신 **in** 을 사용

1. for each 문

```
val numbers = listOf(1L, 2L, 3L)
for (number in numbers) {
    println(number)
```

Java와 동일하게 Iterable이 구현된 타입이라면 모두 들어갈 수 있다.

2. 전통적인 for문

1부터 3까지 출력하는 예제

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

2. 전통적인 for문



```
for (i in 1..3) {  
    println(i)  
}
```

1..3 : 1부터 3까지

2. 전통적인 for문

내려가는 경우는?!

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



```
for (i in 3 downTo 1) {  
    println(i)  
}
```

2. 전통적인 for문

2칸씩 올라가는 경우는?!

```
for (int i = 1; i <= 5; i+=2) {  
    System.out.println(i);  
}
```



```
for (i in 1..5 step 2) {  
    println(i)  
}
```

2. 전통적인 for문

동작 원리가 궁금하시죠?!

3. Progression과 Range

.. 연산자 : 범위를 만들어 내는 연산자

1..3 : 1부터 3의 범위

3. Progression과 Range

IntProgression



IntRange

3. Progression과 Range

IntProgression

등차수열



IntRange

- 1) 시작 값
- 2) 끝 값
- 3) 공차

3. Progression과 Range

사실은 등차수열을 만들어주고 있던 것!

3. Progression과 Range

3 downTo 1 : 시작값 3, 끝값 1, 공차가 -1인 등차수열

1..5 step 2 : 시작값 1, 끝값 5, 공차가 2인 등차수열

3. Progression과 Range

downTo, step 도 함수이다!

3. Progression과 Range

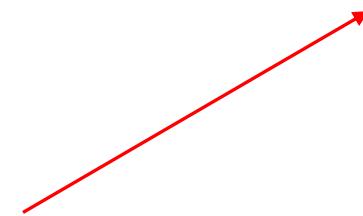
downTo, step 도 함수이다! (중위 호출 함수)

변수.함수이름(argument) 대신

변수 함수이름 argument

3. Progression과 Range

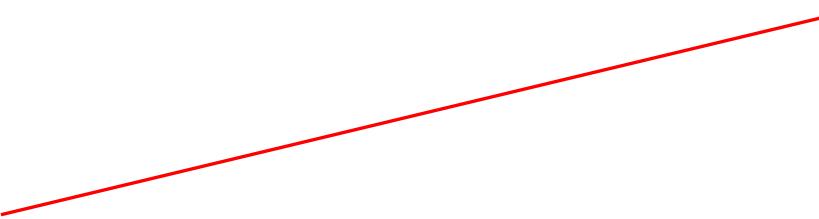
1..5 step 2



1부터 5까지 공차가 1인 등차수열 생성

3. Progression과 Range

1~5, 공차 1 등차수열 step 2



등차수열에 대한 함수 호출, 등차수열.step(2)

3. Progression과 Range

1부터 5까지 공차가 2인 등차수열

1, 3, 5

3. Progression과 Range

한 줄 요약

Kotlin에서 전통적인 for문은 등차수열을 이용한다!

4. While문

1부터 3을 출력하는 예제



```
int i = 1;
while (i <= 3) {
    System.out.println(i);
    i++;
}
```

4. While문

1부터 3을 출력하는 예제



```
var i = 1
while (i <= 3) {
    println(i)
    i++
}
```

4. While문

1부터 3을 출력하는 예제



```
var i = 1
while (i <= 3) {
    println(i)
    i++
}
```

Java와 완전히 동일하죠?! do-while도 똑같습니다!

Lec 06. 코틀린에서 반복문을 다루는 방법

- for each 문에서 Java는 : Kotlin은 in 을 사용한다.
- 전통적인 for문에서 Kotlin은 등차수열과 in을 사용한다.
- 그 외 for문 문법은 모두 동일하다.
- while문과 do while문은 더욱더 놀랍도록 동일하다.

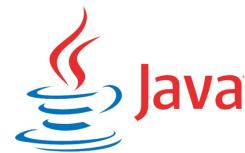
Lec 07. 코틀린에서 예외를 다루는 방법

1. try catch finally 구문
2. Checked Exception과 Unchecked Exception
3. try with resources

1. try catch finally 구문

주어진 문자열을 정수로 변경하는 예제

1. try catch finally 구문



```
private int parseIntOrThrow(@NotNull String str) {
    try {
        return Integer.parseInt(str);
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException(String.format("주어진 %s는 숫자가 아닙니다", str));
    }
}
```

1. try catch finally 구문



```
fun parseIntOrThrow(str: String): Int {  
    try {  
        return str.toInt()  
    } catch (e: NumberFormatException) {  
        throw IllegalArgumentException("주어진 ${str}는 숫자가 아닙니다")  
    }  
}
```

1. try catch finally 구문

```
fun parseIntOrThrow(str: String): Int {  
    try {  
        return str.toInt()  
    } catch (e: NumberFormatException) {  
        throw IllegalArgumentException("주어진 ${str}는 숫자가 아닙니다")  
    }  
}
```

기본타입간의 형변환은 toType()을 사용

1. try catch finally 구문

```
fun parseIntOrThrow(str: String): Int {  
    try {  
        return str.toInt()  
    } catch (e: NumberFormatException) {  
        throw IllegalArgumentException("주어진 ${str}는 숫자가 아닙니다")  
    }  
}
```

- 
- 1) 타입이 뒤에 위치하고 2) new를 사용하지 않음
 - 3) 포맷팅이 간결함

1. try catch finally 구문

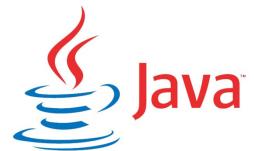
```
fun parseIntOrThrow(str: String): Int {  
    try {  
        return str.toInt()  
    } catch (e: NumberFormatException) {  
        throw IllegalArgumentException("주어진 ${str}는 숫자가 아닙니다")  
    }  
}
```

try catch 구문은 문법적으로 동일하다!

1. try catch finally 구문

주어진 문자열을 정수로 변경하는 예제
실패하면 null을 반환!

1. try catch finally 구문



```
private Integer parseIntOrDefault(String str) {  
    try {  
        return Integer.parseInt(str);  
    } catch (NumberFormatException e) {  
        return null;  
    }  
}
```

1. try catch finally 구문



```
fun parseIntOrThrowV2(str: String): Int? {  
    return try {  
        str.toInt()  
    } catch (e: NumberFormatException) {  
        null  
    }  
}
```

1. try catch finally 구문



```
fun parseIntOrThrowV2(str: String): Int? {  
    return try {  
        str.toInt()  
    } catch (e: NumberFormatException) {  
        null  
    }  
}
```

Kotlin에서는 try catch 구문 역시 expression 이다!

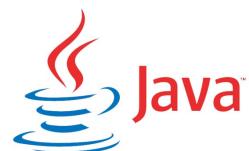
1. try catch finally 구문

try catch finally 역시 동일합니다

2. Checked Exception과 Unchecked Exception

프로젝트 내 파일의 내용물을 읽어오는 예제

2. Checked Exception과 Unchecked Exception



```
public void readFile() throws IOException {
    File currentFile = new File(pathname: ".");
    File file = new File(pathname: currentFile.getAbsolutePath() + "/a.txt");
    BufferedReader reader = new BufferedReader(new FileReader(file));
    System.out.println(reader.readLine());
    reader.close();
}
```

2. Checked Exception과 Unchecked Exception



```
fun readFile() {  
    val currentFile = File(pathname: ".")  
    val file = File(pathname: currentFile.absolutePath + "/a.txt")  
    val reader = BufferedReader(FileReader(file))  
    println(reader.readLine())  
    reader.close()  
}
```

2. Checked Exception과 Unchecked Exception



```
fun readFile() {  
    val currentFile = File(pathname: ".")  
    val file = File(pathname: currentFile.absolutePath + "/a.txt")  
    val reader = BufferedReader(FileReader(file))  
    println(reader.readLine())  
    reader.close()  
}
```

throws 구문이 없다!!

2. Checked Exception과 Unchecked Exception

Kotlin에서는 Checked Exception과
Unchecked Exception을 구분하지 않습니다.

모두 Unchecked Exception입니다.

3. try with resources

프로젝트 내 파일의 내용물을 읽어오는 예제

3. try with resources



```
public void readFile(String path) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
        System.out.println(reader.readLine());
    }
}
```

3. try with resources

Kotlin에는 try with resources 구문이 없습니다!!

3. try with resources

대신 use 라는 inline 확장함수를 사용해야 합니다.

3. try with resources



```
fun readFile(path: String) {
    BufferedReader(FileReader(path)).use { reader ->
        println(reader.readLine())
    }
}
```

Inline 함수, 확장함수 등 문법적인 설명은
Section 4에서 다룹니다!

Lec 07. 코틀린에서 예외를 다루는 방법

- try catch finally 구문은 문법적으로 완전히 동일하다.
 - Kotlin에서는 try catch가 expression이다.
- Kotlin에서 모든 예외는 Unchecked Exception이다.
- Kotlin에서는 try with resources 구문이 없다. 대신 코틀린의 언어적 특징을 활용해 close를 호출해준다.

Lec 08. 코틀린에서 함수를 다루는 방법

1. 함수 선언 문법
2. default parameter
3. named argument (parameter)
4. 같은 타입의 여러 파라미터 받기 (가변인자)

1. 함수 선언 문법

두 정수를 받아 더 큰 정수를 반환하는 예제



```
public int max(int a, int b) {  
    if (a > b) {  
        return a;  
    }  
    return b;  
}
```

1. 함수 선언 문법

두 정수를 받아 더 큰 정수를 반환하는 예제



```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```

1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```

접근 지시어, public은 생략 가능하다.

1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```

함수를 의미하는 키워드

1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```

함수 이름

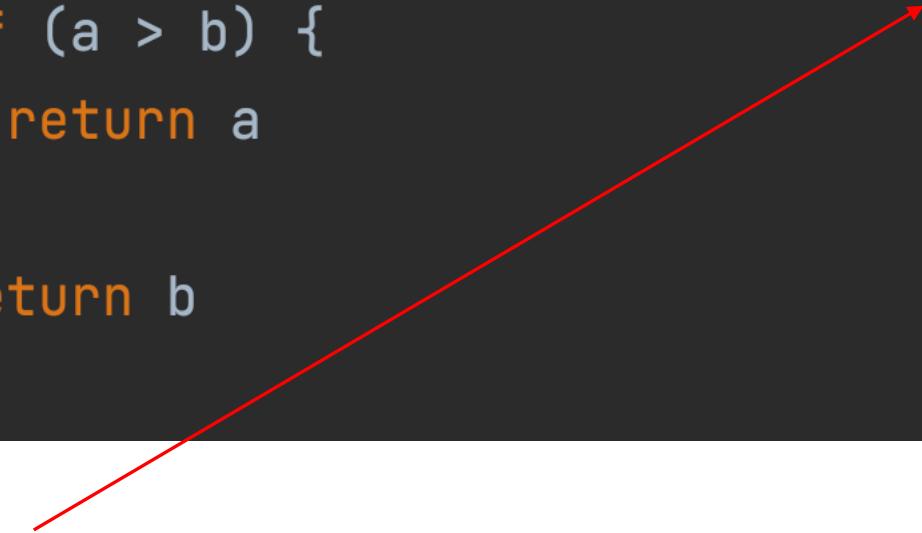
1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```

함수의 매개변수, **매개변수명: 타입**

1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```



함수의 반환 타입 (Unit인 경우 생략 가능)

1. 함수 선언 문법

```
public fun max(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    }  
    return b  
}
```



중괄호 안의 본문

1. 함수 선언 문법

```
fun max(a: Int, b: Int): Int {  
    return if (a > b) {  
        a  
    } else {  
        b  
    }  
}
```

if - else 는 expression

1. 함수 선언 문법

```
fun max(a: Int, b: Int): Int =  
    if (a > b) {  
        a  
    } else {  
        b  
    }
```

함수가 하나의 결과값이면 block 대신 = 사용 가능

1. 함수 선언 문법

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

한 줄로 변경 가능

1. 함수 선언 문법

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

=을 사용하는 경우 반환 타입 생략 가능

1. 함수 선언 문법

block { } 을 사용하는 경우에는 반환 타입이 Unit이 아니면,
명시적으로 작성해주어야 합니다!

1. 함수 선언 문법

함수는 클래스 안에 있을 수도, 파일 최상단에 있을 수도 있습니다.
또한, 한 파일 안에 여러 함수들이 있을 수도 있습니다.

2. default parameter

주어진 문자열을 N번 출력하는 예제

2. default parameter



```
public void repeat(String str, int num, boolean useNewLine) {  
    for (int i = 1; i <= num; i++) {  
        if (useNewLine) {  
            System.out.println(str);  
        } else {  
            System.out.print(str);  
        }  
    }  
}
```

2. default parameter

```
public void repeat(String str, int num, boolean useNewLine) {  
    for (int i = 1; i <= num; i++) {  
        if (useNewLine) {  
            System.out.println(str);  
        } else {  
            System.out.print(str);  
        }  
    }  
}
```

많은 코드에서 userNewLine에 true를 사용한다면?!

2. default parameter

```
public void repeat(String str, int num) {  
    repeat(str, num, useNewLine: true);  
}
```

Java의 오버로딩(OverLoading) 활용!

2. default parameter

```
public void repeat(String str, int num) {  
    repeat(str, num, useNewLine: true);  
}
```

많은 코드에서 출력을 3회씩 사용하고 있다면?!

2. default parameter

```
public void repeat(String str, int num, boolean useNewLine) {  
    for (int i = 1; i <= num; i++) {  
        if (useNewLine) {  
            System.out.println(str);  
        } else {  
            System.out.print(str);  
        }  
    }  
}  
  
public void repeat(String str, int num) {  
    repeat(str, num, useNewLine: true);  
}  
  
public void repeat(String str) {  
    repeat(str, num: 3, useNewLine: true);  
}
```

다시 한 번 오버로딩! 총 3개의 함수가 존재

2. default parameter

메소드를 3개나 만들어야 하나?!

2. default parameter

```
fun repeat(  
    str: String,  
    num: Int = 3,  
    useNewLine: Boolean = true  
) {  
    for (i in 1..num) {  
        if (useNewLine) {  
            println(str)  
        } else {  
            println(str)  
        }  
    }  
}
```



2. default parameter

```
fun repeat(  
    str: String,  
    num: Int = 3,  
    useNewLine: Boolean = true  
) {  
  
    for (i in 1..num) {  
        if (useNewLine) {  
            println(str)  
        } else {  
            println(str)  
        }  
    }  
}
```



밖에서 파라미터를 넣어주지 않으면
기본값을 사용하자!

default parameter

2. default parameter

물론 코틀린에도 Java와 동일하게 오버로드 기능은 있습니다.

3. named argument

```
fun repeat(  
    str: String,  
    num: Int = 3,  
    useNewLine: Boolean = true  
) {  
    for (i in 1..num) {  
        if (useNewLine) {  
            println(str)  
        } else {  
            println(str)  
        }  
    }  
}
```



repeat을 호출할 때,
num은 3을 그대로 쓰고
useNewLine은 false를 쓰고 싶다!

어떻게?!

3. named argument



```
repeat(str: "Hello World", useNewLine = false)
```



매개변수 이름을 통해 직접 지정
지정되지 않은 매개변수는 기본값 사용

3. named argument

builder를 직접 만들지 않고 builder의 장점을 가지게 된다.

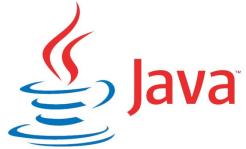
3. named argument

Kotlin에서 Java 함수를 가져다 사용할 때는
named argument를 사용할 수 없다.

4. 같은 타입의 여러 파라미터 받기 (가변인자)

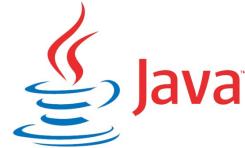
문자열을 N개 받아 출력하는 예제

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
public static void printAll(String... strings) {  
    for (String str : strings) {  
        System.out.println(str);  
    }  
}
```

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
public static void printAll(String... strings) {  
    for (String str : strings) {  
        System.out.println(str);  
    }  
}
```

타입... 을 쓰면 가변인자 사용!

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
String[] array = new String[]{"A", "B", "C"};
printAll(array);
```

```
printAll( ...strings: "A", "B", "C");
```

배열을 직접 넣거나, comma를 이용해 여러 파라미터를 넣거나

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
fun printAll(vararg strings: String) {  
    for (str in strings) {  
        println(str)  
    }  
}
```

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
fun printAll(vararg strings: String) {  
    for (str in strings) {  
        println(str)  
    }  
}
```

...을 타입 뒤에 쓰는 대신 제일 앞에 vararg를 적어주어야 한다!

4. 같은 타입의 여러 파라미터 받기 (가변인자)



```
val array = arrayOf("A", "B", "C")
printAll(*array)

printAll(...strings: "A", "B", "C")
```

배열을 바로 넣는 대신 스프레드 연산자 (*)를 붙여주어야 한다

Lec 08. 코틀린에서 함수를 다루는 방법

- 함수의 문법은 Java와 다르다!

```
접근지시어 fun 함수이름(파라미터): 반환타입 {  
}
```

Lec 08. 코틀린에서 함수를 다루는 방법

- 함수의 문법은 Java와 다르다!
- body가 하나의 값으로 간주되는 경우 block을 없앨 수도 있고, block이 없다면 반환 타입을 없앨 수도 있다.

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

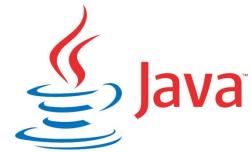
Lec 08. 코틀린에서 함수를 다루는 방법

- 함수의 문법은 Java와 다르다!
- body가 하나의 값으로 간주되는 경우 block을 없앨 수도 있고, block이 없다면 반환 타입을 없앨 수도 있다.
- 함수 파라미터에 기본값을 설정해줄 수 있다.
- 함수를 호출할 때 특정 파라미터를 지정해 넣어줄 수 있다.
- 가변인자에는 vararg 키워드를 사용하며, 가변인자 함수를 배열과 호출할 때는 * 를 붙여주어야 한다.

Lec 09. 코틀린에서 클래스를 다루는 방법

1. 클래스와 프로퍼티
2. 생성자와 init
3. 커스텀 getter, setter
4. backing field

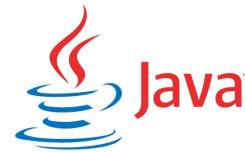
1. 클래스와 프로퍼티



```
public class JavaPerson {  
  
    private final String name;  
    private int age;  
  
}
```

개명이 불가능한 나라에 사는 Person 클래스

1. 클래스와 프로퍼티



```
public class JavaPerson {  
  
    private final String name;  
    private int age;  
  
}
```

현재로써는 name을 초기화할 수 없어 에러가 나옵니다!

1. 클래스와 프로퍼티



```
public class JavaPerson {  
  
    private final String name;  
    private int age;  
  
}
```

생성자와 getter, setter를 만들어 줄게요!

1. 클래스와 프로퍼티



```
public class JavaPerson {  
  
    private final String name;  
    private int age;  
  
    public JavaPerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

name은 setter가 없죠!

1. 클래스와 프로퍼티



```
class Person constructor(name: String, age: Int) {  
    val name = name  
    var age = age  
}
```

프로퍼티 = 필드 + getter + setter

코틀린에서는 필드만 만들면 getter, setter를 자동으로 만들어준다

1. 클래스와 프로퍼티



```
class Person constructor(name: String, age: Int) {  
    val name = name  
    var age = age  
}
```

constructor는 생략할 수 있다

1. 클래스와 프로퍼티



```
class Person constructor(name: String, age: Int) {  
    val name = name  
    var age = age  
}
```

The code snippet shows a Kotlin class definition for 'Person'. It includes a constructor that takes 'name' (String) and 'age' (Int) as parameters. Inside the class body, there are two declarations: 'val name = name' and 'var age = age'. Both of these declarations are highlighted with a red rectangular box. The code is presented on a dark background with white text and syntax highlighting.

클래스의 필드 선언과 생성자를 동시에 선언할 수 있다.

1. 클래스와 프로퍼티



```
public class JavaPerson {  
  
    private final String name;  
    private int age;  
  
    public JavaPerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```



```
class Person(  
    val name: String,  
    var age: Int  
)
```

1. 클래스와 프로퍼티



```
val person = Person(name: "최태현", age: 100)
println(person.name)
person.age = 10
println(person.age)
```

.필드를 통해 getter와 setter를 바로 호출한다

1. 클래스와 프로퍼티



```
val javaPerson = JavaPerson(name: "최태현", age: 100)
println(javaPerson.name)
javaPerson.age = 10
println(javaPerson.age)
```

Java 클래스에 대해서도 .필드로 getter, setter를 사용한다.

2. 생성자와 init

클래스가 생성되는 시점에 나이를 검증해보자!

2. 생성자와 init



```
public JavaPerson(String name, int age) {  
    if (age <= 0) {  
        throw new IllegalArgumentException(String.format("나이(%s)는 10이상이어야 합니다", age));  
    }  
    this.name = name;  
    this.age = age;  
}
```

2. 생성자와 init



```
class Person(  
    val name: String,  
    var age: Int  
)
```

어디서 나이를 검증해야 하지...?!

2. 생성자와 init



```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
}
```

init (초기화) 블록은 생성자가 호출되는 시점에 호출된다.

2. 생성자와 init



```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
}
```

값을 적절히 만들어주거나, validation 하는 로직

2. 생성자와 init

최초로 태어나는 아기는 무조건 1살이니, 생성자를 하나 더 만들자!

2. 생성자와 init



```
public JavaPerson(String name, int age) {  
    if (age <= 0) {  
        throw new IllegalArgumentException(String.format("나이(%s)는 1이상이어야 합니다", age));  
    }  
    this.name = name;  
    this.age = age;  
}
```

```
public JavaPerson(String name) {  
    this(name, age: 1);  
}
```

2. 생성자와 init



```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1)  
}
```

constructor(파라미터)로 생성자를 추가!

2. 생성자와 init



```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, 1)  
}
```

: this(name, 1)로 위에 있는 생성자를 호출!

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1)  
}
```

주생성자
(primary constructor)

반드시 존재해야 한다.

단, 주생성자에 파라미터가
하나도 없다면 생략 가능!

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1)  
}
```

부생성자
(secondary constructor)

있을 수도 있고~
없을 수도 있다.

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1)  
}
```

부생성자
(secondary constructor)

최종적으로 주생성자를
this로 호출해야 한다.

body를 가질 수 있다.

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this(name: "최태현") {  
        println("부생성자 2")  
    }  
}
```

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this( name: "최태현" ) {  
        println("부생성자 2")  
    }  
}
```

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
  
    constructor(name: String) : this(name, age: 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this(name: "최태현") {  
        println("부생성자 2")  
    }  
}
```

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
        println("초기화 블록")  
    }  
  
    constructor(name: String) : this(name, 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this(name: "최태현") {  
        println("부생성자 2")  
    }  
}
```

본문은 역순으로 실행됩니다!

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
        println("초기화 블록")  
    }  
  
    constructor(name: String) : this(name, 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this(name: "최태현") {  
        println("부생성자 2")  
    }  
}
```

본문은 역순으로 실행됩니다!

2. 생성자와 init

```
class Person(  
    val name: String,  
    var age: Int,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
        println("초기화 블록")  
    }  
  
    constructor(name: String) : this(name, 1) {  
        println("부생성자 1")  
    }  
  
    constructor() : this(name: "최태현") {  
        println("부생성자 2")  
    }  
}
```

본문은 역순으로 실행됩니다!

2. 생성자와 init

그런데 사실... 부생성자보다는 default parameter를 권장합니다!

2. 생성자와 init

```
class Person(  
    val name: String = "최태현",  
    var age: Int = 1,  
) {  
  
    init {  
        if (age < 0) {  
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다")  
        }  
    }  
}
```

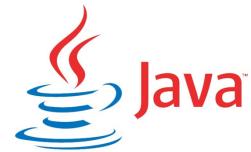
2. 생성자와 init

Converting과 같은 경우 부생성자를 사용할 수 있지만,
그보다는 정적 팩토리 메소드를 추천 드립니다!

3. 커스텀 getter, setter

성인인지 확인하는 기능을 추가해보겠습니다.

3. 커스텀 getter, setter



```
public boolean isAdult() {  
    return this.age >= 20;  
}
```

3. 커스텀 getter, setter



```
fun isAdult(): Boolean {  
    return this.age >= 20  
}
```

함수 대신 프로퍼티로도 만들 수 있다!

3. 커스텀 getter, setter

```
fun isAdult(): Boolean {  
    return this.age >= 20  
}
```

```
val isAdult: Boolean  
    get() = this.age >= 20
```

```
val isAdult: Boolean  
    get() {  
        return this.age >= 20  
    }
```

3. 커스텀 getter, setter

```
fun isAdult(): Boolean {  
    return this.age >= 20  
}
```

Custom getter

```
val isAdult: Boolean  
    get() = this.age >= 20
```

```
val isAdult: Boolean  
    get() {  
        return this.age >= 20  
    }
```

3. 커스텀 getter, setter

모두 동일한 기능이고 표현 방법만 다르다! (가독성)
어떤 방법이 나을까?!

3. 커스텀 getter, setter

(개인적인 의견)

객체의 속성이라면, custom getter
그렇지 않다면 함수

3. 커스텀 getter, setter

Custom getter를 사용하면 자기 자신을 변형해 줄 수도 있다!

3. 커스텀 getter, setter

name을 get할때 무조건 대문자로 바꾸어 줘 볼게요!

3. 커스텀 getter, setter



```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
        get() = field.uppercase()  
  
}
```

3. 커스텀 getter, setter



```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
        get() = field.uppercase()  
  
}
```

주생성자에서 받은 name을
불변 프로퍼티 name에 바로 대입

3. 커스텀 getter, setter



```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
        get() = field.uppercase()  
  
}
```

name에 대한 Custom getter를 만들 때 field를 사용

4. backing field



```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
        get() = field.uppercase()  
}
```

왜 `field`를 사용하는 것일까?! 이게 무엇이지?!

4. backing field



```
val name: String = name
    get() = name.uppercase()
```

4. backing field



```
val name: String = name  
    get() = name.uppercase()
```

name은 name에 대한 getter를 호출 하니까 다시 get을 부른다!

4. backing field



```
val name: String = name
    get() = name.uppercase()
```

getter안에는 다시 name이 있다..!

4. backing field



```
val name: String = name
    get() = name.uppercase()
```



다시 getter를 부른다...!! **무한루프** 발생!

4. backing field



```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
        get() = field.uppercase()  
  
}
```

무한루프를 막기 위한 예약어, 자기 자신을 가리킨다

4. backing field



backing field

```
class Person(  
    name: String,  
    var age: Int,  
) {  
  
    val name: String = name  
    get() = field.uppercase()  
}
```

무한루프를 막기 위한 예약어, 자기 자신을 가리킨다

4. backing field

개인적으로는 custom getter에서
backing field를 쓰는 경우는 드물었습니다!

4. backing field

```
val upperCaseName: String  
    get() = this.name.uppercase()
```

3. 커스텀 getter, setter

name을 **set**할때 무조건 대문자로 바꾸어 줘 볼게요!

3. 커스텀 getter, setter

```
var name: String = name
    set(value) {
        field = value.uppercase()
    }
```

3. 커스텀 getter, setter

```
var name: String = name
    set(value) {
        field = value.uppercase()
    }
```

3. 커스텀 getter, setter

사실은 Setter 자체를 지양하기 때문에
custom setter도 잘 안쓴다...!

Lec 09. 코틀린에서 클래스를 다루는 방법

- 코틀린에서는 필드를 만들면 getter와 (필요에 따라) setter가 자동으로 생긴다.
- 때문에 이를 **프로퍼티**라고 부른다.

```
class Person(  
    val name: String = "최태현",  
    var age: Int = 1  
)
```

Lec 09. 코틀린에서 클래스를 다루는 방법

- 코틀린에서는 필드를 만들면 getter와 (필요에 따라) setter가 자동으로 생긴다.
 - 때문에 이를 프로퍼티 라고 부른다.
- 코틀린에서는 **주생성자**가 필수이다.

Lec 09. 코틀린에서 클래스를 다루는 방법

- 코틀린에서는 필드를 만들면 getter와 (필요에 따라) setter가 자동으로 생긴다.
 - 때문에 이를 프로퍼티 라고 부른다.
- 코틀린에서는 주생성자가 필수이다.
- 코틀린에서는 consturctor 키워드를 사용해 **부생성자**를 추가로 만들 수 있다.
 - 단, default parameter나 정적 팩토리 메소드를 추천한다.

Lec 09. 코틀린에서 클래스를 다루는 방법

- 실제 메모리에 존재하는 것과 무관하게 **custom getter**와 **custom setter**를 만들 수 있다.

Lec 09. 코틀린에서 클래스를 다루는 방법

- 실제 메모리에 존재하는 것과 무관하게 custom getter와 custom setter를 만들 수 있다.
- custom getter, custom setter에서 무한루프를 막기 위해 **field**라는 키워드를 사용한다.
 - 이를 **backing field** 라고 부른다.

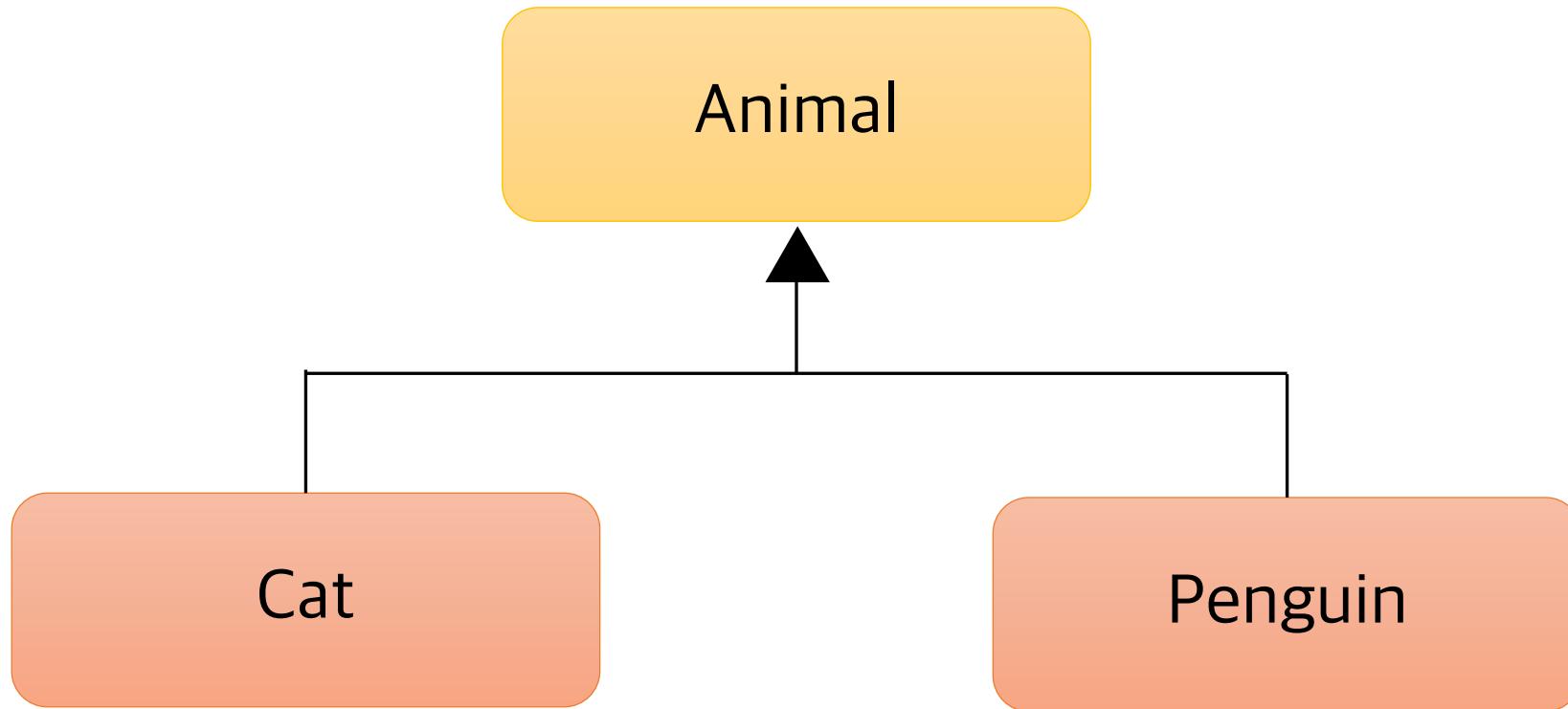
Lec 10. 코틀린에서 상속을 다루는 방법

1. 추상 클래스
2. 인터페이스
3. 클래스를 상속할 때 주의할 점
4. 상속 관련 지시어 정리

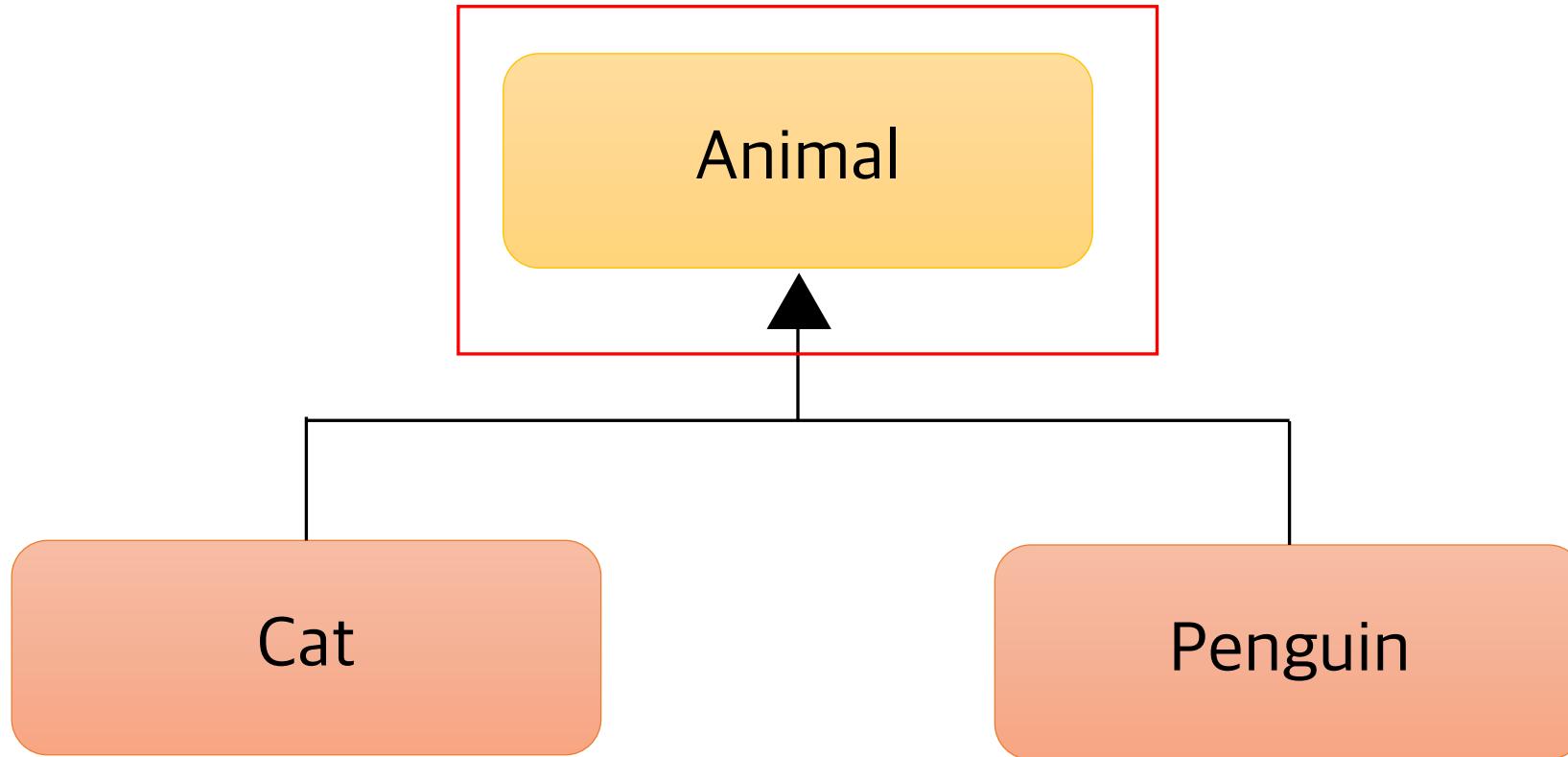
1. 추상 클래스

Animal이란 추상클래스를 구현한 Cat, Penguin

1. 추상 클래스



1. 추상 클래스



1. 추상 클래스

```
① public abstract class JavaAnimal {  
  
    protected final String species;  
    protected final int legCount;  
  
    public JavaAnimal(String species, int legCount) {  
        this.species = species;  
        this.legCount = legCount;  
    }  
  
    ② abstract public void move();  
  
    public String getSpecies() {  
        return species;  
    }  
  
    ③ public int getLegCount() {  
        return legCount;  
    }  
  
}
```



1. 추상 클래스



```
abstract class Animal(
    protected val species: String,
    protected val legCount: Int,
) {
    abstract fun move()
}
```

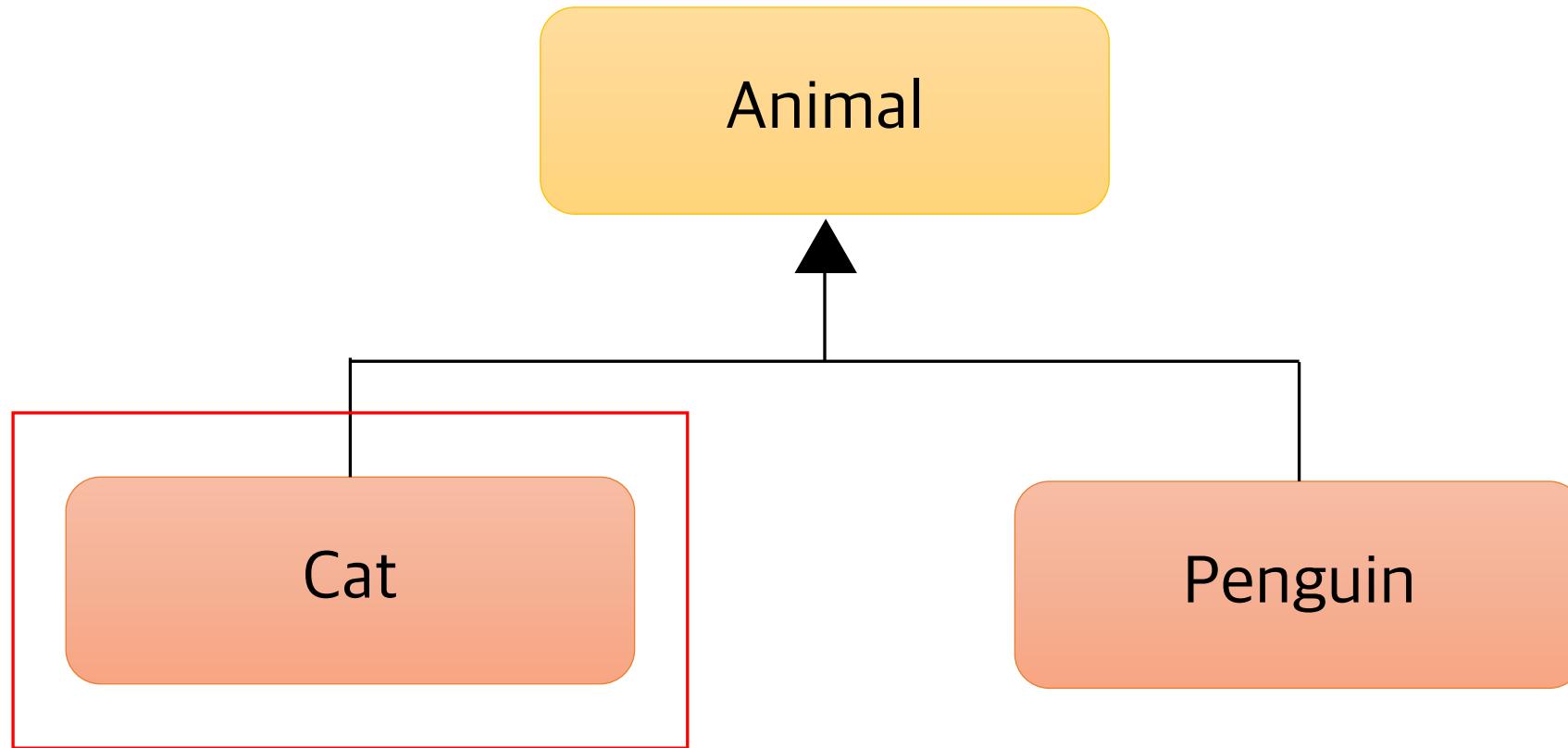
1. 추상 클래스



```
abstract class Animal(  
    protected val species: String,  
    protected val legCount: Int,  
) {  
  
    abstract fun move()  
  
}
```

크게 다른건 아직 보이지 않네요!

1. 추상 클래스



1. 추상 클래스



```
public class JavaCat extends JavaAnimal {

    public JavaCat(String species) {
        super(species, legCount: 4);
    }

    @Override
    public void move() {
        System.out.println("고양이가 사뿐 사뿐 걸어가~");
    }
}
```

1. 추상 클래스



```
class Cat(  
    species: String  
) : Animal(species, legCount: 4) {  
  
    override fun move() {  
        println("고양이가 사뿐 사뿐 걸어가~")  
    }  
}
```

1. 추상 클래스



```
class Cat(  
    species: String  
) : Animal(species, legCount: 4) {  
  
    override fun move() {  
        println("고양이가 사뿐 사뿐 걸어가~")  
    }  
}
```

extends 키워드를 사용하지 않고 : 을 사용한다

1. 추상 클래스



```
class Cat(  
    species: String  
) : Animal(species, legCount: 4) {  
  
    override fun move() {  
        println("고양이가 사뿐 사뿐 걸어가~")  
    }  
}
```

상위 클래스의 생성자를 바로 호출한다

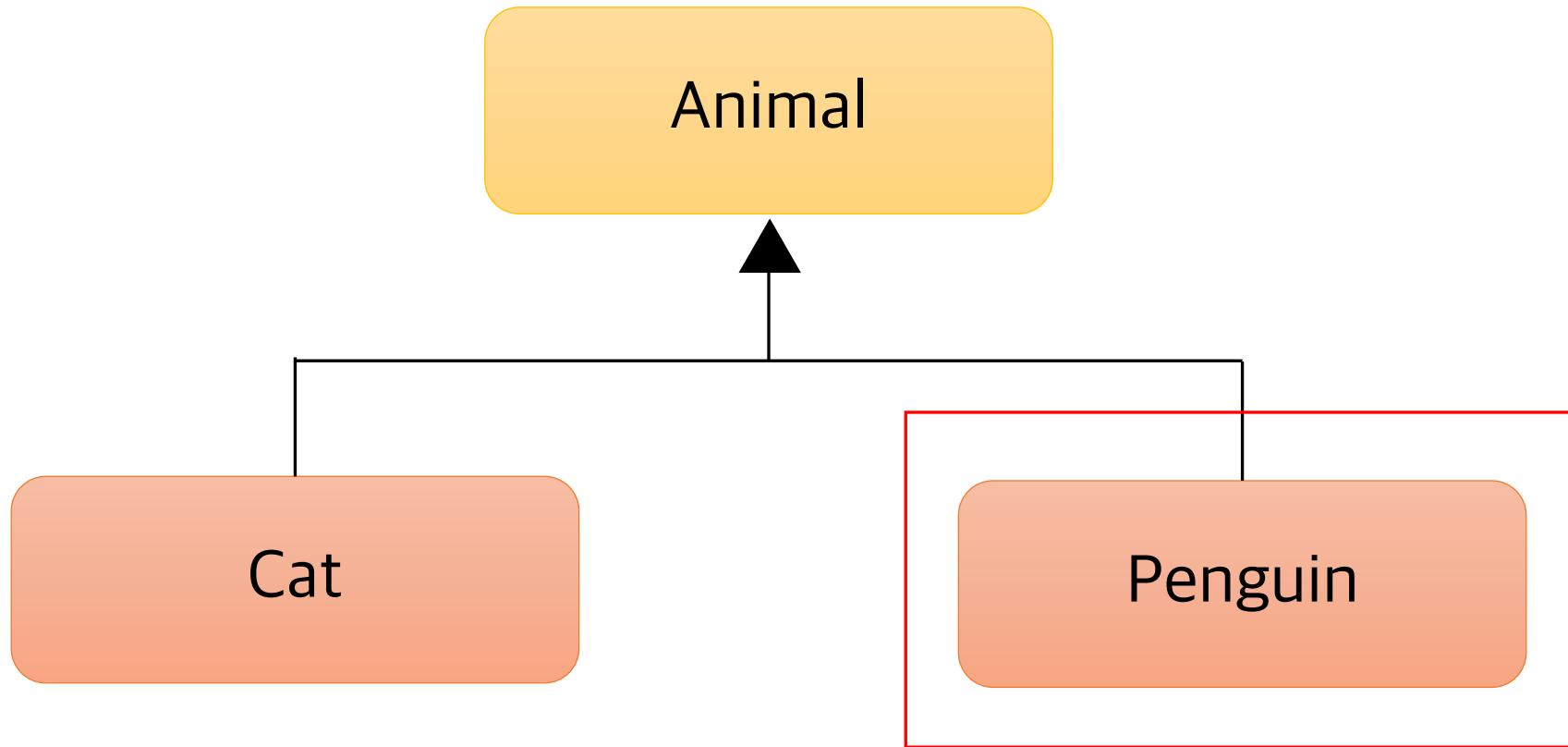
1. 추상 클래스



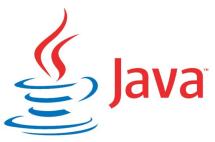
```
class Cat(  
    species: String  
) : Animal(species, legCount: 4) {  
  
    override fun move() {  
        println("고양이가 사뿐 사뿐 걸어가~")  
    }  
}
```

override를 필수적으로 붙여 주어야 한다

1. 추상 클래스



1. 추상 클래스



```
public final class JavaPenguin extends JavaAnimal {

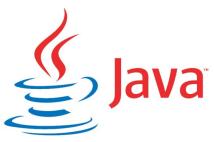
    private final int wingCount;

    public JavaPenguin(String species) {
        super(species, legCount: 2);
        this.wingCount = 2;
    }

    @Override
    public void move() {
        System.out.println("펭귄이 움직입니다~ 펙페");
    }

    @Override
    public int getLegCount() {
        return super.legCount + this.wingCount;
    }
}
```

1. 추상 클래스



```
public final class JavaPenguin extends JavaAnimal {  
  
    private final int wingCount;  
  
    public JavaPenguin(String species) {  
        super(species, legCount: 2);  
        this.wingCount = 2;  
    }  
  
    @Override  
    public void move() {  
        System.out.println("펭귄이 움직입니다~ 펙페");  
    }  
  
    @Override  
    public int getLegCount() {  
        return super.legCount + this.wingCount;  
    }  
}
```

1. 추상 클래스



```
abstract class Animal(  
    protected val species: String,  
    protected open val legCount: Int,  
) {  
    abstract fun move()  
}
```

추상 프로퍼티가 아니라면, 상속받을때 **open**을 꼭 붙여야 한다.

1. 추상 클래스



```
class Penguin(  
    species: String,  
) : Animal(species, legCount: 2) {  
  
    private val wingCount: Int = 2  
  
    override fun move() {  
        println("펭귄이 움직입니다~ 펙페")  
    }  
  
    override val legCount: Int  
        get() = super.legCount + this.wingCount  
}
```

추상클래스에서 자동으로 만들어진 getter를 override

1. 추상 클래스



```
class Penguin(  
    species: String,  
) : Animal(species, legCount: 2) {  
  
    private val wingCount: Int = 2  
  
    override fun move() {  
        println("펭귄이 움직입니다~ 펙페")  
    }  
  
    override val legCount: Int  
        get() = super.legCount + this.wingCount  
  
}
```

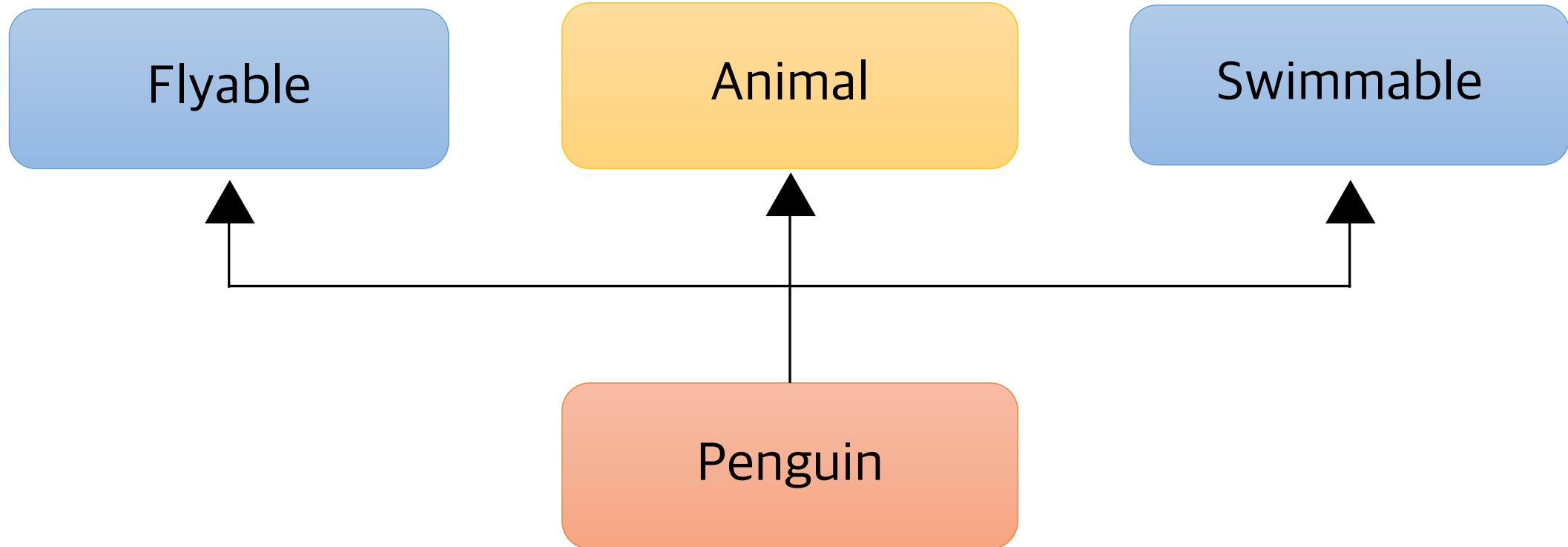
1. 추상 클래스

Java, Kotlin 모두 추상 클래스는 인스턴스화 할 수 없다!

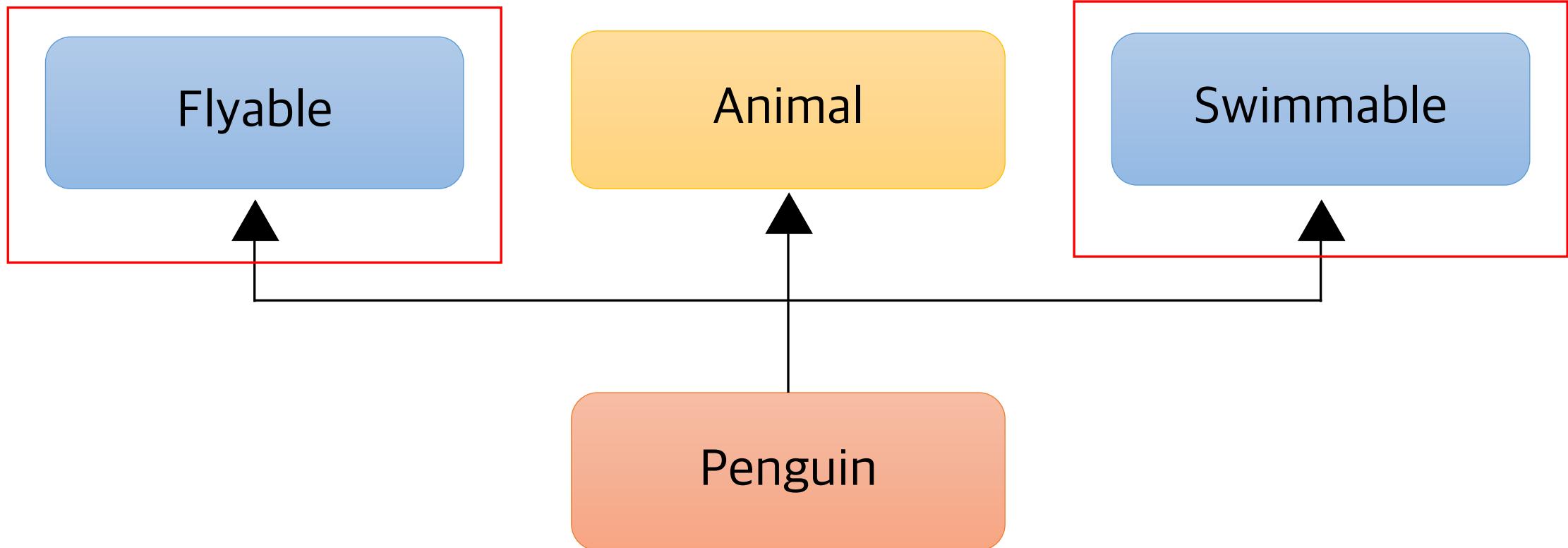
2. 인터페이스

Flyable과 Swimmable을 구현한 Penguin

2. 인터페이스



2. 인터페이스



2. 인터페이스



```
public interface JavaSwimable {  
    default void act() {  
        System.out.println("어푸 어푸");  
    }  
}
```

```
public interface JavaFlyable {  
    default void act() {  
        System.out.println("파닥 파닥");  
    }  
}
```

2. 인터페이스



```
public interface JavaSwimable {  
    default void act() {  
        System.out.println("어푸 어푸");  
    }  
}
```

```
public interface JavaFlyable {  
    default void act() {  
        System.out.println("파닥 파닥");  
    }  
}
```

2. 인터페이스



```
interface Swimable {  
    fun act() {  
        println("어푸 어푸")  
    }  
}
```

```
interface Flyable {  
    fun act() {  
        println("파닥 파닥")  
    }  
}
```

2. 인터페이스



```
interface Swimable {  
    fun act() {  
        println("어푸 어푸")  
    }  
}
```

```
interface Flyable {  
    fun act() {  
        println("파닥 파닥")  
    }  
}
```

default 키워드 없이 메소드 구현이 가능하다

2. 인터페이스

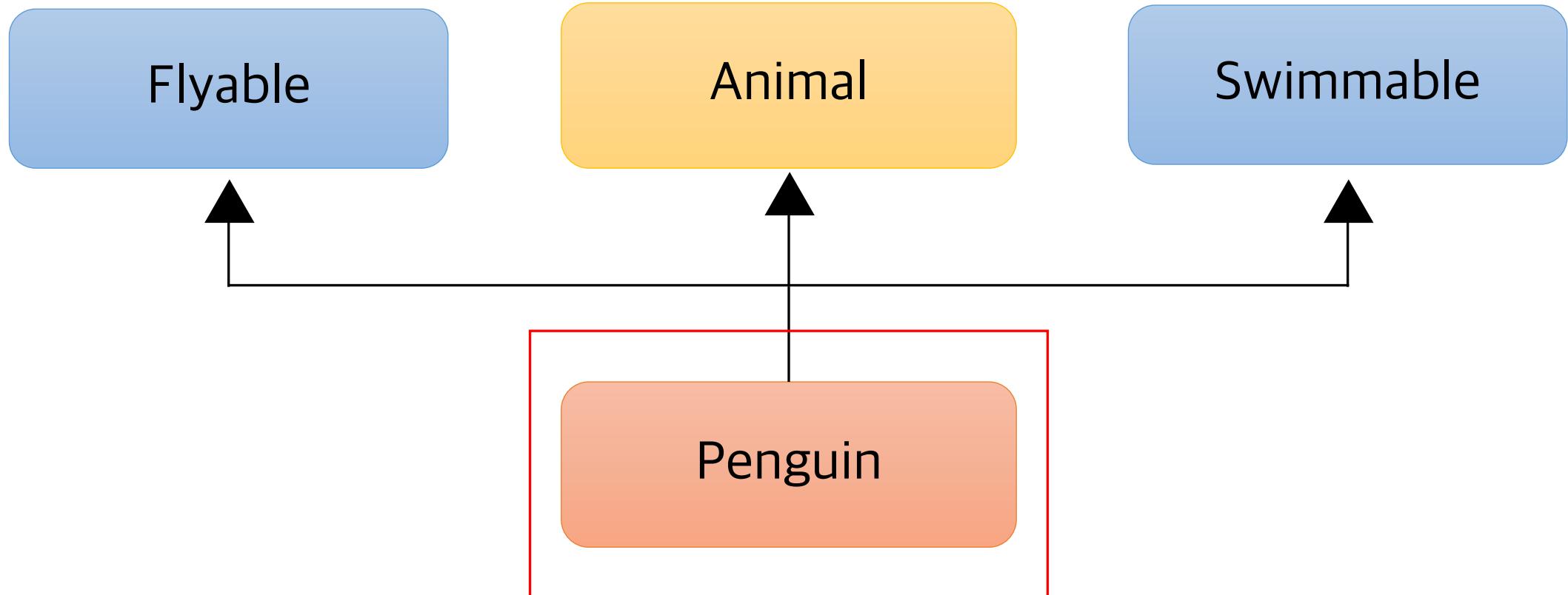


```
interface Swimable {  
    fun act() {  
        println("어푸 어푸")  
    }  
}
```

```
interface Flyable {  
    fun act() {  
        println("파닥 파닥")  
    }  
}
```

Kotlin에서도 추상 메소드를 만들 수 있다.

2. 인터페이스



2. 인터페이스



```
public final class JavaPenguin extends JavaAnimal implements JavaFlyable, JavaSwimable {  
    @Override  
    public void act() {  
        JavaSwimable.super.act();  
        JavaFlyable.super.act();  
    }  
}
```

2. 인터페이스



```
class Penguin(  
    species: String,  
) : Animal(species, legCount: 2), Swimable, Flyable {  
  
    override fun act() {  
        super<Swimable>.act()  
        super<Flyable>.act()  
    }  
}
```

2. 인터페이스



```
class Penguin(  
    species: String,  
) : Animal(species, legCount: 2), Swimable, Flyable {  
  
    override fun act() {  
        super<Swimable>.act()  
        super<Flyable>.act()  
    }  
}
```

인터페이스 구현도 `:` 을 사용한다.

2. 인터페이스



```
class Penguin(  
    species: String,  
) : Animal(species, legCount: 2), Swimable, Flyable {  
  
    override fun act() {  
        super<Swimable>.act()  
        super<Flyable>.act()  
    }  
}
```

중복되는 인터페이스를 특정할때 **super<타입>.함수** 사용

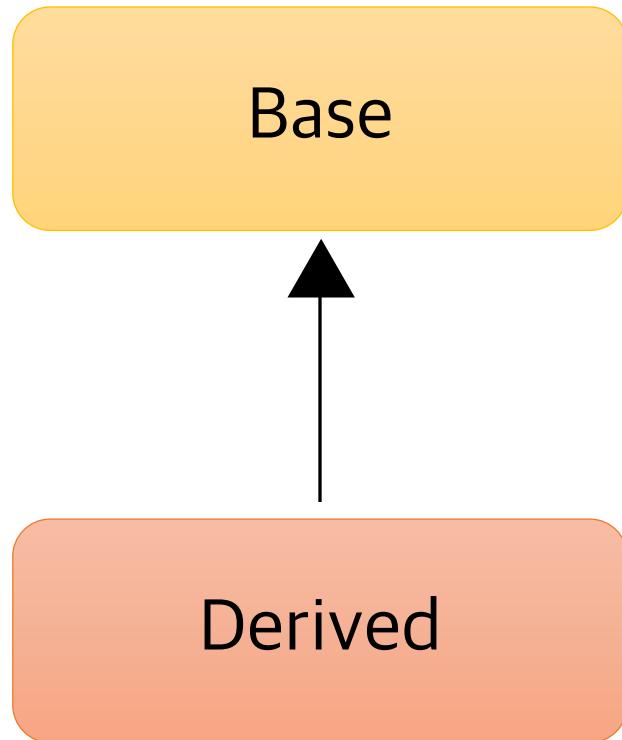
2. 인터페이스

Java, Kotlin 모두 인터페이스를 인스턴스화 할 수 없습니다!

2. 인터페이스

Kotlin에서는 backing field가 없는 프로퍼티를
Interface에 만들 수 있다

3. 클래스를 상속받을 때 주의할 점



3. 클래스를 상속받을 때 주의할 점

```
open class Base(  
    open val number: Int = 100  
) {  
    init {  
        println("Base Class")  
        println(number)  
    }  
}  
  
class Derived(  
    override val number: Int  
) : Base(number) {  
    init {  
        println("Derived Class")  
    }  
}
```

3. 클래스를 상속받을 때 주의할 점

```
open class Base(  
    open val number: Int = 100  
) {  
    init {  
        println("Base Class")  
        println(number)  
    }  
}  
  
class Derived(  
    override val number: Int  
) : Base(number) {  
    init {  
        println("Derived Class")  
    }  
}
```

3. 클래스를 상속받을 때 주의할 점

```
open class Base(  
    open val number: Int = 100  
) {  
    init {  
        println("Base Class")  
        println(number)  
    }  
}  
  
class Derived(  
    override val number: Int  
) : Base(number) {  
    init {  
        println("Derived Class")  
    }  
}
```

3. 클래스를 상속받을 때 주의할 점

```
open class Base(  
    open val number: Int = 100  
) {  
    init {  
        println("Base Class")  
        println(number)  
    }  
}  
  
class Derived(  
    override val number: Int  
) : Base(number) {  
    init {  
        println("Derived Class")  
    }  
}
```

3. 클래스를 상속받을 때 주의할 점

```
open class Base(  
    open val number: Int = 100  
) {  
    init {  
        println("Base Class")  
        println(number)  
    }  
}  
  
class Derived(  
    override val number: Int  
) : Base(number) {  
    init {  
        println("Derived Class")  
    }  
}
```

3. 클래스를 상속받을 때 주의할 점

상위 클래스를 설계할 때
생성자 또는 초기화 블록에 사용되는 프로퍼티에는
open을 피해야 한다

4. 상속 관련 키워드 4가지 정리

1. final : override를 할 수 없게 한다. default로 보이지 않게 존재한다.
2. open : override를 열어 준다.
3. abstract : 반드시 override 해야 한다.
4. override : 상위 타입을 오버라이드 하고 있다.

Lec 10. 코틀린에서 상속을 다루는 방법

- 상속 또는 구현을 할 때에 `:` 을 사용해야 한다.

Lec 10. 코틀린에서 상속을 다루는 방법

- 상속 또는 구현을 할 때에 : 을 사용해야 한다.
- 상위 클래스 상속을 구현할 때 생성자를 반드시 호출해야 한다.
- override를 필수로 붙여야 한다.
- 추상 멤버가 아니면 기본적으로 오버라이드가 불가능하다.
 - **open**을 사용해주어야 한다.

Lec 10. 코틀린에서 상속을 다루는 방법

- 상속 또는 구현을 할 때에 : 을 사용해야 한다.
- 상위 클래스 상속을 구현할 때 생성자를 반드시 호출해야 한다.
- override를 필수로 붙여야 한다.
- 추상 멤버가 아니면 기본적으로 오버라이드가 불가능하다.
 - open을 사용해주어야 한다.
- 상위 클래스의 생성자 또는 초기화 블록에서 open 프로퍼티를 사용하면 얘기치 못한 버그가 생길 수 있다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

1. 자바와 코틀린의 가시성 제어
2. 코틀린 파일의 접근 제어
3. 다양한 구성요소의 접근 제어
4. Java와 Kotlin을 함께 사용할 경우 주의할 점

1. Java와 코틀린의 가시성 제어

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



1. Java와 코틀린의 가시성 제어

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



public	모든 곳에서 접근 가능



1. Java와 코틀린의 가시성 제어

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



public	모든 곳에서 접근 가능
protected	선언된 클래스 또는 하위 클래스에서만 접근 가능



1. Java와 코틀린의 가시성 제어

Kotlin에서는 패키지를 namespace를 관리하기 위한 용도로만 사용!
가시성 제어에는 사용되지 않는다.

1. Java와 코틀린의 가시성 제어

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



public	모든 곳에서 접근 가능
protected	선언된 클래스 또는 하위 클래스에서만 접근 가능
internal	같은 모듈에서만 접근 가능



1. Java와 코틀린의 차이성 제어

모듈 : 한 번에 컴파일 되는 Kotlin 코드

IDEA Module

Maven Project

Gradle Source Set

Ant Task <kotlinc>의 호출로 컴파일 파일의 집합

1. Java와 코틀린의 가시성 제어

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



public	모든 곳에서 접근 가능
protected	선언된 클래스 또는 하위 클래스에서만 접근 가능
internal	같은 모듈에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능



1. Java와 코틀린의 가시성 제어

Java의 기본 접근 지시어는 default
Kotlin의 기본 접근 지시어는 **public**

2. 코틀린 파일의 접근 제어

코틀린은 .kt 파일에 변수, 함수, 클래스 여러개를 바로 만들 수 있다.

2. 코틀린 파일의 접근 제어



public	기본값 어디서든 접근할 수 있다.
protected	파일(최상단)에는 사용 불가능
internal	같은 모듈에서만 접근 가능
private	같은 파일 내에서만 접근 가능

3. 다양한 구성요소의 접근 제어

클래스, 생성자, 프로퍼티에 대해 말씀드릴게요!

3. 다양한 구성요소의 접근 제어 - 클래스 안의 멤버



public	모든 곳에서 접근 가능
protected	선언된 클래스 또는 하위 클래스에서만 접근 가능
internal	같은 모듈에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능

3. 다양한 구성요소의 접근 제어 - 생성자

생성자도 가시성 범위는 동일합니다. 단!

3. 다양한 구성요소의 접근 제어 - 생성자



```
class Bus internal constructor(  
    val price: Int  
)
```

A screenshot of a code editor showing a Kotlin class definition. The class name 'Bus' is in orange, and the 'internal constructor' part is highlighted with a red rectangular box. A red arrow points from the explanatory text below to this highlighted area.

생성자에 접근 지시어를 붙이려면, constructor를 써주셔야 합니다!

3. 다양한 구성요소의 접근 제어 - 생성자



Java에서 유틸성 코드를 만들때
abstract class + private constructor를 사용해서 인스턴스화를 막았죠?!

```
public abstract class StringUtils {  
    private StringUtils() {}  
  
    public boolean isDirectoryPath(String path) {  
        return path.endsWith("/");  
    }  
}
```

3. 다양한 구성요소의 접근 제어 - 생성자

Kotlin에서도 비슷하게 가능합니다만...

3. 다양한 구성요소의 접근 제어 - 생성자



파일 최상단에 바로 유틸 함수를 작성하면 편합니다!

```
fun isDirectoryPath(path: String): Boolean {  
    return path.endsWith(suffix: "/")  
}
```

3. 다양한 구성요소의 접근 제어 - 프로퍼티

프로퍼티도 가시성 범위는 동일합니다. 단!
프로퍼티의 가시성을 제어하는 방법으로는...

3. 다양한 구성요소의 접근 제어 - 프로퍼티

```
class Car(  
    internal val name: String,  
    _price: Int  
) {  
  
    var price = _price  
        private set  
}
```

getter, setter 한 번에 접근 지시어를 정하거나

3. 다양한 구성요소의 접근 제어 - 프로퍼티

```
class Car(  
    internal val name: String,  
    _price: Int  
) {  
  
    var price = _price  
        private set  
}
```

Setter에만 추가로 가시성을 부여할 수 있습니다.

4. Java와 Kotlin을 함께 사용할 때 주의할 점

Internal은 바이트 코드 상 public이 된다.
때문에 Java 코드에서는 Kotlin 모듈의 internal 코드를 가져올 수 있다.

4. Java와 Kotlin을 함께 사용할 때 주의할 점

Kotlin의 protected와 Java의 protected는 다르다.
Java는 같은 패키지의 Kotlin protected 멤버에 접근할 수 있다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

public	모든 곳에서 접근 가능
protected	같은 패키지 또는 하위 클래스에서만 접근 가능
default	같은 패키지에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능

public	모든 곳에서 접근 가능
protected	선언된 클래스 또는 하위 클래스에서만 접근 가능
internal	같은 모듈에서만 접근 가능
private	선언된 클래스 내에서만 접근 가능

Lec 11. 코틀린에서 접근 제어를 다루는 방법

- Kotlin에서 패키지는 namespace 관리용이기 때문에 **protected**는 의미가 달라졌다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

- Kotlin에서 패키지는 namespace 관리용이기 때문에 protected는 의미가 달라졌다.
- Kotlin에서는 **default**가 사라지고, 모듈간의 접근을 통제하는 **internal**이 새로 생겼다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

- Kotlin에서 패키지는 namespace 관리용이기 때문에 protected는 의미가 달라졌다.
- Kotlin에서는 default가 사라지고, 모듈간의 접근을 통제하는 internal이 새로 생겼다.
- 생성자에 접근 지시어를 붙일 때는 **constructor**를 명시적으로 써주어야 한다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

- Kotlin에서 패키지는 namespace 관리용이기 때문에 protected는 의미가 달라졌다.
- Kotlin에서는 default가 사라지고, 모듈간의 접근을 통제하는 internal이 새로 생겼다.
- 생성자에 접근 지시어를 붙일 때는 constructor를 명시적으로 써주어야 한다.
- 유틸성 함수를 만들 때는 파일 최상단을 이용하면 편리하다.

Lec 11. 코틀린에서 접근 제어를 다루는 방법

- 프로퍼티의 custom setter에 접근 지시어를 붙일 수 있다.
- Java에서 Kotlin 코드를 사용할 때 internal과 protected는 주의해야 한다.

Lec 12. 코틀린에서 object 키워드를 다루는 방법

1. static 함수와 변수
2. 싱글톤
3. 익명 클래스

1. static 함수와 변수

오늘도 사람을 가져왔습니다

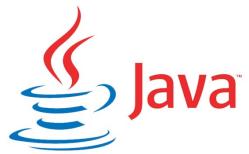


1. static 함수와 변수



```
public class JavaPerson {  
  
    private static final int MIN_AGE = 1;  
  
    public static JavaPerson newBaby(String name) {  
        return new JavaPerson(name, MIN_AGE);  
    }  
  
    private String name;  
  
    private int age;  
  
    private JavaPerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
}
```

1. static 함수와 변수



```
public class JavaPerson {  
    private static final int MIN_AGE = 1;  
  
    public static JavaPerson newBaby(String name) {  
        return new JavaPerson(name, MIN_AGE);  
    }  
  
    private String name;  
  
    private int age;  
  
    private JavaPerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

1. static 함수와 변수



```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object {  
        private val MIN_AGE = 0  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
    }  
}
```

1. static 함수와 변수



```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object {  
        private val MIN_AGE = 0  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
    }  
}
```

static 대신
companion object를 사용

1. static 함수와 변수

static : 클래스가 인스턴스화 될 때 새로운 값이 복제되는게 아니라 정적으로 인스턴스끼리의 값을 공유

companion object : 클래스와 동행하는 유일한 오브젝트

1. static 함수와 변수

```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object {  
        private val MIN_AGE = 0;  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
    }  
}
```

런타임 시에 변수가 할당된다.

1. static 함수와 변수

```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object {  
        private const val MIN_AGE = 0  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
    }  
}
```

컴파일 시에 변수가 할당된다.

진짜 상수에 붙이기 위한 용도.
기본 타입과 String에 붙일 수 있음.

1. static 함수와 변수

```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object {  
        private const val MIN_AGE = 0  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
    }  
}
```

사용법은 Java와 동일하다!

```
println(Person.newBaby("최태현"))
```

1. static 함수와 변수

여기서 Java와 다른 점 한 가지!

1. static 함수와 변수

companion object, 즉 동반객체도 하나의 객체로 간주된다.
때문에 이름을 붙일 수도 있고, interface를 구현할 수도 있다.

1. static 함수와 변수

```
class Person private constructor(  
    private val name: String,  
    private val age: Int,  
) {  
  
    companion object Factory : Log {  
        private const val MIN_AGE = 0  
        fun newBaby(name: String): Person {  
            return Person(name, MIN_AGE)  
        }  
  
        override fun log() {  
            println("LOG")  
        }  
    }  
}
```

1. static 함수와 변수

companion object에 유틸성 함수들을 넣어도 되지만,
최상단 파일을 활용하는 것을 추천 드립니다!

1. static 함수와 변수

Java에서 Kotlin companion object를 사용하려면
@JvmStatic 을 붙여야 합니다!



```
Person person = Person.newBaby("A");
```



```
companion object {
    private const val MIN_AGE = 0

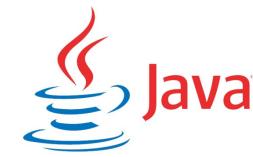
    @JvmStatic
    fun newBaby(name: String): Person {
        return Person(name, MIN_AGE)
    }
}
```

1. static 함수와 변수

만약 companion object 이름이 있으면,
이름을 사용하시면 됩니다!

```
Person person = Person.Factory.newBaby(name: "A");
```

2. 싱글톤



```
public class JavaSingleton {  
  
    private static final JavaSingleton INSTANCE = new JavaSingleton();  
  
    private JavaSingleton() { }  
  
    public static JavaSingleton getInstance() {  
        return INSTANCE;  
    }  
}
```

2. 싱글톤

보여드린 코드에서 동시성 처리를 조금더 해주거나
enum class를 활용하는 방법도 있었습니다!

2. 싱글톤



```
| object Singleton
```

끝입니다...

3. 익명 클래스

특정 인터페이스나 클래스를 상속받은 구현체를
일회성으로 사용할 때 쓰는 클래스

3. 익명 클래스



```
public static void main(String[] args) {
    moveSomething(new Movable() {
        @Override
        public void move() { System.out.println("움직인다~~"); }

        @Override
        public void fly() { System.out.println("난다~~"); }
    });
}

private static void moveSomething(Movable movable) {
    movable.move();
    movable.fly();
}
```

3. 익명 클래스



```
public interface Movable {  
  
    void move();  
  
    void fly();  
  
}
```

3. 익명 클래스



```
fun main() {
    moveSomething(object : Movable {
        override fun move() {
            println("움직인다")
        }

        override fun fly() {
            println("난다~~")
        }
    })
}

private fun moveSomething(movable: Movable) {
    movable.move()
    movable.fly()
}
```

Java에서는 new 타입이름()
Kotlin에서는 object : 타입이름

Lec 12. 코틀린에서 object 키워드를 다루는 방법

- Java의 static 변수와 함수를 만드려면,
Kotlin에서는 **companion object**를 사용해야 한다.

Lec 12. 코틀린에서 object 키워드를 다루는 방법

- Java의 static 변수와 함수를 만드려면,
Kotlin에서는 companion object를 사용해야 한다.
- companion object도 하나의 객체로 간주되기 때문에 이름을
붙일 수 있고, 다른 타입을 상속받을 수도 있다.
- Kotlin에서 싱글톤 클래스를 만들 때 **object** 키워드를 사용한다.

Lec 12. 코틀린에서 object 키워드를 다루는 방법

- Java의 static 변수와 함수를 만드려면,
Kotlin에서는 companion object를 사용해야 한다.
- companion object도 하나의 객체로 간주되기 때문에 이름을
붙일 수 있고, 다른 타입을 상속받을 수도 있다.
- Kotlin에서 싱글톤 클래스를 만들 때 object 키워드를 사용한다.
- Kotlin에서 익명 클래스를 만들 때 **object : 타입**을 사용한다.

Lec 13. 코틀린에서 중첩 클래스를 다루는 방법

1. 중첩 클래스의 종류
2. 코틀린의 중첩 클래스와 내부 클래스

1. 중첩 클래스의 종류



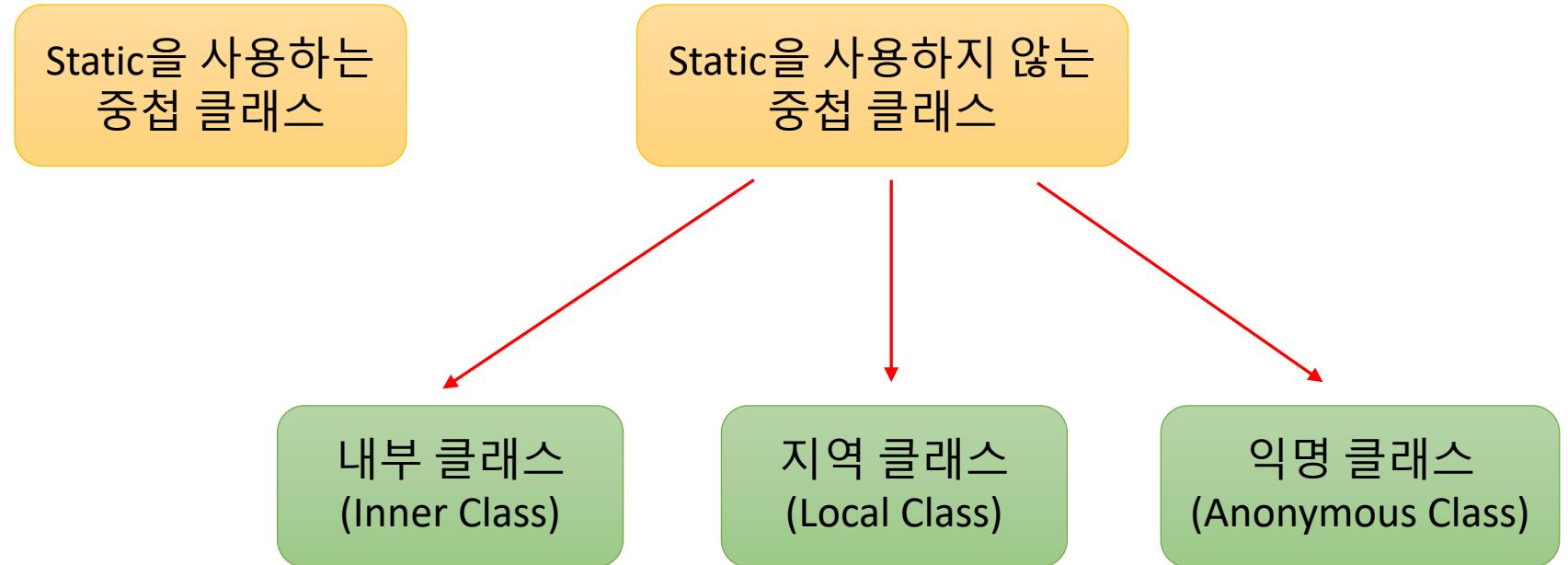
어딘가에 소속되어 있는 클래스, 여러 종류가 있었다!

1. 중첩 클래스의 종류

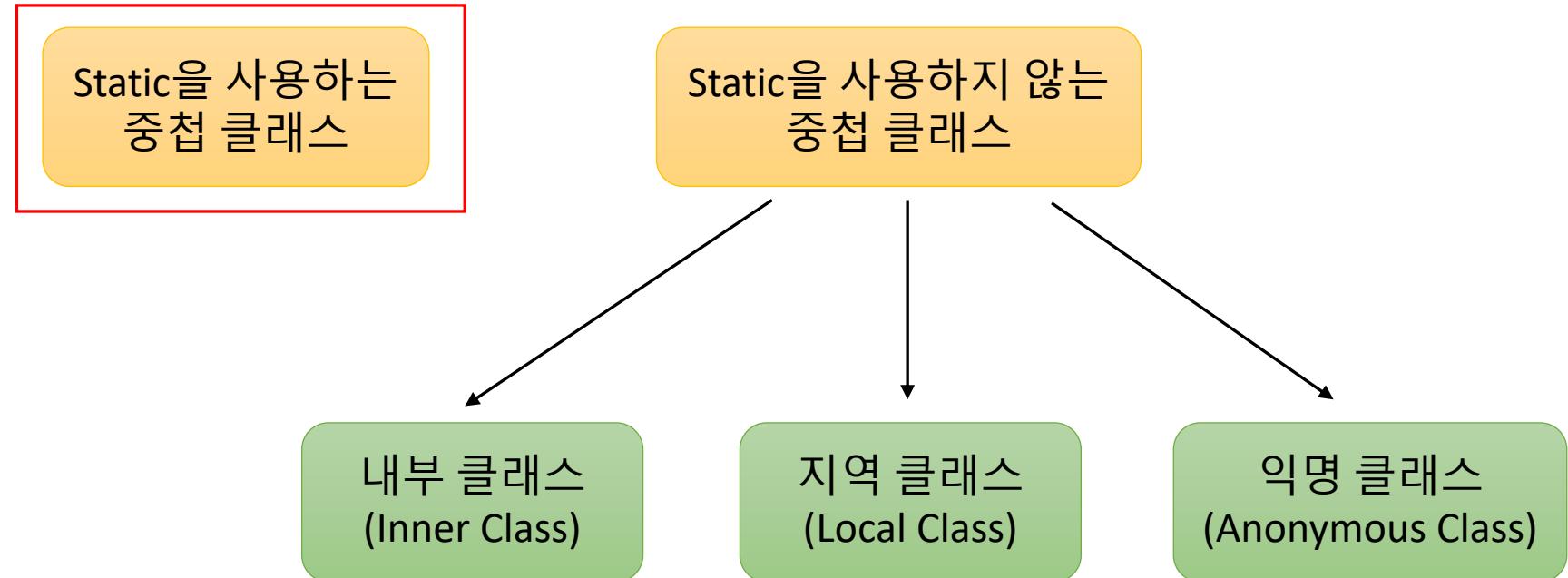
Static을 사용하는
중첩 클래스

Static을 사용하지 않는
중첩 클래스

1. 중첩 클래스의 종류

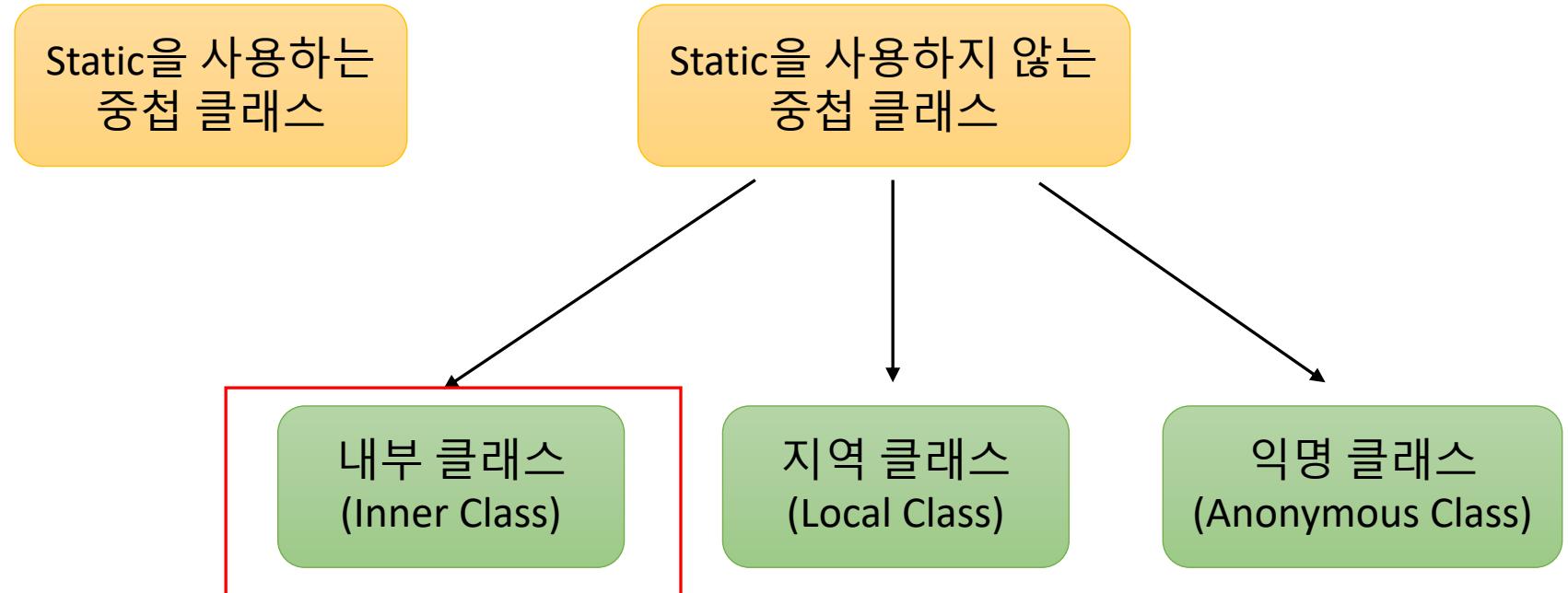


1. 중첩 클래스의 종류



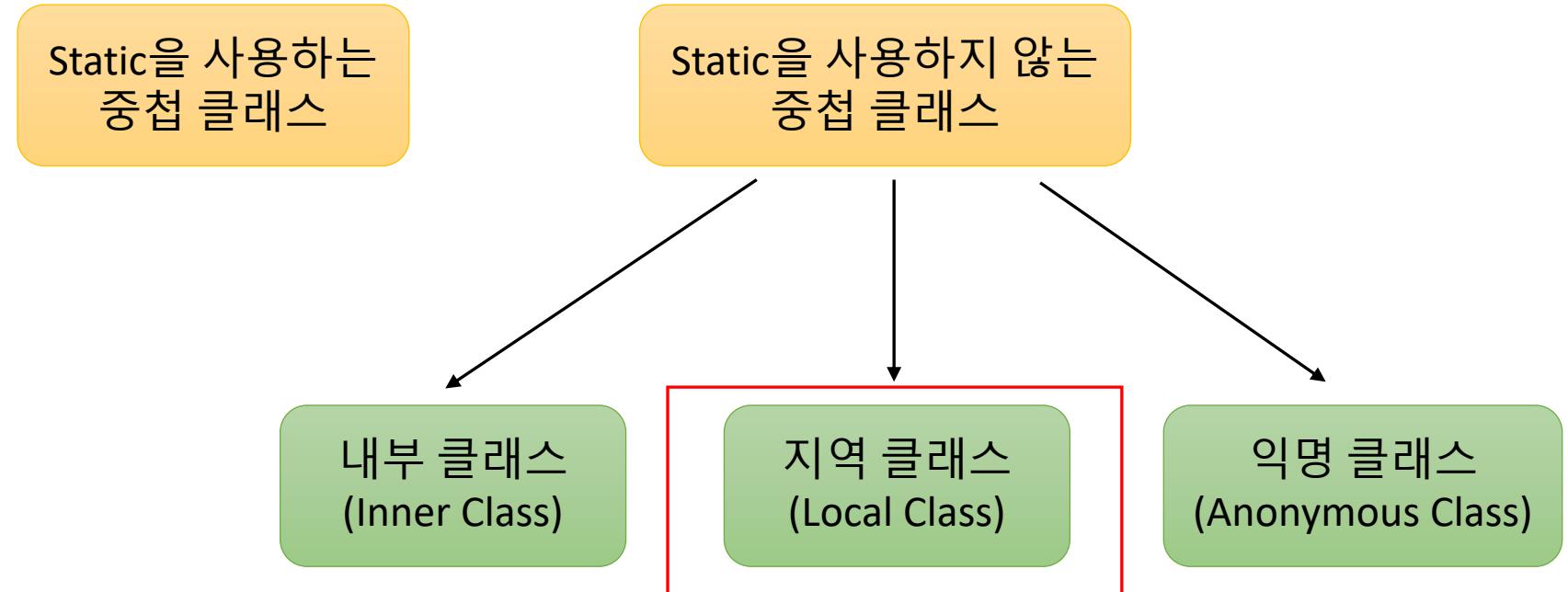
클래스 안에 static을 붙인 클래스! **밖의 클래스 직접 참조 불가**

1. 중첩 클래스의 종류



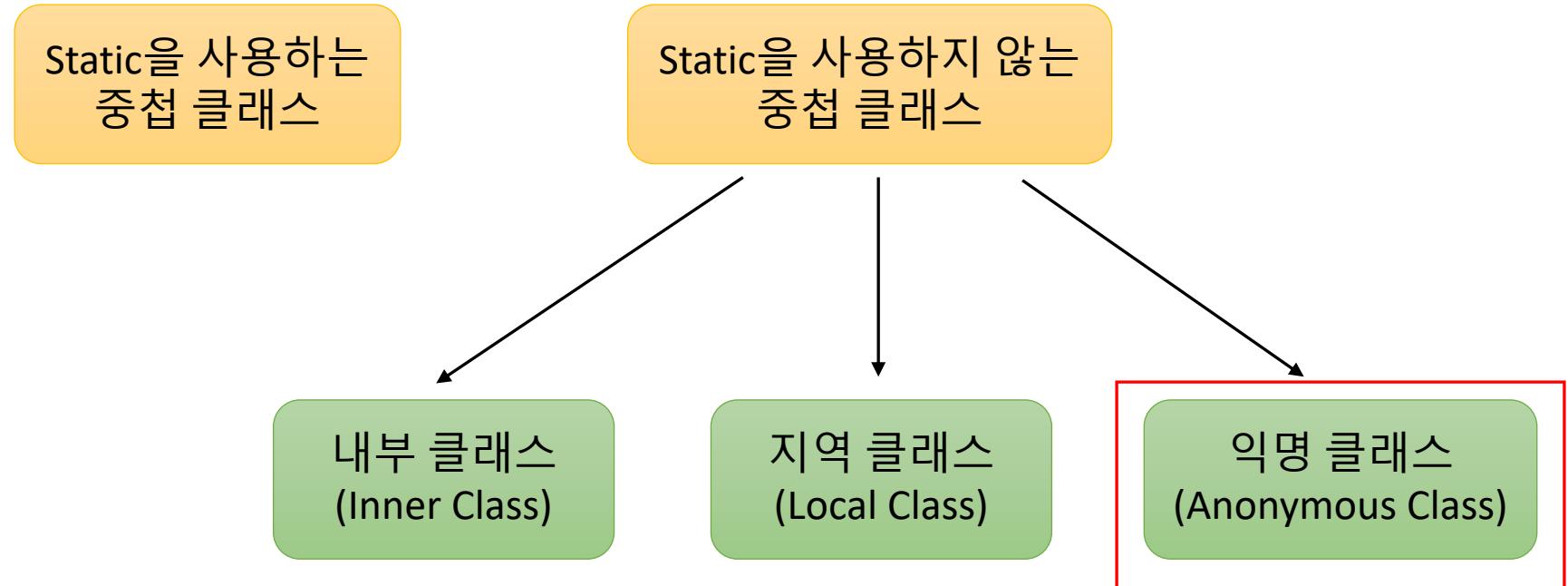
클래스 안의 클래스, 밖의 클래스 직접 참조 가능!

1. 중첩 클래스의 종류



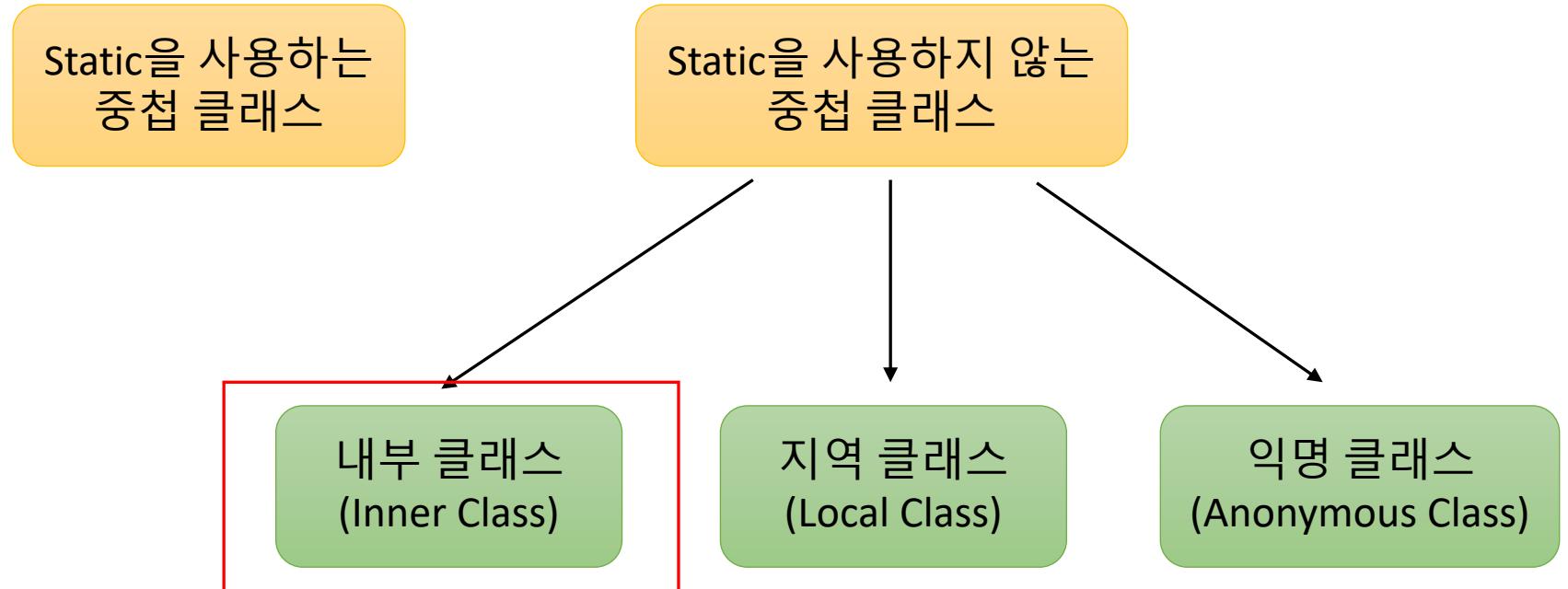
메소드 내부에 클래스를 정의 (실제 본적은 없습니다..)

1. 중첩 클래스의 종류



지난 시간에 다루었던, 일회성 클래스

1. 중첩 클래스의 종류

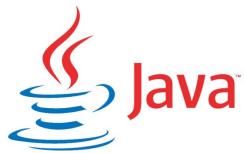


1. 중첩 클래스의 종류



```
public class JavaHouse {  
  
    private String address;  
    private LivingRoom livingRoom;  
  
    public JavaHouse(String address) {  
        this.address = address;  
        this.livingRoom = new LivingRoom(area: 10);  
    }  
  
    public LivingRoom getLivingRoom() {  
        return livingRoom;  
    }  
  
    public class LivingRoom {  
        private double area;  
  
        public LivingRoom(double area) {  
            this.area = area;  
        }  
  
        public String getAddress() {  
            return JavaHouse.this.address;  
        }  
    }  
}
```

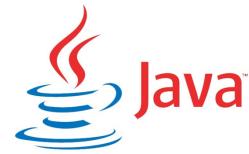
1. 중첩 클래스의 종류



```
public class JavaHouse {  
  
    private String address;  
    private LivingRoom livingRoom;  
  
    public JavaHouse(String address) {  
        this.address = address;  
        this.livingRoom = new LivingRoom(area: 10);  
    }  
  
    public LivingRoom getLivingRoom() {  
        return livingRoom;  
    }  
  
    public class LivingRoom {  
        private double area;  
  
        public LivingRoom(double area) {  
            this.area = area;  
        }  
  
        public String getAddress() {  
            return JavaHouse.this.address;  
        }  
    }  
}
```

바깥 클래스와
연결되어 있다.

1. 중첩 클래스의 종류



```
JavaHouse house = new JavaHouse( address: "제주도");
System.out.println(house.getLivingRoom().getAddress());
```

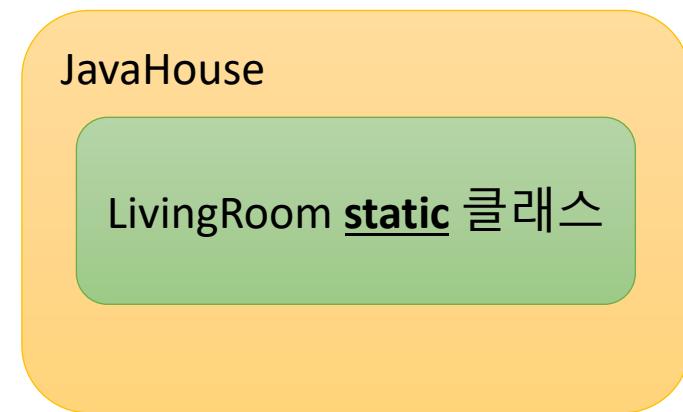
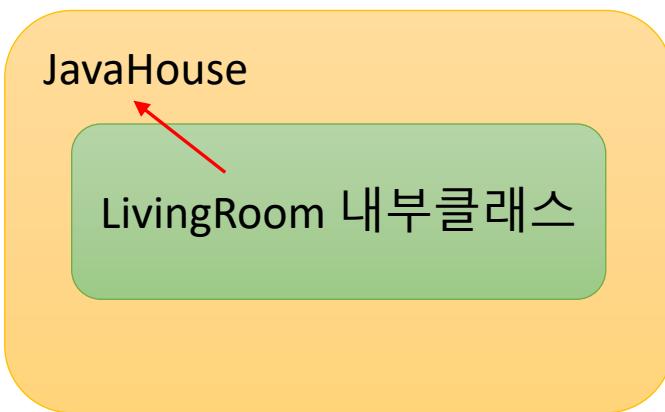
1. 중첩 클래스의 종류



```
public class JavaHouse {  
  
    private String address;  
    private LivingRoom livingRoom;  
  
    public JavaHouse(String address) {  
        this.address = address;  
        this.livingRoom = new LivingRoom(area: 10);  
    }  
  
    public LivingRoom getLivingRoom() {  
        return livingRoom;  
    }  
  
    public static class LivingRoom {  
        private double area;  
  
        public LivingRoom(double area) {  
            this.area = area;  
        }  
  
        public String getAddress() {  
            return JavaHouse.this.address;  
        }  
    }  
}
```

바깥 클래스를
바로 불러올 수 없다.

1. 중첩 클래스의 종류



1. 중첩 클래스의 종류

Effective Java 3rd Edition - Item24, Item86

1. 내부 클래스는 숨겨진 외부 클래스 정보를 가지고 있어, 참조를 해지하지 못하는 경우 메모리 누수가 생길 수 있고, 이를 디버깅 하기 어렵다.
2. 내부 클래스의 직렬화 형태가 명확하게 정의되지 않아 직렬화에 있어 제한이 있다.

1. 중첩 클래스의 종류

클래스 안에 클래스를 만들 때는 static 클래스를 사용하라

2. 코틀린의 중첩 클래스와 내부 클래스

Kotlin에서는 이러한 Guide를 충실히 따르고 있다.

2. 코틀린의 중첩 클래스와 내부 클래스

Java의 static 중첩 클래스 (권장되는 클래스 안의 클래스)

```
class House(  
    var address: String,  
    var livingRoom: LivingRoom = LivingRoom(area: 10.0)  
) {  
  
    class LivingRoom(  
        private var area: Double,  
    )  
  
}
```

2. 코틀린의 중첩 클래스와 내부 클래스

Java의 static 중첩 클래스 (권장되는 클래스 안의 클래스)

```
class House(  
    var address: String,  
    var livingRoom: LivingRoom = LivingRoom(area: 10.0)  
) {  
  
    class LivingRoom(  
        private var area: Double,  
    )  
}  
}
```

기본적으로 바깥 클래스에 대한 연결이 없는 중첩 클래스가 만들어진다.

2. 코틀린의 중첩 클래스와 내부 클래스

Java의 내부 클래스 (권장되지 않는 클래스 안의 클래스)

```
class House(  
    var address: String,  
) {  
  
    var livingRoom = this.LivingRoom(area: 10.0)  
  
    inner class LivingRoom(  
        private var area: Double,  
    ) {  
        val address: String  
            get() = this@House.address  
    }  
}
```

2. 코틀린의 중첩 클래스와 내부 클래스

Java의 내부 클래스 (권장되지 않는 클래스 안의 클래스)

```
class House(  
    var address: String,  
) {  
  
    var livingRoom = this.LivingRoom(area: 10.0)  
  
    inner class LivingRoom(  
        private var area: Double,  
    ) {  
        val address: String  
        get() = this@House.address  
    }  
}
```

2. 코틀린의 중첩 클래스와 내부 클래스

Java의 내부 클래스 (권장되지 않는 클래스 안의 클래스)

```
class House(  
    var address: String,  
) {  
  
    var livingRoom = this.LivingRoom(area: 10.0)  
  
    inner class LivingRoom(  
        private var area: Double,  
    ) {  
        val address: String  
        get() = this@House.address  
    }  
}
```

바깥클래스
참조를 위해
this@바깥클래스
를 사용한다.

2. 코틀린의 중첩 클래스와 내부 클래스

Kotlin에서는 이러한 Guide를 충실히 따르고 있다.

기본적으로 바깥 클래스 참조하지 않는다.

바깥 클래스를 참조하고 싶다면 inner 키워드를 추가한다.

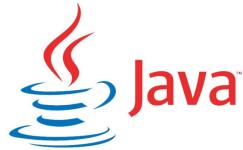
Lec 13. 코틀린에서 중첩 클래스를 다루는 방법

- 클래스 안에 클래스가 있는 경우 종류는 두 가지 였다.
 - (Java기준) static을 사용하는 클래스
 - (Java기준) static을 사용하지 않는 클래스
- 권장되는 클래스는 static을 사용하는 클래스이다.

Lec 13. 코틀린에서 중첩 클래스를 다루는 방법

- 코틀린에서는 이러한 가이드를 따르기 위해
 - 클래스 안에 기본 클래스를 사용하면 바깥 클래스에 대한 참조가 없고
 - 바깥 클래스를 참조하고 싶다면, `inner` 키워드를 붙여야 한다.
- 코틀린 `inner class`에서 바깥 클래스를 참조하려면 **this@바깥클래스**를 사용해야 한다.

Lec 13. 코틀린에서 중첩 클래스를 다루는 방법



클래스 안의 static 클래스	바깥 클래스 참조 없음 권장되는 유형
클래스 안의 클래스	바깥 클래스 참조 있음

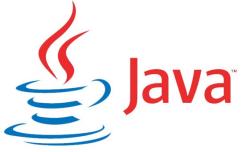


클래스 안의 클래스	바깥 클래스 참조 없음 권장되는 유형
클래스 안의 inner 클래스	바깥 클래스 참조 있음

Lec 14. 코틀린에서 다양한 클래스를 다루는 방법

1. Data Class
2. Enum Class
3. Sealed Class, Sealed Interface

1. Data Class



```
public class JavaPersonDto {  
  
    private final String name;  
    private final int age;  
  
    public JavaPersonDto(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

계층간의 데이터를 전달하기 위한 DTO(Data Transfer Object)

1. Data Class

계층간의 데이터를 전달하기 위한 DTO(Data Transfer Object)

데이터(필드)
생성자와 getter
equals, hashCode
toString

1. Data Class

```
2 import java.util.Objects;
3
4 public class JavaPersonDto {
5
6     private final String name;
7     private final int age;
8
9
10    public JavaPersonDto(String name, int age) {
11        this.name = name;
12        this.age = age;
13    }
14
15    public String getName() {
16        return name;
17    }
18
19    public int getAge() {
20        return age;
21    }
22
23    @Override
24    public boolean equals(Object o) {
25        if (this == o) return true;
26        if (o == null || getClass() != o.getClass()) return false;
27        JavaPersonDto that = (JavaPersonDto) o;
28        return age == that.age && Objects.equals(name, that.name);
29    }
30
31    @Override
32    public int hashCode() {
33        return Objects.hash(name, age);
34    }
35
36    @Override
37    public String toString() {
38        return "JavaPersonDto{" +
39                  "name='" + name + '\'' +
40                  ", age=" + age +
41                  '}';
42    }
43
44}
```

1. Data Class



IDE를 활용할 수도 있고, lombok을 활용할 수도 있지만

클래스가 장황해지거나,
클래스 생성 이후 추가적인 처리를 해줘야 하는 단점이 있다.

1. Data Class



```
data class PersonDto(  
    val name: String,  
    val age: Int,  
)
```

1. Data Class



```
data class PersonDto(  
    val name: String,  
    val age: Int,  
)
```

data 키워드를 붙여주면
equals, hashCode, toString을 자동으로 만들어줍니다!

1. Data Class



```
data class PersonDto(  
    val name: String,  
    val age: Int,  
)
```

참 간단하죠?!

1. Data Class



```
data class PersonDto(  
    val name: String,  
    val age: Int,  
)
```

여기에 named argument까지 활용하면
builder pattern을 쓰는 것 같은 효과도 있습니다!

1. Data Class



```
data class PersonDto(  
    val name: String,  
    val age: Int,  
)
```

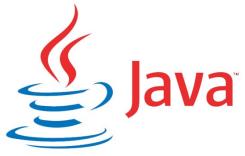
사실상 builder, equals, hashCode, toString이 모두 있는것!

1. Data Class



Java에서는 JDK16부터
Kotlin의 data class 같은 record class를 도입

2. Enum Class



```
public enum JavaCountry {  
  
    KOREA(code: "K0"),  
    AMERICA(code: "US");  
  
    private final String code;  
  
    JavaCountry(String code) { this.code = code; }  
  
    public String getCode() { return code; }  
}
```

추가적인 클래스를 상속받을 수 없다.
인터페이스는 구현할 수 있으며, 각 코드가 싱글톤이다.

2. Enum Class



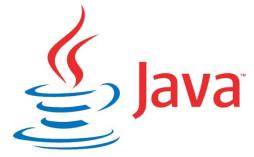
```
enum class Country(  
    val code: String  
) {  
  
    KOREA(code: "KO"),  
    AMERICA(code: "US"),  
    ;  
}
```

딱히 다른게 없다!!

2. Enum Class

when은 **Enum Class** 혹은 Sealed Class와 함께 사용할 경우, 더욱더 진가를 발휘한다.

2. Enum Class



```
private static void handleCountry(JavaCountry country) {  
    if (country == JavaCountry.KOREA) {  
        // 로직 처리  
    }  
  
    if (country == JavaCountry.AMERICA) {  
        // 로직 처리  
    }  
}
```

코드가 많아지게 된다면..?! else 로직 처리에 대한 애매함

2. Enum Class



```
private fun handleCountry(country: Country) {  
    when (country) {  
        Country.KOREA -> TODO()  
        Country.AMERICA -> TODO()  
    }  
}
```

조금 더 읽기 쉬운 코드

2. Enum Class



```
private fun handleCountry(country: Country) {  
    when (country) {  
        Country.KOREA -> TODO()  
        Country.UNITED STATES -> TODO()  
    }  
}
```

컴파일러가 country의 모든 타입을 알고 있어
다른 타입에 대한 로직(else)을 작성하지 않아도 된다.

2. Enum Class



```
private fun handleCountry(country: Country) {  
    when (country) {  
        Country.KOREA -> TODO()  
        Country.UNITED STATES -> TODO()  
    }  
}
```

Enum에 변화가 있으면 알 수 있다.

3. Sealed Class, Sealed Interface

sealed

미국·영국 [sí:ld]  영국식 

형용사

1 봉인을 한

2 <도로가> 포장된

3 해결[처리]된

영어사전 다른 뜻 5

3. Sealed Class, Sealed Interface

상속이 가능하도록 추상클래스를 만들까 하는데...
외부에서는 이 클래스를 상속받지 않았으면 좋겠어!!

하위 클래스를 봉인하자!!!

3. Sealed Class, Sealed Interface

컴파일 타임 때 하위 클래스의 타입을 모두 기억한다.
즉, 런타임때 클래스 타입이 추가될 수 없다.

하위 클래스는 같은 패키지에 있어야 한다.

Enum과 다른 점

- 클래스를 상속받을 수 있다.
- 하위 클래스는 멀티 인스턴스가 가능하다.

3. Sealed Class, Sealed Interface



```
sealed class HyundaiCar(  
    val name: String,  
    val price: Long  
)  
  
class Avante : HyundaiCar(name: "아반떼", price: 1_000L)  
  
class Sonata : HyundaiCar(name: "소나타", price: 2_000L)  
  
class Grandeur : HyundaiCar(name: "그랜저", price: 3_000L)
```

3. Sealed Class, Sealed Interface

컴파일 타임 때 하위 클래스의 타입을 모두 기억한다.
즉, 런타임때 클래스 타입이 추가될 수 없다.

3. Sealed Class, Sealed Interface



```
fun main() {
    handleCar(Avante())
}

private fun handleCar(car: HyundaiCar) {
    when (car) {
        is Avante -> TODO()
        is Grandeur -> TODO()
        is Sonata -> TODO()
    }
}
```

3. Sealed Class, Sealed Interface

추상화가 필요한 Entity or DTO에 sealed class를 활용

3. Sealed Class, Sealed Interface

추가로, JDK17에서도 Sealed Class가 추가되었습니다.

Lec 14. 코틀린에서 다양한 클래스를 다루는 방법

- Kotlin의 **Data class**를 사용하면 equals, hashCode, toString을 자동으로 만들어준다.

Lec 14. 코틀린에서 다양한 클래스를 다루는 방법

- Kotlin의 Data class를 사용하면 equals, hashCode, toString을 자동으로 만들어준다.
- Kotlin의 **Enum Class**는 Java의 Enum Class와 동일하지만, when과 함께 사용함으로써 큰 장점을 갖게 된다.

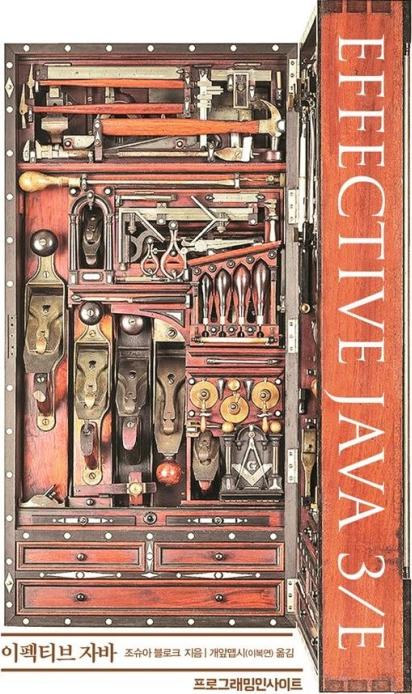
Lec 14. 코틀린에서 다양한 클래스를 다루는 방법

- Kotlin의 Data class를 사용하면 equals, hashCode, toString을 자동으로 만들어준다.
- Kotlin의 Enum Class는 Java의 Enum Class와 동일하지만, when과 함께 사용함으로써 큰 장점을 갖게 된다.
- Enum Class보다 유연하지만, 하위 클래스를 제한하는 **Sealed Class** 역시 when과 함께 주로 사용된다.

Lec 15. 코틀린에서 배열과 컬렉션을 다루는 방법

1. 배열
2. 코틀린에서의 Collection - List, Set, Map
3. 컬렉션의 null 가능성, Java와 함께 사용하기

1. 배열



사실 배열은 잘 사용하지 않습니다!

1. 배열

사실 배열은 잘 사용하지 않습니다!

하지만 문법을 간략히 소개해드릴게요~

1. 배열



```
int[] array = {100, 200};  
  
for (int i = 0; i < array.length; i++) {  
    System.out.printf("%s %s", i, array[i]);  
}
```

1. 배열



```
val array = arrayOf(100, 200)

for (i in array.indices) {
    println("${i} ${array[i]}")
}
```

array.indices는 0부터 마지막 index까지의 Range이다.

1. 배열



```
val array = arrayOf(100, 200)

for (i in array.indices) {
    println("${i} ${array[i]}")
}
```

1. 배열



```
val array = arrayOf(100, 200)

for ((idx, value) in array.withIndex()) {
    println("${idx} ${value}")
}
```

withIndex() 를 사용하면, 인덱스와 값을 한 번에 가져올 수 있다.

1. 배열



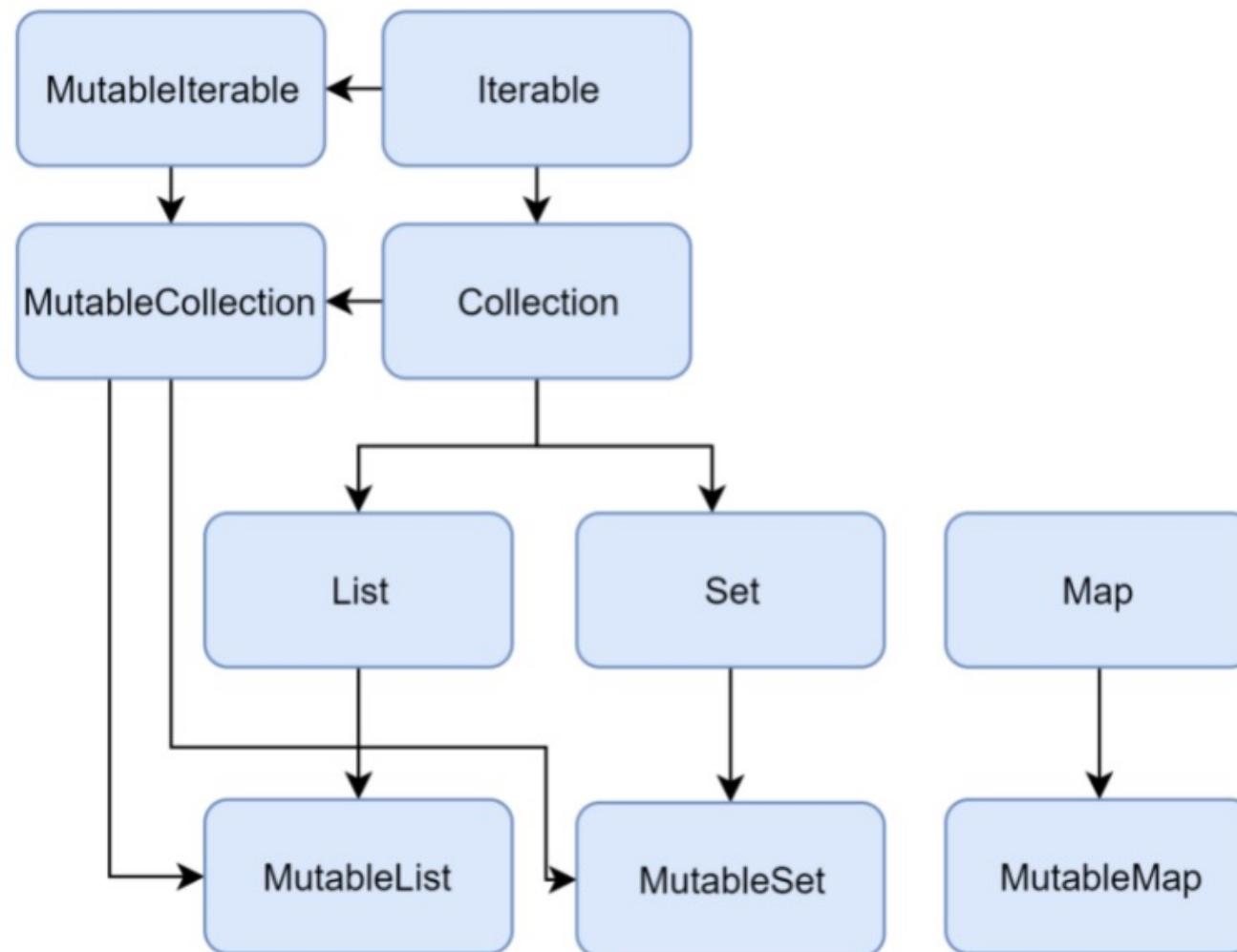
```
val array = arrayOf(100, 200)
array.plus(element: 300)
for ((idx, value) in array.withIndex()) {
    println("${idx} ${value}")
}
```

값을 쉽게 넣을 수도 있다.

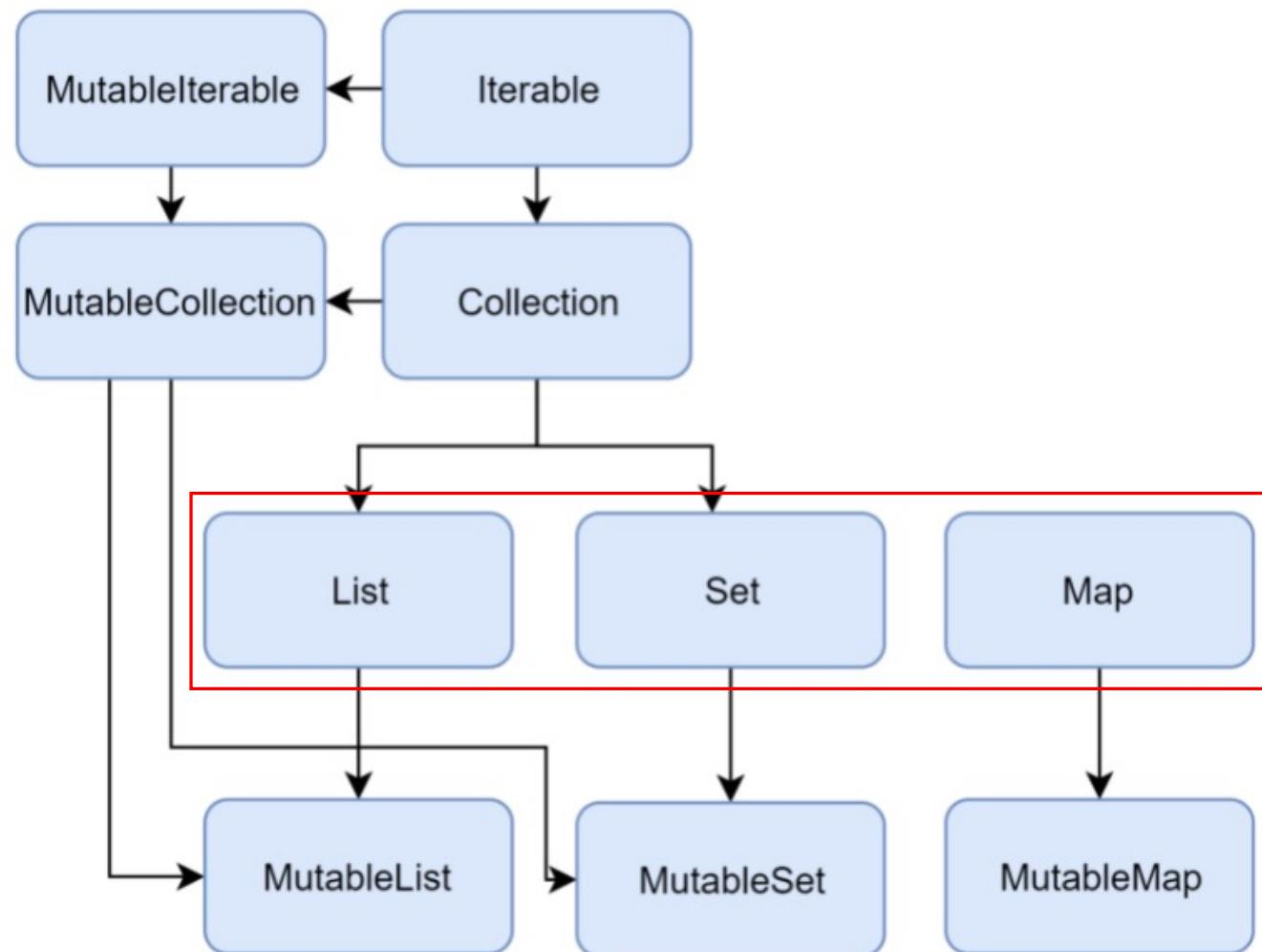
2. 코틀린에서의 Collection

컬렉션을 만들어줄 때 **불변인지, 가변인지**를 설정해야 한다.

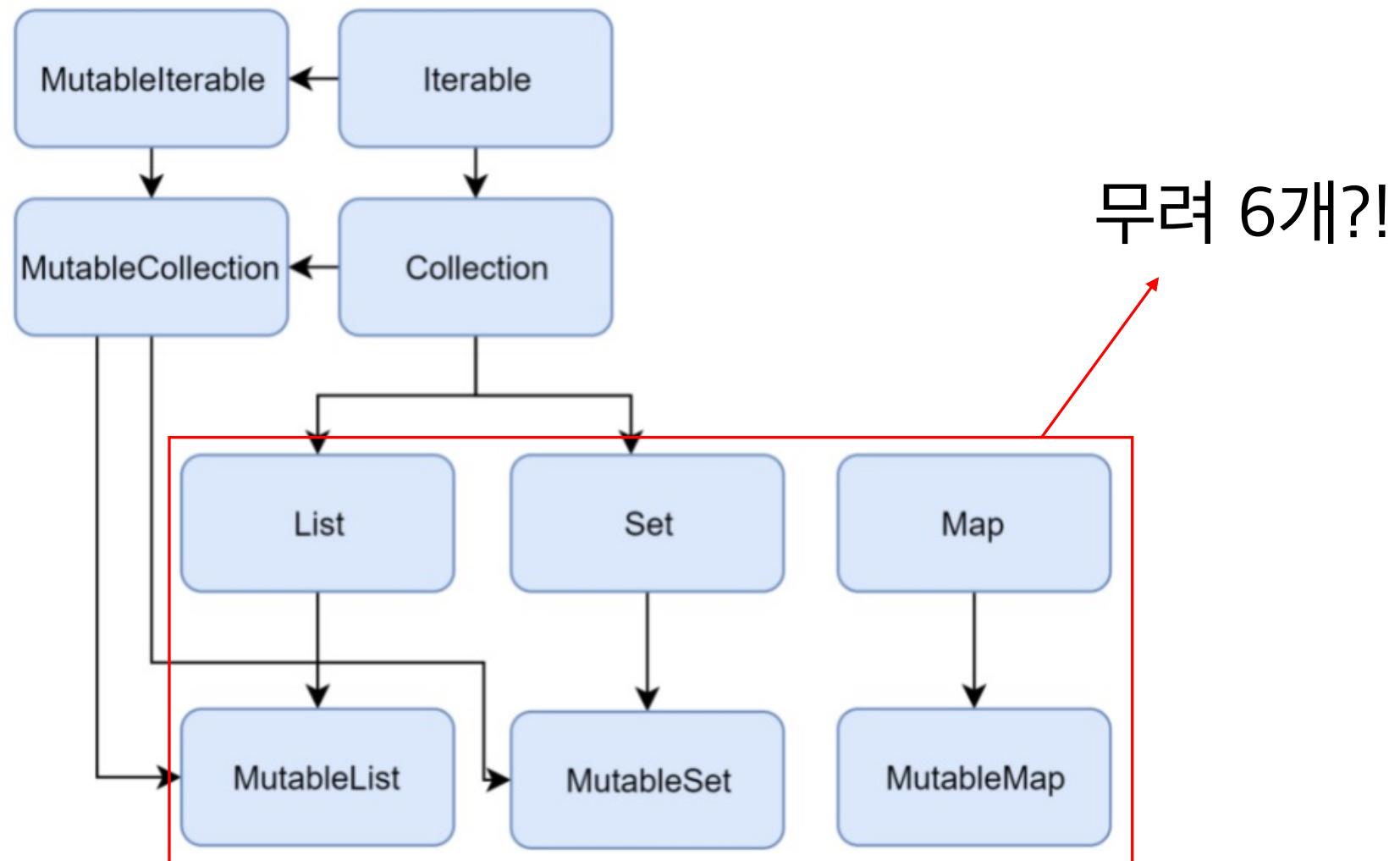
2. 코틀린에서의 Collection



2. 코틀린에서의 Collection



2. 코틀린에서의 Collection



2. 코틀린에서의 Collection

가변 (Mutable) 컬렉션 : 컬렉션에 element를 추가, 삭제할 수 있다.

불변 컬렉션 : 컬렉션에 element를 추가, 삭제할 수 없다.

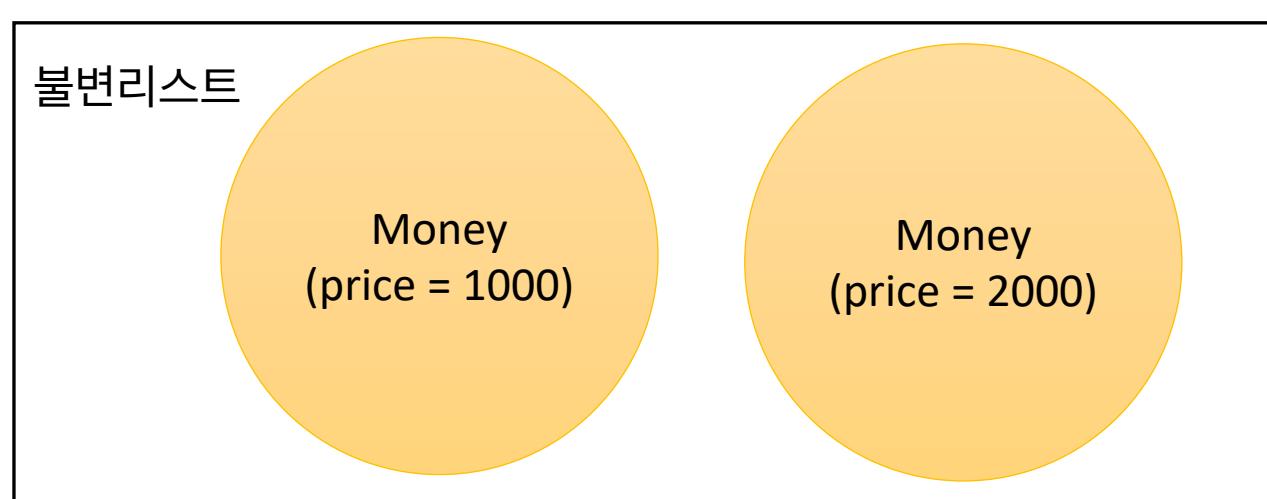
2. 코틀린에서의 Collection

Collection을 만들자 마자
Collections.unmodifiableList() 등을 붙여준다!

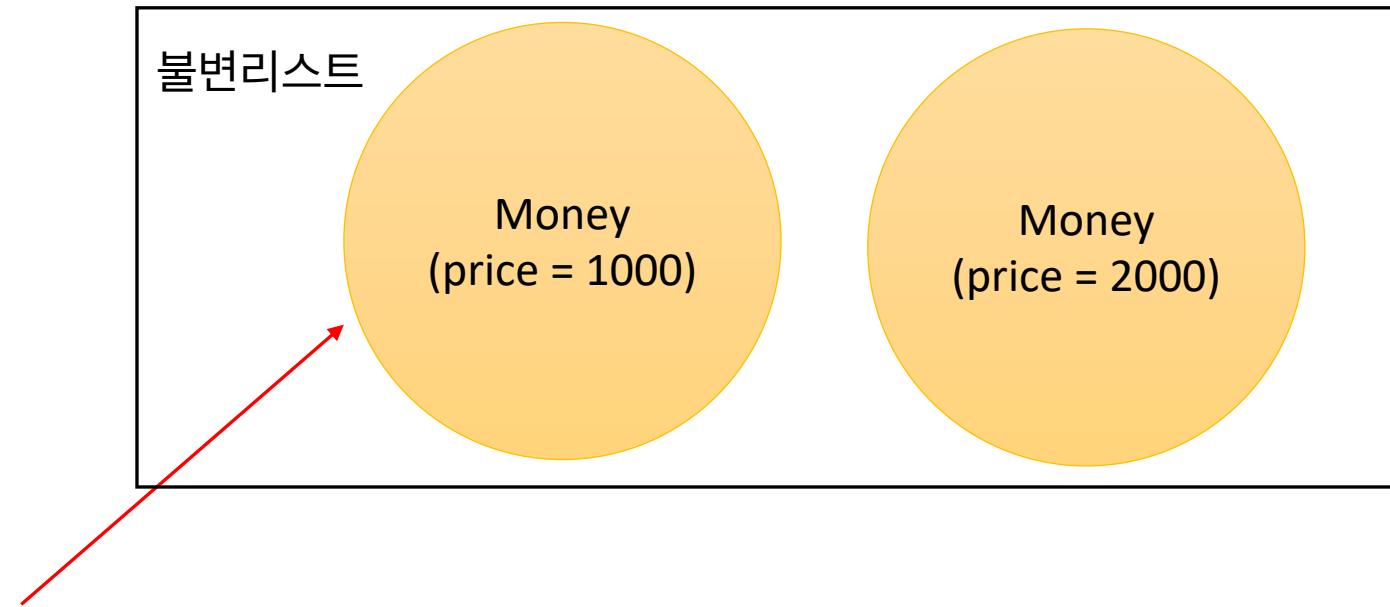
2. 코틀린에서의 Collection

불변 컬렉션이라 하더라도
Reference Type인 Element의 필드는 바꿀 수 있다.

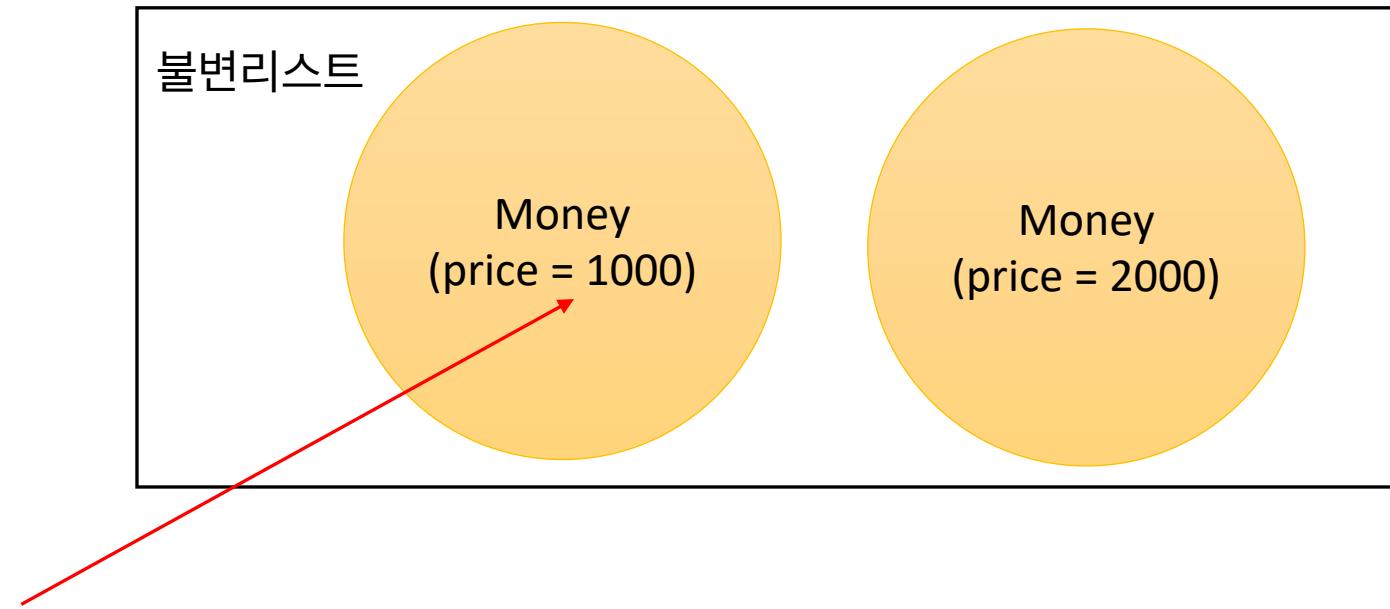
2. 코틀린에서의 Collection



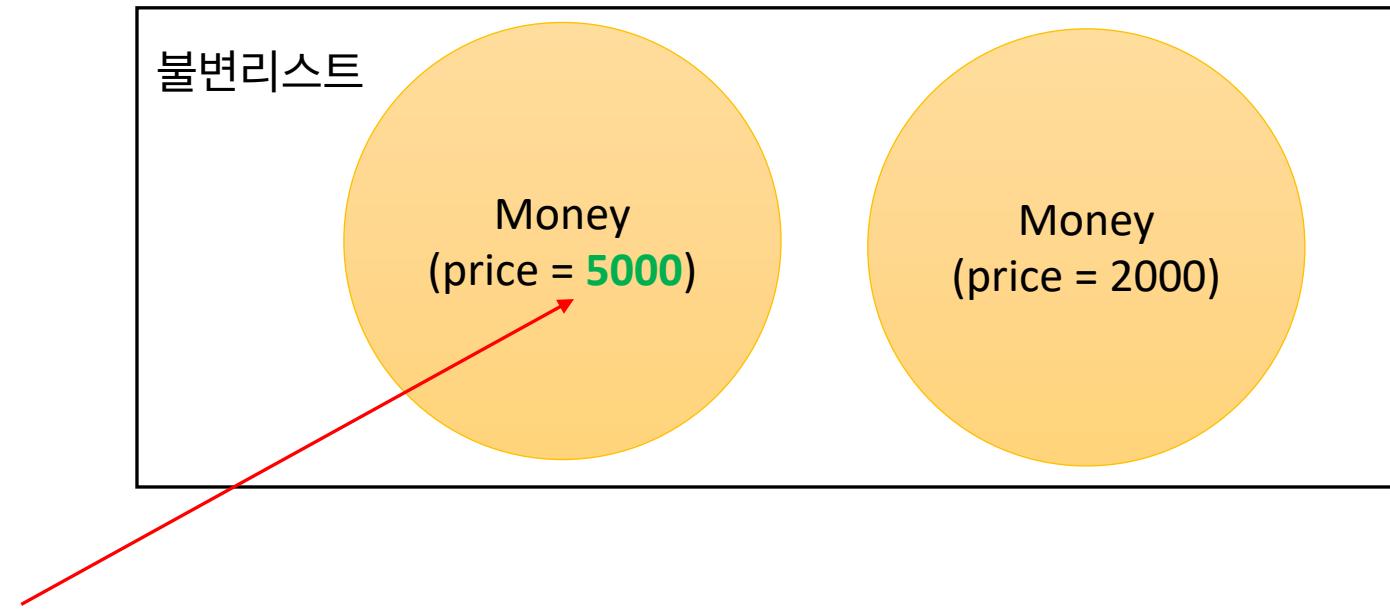
2. 코틀린에서의 Collection



2. 코틀린에서의 Collection



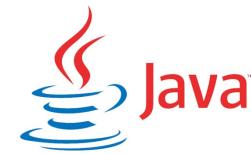
2. 코틀린에서의 Collection



2. 코틀린에서의 Collection

Kotlin은 불변/가변을 지정해 주어야 한다는 사실을 꼭 기억해주시고
본격적으로 List부터 설명드리겠습니다!

2. 코틀린에서의 Collection - List



```
final List<Integer> numbers = Arrays.asList(100, 200);
```

2. 코틀린에서의 Collection - List



```
val numbers = listOf(100, 200)
```

listOf 를 통해 ‘불변 리스트’를 만든다.

2. 코틀린에서의 Collection - List



```
val emptyList = emptyList<Int>()
```

emptyList<타입>()

2. 코틀린에서의 Collection - List



```
fun main() {  
    useNumbers(emptyList())  
}  
  
private fun useNumbers(numbers: List<Int>) {  
}
```



타입을 추론할 수 있다면 생략 가능하다

2. 코틀린에서의 Collection - List



```
// 하나를 가져오기
System.out.println(numbers.get(0));

// For Each
for (int number : numbers) {
    System.out.println(number);
}

// 전통적인 For문
for (int i = 0; i < numbers.size(); i++) {
    System.out.printf("%s %s", i, numbers.get(i));
}
```

2. 코틀린에서의 Collection - List



```
// 하나를 가져오기
println(numbers[0])

// For Each
for (number in numbers) {
    println(number)
}

// 전통적인 For문 느낌
for ((index, number) in numbers.withIndex()) {
    println("$index $number")
}
```

2. 코틀린에서의 Collection - List



```
// 하나를 가져오기
println(numbers[0])

// For Each
for (number in numbers) {
    println(number)
}

// 전통적인 For문 느낌
for ((index, number) in numbers.withIndex()) {
    println("$index $number")
}
```

2. 코틀린에서의 Collection - List



가변(Mutable) 리스트를 만들고 싶다면?!!

```
val numbers = mutableListOf(100, 200)  
numbers.add(300)
```

기본 구현체는 `ArrayList`이고
기타 사용법은 Java와 동일합니다!

2. 코틀린에서의 Collection - List

간단한 TIP

우선 불변 리스트를 만들고, 꼭 필요한 경우 가변 리스트로 바꾸자!

2. 코틀린에서의 Collection - Set

집합은 List와 다르게 순서가 없고,
같은 element는 하나만 존재할 수 있다.

자료구조적 의미만 제외하면 모든 기능이 List와 비슷합니다!

2. 코틀린에서의 Collection - Set



```
val numbers = setOf(100, 200)

// For Each
for (number in numbers) {
    println(number)
}

// 전통적인 For문
for ((index, number) in numbers.withIndex()) {
    println("$index $number")
}
```

2. 코틀린에서의 Collection - Set



가변(Mutable) 집합을 만들고 싶다면?!!

```
val numbers = mutableSetOf(100, 200)
```

기본 구현체는 LinkedHashSet 입니다.

2. 코틀린에서의 Collection - Map



```
// JDK 8까지
Map<Integer, String> map = new HashMap<>();
map.put(1, "MONDAY");
map.put(2, "TUESDAY");

// JDK 9부터
Map.of(k1: 1, v1: "MONDAY", k2: 2, v2: "TUESDAY");
```

2. 코틀린에서의 Collection - Map



Kotlin도 동일하게 MutableMap을 만들어 넣을 수도 있고,
정적 팩토리 메소드를 바로 활용할 수도 있다.

2. 코틀린에서의 Collection - Map



```
val map = mutableMapOf<Int, String>()
map[1] = "MONDAY"
map[2] = "TUESDAY"

mapOf(1 to "MONDAY", 2 to "TUESDAY")
```

2. 코틀린에서의 Collection - Map



```
val map = mutableMapOf<Int, String>()
map[1] = "MONDAY"
map[2] = "TUESDAY"

mapOf(1 to "MONDAY", 2 to "TUESDAY")
```

타입을 추론할 수 없어, 타입을 지정해주었다.

2. 코틀린에서의 Collection - Map



```
val map = mutableMapOf<Int, String>()
map[1] = "MONDAY"
map[2] = "TUESDAY"

mapOf(1 to "MONDAY", 2 to "TUESDAY")
```

가변 Map 이기 때문에 (key, value)를 넣을 수 있다.
Java처럼 put을 쓸 수도 있고, **map[key] = value** 을 쓸 수도 있다.

2. 코틀린에서의 Collection - Map



```
val map = mutableMapOf<Int, String>()
map[1] = "MONDAY"
map[2] = "TUESDAY"

mapOf(1 to "MONDAY", 2 to "TUESDAY")
```

mapOf(key to value) 를 사용해 불변 map을 만들 수 있다.

2. 콜렉션에서의 Collection - Map



```
for (int key : map.keySet()) {  
    System.out.println(key);  
    System.out.println(map.get(key));  
}  
  
for (Map.Entry<Integer, String> entry : map.entrySet()) {  
    System.out.println(entry.getKey());  
    System.out.println(entry.getValue());  
}
```

2. 코틀린에서의 Collection - Map



```
for (key in map.keys) {  
    println(key)  
    println(map[key])  
}  
  
for ((key, value) in map.entries) {  
    println(key)  
    println(value)  
}
```

2. 코틀린에서의 Collection - Map



```
for (key in map.keys) {  
    println(key)  
    println(map[key])  
}  
  
for ((key, value) in map.entries) {  
    println(key)  
    println(value)  
}
```

2. 코틀린에서의 Collection - Map



```
for (key in map.keys) {  
    println(key)  
    println(map[key])  
}  
  
for ((key, value) in map.entries) {  
    println(key)  
    println(value)  
}
```

2. 코틀린에서의 Collection - Map



```
for (key in map.keys) {  
    println(key)  
    println(map[key])  
}  
  
for ((key, value) in map.entries) {  
    println(key)  
    println(value)  
}
```

3. 컬렉션의 null 가능성, Java와 함께 사용하기

`List<Int?>` : 리스트에 null이 들어갈 수 있지만, 리스트는 절대 null이 아님

`List<Int>?` : 리스트에는 null이 들어갈 수 없지만, 리스트는 null일 수 있음

`List<Int?>?` : 리스트에 null이 들어갈 수도 있고, 리스트가 null일 수도 있음

3. 컬렉션의 null 가능성, Java와 함께 사용하기

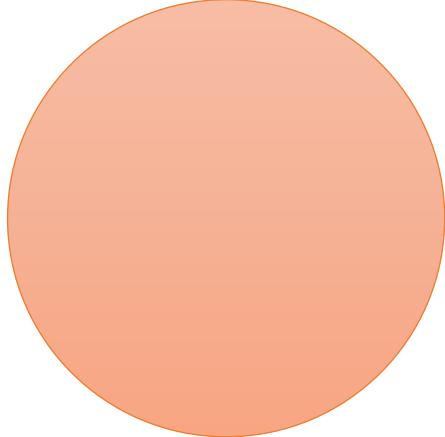
? 위치에 따라 null가능성 의미가 달라지므로
차이를 잘 이해하셔야 합니다!

3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않는다.

3. 컬렉션의 null 가능성, Java와 함께 사용하기

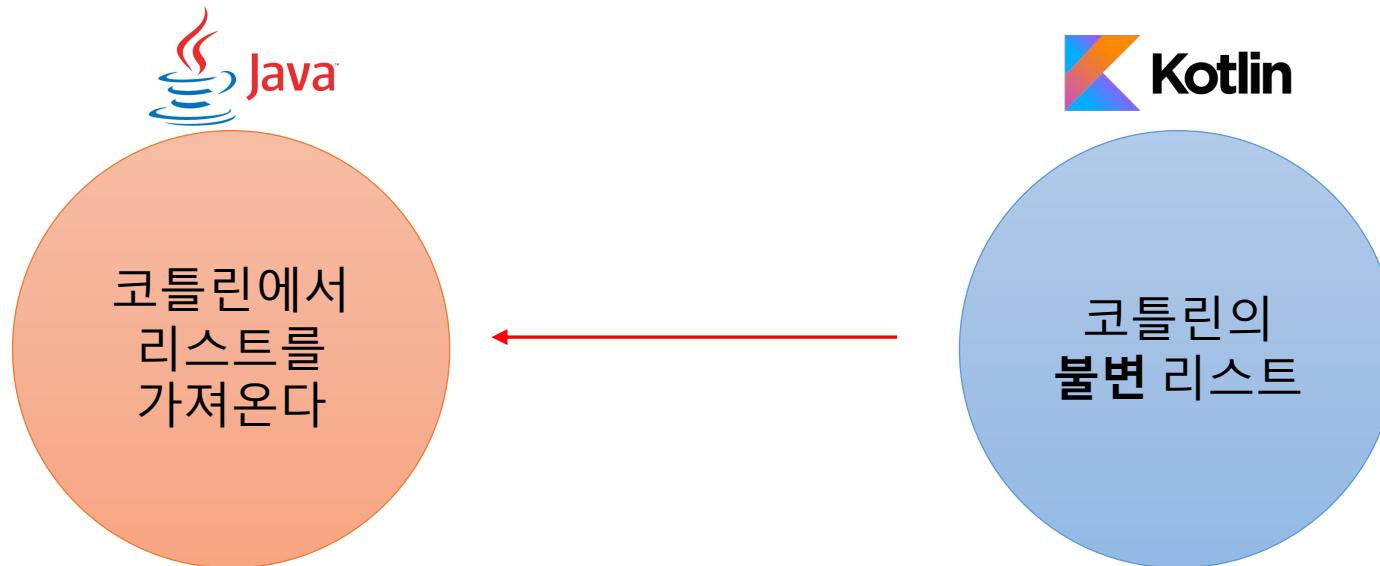
Java는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않는다.



코틀린의
불변 리스트

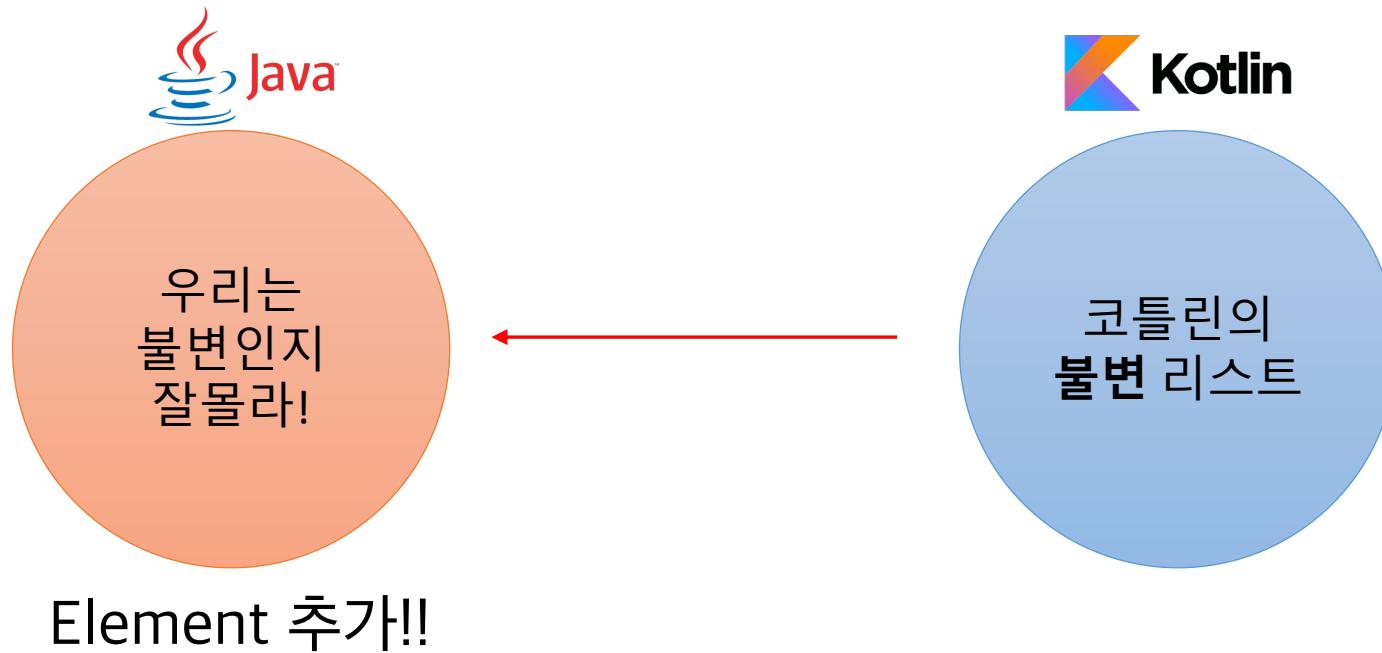
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않는다.



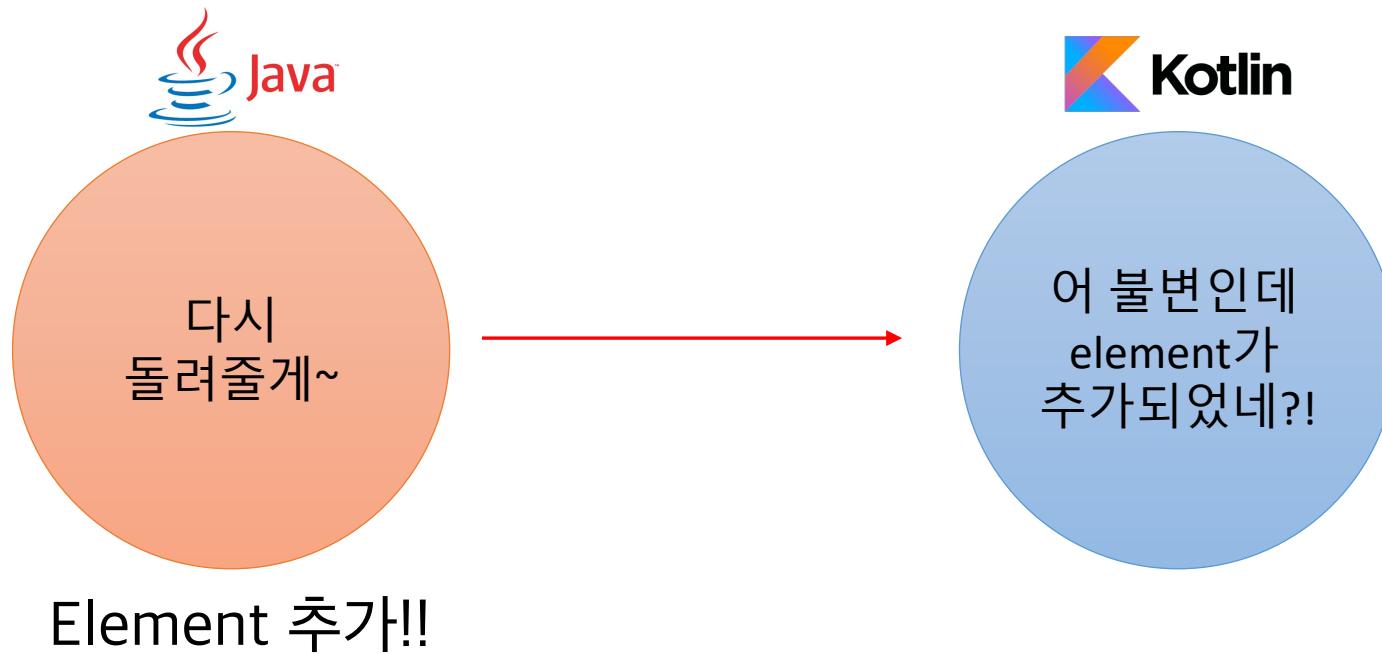
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않는다.



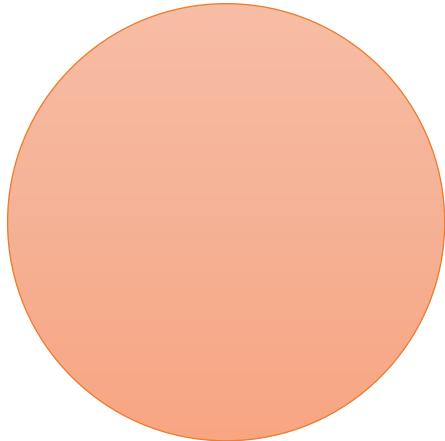
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않는다.



3. 컬렉션의 null 가능성, Java와 함께 사용하기

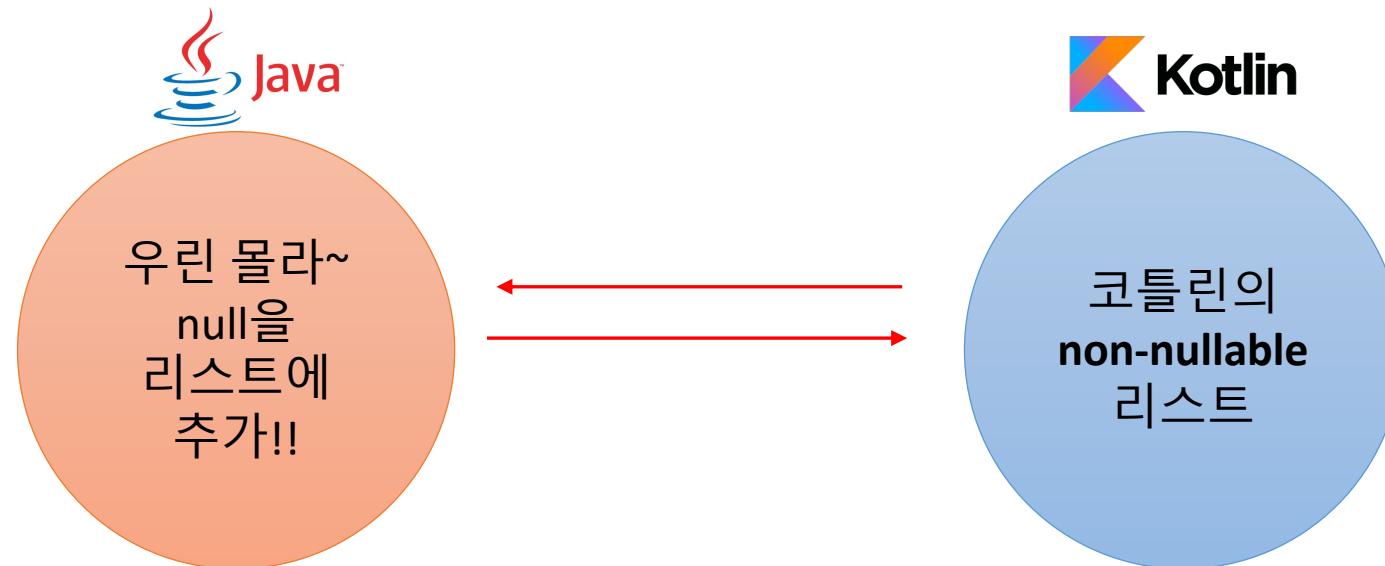
Java는 nullable 타입과 non-nullable 타입을 구분하지 않는다.



코틀린의
non-nullable
리스트

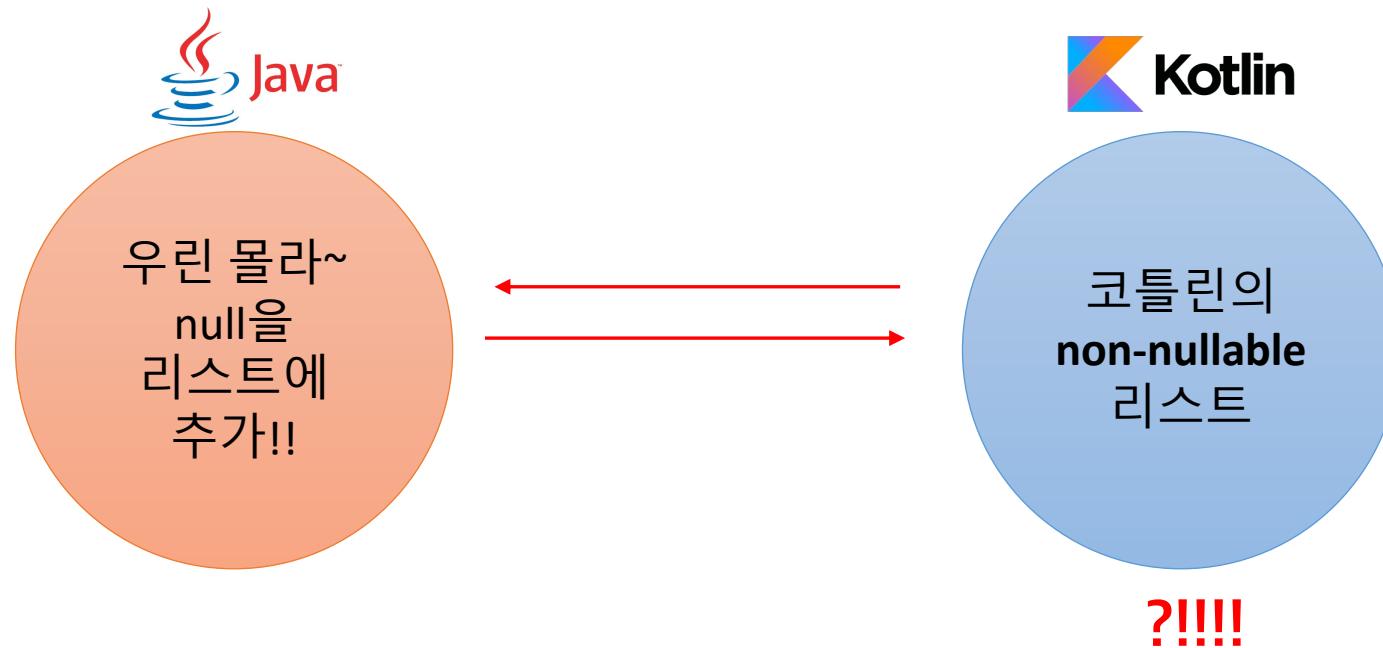
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 nullable 타입과 non-nullable 타입을 구분하지 않는다.



3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java는 nullable 타입과 non-nullable 타입을 구분하지 않는다.



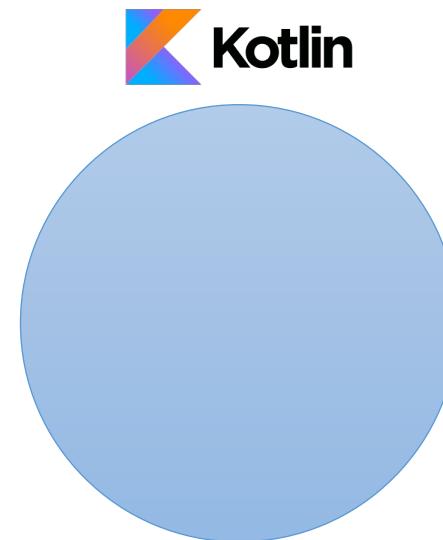
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Kotlin 쪽의 컬렉션이 Java에서 호출되면
컬렉션 내용이 변할 수 있음을 감안해야 한다.

코틀린 쪽에서 **Collections.unmodifiableXXX()**를 활용하면
변경 자체를 막을 수는 있다!

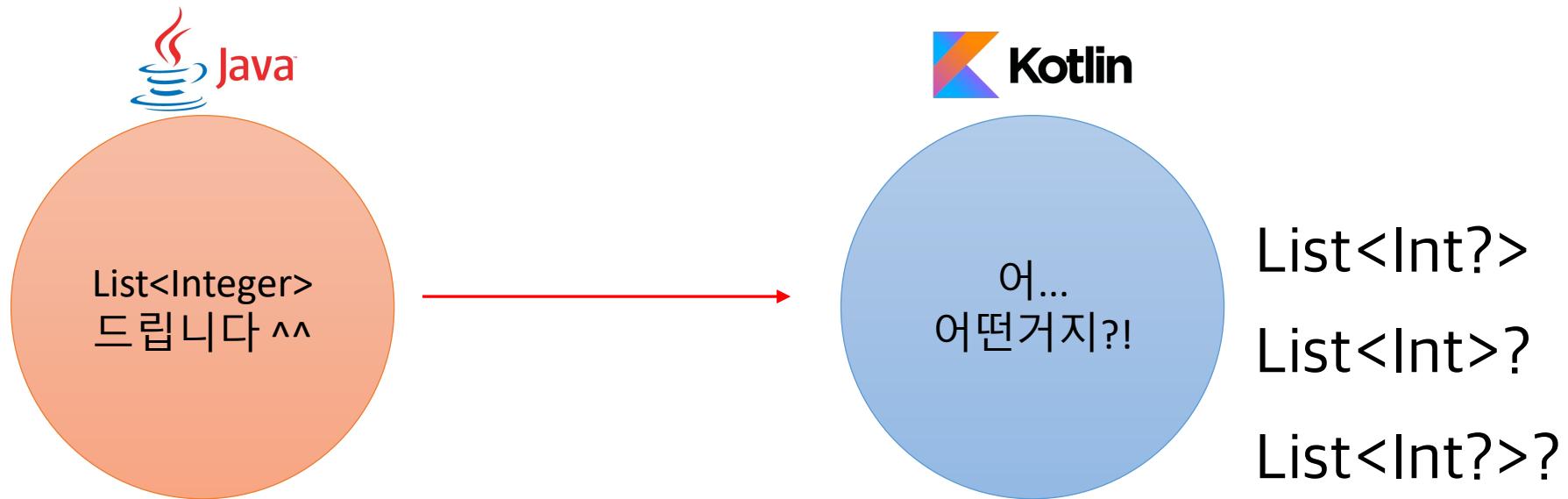
3. 컬렉션의 null 가능성, Java와 함께 사용하기

Kotlin에서 Java 컬렉션을 가져다 사용할때 **플랫폼 타입**을 신경써야 한다.



3. 컬렉션의 null 가능성, Java와 함께 사용하기

Kotlin에서 Java 컬렉션을 가져다 사용할때 **플랫폼 타입**을 신경써야 한다.



3. 컬렉션의 null 가능성, Java와 함께 사용하기

Java 코드를 보며, 맥락을 확인하고
Java 코드를 가져오는 지점을 wrapping한다

Lec 15. 코틀린에서 배열과 컬렉션을 다루는 방법

- 배열의 사용법이 약간 다르다!

```
val array = arrayOf(100, 200)
```

```
for (i in array.indices) {  
    println("${i} ${array[i]}")  
}
```

```
for ((idx, value) in array.withIndex()) {  
    println("${idx} ${value}")  
}
```

Lec 15. 코틀린에서 배열과 컬렉션을 다루는 방법

- 코틀린에서는 컬렉션을 만들 때도 불변/가변을 지정해야 한다.
- List, Set, Map 에 대한 사용법이 변경, 확장되었다.
- Java와 Kotlin 코드를 섞어 컬렉션을 사용할 때에는 주의해야 한다.
 - Java에서 Kotlin 컬렉션을 가져갈 때는 **불변 컬렉션을 수정할 수도 있고, non-nullable 컬렉션에 null을 넣을 수도 있다.**

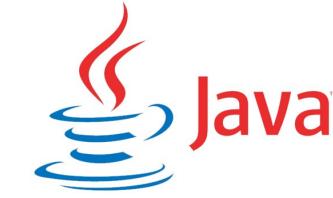
Lec 15. 코틀린에서 배열과 컬렉션을 다루는 방법

- 코틀린에서는 컬렉션을 만들 때도 불변/가변을 지정해야 한다.
- List, Set, Map 에 대한 사용법이 변경, 확장되었다.
- Java와 Kotlin 코드를 섞어 컬렉션을 사용할 때에는 주의해야 한다.
 - Java에서 Kotlin 컬렉션을 가져갈 때는 불변 컬렉션을 수정할 수도 있고, non-nullable 컬렉션에 null을 넣을 수도 있다.
 - Kotlin에서 Java 컬렉션을 가져갈 때는 **플랫폼타입**을 주의해야 한다.

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

1. 확장함수
2. infix 함수
3. inline 함수
4. 지역함수

1. 확장함수



우리는 Java와 **100% 호환하는 것을 목표로 하고 있어요!**

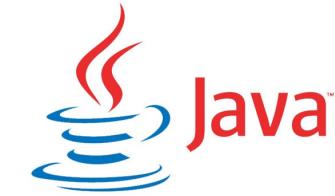
1. 확장함수



기존 Java 코드 위에 **자연스럽게** 코틀린 코드를 추가할 수는 없을까?!

Java로 만들어진 라이브러리를 유지보수, 확장할 때
Kotlin 코드를 덧붙이고 싶어!!!

1. 확장함수



어떤 클래스안에 있는 메소드처럼 호출할 수 있지만,
함수는 밖에 만들 수 있게 하자!!!

1. 확장함수



```
fun String.lastChar(): Char {  
    return this[this.length - 1]  
}
```

1. 확장함수



```
fun String.lastChar(): Char {  
    return this[this.length - 1]  
}
```

이건 함수야~

1. 확장함수



```
fun String.lastChar(): Char {  
    return this[this.length - 1]  
}
```

String 클래스를 확장하는구나!

1. 확장함수



```
fun String.lastChar(): Char {  
    return this[this.length - 1]  
}
```

함수 안에서는 **this**를 통해 인스턴스에 접근 가능하다!

1. 확장함수



```
fun 확장하려는클래스.함수이름(파라미터): 리턴타입 {  
    // this를 이용해 실제 클래스 안의 값에 접근  
}
```

1. 확장함수



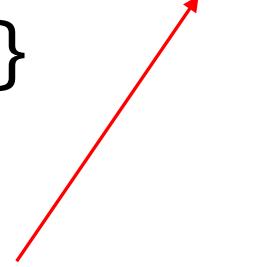
```
fun 확장하려는클래스.함수이름(파라미터): 리턴타입 {  
    // this를 이용해 실제 클래스 안의 값에 접근  
}
```

수신객체

this

}

}

수신객체

1. 확장함수



```
fun 확장하려는클래스.함수이름(파라미터): 리턴타입 {  
    // this를 이용해 실제 클래스 안의 값에 접근  
}
```

수신객체 타입

1. 확장함수



```
val str: String = "ABC"  
str.lastChar()
```

원래 String에 있는 멤버함수 처럼 사용할 수 있다.

1. 확장함수

엇 확장함수가 public이고,
확장함수에서 수신객체클래스의 private 함수를 가져오면
캡슐화가 깨지는거 아닌가??!

1. 확장함수

확장함수는 클래스에 있는
private 또는 **protected** 멤버를
가져올 수 없다!!

1. 확장함수

멤버함수와 확장함수의 시그니처가 같다면?!

1. 확장함수



```
public class Person {  
  
    private final String firstName;  
    private int age;  
  
    public Person(String firstName, int age) {  
        this.firstName = firstName;  
        this.age = age;  
    }  
  
    public int nextYearAge() {  
        System.out.println("멤버 함수");  
        return this.age + 1;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```



```
fun Person.nextYearAge(): Int {  
    println("확장 함수")  
    return this.age + 1  
}  
  
fun main() {  
    val person = Person(firstName: "A", age: 100)  
    println(person.nextYearAge())  
}
```

1. 확장함수

멤버함수가 우선적으로 호출된다.

확장함수를 만들었지만, 다른 기능의 똑같은 멤버함수가 생기면?
오류가 발생할 수 있다!!

1. 확장함수

확장함수가 오버라이드 된다면?!!

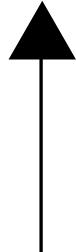
1. 확장함수



```
open class Train(  
    val name: String = "새마을기차",  
    val price: Int = 5_000,  
)  
  
fun Train.isExpensive(): Boolean {  
    println("Train의 확장함수")  
    return this.price >= 10000  
}  
  
class Srt : Train(name: "SRT", price: 40_000)  
  
fun Srt.isExpensive(): Boolean {  
    println("Srt의 확장함수")  
    return this.price >= 10000  
}
```

Train

Srt



1. 확장함수

```
val train: Train = Train()  
train.isExpensive() // Train의 확장함수
```

```
val srt1: Train = Srt()  
srt1.isExpensive() // Train의 확장함수
```

```
val srt2: Srt = Srt()  
srt2.isExpensive() // Srt의 확장함수
```

1. 확장함수

```
val train: Train = Train()  
train.isExpensive() // Train의 확장함수
```

```
val srt1: Train = Srt()  
srt1.isExpensive() // Train의 확장함수
```

```
val srt2: Srt = Srt()  
srt2.isExpensive() // Srt의 확장함수
```

1. 확장함수

```
val train: Train = Train()  
train.isExpensive() // Train의 확장함수
```

```
val srt1: Train = Srt()  
srt1.isExpensive() // Train의 확장함수
```

```
val srt2: Srt = Srt()  
srt2.isExpensive() // Srt의 확장함수
```

1. 확장함수

```
val train: Train = Train()  
train.isExpensive() // Train의 확장함수
```

```
val srt1: Train = Srt()  
srt1.isExpensive() // Train의 확장함수
```

```
val srt2: Srt = Srt()  
srt2.isExpensive() // Srt의 확장함수
```

1. 확장함수

해당 변수의 **현재 타입**

즉, 정적인 타입에 의해 어떤 확장함수가 호출될지 결정된다.

1. 확장함수 - 중간정리

1. 확장함수는 원본 클래스의 private, protected 멤버 접근이 안된다!
2. 멤버함수, 확장함수 중 멤버함수에 우선권이 있다!
3. 확장함수는 현재 타입을 기준으로 호출된다!

1. 확장함수

Java에서 Kotlin 확장함수를 가져다 사용할 수 있나?!

1. 확장함수

```
public static void main(String[] args) {  
    StringUtilsKt.lastChar( $this$lastChar: "ABC");  
}
```

정적 메소드를 부르는 것처럼 사용 가능하다.

1. 확장함수

확장함수 라는 개념은 **확장프로퍼티**와도 연결됩니다.

1. 확장함수

```
fun String.lastChar(): Char {  
    return this[this.length - 1]  
}  
  
val String.lastChar: Char  
    get() = this[this.length - 1]
```

확장 프로퍼티의 원리는 확장함수 + custom getter와 동일하다!

2. infix 함수

중위함수, 함수를 호출하는 새로운 방법!!

2. infix 함수

downTo, step 도 함수이다! (중위 호출 함수)

변수.함수이름(argument) 대신

변수 함수이름 argument

2. infix 함수



```
fun Int.add(other: Int): Int {  
    return this + other  
}  
  
infix fun Int.add2(other: Int): Int {  
    return this + other  
}
```

2. infix 함수



```
fun Int.add(other: Int): Int {  
    return this + other  
}  
  
infix fun Int.add2(other: Int): Int {  
    return this + other  
}
```

2. infix 함수



```
fun Int.add(other: Int): Int {  
    return this + other  
}  
  
infix fun Int.add2(other: Int): Int {  
    return this + other  
}
```

3.add(4)

3.add2(
 other: 4)
3 add2 4

2. infix 함수



```
fun Int.add(other: Int): Int {  
    return this + other  
}  
  
infix fun Int.add2(other: Int): Int {  
    return this + other  
}
```

3.add(4)

3.add2(other: 4)
3 add2 4

Infix는 멤버함수에도 붙일 수 있습니다!

2. infix 함수



```
fun Int.add(other: Int): Int {  
    return this + other  
}  
  
infix fun Int.add2(other: Int): Int {  
    return this + other  
}
```

3.add(4)

3.add2(
 other: 4)
3 add2 4

3. inline 함수

함수가 호출되는 대신,
함수를 호출한 지점에 함수 본문을 그대로 복붙하고 싶은 경우!

3. inline 함수



```
fun main() {  
    3.add(4)  
}  
  
inline fun Int.add(other: Int): Int {  
    return this + other  
}
```



```
public static final void main() {  
    byte $this$add$iv = 3;  
    int other$iv = 4;  
    int $i$f$add = false;  
    int var10000 = $this$add$iv + other$iv;  
}
```

3. inline 함수



```
fun main() {  
    3.add(4)  
}  
  
inline fun Int.add(other: Int): Int {  
    return this + other  
}
```



```
public static final void main() {  
    byte $this$add$iv = 3;  
    int other$iv = 4;  
    int $i$f$add = false;  
    int var10000 = $this$add$iv + other$iv;  
}
```

3. inline 함수

함수를 파라미터로 전달할 때에 오버헤드를 줄일 수 있다.

하지만 inline 함수의 사용은
성능 측정과 함께 신중하게 사용되어야 합니다!

4. 지역함수

함수 안에 함수를 선언할 수 있다.

4. 지역함수



```
fun createPerson(firstName: String, lastName: String): Person {  
    if (firstName.isEmpty()) {  
        throw IllegalArgumentException("firstName은 비어있을 수 없습니다! 현재 값 : $firstName")  
    }  
  
    if (lastName.isEmpty()) {  
        throw IllegalArgumentException("lastName은 비어있을 수 없습니다! 현재 값 : $lastName")  
    }  
  
    return Person(firstName, lastName, age: 1)  
}
```

4. 지역함수



```
fun createPerson(firstName: String, lastName: String): Person {  
    if (firstName.isEmpty()) {  
        throw IllegalArgumentException("firstName은 비어있을 수 없습니다! 현재 값 : $firstName")  
    }  
  
    if (lastName.isEmpty()) {  
        throw IllegalArgumentException("lastName은 비어있을 수 없습니다! 현재 값 : $lastName")  
    }  
  
    return Person(firstName, lastName, age: 1)  
}
```

4. 지역함수



```
fun createPerson(firstName: String, lastName: String): Person {  
    fun validateName(name: String, fieldName: String) {  
        if (name.isEmpty()) {  
            throw IllegalArgumentException("${fieldName}은 비어있을 수 없습니다! 현재 값 : $name")  
        }  
    }  
    validateName(firstName, fieldName: "firstName")  
    validateName(lastName, fieldName: "lastName")  
  
    return Person(firstName, lastName, age: 1)  
}
```

4. 지역함수



```
fun createPerson(firstName: String, lastName: String): Person {  
    fun validateName(name: String, fieldName: String) {  
        if (name.isEmpty()) {  
            throw IllegalArgumentException("${fieldName}은 비어있을 수 없습니다! 현재 값 : $name")  
        }  
    }  
  
    validateName(firstName, fieldName: "firstName")  
    validateName(lastName, fieldName: "lastName")  
  
    return Person(firstName, lastName, age: 1)  
}
```

4. 지역함수

함수로 추출하면 좋을 것 같은데,
이 함수를 지금 함수 내에서만 사용하고 싶을 때

4. 지역함수

depth가 깊어지기도 하고, 코드가 그렇게 깔끔하지는 않다..

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- Java 코드가 있는 상황에서, Kotlin 코드로 추가 기능 개발을 하기 위해 **확장함수**와 **확장프로퍼티**가 등장했다.

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- Java 코드가 있는 상황에서, Kotlin 코드로 추가 기능 개발을 하기 위해 확장함수와 확장프로퍼티가 등장했다.

```
fun 확장하려는클래스.함수이름(파라미터): 리턴타입 {  
    // this를 이용해 실제 클래스 안의 값에 접근  
}
```

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- Java 코드가 있는 상황에서, Kotlin 코드로 추가 기능 개발을 하기 위해 확장함수와 확장프로퍼티가 등장했다.
- 확장함수는 원본 클래스의 private, protected 멤버 접근이 안된다!
- 멤버함수, 확장함수 중 멤버함수에 우선권이 있다!
- 확장함수는 현재 타입을 기준으로 호출된다!
- Java에서는 **static 함수를 쓰는 것처럼**
Kotlin의 확장함수를 쓸 수 있다.

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- 함수 호출 방식을 바꿔주는 **infix 함수**가 존재한다.

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- 함수 호출 방식을 바꿔주는 infix 함수가 존재한다.
- 함수를 복사-붙여넣기 하는 **inline 함수**가 존재한다.

Lec 16. 코틀린에서 다양한 함수를 다루는 방법

- 함수 호출 방식을 바꿔주는 infix 함수가 존재한다.
- 함수를 복사-붙여넣기 하는 inline 함수가 존재한다.
- Kotlin에서는 함수 안에 함수를 선언할 수 있고,
지역함수라고 부른다.

Lec 17. 코틀린에서 람다를 다루는 방법

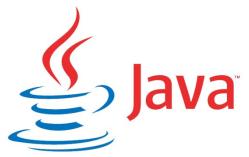
1. Java에서 람다를 다루기 위한 노력
2. 코틀린에서의 람다
3. Closure
4. 다시 try with resources

1. Java에서 람다를 다루기 위한 노력



```
public class Fruit {  
  
    private final String name;  
    private final int price;  
  
    public Fruit(String name, int price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    public String getName() { return name; }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

1. Java에서 람다를 다루기 위한 노력



과일가게

```
List<Fruit> fruits = Arrays.asList(  
    new Fruit( name: "사과", price: 1_000 ),  
    new Fruit( name: "사과", price: 1_200 ),  
    new Fruit( name: "사과", price: 1_200 ),  
    new Fruit( name: "사과", price: 1_500 ),  
    new Fruit( name: "바나나", price: 3_000 ),  
    new Fruit( name: "바나나", price: 3_200 ),  
    new Fruit( name: "바나나", price: 2_500 ),  
    new Fruit( name: "수박", price: 10_000 )  
);
```

1. Java에서 람다를 다루기 위한 노력

사장님~ 사과만 보여주세요~

```
private List<Fruit> findApples(List<Fruit> fruits) {  
    List<Fruit> apples = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruit.getName().equals("사과")) {  
            apples.add(fruit);  
        }  
    }  
    return apples;  
}
```

1. Java에서 람다를 다루기 위한 노력

사장님~ 바나나만 보여주세요~

```
private List<Fruit> findBananas(List<Fruit> fruits) {  
    List<Fruit> bananas = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruit.getName().equals("바나나")) {  
            bananas.add(fruit);  
        }  
    }  
    return bananas;  
}
```

1. Java에서 람다를 다루기 위한 노력

엇?! 중복이 존재하는군?!

```
private List<Fruit> findFruitsWithName(List<Fruit> fruits, String name) {  
    List<Fruit> results = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruit.getName().equals(name)) {  
            results.add(fruit);  
        }  
    }  
    return results;  
}
```

1. Java에서 람다를 다루기 위한 노력

사장님~ 사과랑 바나나 같이 보여주세요~

사과인데 가격이 1200원을 넘지 않는 사과만 주세요~

10000원 이하의 수박과 1000원 이상의 바나나 보여주세요~

1. Java에서 람다를 다루기 위한 노력

파라미터를 늘리는 것으로는 안되겠다!!

1. Java에서 람다를 다루기 위한 노력

인터페이스와 익명 클래스를 사용하자!!

1. Java에서 람다를 다루기 위한 노력

```
public interface FruitFilter {  
    boolean isSelected(Fruit fruit);  
}
```

```
private List<Fruit> filterFruits(List<Fruit> fruits, FruitFilter fruitFilter) {  
    List<Fruit> results = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruitFilter.isSelected(fruit)) {  
            results.add(fruit);  
        }  
    }  
    return results;  
}
```

1. Java에서 람다를 다루기 위한 노력

```
filterFruits(fruits, new FruitFilter() {  
    @Override  
    public boolean isSelected(Fruit fruit) {  
        return Arrays.asList("사과", "바나나").contains(fruit.getName()) &&  
            fruit.getPrice() > 5_000;  
    }  
});
```

1. Java에서 람다를 다루기 위한 노력

매우 좋습니다~ 무수한 메소드를 막았군요~

1. Java에서 람다를 다루기 위한 노력

하지만 ‘익명 클래스’를 사용하는 것은 복잡합니다.

1. Java에서 람다를 다루기 위한 노력

또한 다양한 Filter가 필요할 수도 있습니다.

과일 간의 무게 비교를 한다거나,
N개의 과일을 한 번에 비교한다거나 등등..

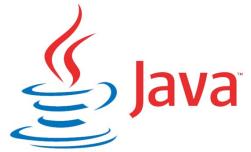
1. Java에서 람다를 다루기 위한 노력

JDK8부터 람다 (이름이 없는 함수) 등장!

1. Java에서 람다를 다루기 위한 노력

FruitFilter와 같은 인터페이스
Predicate, Consumer 등을 많이 만들어 두었다!

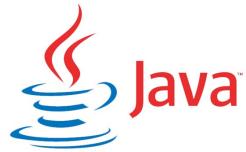
1. Java에서 람다를 다루기 위한 노력



```
filterFruits(fruits, fruit -> fruit.getName().equals("사과"));
```

```
private List<Fruit> filterFruits(List<Fruit> fruits, Predicate<Fruit> fruitFilter) {  
    List<Fruit> results = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruitFilter.test(fruit)) {  
            results.add(fruit);  
        }  
    }  
    return results;  
}
```

1. Java에서 람다를 다루기 위한 노력

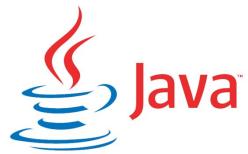


```
filterFruits(fruits, fruit -> fruit.getName().equals("사과"));
```

변수 -> 변수를 이용한 함수

(변수1, 변수2) -> 변수1과 변수 2를 이용한 함수

1. Java에서 람다를 다루기 위한 노력

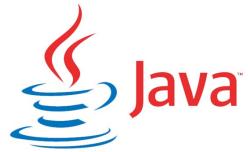


```
private List<Fruit> filterFruits(List<Fruit> fruits, Predicate<Fruit> fruitFilter) {  
    List<Fruit> results = new ArrayList<>();  
    for (Fruit fruit : fruits) {  
        if (fruitFilter.test(fruit)) {  
            results.add(fruit);  
        }  
    }  
    return results;  
}
```

1. Java에서 람다를 다루기 위한 노력

간결한 스트림이 등장했다.
(병렬처리에도 강점이 생김)

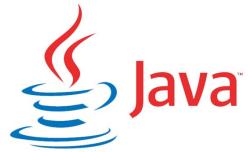
1. Java에서 람다를 다루기 위한 노력



```
filterFruits(fruits, fruit -> fruit.getName().equals("사과"));
```

```
private static List<Fruit> filterFruits(List<Fruit> fruits, Predicate<Fruit> fruitFilter) {  
    return fruits.stream()  
        .filter(fruitFilter)  
        .collect(Collectors.toList());  
}
```

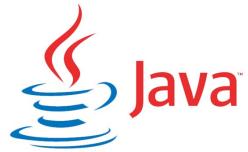
1. Java에서 람다를 다루기 위한 노력



```
filterFruits(fruits, Fruit::isApple);
```

```
private static List<Fruit> filterFruits(List<Fruit> fruits, Predicate<Fruit> fruitFilter) {  
    return fruits.stream()  
        .filter(fruitFilter)  
        .collect(Collectors.toList());  
}
```

1. Java에서 람다를 다루기 위한 노력



```
filterFruits(fruits, Fruit::isApple);
```

```
private static List<Fruit> filterFruits(List<Fruit> fruits, Predicate<Fruit> fruitFilter) {  
    return fruits.stream()  
        .filter(fruitFilter)  
        .collect(Collectors.toList());  
}
```

메소드 레퍼런스

1. Java에서 람다를 다루기 위한 노력

‘메소드 자체를 직접 넘겨주는 것처럼’ 쓸 수 있다

1. Java에서 람다를 다루기 위한 노력

(2급시민)

바꿔 말하면, Java에서 함수는
변수에 할당되거나 파라미터로 전달할 수 없다.

2. 코틀린에서의 람다

Java와는 근본적으로 다른 한 가지가 있다.

코틀린에서는 함수가 그 자체로 값이 될 수 있다.
변수에 할당할 수도, 파라미터로 넘길 수도 있다.

2. 코틀린에서의 람다



제가 한 번 람다(익명함수)를 변수에 넣어보겠습니다.

2. 코틀린에서의 람다



```
// 람다를 만드는 방법 1
val isApple = fun(fruit: Fruit): Boolean {
    return fruit.name == "사과"
}

// 람다를 만드는 방법 2
val isApple2 = { fruit: Fruit -> fruit.name == "사과" }
```

2. 코틀린에서의 람다



```
// 람다를 직접 호출하는 방법 1  
isApple(Fruit(name: "사과", price: 1000))  
  
// 람다를 직접 호출하는 방법 2  
isApple.invoke(Fruit(name: "사과", price: 1000))
```

2. 코틀린에서의 람다



```
val isApple: (Fruit) -> Boolean = fun(fruit: Fruit): Boolean {  
    return fruit.name == "사과"  
}  
  
val isApple2: (Fruit) -> Boolean = { fruit: Fruit -> fruit.name == "사과" }
```

함수의 타입 : (파라미터 타입...) → 반환 타입

2. 코틀린에서의 람다



filterFruits를 옮겨보겠습니다

2. 코틀린에서의 람다



```
val isApple: (Fruit) -> Boolean = { fruit: Fruit -> fruit.name == "사과" }
filterFruits(fruits, isApple)
```

```
private fun filterFruits(fruits: List<Fruit>, filter: (Fruit) -> Boolean): List<Fruit> {
    val results = mutableListOf<Fruit>()
    for (fruit in fruits) {
        if (filter.invoke(fruit)) {
            results.add(fruit)
        }
    }
    return results
}
```

2. 코틀린에서의 람다



```
private fun filterFruits(fruits: List<Fruit>, filter: (Fruit) -> Boolean): List<Fruit> {  
    val results = mutableListOf<Fruit>()  
    for (fruit in fruits) {  
        if (filter.invoke(fruit)) {  
            results.add(fruit)  
        }  
    }  
    return results  
}
```

Kotlin에서는 함수가 1급 시민이다. (Java에서는 2급 시민)

2. 코틀린에서의 람다



```
filterFruits(fruits) { it.name == "사과" }
```

```
filterFruits(fruits) { fruit ->
    println("사과만 받는다..!!!")
    fruit.name == "사과" ^filterFruits
}
```

마지막 파라미터가 함수인 경우, 소괄호 밖에 람다 사용 가능

2. 코틀린에서의 람다



```
filterFruits(fruits) { it.name == "사과" }
```

```
filterFruits(fruits) { fruit ->
    println("사과만 받는다..!!!")
    fruit.name == "사과" ^filterFruits
}
```

람다를 작성할때, 람다의 파라미터를 **it** 으로 직접 참조할 수 있다.

2. 코틀린에서의 람다



```
filterFruits(fruits) { it.name == "사과" }

filterFruits(fruits) { fruit ->
    println("사과만 받는다..!!!")
    fruit.name == "사과" ^filterFruits
}
```

람다를 여러줄 작성할 수 있고, 마지막 줄의 결과가 람다의 반환값이다.

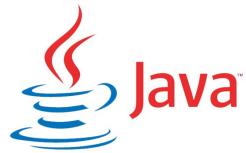
2. 코틀린에서의 람다



```
private fun filterFruits(fruits: List<Fruit>, filter: (Fruit) -> Boolean): List<Fruit> {  
    val results = mutableListOf<Fruit>()  
    for (fruit in fruits) {  
        if (filter.invoke(fruit)) {  
            results.add(fruit)  
        }  
    }  
    return results  
}
```

JDK8+ 처럼 선언식으로 변경 가능합니다!! 다음 시간에 말씀드릴게요!

3. Closure



```
String targetFruitName = "바나나";
targetFruitName = "수박";
filterFruits(fruits, (fruit) -> targetFruitName.equals(fruit.getName()));
```

Variable used in lambda expression should be final or effectively final

Java에서는 람다를 쓸 때 사용할 수 있는 변수에 제약이 있다!

3. Closure



```
var targetFruitName = "바나나"  
targetFruitName = "수박"  
filterFruits(fruits) { it.name == targetFruitName }
```

코틀린에서는 아무런 문제 없이 동작한다!

3. Closure

어떻게 이것이 가능한가?!

3. Closure

코틀린에서는 람다가 시작하는 지점에 참조하고 있는 변수들을
모두 포획하여 그 정보를 가지고 있다.

3. Closure

이렇게 해야만, 람다를 진정한 일급 시민으로 간주할 수 있다.
이 데이터 구조를 **Closure**라고 부른다.

4. 다시 try with resources



```
fun readFile(path: String) {
    BufferedReader(FileReader(path)).use { reader ->
        println(reader.readLine())
    }
}
```

4. 다시 try with resources



```
public inline fun <T : Closeable?, R> T.use(block: (T) -> R): R {
```

Closeable 구현체에 대한 확장함수이다

4. 다시 try with resources



```
public inline fun <T : Closeable?, R> T.use(block: (T) -> R): R {
```

Inline 함수이다.

4. 다시 try with resources



```
public inline fun <T : Closeable?, R> T.use(block: (T) -> R): R {
```

람다를 받게 만들어진 함수이다.

4. 다시 try with resources



```
fun readFile(path: String) {  
    BufferedReader(FileReader(path)).use { reader ->  
        println(reader.readLine())  
    }  
}
```

실제 람다를 전달하고 있다.

Lec 17. 코틀린에서 람다를 다루는 방법

- 함수는 Java에서 2급시민이지만, 코틀린에서는 **1급시민**이다.
 - 때문에, 함수 자체를 변수에 넣을 수도 있고
파라미터로 전달할 수도 있다.

Lec 17. 코틀린에서 람다를 다루는 방법

- 함수는 Java에서 2급시민이지만, 코틀린에서는 1급시민이다.
 - 때문에, 함수 자체를 변수에 넣을 수도 있고
파라미터로 전달할 수도 있다.
 - 코틀린에서 함수 타입은 (**파라미터 타입, ...**) → **반환타입** 이었다.

Lec 17. 코틀린에서 람다를 다루는 방법

- 코틀린에서 람다는 두 가지 방법으로 만들 수 있고, {} 방법이 더 많이 사용된다.

```
// 람다를 만드는 방법 1
val isApple = fun(fruit: Fruit): Boolean {
    return fruit.name == "사과"
}

// 람다를 만드는 방법 2
val isApple2 = { fruit: Fruit -> fruit.name == "사과" }
```

Lec 17. 코틀린에서 람다를 다루는 방법

- 코틀린에서 람다는 두 가지 방법으로 만들 수 있고, {} 방법이 더 많이 사용된다.
- 함수를 호출하며, 마지막 파라미터인 람다를 쓸 때는 소괄호 밖으로 람다를 뺄 수 있다.

```
filterFruits(fruits) { fruit -> fruit.name == "사과" }
```

```
filterFruits(fruits) { it.name == "사과" }
```

Lec 17. 코틀린에서 람다를 다루는 방법

- 코틀린에서 람다는 두 가지 방법으로 만들 수 있고, {} 방법이 더 많이 사용된다.
- 함수를 호출하며, 마지막 파라미터인 람다를 쓸 때는 소괄호 밖으로 람다를 뺄 수 있다.
- 람다의 마지막 expression 결과는 람다의 반환 값이다.

```
filterFruits(fruits) { fruit ->
    println("사과만 받는다...!!")
    fruit.name == "사과" ^filterFruits
}
```

Lec 17. 코틀린에서 람다를 다루는 방법

- 코틀린에서 람다는 두 가지 방법으로 만들 수 있고, {} 방법이 더 많이 사용된다.
- 함수를 호출하며, 마지막 파라미터인 람다를 쓸 때는 소괄호 밖으로 람다를 뺄 수 있다.
- 람다의 마지막 expression 결과는 람다의 반환 값이다.
- 코틀린에서는 Closure를 사용하여 non-final 변수도 람다에서 사용 할 수 있다.

Lec 18. 코틀린에서 컬렉션을 함수형으로 다루는 방법

1. 필터와 맵
2. 다양한 컬렉션 처리 기능
3. List를 Map으로
4. 중첩된 컬렉션 처리

1. 필터와 맵



오늘도 Fruit를 가져왔습니다.

```
data class Fruit(  
    val id: Long,  
    val name: String,  
    val factoryPrice: Long,  
    val currentPrice: Long,  
)
```

1. 필터와 맵



사과만 주세요!

1. 필터와 맵



사과의 가격들을 알려주세요!

1. 필터와 맵



filter : 사과만 주세요!

```
val apples = fruits.filter { fruit -> fruit.name == "사과" }
```

1. 필터와 맵



필터에서 인덱스가 필요하면?!

```
val apples = fruits.filterIndexed { idx, fruit ->
    println(idx)
    fruit.name == "사과" ^filterIndexed
}
```

1. 필터와 맵



map : 사과의 가격들을 알려주세요!

```
val applePrices = fruits.filter { fruit -> fruit.name == "사과" }  
    .map { fruit -> fruit.currentPrice }
```

1. 필터와 맵



맵에서 인덱스가 필요하다면?!

```
val applePrices = fruits.filter { fruit -> fruit.name == "사과" }  
    .mapIndexed { idx, fruit ->  
        println(idx)  
        fruit.currentPrice ^mapIndexed  
    }
```

1. 필터와 맵



Mapping의 결과가 null이 아닌 것만 가져오고 싶다면?!

```
val values = fruits.filter { fruit -> fruit.name == "사과" }  
    .mapNotNull { fruit -> fruit.nullOrValue() }
```

1. 필터와 맵



filter / filterIndexed
map / mapIndexed / mapNotNull

2. 다양한 컬렉션 처리 기능



모든 과일이 사과인가요?!

2. 다양한 컬렉션 처리 기능



혹시 출고가 10,000원 이상의 과일이 하나라도 있나요?!

2. 다양한 컬렉션 처리 기능



all : 조건을 모두 만족하면 true 그렇지 않으면 false

```
val isAllApple = fruits.all { fruit -> fruit.name == "사과" }
```

2. 다양한 컬렉션 처리 기능



none : 조건을 모두 불만족하면 true 그렇지 않으면 false

```
val isNoApple = fruits.none { fruit -> fruit.name == "사과" }
```

2. 다양한 컬렉션 처리 기능



any : 조건을 하나라도 만족하면 true 그렇지 않으면 false

```
val isNoApple = fruits.any { fruit -> fruit.factoryPrice >= 10_000 }
```

2. 다양한 컬렉션 처리 기능



총 과일 개수가 몇 개인가요?!

2. 다양한 컬렉션 처리 기능



낮은 가격 순으로 보여주세요!

2. 다양한 컬렉션 처리 기능



과일이 몇 종류 있죠?!

2. 다양한 컬렉션 처리 기능



count : 개수를 센다

```
val fruitCount = fruits.count()
```

2. 다양한 컬렉션 처리 기능



sortedBy : (오름차순) 정렬을 한다

```
val fruitCount = fruits.sortedBy { fruit -> fruit.currentPrice }
```

2. 다양한 컬렉션 처리 기능



sortedByDescending : (내림차순) 정렬을 한다

```
val fruitCount = fruits.sortedByDescending { fruit -> fruit.currentPrice }
```

2. 다양한 컬렉션 처리 기능



distinctBy : 변형된 값을 기준으로 중복을 제거한다

```
val distinctFruitNames = fruits.distinctBy { fruit -> fruit.name }  
    .map { fruit -> fruit.name }
```

2. 다양한 컬렉션 처리 기능



첫번째 과일만 주세요!
마지막 과일만 주세요!

2. 다양한 컬렉션 처리 기능



first : 첫번째 값을 가져온다 (무조건 null이 아니어야함)

firstOrNull : 첫번째 값 또는 null을 가져온다

```
fruits.first()
```

```
fruits.firstOrNull()
```

2. 다양한 컬렉션 처리 기능



last : 마지막 값을 가져온다 (무조건 null이 아니어야함)

lastOrNull : 첫번째 값 또는 null을 가져온다

```
fruits.last()
```

```
fruits.lastOrNull()
```

2. 다양한 컬렉션 처리 기능



all / none / any

count / sortedBy / sortedByDescending / distinctBy
first / firstOrNull / last / lastOrNull

3. List를 Map으로



과일이름 → List<과일>
Map이 필요해요!

```
val map: Map<String, List<Fruit>> = fruits.groupBy { fruit -> fruit.name }
```

3. List를 Map으로



id → 과일
Map이 필요해요!

```
val map: Map<Long, Fruit> = fruits.associateBy { fruit -> fruit.id }
```

3. List를 Map으로



Key와 value를 동시에 처리할 수도 있습니다.

3. List를 Map으로



과일이름 → List<출고가> Map이 필요해요!

```
val map: Map<String, List<Long>> = fruits
    .groupBy({ fruit -> fruit.name }, { fruit -> fruit.factoryPrice })
```

3. List를 Map으로



id → 출고가 Map이 필요해요!

```
val map: Map<Long, Long> = fruits  
    .associateBy({ fruit -> fruit.id }, { fruit -> fruit.factoryPrice })
```

3. List를 Map으로



Map에 대해서도 앞선 기능들을 대부분 사용할 수 있습니다.

```
val map: Map<String, List<Fruit>> = fruits.groupBy { fruit -> fruit.name }  
.filter { (key, value) -> key == "사과" }
```

3. List를 Map으로



groupBy / associateBy

4. 중첩된 컬렉션 처리



```
val fruitsInList: List<List<Fruit>> = listOf(
    listOf(
        Fruit(id: 1L, name: "사과", factoryPrice: 1_000, currentPrice: 1_500),
        Fruit(id: 2L, name: "사과", factoryPrice: 1_200, currentPrice: 1_500),
        Fruit(id: 3L, name: "사과", factoryPrice: 1_200, currentPrice: 1_500),
        Fruit(id: 4L, name: "사과", factoryPrice: 1_500, currentPrice: 1_500),
    ),
    listOf(
        Fruit(id: 5L, name: "바나나", factoryPrice: 3_000, currentPrice: 3_200),
        Fruit(id: 6L, name: "바나나", factoryPrice: 3_200, currentPrice: 3_200),
        Fruit(id: 7L, name: "바나나", factoryPrice: 2_500, currentPrice: 3_200),
    ),
    listOf(
        Fruit(id: 8L, name: "수박", factoryPrice: 10_000, currentPrice: 10_000),
    )
)
```

4. 중첩된 컬렉션 처리



이 상황에서, 출고가와 현재가가 동일한 과일을 골라주세요!

```
val samePriceFruits = fruitsInList.flatMap { list ->
    list.filter { fruit -> fruit.factoryPrice == fruit.currentPrice }
}
```

4. 중첩된 컬렉션 처리



```
val samePriceFruits = fruitsInList.flatMap { list -> list.samePriceFilter }
```

```
    val List<Fruit>.samePriceFilter: List<Fruit>
        get() = this.filter(Fruit::isSamePrice)
```

```
data class Fruit(
    val id: Long,
    val name: String,
    val factoryPrice: Long,
    val currentPrice: Long,
) {

    val isSamePrice: Boolean
        get() = factoryPrice == currentPrice
}
```

4. 중첩된 컬렉션 처리



List<List<Fruit>>를 List<Fruit>로 그냥 바꾸어주세요!

```
fruitsInList.flatten()
```

4. 중첩된 컬렉션 처리



flatMap / flatten

Lec 18. 코틀린에서 컬렉션을 함수형으로 다루는 방법

filter / filterIndexed

map / mapIndexed / mapNotNull

all / none / any

count / sortedBy / sortedByDescending / distinct

first / firstOrNull / last / lastOrNull

groupBy / associateBy

flatMap / flatten

Lec 19. 코틀린의 이모저모

1. Type Alias와 as import
2. 구조분해와 componentN 함수
3. Jump와 Label
4. TakeIf와 TakeUnless

1. Type Alias와 as import

긴 이름의 클래스 혹은 함수 타입이 있을때
축약하거나 더 좋은 이름을 쓰고 싶다!

1. Type Alias와 as import



```
fun filterFruits(fruits: List<Fruit>, filter: (Fruit) -> Boolean) {  
}
```

(Fruit) -> Boolean 이란 타입이 너무 길어!!
파라미터가 더 많아진다면..?!!

1. Type Alias 와 as import



```
typealias FruitFilter = (Fruit) -> Boolean  
fun filterFruits(fruits: List<Fruit>, filter: FruitFilter) {  
}
```

A red arrow points from the word "FruitFilter" in the first line of code to the parameter "filter" in the second line of code, highlighting the type alias definition and its usage.

1. Type Alias와 as import



```
data class UltraSuperGuardianTribe(  
    val name: String  
)  
  
typealias USGTMap = Map<String, UltraSuperGuardianTribe>
```

이름 긴 클래스를 컬렉션에 사용할 때도 간단히 줄일 수 있다.

1. Type Alias와 as import



```
b/Print.kt
1 package com.lannstark.lec19.b
2
3     fun printHelloWorld() {
4         println("Hell World B")
5     }
a/Print.kt
1 package com.lannstark.lec19.a
2
3     fun printHelloWorld() {
4         println("Hell World A")
5     }
```

다른 패키지의 같은 이름 함수를 동시에 가져오고 싶다면?!

1. Type Alias와 as import

as import : 어떤 클래스나 함수를 임포트 할 때 이름을 바꾸는 기능

1. Type Alias와 as import



```
import com.lannstark.lec19.a.printHelloWorld as printHelloWorldA
import com.lannstark.lec19.b.printHelloWorld as printHelloWorldB

fun main() {
    printHelloWorldA()
    printHelloWorldB()
}
```

1. Type Alias와 as import



```
import com.lannstark.lec19.a.printHelloWorld as printHelloWorldA
import com.lannstark.lec19.b.printHelloWorld as printHelloWorldB

fun main() {
    printHelloWorldA()
    printHelloWorldB()
}
```

Import와 동시에 이름을 바꿀 수 있다.

2. 구조분해와 componentN 함수

구조분해 : 복합적인 값을 분해하여 여러 변수를 한 번에 초기화하는 것

2. 구조분해와 componentN 함수



```
val person = Person(name: "최태현", age: 100)  
val (name, age) = person
```

2. 구조분해와 componentN 함수



```
data class Person(  
    val name: String,  
    val age: Int  
)
```

```
val person = Person(name: "최태현", age: 100)  
val (name, age) = person
```

Data Class는 componentN이란 함수도 자동으로 만들어준다!

2. 구조분해와 componentN 함수



```
val person = Person(name: "최태현", age: 100)  
val name = person.component1()  
val age = person.component2()
```

```
val person = Person(name: "최태현", age: 100)  
val (name, age) = person
```

둘은 같은 코드이다!

2. 구조분해와 componentN 함수



Data Class가 아닌데 구조분해를 사용하고 싶다면,
componentN 함수를 직접 구현해줄 수도 있다.

2. 구조분해와 componentN 함수



```
class Person(  
    val name: String,  
    val age: Int  
) {  
  
    operator fun component1(): String {  
        return this.name  
    }  
  
    operator fun component2(): Int {  
        return this.age  
    }  
}
```

2. 구조분해와 componentN 함수



```
val map = mapOf(1 to "A", 2 to "B")
for ((key, value) in map.entries) {
}
```

이 문법 역시 구조분해 입니다!

3. Jump Label

return / break / continue

3. Jump와 Label

- return : 기본적으로 가장 가까운 enclosing function 또는 익명함수로 값이 반환된다
- break : 가장 가까운 루프가 제거된다
- continue : 가장 가까운 루프를 다음 step으로 보낸다

3. Jump와 Label

for문 및 while 문에서
break, continue 기능은 동일합니다.
단!!!

3. Jump와 Label



```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.forEach { number ->
    if (number == 3) continue
    println(number)
}
```

3. Jump와 Label



```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.forEach { number ->
    if (number == 3) continue
    println(number)
}
```

forEach문과 함께 break 또는 continue를 꼭 쓰고 싶다면?!

3. Jump와 Label



```
val numbers = listOf(1, 2, 3, 4, 5)
run {
    numbers.forEach { number ->
        if (number == 3) {
            return@run
        }
        println(number)
    }
}
```

break

3. Jump 와 Label



```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.forEach{ it: Int
    println(it)
    if (it == 3) {
        return@forEach
    }
}
```

continue

3. Jump와 Label

break, continue를 사용할 때엔
가급적 익숙하신 for문 사용을 추천드립니다!

3. Jump와 Label

코틀린에는 라벨이라는 기능이 있다.

3. Jump와 Label

특정 expression에 라벨이름@ 을 붙여 하나의 라벨로 간주하고
break, continue, return 등을 사용하는 기능

3. Jump 와 Label



```
loop@ for (i in 1..100) {  
    for (j in 1..100) {  
        if (j == 2) {  
            break@loop  
        }  
        print("${i} ${j}")  
    }  
}
```

3. Jump 와 Label



```
loop@ for (i in 1..100) {  
    for (j in 1..100) {  
        if (j == 2) {  
            break@loop  
        }  
        print("${i} ${j}")  
    }  
}
```

A red box highlights the label 'loop@' at the start of the first loop. A red arrow points from the 'break' keyword in the inner loop to the 'loop@' label, illustrating how the program exits the outer loop when it reaches the second iteration of the inner loop.

3. Jump와 Label

라벨을 사용한 Jump는 사용하지 않는 것을 강력 추천드립니다!

4. TakeIf와 TakeUnless



```
fun getNumberOrNull(): Int? {  
    return if (number <= 0) {  
        null  
    } else {  
        number  
    }  
}
```

Kotlin에서는 method chaning을 위한 특이한 함수를 제공합니다.

4. TakeIf와 TakeUnless



```
fun getNumberOrNullV2(): Int? {  
    return number.takeIf { it > 0 }  
}
```

주어진 조건을 만족하면 그 값이, 그렇지 않으면 null이 반환된다.

4. TakeIf와 TakeUnless



```
fun getNumberOrNullV3(): Int? {  
    return number.takeUnless { it <= 0 }  
}
```

주어진 조건을 만족하지 않으면 그 값이
그렇지 않으면 null이 반환된다.

Lec 19. 코틀린의 이모저모

- 타입에 대한 별칭을 줄 수 있는 **typealias**라는 키워드가 존재한다.

Lec 19. 코틀린의 이모저모

- 타입에 대한 별칭을 줄 수 있는 typealias라는 키워드가 존재한다.
- Import 당시 이름을 바꿀 수 있는 **as import** 기능이 존재한다.

Lec 19. 코틀린의 이모저모

- 타입에 대한 별칭을 줄 수 있는 typealias라는 키워드가 존재한다.
- Import 당시 이름을 바꿀 수 있는 as import 기능이 존재한다.
- 변수를 한 번에 선언할 수 있는 구조분해 기능이 있으며 componentN 함수를 사용한다.

Lec 19. 코틀린의 이모저모

- 타입에 대한 별칭을 줄 수 있는 typealias라는 키워드가 존재한다.
- Import 당시 이름을 바꿀 수 있는 as import 기능이 존재한다.
- 변수를 한 번에 선언할 수 있는 구조분해 기능이 있으며 componentN 함수를 사용한다.
- for문, while문과 달리 **forEach**에는 break와 continue를 사용할 수 없다.

Lec 19. 코틀린의 이모저모

- 타입에 대한 별칭을 줄 수 있는 typealias라는 키워드가 존재한다.
- Import 당시 이름을 바꿀 수 있는 as import 기능이 존재한다.
- 변수를 한 번에 선언할 수 있는 구조분해 기능이 있으며 componentN 함수를 사용한다.
- for문, while문과 달리 forEach에는 break와 continue를 사용할 수 없다.
- **takelf**와 **takeUnless**를 활용해 코드양을 줄이고 method chaning을 활용할 수 있다.

Lec 20. 코틀린의 scope function

1. scope function이란 무엇인가?!
2. scope function의 분류
3. 언제 어떤 scope function을 사용해야 할까?!
4. scope function과 가독성

1. scope function이란 무엇인가?!

scope : 영역

function : 함수

1. scope function이란 무엇인가?!

scope function : 일시적인 영역을 형성하는 함수

1. scope function이란 무엇인가?!



```
fun printPerson(person: Person?) {  
    if (person != null) {  
        println(person.name)  
        println(person.age)  
    }  
}
```

이 코드를 일시적인 영역과 함께 리팩토링 해보겠습니다.

1. scope function이란 무엇인가?!



```
fun printPerson(person: Person?) {  
    person?.let { it: Person  
        println(it.name)  
        println(it.age)  
    }  
}
```

1. scope function이란 무엇인가?!



```
fun printPerson(person: Person?) {  
    person?.let { it: Person  
        println(it.name)  
        println(it.age)  
    }  
}
```

Safe Call (?.) 을 사용 : person이 null이 아닐때에 let을 호출

1. scope function이란 무엇인가?!



```
fun printPerson(person: Person?) {  
    person?.let { it: Person  
        println(it.name)  
        println(it.age)  
    }  
}
```

let : scope function의 한 종류

1. scope function이란 무엇인가?!



```
public inline fun <T, R> T.let(block: (T) -> R): R {  
    return block(this)  
}
```

let : 확장함수. 람다를 받아, 람다 결과를 반환한다.

1. scope function이란 무엇인가?!



```
fun printPerson(person: Person?) {  
    person?.let { it: Person  
        println(it.name)  
        println(it.age)  
    }  
}
```

람다를 사용하고 있다. 람다 안에서 it을 통해 person에 접근

1. scope function이란 무엇인가?!

람다를 사용해 일시적인 영역을 만들고
코드를 더 간결하게 만들거나, method chaning에 활용하는 함수를
scope function이라고 합니다.

2. scope function의 분류

절대 외우지 마세요!

2. scope function의 분류

let

run

also

apply

with

2. scope function의 분류

let

run

also

apply

with

2. scope function의 분류

람다의 결과 let run

also apply

with

2. scope function의 분류

람다의 결과 let run

객체 그 자체 also apply

with

2. scope function의 분류



```
val value1 = person.let { it: Person
    it.age
}

val value2 = person.run { this: Person
    this.age
}

val value3 = person.also { it: Person
    it.age
}

val value4 = person.apply { this: Person
    this.age
}
```

2. scope function의 분류



age

```
val value1 = person.let { it: Person
    it.age
}

val value2 = person.run { this: Person
    this.age
}

val value3 = person.also { it: Person
    it.age
}

val value4 = person.apply { this: Person
    this.age
}
```

2. scope function의 분류



person

```
val value1 = person.let { it: Person
    it.age
}

val value2 = person.run { this: Person
    this.age
}

val value3 = person.also { it: Person
    it.age
}

val value4 = person.apply { this: Person
    this.age
}
```

2. scope function의 분류

람다의 결과 let run

객체 그 자체 also apply

with

2. scope function의 분류

it 사용

람다의 결과 **let** run

객체 그 자체 **also** apply

with

2. scope function의 분류

it 사용

람다의 결과 let

객체 그 자체 also

with

this 사용

run

apply

2. scope function의 분류



```
val value1 = person.let { it: Person  
    it.age  
}
```

```
val value2 = person.run { this: Person  
    this.age  
}
```

```
val value3 = person.also { it: Person  
    it.age  
}
```

```
val value4 = person.apply { this: Person  
    this.age  
}
```

2. scope function의 분류



```
val value1 = person.let { it: Person  
    it.age  
}
```

```
val value2 = person.run { this: Person  
    this.age  
}
```

```
val value3 = person.also { it: Person  
    it.age  
}
```

```
val value4 = person.apply { this: Person  
    this.age  
}
```

2. scope function의 분류

this : 생략이 가능한 대신, 다른 이름을 붙일 수 없다.

it : 생략이 불가능한 대신, 다른 이름을 붙일 수 있다.

2. scope function의 분류



```
val value1 = person.let { p ->
    p.age
}
```

```
val value2 = person.run { this: Person
    age
}
```

2. scope function의 분류

왜 이런 차이가 발생할까?!

2. scope function의 분류



```
public inline fun <T, R> T.let(block: (T) -> R): R {  
    return block(this)  
}  
  
public inline fun <T, R> T.run(block: T.() -> R): R {  
    return block()  
}
```

2. scope function의 분류



```
public inline fun <T, R> T.let(block: (T) -> R): R {  
    return block(this)  
}  
  
public inline fun <T, R> T.run(block: T.() -> R): R {  
    return block()  
}
```

let은 일반 함수를 받는다.

2. scope function의 분류



```
public inline fun <T, R> T.let(block: (T) -> R): R {  
    return block(this)  
}  
  
public inline fun <T, R> T.run(block: T.() -> R): R {  
    return block()  
}
```

run은 **확장 함수**를 받는다.

2. scope function의 분류

확장함수에서는 본인 자신을 this로 호출하고, 생략할 수 있었다.

2. scope function의 분류



```
val person = Person(name: "최태현", age: 100)
with(person) { this: Person
    println(name)
    println(this.age)
}
```

with(파라미터, 람다) : this를 사용해 접근하고, this는 생략 가능하다.

2. scope function의 분류

it 사용

람다의 결과

let

this 사용

run

객체 그 자체

also

apply

with

3. 언제 어떤 scope function을 사용해야 할까?! - let



```
val strings = listOf("APPLE", "CAR")
strings.map { it.length }
    .filter { it > 3 }
    .let(::println)
```

하나 이상의 함수를 call chain 결과로 호출 할 때

3. 언제 어떤 scope function을 사용해야 할까?! - let



```
val strings = listOf("APPLE", "CAR")
strings.map { it.length }
    .filter { it > 3 }
    .let { lengths -> println(lengths) }
```

동일한 코드입니다.

3. 언제 어떤 scope function을 사용해야 할까?! - let



```
val length = str?.let { it: String  
    println(it.uppercase())  
    it.length ^let  
}
```

non-null 값에 대해서만 code block을 실행시킬 때

3. 언제 어떤 scope function을 사용해야 할까?! - let



```
val numbers = listOf("one", "two", "three", "four")
val modifiedFirstItem = numbers.first()
    .let { firstItem ->
        if (firstItem.length >= 5) firstItem else "!$firstItem!"
    }.uppercase()
println(modifiedFirstItem)
```

일회성으로 제한된 영역에 지역 변수를 만들 때

3. 언제 어떤 scope function을 사용해야 할까?! - run

객체 초기화와 반환 값의 계산을 동시에 해야 할 때

3. 언제 어떤 scope function을 사용해야 할까?! - run



```
val person = Person(name: "최태현", age: 100).run(personRepository::save)
```

객체를 만들어 DB에 바로 저장하고, 그 인스턴스를 활용할 때

3. 언제 어떤 scope function을 사용해야 할까?! - run



```
val person = Person(name: "최태현", age: 100).run { this: Person
    hobby = "독서"
    personRepository.save(person: this) ^run
}
```

객체를 만들어 DB에 바로 저장하고, 그 인스턴스를 활용할 때

3. 언제 어떤 scope function을 사용해야 할까?! - run

개인적으로는 잘 사용하지 않습니다!

아래 코드가 익숙하기도 하고,

반복되는 생성 후처리는 생성자, 프로퍼티, init block으로 넣는 것이 좋다.

```
val person = personRepository.save(Person(name: "최태현", age: 100))
```

3. 언제 어떤 scope function을 사용해야 할까?! - apply

apply 특징 : 객체 그 자체가 반환된다.

객체 설정을 할 때에
객체를 수정하는 로직이 call chain 중간에 필요할 때

3. 언제 어떤 scope function을 사용해야 할까?! - apply



```
fun createPerson(  
    name: String,  
    age: Int,  
    hobby: String,  
): Person {  
    return Person(  
        name = name,  
        age = age,  
    ).apply { this: Person  
        this.hobby = hobby  
    }  
}
```

Test Fixture를 만들 때

3. 언제 어떤 scope function을 사용해야 할까?! - apply



```
val person = Person(name: "최태현", age: 100)
person.apply { this.growOld() }
    .let { println(it) }
```

이런 코드도 가능한 합니다!!

3. 언제 어떤 scope function을 사용해야 할까?! - also

also 특징 : 객체 그 자체가 반환된다.

객체를 수정하는 로직이 call chain 중간에 필요할 때

3. 언제 어떤 scope function을 사용해야 할까?! - also



```
mutableListOf("one", "two", "three")
    .also { println("four 추가 이전 지금 값: $it") }
    .add("four")
```

3. 언제 어떤 scope function을 사용해야 할까?! - with

특정 객체를 다른 객체로 변환해야 하는데,
모듈 간의 의존성에 의해
정적 팩토리 혹은 toClass 함수를 만들기 어려울 때

3. 언제 어떤 scope function을 사용해야 할까?! - with



```
return with(person) { this: Person  
    PersonDto(  
        name = name,  
        age = age,  
    )  
}
```

this를 생략할 수 있어 필드가 많아도 코드가 간결해진다.

4. scope function과 가독성

scope function을 사용한 코드가
그렇지 않은 코드보다 가독성 좋은 코드일까?!

4. scope function과 가독성



```
// 1번 코드  
if (person != null && person.isAdult) {  
    view.showPerson(person)  
} else {  
    view.showError()  
}  
  
// 2번 코드  
person?.takeIf { it.isAdult }  
    ?.let(view::showPerson)  
    ?: view.showError()
```

1번 코드 : 전통적인 if와 else를 활용

4. scope function과 가독성



```
// 1번 코드  
if (person != null && person.isAdult) {  
    view.showPerson(person)  
} else {  
    view.showError()  
}  
  
// 2번 코드  
person?.takeIf { it.isAdult }  
    ?.let(view::showPerson)  
    ?: view.showError()
```

2번 코드 : scope function을 활용한 코틀린스러운 코드

4. scope function과 가독성

개인적으로 1번 코드가 훨씬 좋은 코드라고 생각합니다.

4. scope function과 가독성

1. 구현 2는 숙련된 코틀린 개발자만 더 알아보기 쉽다.
어쩌면 숙련된 코틀린 개발자도 잘 이해하지 못할 수 있다.
2. 구현 1의 디버깅이 쉽다.
3. 구현이 10이 수정도 더 쉽다.

4. scope function과 가독성



```
// 1번 코드  
if (person != null && person.isAdult) {  
    view.showPerson(person)  
} else {  
    view.showError()  
}  
  
// 2번 코드  
person?.takeIf { it.isAdult }  
    ?.let(view::showPerson)  
    ?: view.showError()
```

view.showPerson()이 null을 반환한다면?!

4. scope function과 가독성

사용 빈도가 적은 관용구는 코드를 더 복잡하게 만들고
이런 관용구들을 한 문장 내에서 조합해 사용하면
복잡성이 훨씬 증가한다.

4. scope function과 가독성

하지만 scope function을 사용하면 안되는 것도 아니다!
적절한 convention을 적용하면 유용하게 활용할 수 있다.

Lec 20. 코틀린의 scope function

- 코틀린의 **scope function**은 일시적인 영역을 만들어 코드를 더 간결하게 하거나, method chain에 활용된다.

Lec 20. 코틀린의 scope function

- 코틀린의 scope function은 일시적인 영역을 만들어 코드를 더 간결하게 하거나, method chain에 활용된다.
- scope function의 종류에는 **let / run / also / apply / with**가 있었다.

Lec 20. 코틀린의 scope function

it 사용

this 사용

람다의 결과 let

run

객체 그 자체 also apply

with

Lec 20. 코틀린의 scope function

- scope function을 사용한 코드는 사람에 따라 가독성을 다르게 느낄 수 있기 때문에, 함께 프로덕트를 만들어 가는 팀끼리 convention을 잘 정해야 한다.

감사합니다