

3. 코틀린

1. 코틀린 언어 소개
2. 변수와 함수
3. 조건문과 반복문
4. 객체지향
5. 람다 함수와 고차함수
6. 널 안전성

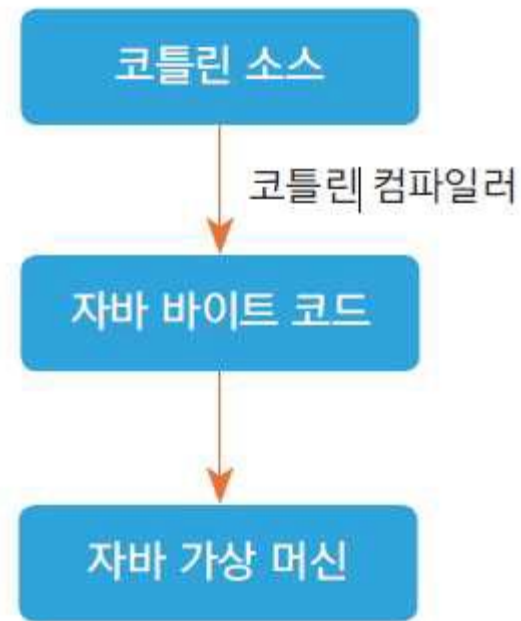
1. 코틀린 언어 소개

■ 코틀린의 등장 배경

- 코틀린은 젯브레인스(JetBrains)에서 오픈소스 그룹을 만들어 개발한 프로그래밍 언어
- 2017년 구글에서 안드로이드 공식 언어로 지정
- JVM에 기반을 둔 언어

■ 코틀린의 이점

- 최신 언어 기법을 이용하면 훨씬 간결한 구문으로 프로그램을 작성
- 코틀린은 널 안전성null safety을 지원하는 언어
- 코틀린은 자바와 100% 호환합니다.
- 코루틴coroutines이라는 기법을 이용하면 비동기 프로그래밍을 간소화할 수 있음



1. 코틀린 언어 소개

■ 코틀린 파일 구성

```
package com.example.lab3.test1

import java.text.SimpleDateFormat
import java.util.*

var data = 10

fun formatDate(date: Date): String {
    val sdfformat = SimpleDateFormat("yyyy-
mm-dd")
    return sdfformat.format(date)
}

class User {
    var name="hello"
    fun sayHello(){

    }
}
```

```
package com.example.lab3.test1.aaa

import java.util.*

fun main() {
    data=20
    formatDate(Date())
    User()
}
```

동일 패키지

```
package com.example.lab3.test1.aaa

import com.example.lab3.test1.User
import com.example.lab3.test1.data
import com.example.lab3.test1.formatDate
import java.util.*

fun main() {
    data=20
    formatDate(Date())
    User()
}
```

다른 패키지

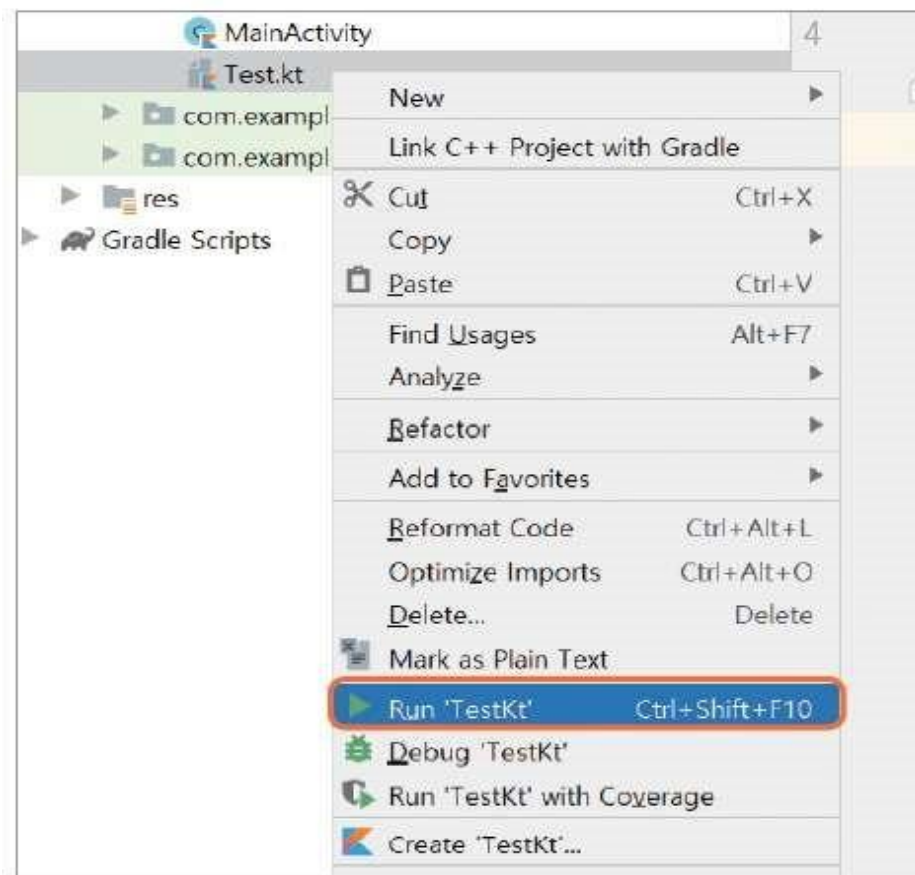
1. 코틀린 언어 소개

■ 코틀린 소스를 테스트하는 방법

- 테스트할 코틀린 소스 파일에는 `main()` 함수가 있어야 하며, 실행하면 `main()` 함수가 자동으로 실행됐다가 끝나면 프로그램이 종료됨



```
• 코틀린 소스 파일 • Test.kt  
  
fun main() {  
    println("hello world")  
}
```



2. 변수와 함수

■ 변수 선언하기

- 변수는 val, var 키워드로 선언
- val: value의 줄임말로 초깃 값이 할당되면 바꿀 수 없는 변수를 선언
- var: variable의 줄임말로 초깃 값이 할당된 후에도 값을 바꿀 수 있는 변수를 선언

변수 선언 형식

val(or var) 변수명 : 타입=값

val과 var 변수의 차이

```
val data1=10
```

```
var data2=10
```

```
fun main(){
```

```
    data1=20 //오류
```

```
    data2=20 //성공
```

```
}
```

2. 변수와 함수

■ 타입 지정과 타입 추론

- 변수명 뒤에는 콜론(:)을 추가해 타입을 명시
- 대입하는 값에 따라 타입을 유추(타입 추론)할 수 있을 때는 생략 가능

■ 초기값 할당

- 최상위에 선언한 변수나 클래스의 멤버 변수는 선언과 동시에 초기값을 할당해야 하며, 함수 내부에 선언한 변수는 선언과 동시에 초기값을 할당하지 않아도 됨.

변수에 타입 지정과 타입 추론

```
val data1:Int=10  
val data2=10
```

초기값 할당

```
val data1:Int //오류  
var data2=10 //성공
```

```
fun someFunc(){  
    val data3:Int  
    println("data3: $data3") //오류  
    data3=10  
    println("data3: $data3") //성공  
}
```

```
class User{  
    val data4: Int //오류  
    val data5 : Int=10 //성공  
}
```

2. 변수와 함수

■ 초기화 미루기

- lateinit 키워드는 이후에 초기값을 할당할 것임을 명시적으로 선언
- lateinit은 var 키워드로 선언한 변수에만 사용할 수 있음.
- Int, Long, Short, Double, Float, Boolean, Byte 타입에는 사용할 수 없음.
- by lazy { } 형식으로 선언하며, 소스에서 변수가 최초로 이용되는 순간 중괄호로 묶은 부분이 자동으로 실행되어 그 결과 값이 변수의 초기 값으로 할당

초기화 미루기-1

```
lateinit var data1: Int //error
lateinit val data2: String //error
lateinit var data3: String
class User
lateinit var user: User
lateinit var data4: String
```

초기화 미루기-2

```
val data5 by lazy {
    println("in lazy....")
    10
}
fun main() {
    println("main....")
    println(data5)
    println(data5)
}
```

▶ 실행 결과

```
in main.....
in lazy.....
20
20
```

2. 변수와 함수

■ 데이터 타입

- 코틀린의 모든 변수는 객체
- Int, Short, Long, Double, Float, Byte, Boolean — 기초 타입 객체
- Char, String — 문자와 문자열
- String 타입의 데이터는 문자열을 큰따옴표(")나 삼중 따옴표(""")로 감싸서 표현

• 문자 표현

```
val a: Char = 'a'
if (a == 1) {    // 오류!
}
```

• Int 타입에 null 대입과 메서드 이용

```
fun someFun() {
    var data1: Int = 10
    var data2: Int? = null    // null 대입 가능

    data1 = data1 + 10
    data1 = data1.plus(10)    // 객체의 메서드 이용 가능
}
```

• 기초 데이터 타입

```
val a1: Byte = 0b00001011

val a2: Int = 123
val a3: Short = 123
val a4: Long = 10L
val a5: Double = 10.0
val a6: Float = 10.0f

val a7: Boolean = true
```

• 문자열 표현 - 큰따옴표와 삼중 따옴표의 차이

```
fun main() {
    val str1 = "Hello \n World"
    val str2 = """
Hello
World
"""
    println("str1 : $str1")
    println("str2 : $str2")
}
```

▶ 실행 결과

```
str1 : Hello
      World
str2 :
Hello
World
```


2. 변수와 함수

■ 데이터 타입

- 문자열 템플릿 : String 타입의 데이터에 변수값이나 어떤 연산식의 결과값을 포함해야 할 때는 \$ 기호를 이용
- Any — 모든 타입 가능
- Unit — 반환문이 없는 함수

• 문자열 템플릿 사용 예

```
fun main() {  
    fun sum(no: Int): Int {  
        var sum = 0  
        for (i in 1..no) {  
            sum += i  
        }  
        return sum  
    }  
  
    val name: String = "kkang"  
    println("name : $name, sum : ${sum(10)}, plus : ${10 + 20}")  
}
```

▶ 실행 결과

```
name : kkang, sum : 55, plus : 30
```

• Any 타입 사용 예

```
val data1: Any = 10  
val data2: Any = "hello"  
  
class User  
val data3: Any = User()
```

• Unit 타입 사용 예

```
val data1: Unit = Unit
```

• Unit 타입 사용 예 - 반환문이 없는 함수

```
fun some(): Unit {  
    println(10 + 20)  
}
```

• 반환 타입을 생략한 예

```
fun some() {  
    println(10 + 20)  
}
```

2. 변수와 함수

■ 데이터 타입

- Nothing — null이나 예외를 반환하는 함수
- 널 허용과 불허용

• Nothing 사용 예

```
val data1: Nothing? = null
```

• null 반환 함수와 예외를 던지는 함수

```
fun some1(): Nothing? {  
    return null  
}  
  
fun some2(): Nothing {  
    throw Exception()  
}
```

• 널 허용과 불허용

```
var data1: Int = 10  
data1 = null    // 오류!  
  
var data2: Int? = 10  
data2 = null    // 성공!
```

2. 변수와 함수

■ 함수 선언하기

- 함수를 선언하려면 fun이라는 키워드를 이용
- 반환 타입을 선언할 수 있으며 생략하면 자동으로 Unit 타입이 적용
- 함수의 매개변수에는 var나 val 키워드를 사용할 수 없으며 val이 자동으로 적용

• 함수 선언 형식

```
fun 함수명(매개변수명: 타입): 반환 타입 { ... }
```

• 반환 타입이 있는 함수 선언

```
fun some(data1: Int): Int {  
    return data1 * 10  
}
```

• 매개변숫값 변경 오류

```
fun some(data1: Int) {  
    data1 = 20    // 오류!  
}
```

2. 변수와 함수

■ 함수 선언하기

- 함수의 매개변수에는 기본값 선언 가능

• 기본값 활용

```
fun main() {  
    fun some(data1: Int, data2: Int = 10): Int {  
        return data1 * data2  
    }  
    println(some(10))  
    println(some(10, 20))  
}
```

▶ 실행 결과

```
100  
200
```

- 매개변수명을 지정하여 호출하는 것을 명명된 매개변수라고 하며, 이렇게 하면 함수 선언문의 매개변수 순서에 맞춰 호출하지 않아도 됨.

• 매개변수명 생략 - 매개변수 순서대로 할당

```
fun some(data1: Int, data2: Int): Int {  
    return data1 * data2  
}  
println(some(10, 20))
```

• 매개변수명을 지정하여 호출

```
some(data2 = 20, data1 = 10)
```

2. 변수와 함수

■ 컬렉션 타입

- Array — 배열 표현
 - 배열은 Array 클래스로 표현
 - 배열의 데이터에 접근할 때는 대괄호([])를 이용해도 되고 set()이나 get() 함수를 이용할 수도 있음.

• 배열의 데이터에 접근하는 예

```
fun main() {  
    val data1: Array<Int> = Array(3, { 0 })  
    data1[0] = 10  
    data1[1] = 20  
    data1.set(2, 30)  
  
    println(  
        """"  
        array size : ${data1.size}  
        array data : ${data1[0]}, ${data1[1]}, ${data1.get(2)}  
        """"  
    )  
}
```

실행 결과

array size : 3
array data : 10, 20, 30

배열에서 2번째 데이터를 30으로 설정

2번째 데이터 가져오기

2. 변수와 함수

■ 컬렉션 타입

- 기초 타입의 배열
 - 기초 타입이라면 Array를 사용하지 않고 BooleanArray, ByteArray, CharArray, DoubleArray, FloatArray, IntArray, LongArray, ShortArray 클래스를 이용할 수도 있음
 - arrayOf()라는 함수를 이용하면 배열을 선언할 때 값을 할당할 수도 있음.
 - 기초 타입을 대상으로 하는 booleanArrayOf(), byteArrayOf(), charArrayOf(), doubleArrayOf(), floatArrayOf(), intArrayOf(), longArrayOf(), shortArrayOf() 함수를 제공

• 기초 타입 배열 선언

```
val data1: IntArray = IntArray(3, { 0 })  
val data2: BooleanArray = BooleanArray(3, { false })
```

• 기초 타입 arrayOf() 함수

```
val data1 = intArrayOf(10, 20, 30)  
val data2 = booleanArrayOf(true, false, true)
```

• 배열 선언과 동시에 값 할당

```
fun main() {  
    val data1 = arrayOf<Int>(10, 20, 30) — 크기가 3인 Int 배열을 선언하고 10, 20, 30으로 할당  
    println(  
        ""  
        array size : ${data1.size}  
        array data : ${data1[0]}, ${data1[1]}, ${data1.get(2)}  
        ""  
    )  
}
```

▶ 실행 결과

```
array size : 3  
array data : 10, 20, 30
```

2. 변수와 함수

■ 컬렉션 타입

■ List, Set, Map

- List: 순서가 있는 데이터 집합으로 데이터의 중복을 허용
- Set: 순서가 없으며 데이터의 중복을 허용하지 않습니다.
- Map: 키와 값으로 이루어진 데이터 집합으로 순서가 없으며 키의 중복은 허용하지 않음.
- Collection 타입의 클래스는 가변 클래스와 불변 클래스로 나뉨.
- 불변 클래스는 초기에 데이터를 대입하면 더 이상 변경할 수 없는 타입임.
- 가변 클래스는 초기값을 대입한 이후에도 데이터를 추가하거나 변경할 수 있음.

| 구분 | 타입 | 함수 | 특징 |
|------|-------------|-----------------|----|
| List | List | listOf() | 불변 |
| | MutableList | mutableListOf() | 가변 |
| Set | Set | setOf() | 불변 |
| | MutableSet | mutableSetOf() | 가변 |
| Map | Map | mapOf() | 불변 |
| | MutableMap | mutableMapOf() | 가변 |

2. 변수와 함수

■ 컬렉션 타입(List)

• 리스트 사용 예

```
fun main() {  
    var list = listOf<Int>(10, 20, 30)  
    println(  
        """  
list size : ${list.size}  
list data : ${list[0]}, ${list.get(1)}, ${list.get(2)}  
        """)  
}
```

▶ 실행 결과

```
list size : 3  
list data : 10, 20, 30
```

• 가변 리스트 사용 예

```
fun main() {  
    var mutableList = mutableListOf<Int>(10, 20, 30)  
    mutableList.add(3, 40)  
    mutableList.set(0, 50)  
    println(  
        """  
list size : ${mutableList.size}  
list data : ${mutableList[0]}, ${mutableList.get(1)},  
            ${mutableList.get(2)}, ${mutableList.get(3)}  
        """)  
}
```

▶ 실행 결과

```
list size : 4  
list data : 50, 20, 30, 40
```


2. 변수와 함수

■ 컬렉션 타입(map)

- Map 객체는 키와 값으로 이루어진 데이터의 집합
- Map 객체의 키와 값은 Pair 객체를 이용할 수도 있고 '키 to 값' 형태로 이용할 수도 있음

• 집합 사용 예

```
fun main() {  
    var map = mapOf<String, String>(Pair("one", "hello"), "two" to "world")  
    println(  
        """"  
        map size : ${map.size}  
        map data : ${map.get("one")}, ${map.get("two")}  
        """"  
    )  
}
```

▶ 실행 결과

```
map size : 2  
map data : hello, world
```

3. 조건문과 반복문

■ 조건문 if~else와 표현식

if-else 문

```
fun main() {  
    var data = 10  
    if (data > 0) {  
        println("data > 0")  
    } else {  
        println("data <= 0")  
    }  
}
```

▶ 실행 결과

data > 0

multi_if-else 문

```
fun main() {  
    var data = 10  
    if (data > 10) {  
        println("data > 10")  
    } else if (data > 0 && data <= 10) {  
        println("data > 0 && data <= 10")  
    } else {  
        println("data <= 0")  
    }  
}
```

▶ 실행 결과

data > 0 && data <= 10

3. 조건문과 반복문

- 코틀린에서 if~else는 표현식으로도 사용 가능
- 표현식이란 결과값을 반환하는 계산식을 말함.

```
fun main() {  
    var data = 10  
    val result = if (data > 0) {  
        println("data > 0")  
        true — 참일 때 반환값  
    } else {  
        println("data <= 0")  
        false — 거짓일 때 반환값  
    }  
    println(result)  
}
```

▶ 실행 결과

```
data > 0  
true
```

3. 조건문과 반복문

■ 조건문 when

정수타입 조건문

```
fun main() {  
    var data = 10  
    when (data) {  
        10 -> println("data is 10")  
        20 -> println("data is 20")  
        else -> {  
            println("data is not valid data")  
        }  
    }  
}
```

▶ 실행 결과

data is 10

문자열타입 조건문

```
fun main() {  
    var data = "hello"  
    when (data) {  
        "hello" -> println("data is hello")  
        "world" -> println("data is world")  
        else -> {  
            println("data is not valid data")  
        }  
    }  
}
```

▶ 실행 결과

data is hello

3. 조건문과 반복문

■ 조건문 when

- when 문에서는 조건을 데이터 타입, 범위 등으로 다양하게 명시할 수 있음
- is는 타입을 확인하는 연산자이며 in은 범위 지정 연산자임.

```
fun main() {  
    var data: Any = 10  
    when (data) {  
        is String -> println("data is String")  
        20, 30 -> println("data is 20 or 30")  
        in 1..10 -> println("data is 1..10")  
        else -> println("data is not valid")  
    }  
}
```

// data가 문자열 타입이면
// data가 20 또는 30이면
// data가 1~10의 값이면

▶ 실행 결과

data is 1..10

3. 조건문과 반복문

- 조건문 when

- when은 if 문과 마찬가지로 표현식으로도 사용 가능

```
fun main() {  
    var data = 10  
    val result = when {  
        data <= 0 -> "data is <= 0"  
        data > 100 -> "data is > 100"  
        else -> "data is valid"  
    }  
    println(result)  
}
```

3. 조건문과 반복문

■ 반복문 for

- for 문은 제어 변수값을 증감하면서 특정 조건이 참일 때까지 구문을 반복해서 실행
- or 문의 조건에는 주로 범위 연산자인 in을 사용
- for (i in 1..10) { ... } → 1부터 10까지 1씩 증가
- for (i in 1 until 10) { ... } → 1부터 9까지 1씩 증가(10은 미포함)
- for (i in 2..10 step 2) { ... } → 2부터 10까지 2씩 증가
- for (i in 10 downTo 1) { ... } → 10부터 1까지 1씩 감소

```
fun main() {  
    var sum: Int = 0  
    for (i in 1..10) {  
        sum += i  
    }  
    println(sum)  
}
```

▶ 실행 결과

55

3. 조건문과 반복문

■ 반복문 for

- 컬렉션 타입의 데이터 개수만큼 반복
- indices는 컬렉션 타입의 인덱스값을 의미
- 인덱스와 실제 데이터를 함께 가져오려면 withIndex() 함수를 이용

반복문에 컬렉션 타입 활용

```
fun main() {  
    var data = arrayOf<Int>(10, 20, 30)  
    for (i in data.indices) {  
        print(data[i])  
        if (i != data.size - 1) print(",")  
    }  
}
```

▶ 실행 결과

10,20,30

인덱스와 데이터를 가져오는 withIndex() 함수

```
fun main() {  
    var data = arrayOf<Int>(10, 20, 30)  
    for ((index, value) in data.withIndex()){  
        print(value)  
        if (index != data.size - 1) print(",")  
    }  
}
```

▶ 실행 결과

10,20,30

3. 조건문과 반복문

■ while문

- 조건이 참이면 중괄호 {}로 지정한 영역을 반복해서 실행

```
fun main(args: Array<String>) {  
    var x = 0  
    var sum1 = 0  
    while (x < 10) {  
        sum1 += ++x  
    }  
    println(sum1)  
}
```

▶ 실행 결과

55

4. 객체지향

■ 클래스와 생성자

- 클래스 선언
 - 클래스는 class 키워드로 선언
 - 클래스의 본문에 입력하는 내용이 없다면 {}를 생략
 - 클래스의 멤버는 생성자, 변수, 함수, 클래스로 구성
 - 생성자는 constructor라는 키워드로 선언하는 함수
- 객체 생성 및 멤버접근
 - 객체를 생성해 사용하며 객체로 클래스의 멤버에 접근
 - 객체를 생성할 때 new 키워드를 사용하지 않는다.

클래스의 멤버

```
class User {  
    var name = "kkang"  
    constructor(name: String) {  
        this.name = name  
    }  
    fun someFun() {  
        println("name : $name")  
    }  
    class SomeClass { }  
}
```

객체 생성과 멤버 접근

```
val user = User("kim")  
user.someFun()
```

4. 객체지향

■ 클래스와 생성자 : 주 생성자

- 주 생성자
 - 생성자를 주 생성자와 보조 생성자로 구분
 - 주 생성자는 constructor 키워드로 클래스 선언부에 선언
 - 주 생성자 선언은 필수는 아니며 한 클래스에 하나만 가능
 - constructor 키워드는 생략할 수 있다.

주생성자 선언

```
class User constructor() {  
}
```

constructor 생략 예

```
class User() {  
}
```

매개변수가 없는 주생성자 자동 선언

```
class User {  
}
```

4. 객체지향

■ 클래스와 생성자 : 주생성자

- 주 생성자의 본문 — init 영역
 - init 키워드를 이용해 주 생성자의 본문을 구현
 - init 키워드로 지정한 영역은 객체를 생성할 때 자동으로 실행
- 생성자의 매개변수를 클래스의 멤버 변수로 선언하는 방법
 - 생성자의 매개변수는 기본적으로 생성자에서만 사용할 수 있는 지역 변수

생성자의 매개변수를 init 영역에서 사용 예

```
class User(name: String, count: Int) {  
    init {  
        println("name : $name, count : $count")    // 성공!  
    }  
    fun someFun() {  
        println("name : $name, count : $count")    // 오류!  
    }  
}
```

init 키워드로 주 생성자의 본문 지정

```
class User(name: String, count: Int) {  
    init {  
        println("i am init...")  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
}
```

▶ 실행 결과

i am init...

4. 객체지향

■ 클래스와 생성자 : 주 생성자

- 매개변수를 var나 val 키워드로 선언하면 클래스의 멤버 변수

생성자의 매개변수를 클래스의 멤버 변수로 선언하는 방법

```
class User(val name: String, val count: Int) {  
    fun someFun() {  
        println("name : $name, count : $count")    // 성공!  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
    user.someFun()  
}
```

▶ 실행 결과

name : kkang, count : 10

4. 객체지향

■ 클래스와 생성자 : 보조 생성자

■ 보조 생성자

- 보조 생성자는 클래스의 본문에 constructor 키워드로 선언하는 함수
- 여러 개를 추가할 수 있다.

보조 생성자 선언

```
class User {  
    constructor(name: String) {  
        println("constructor(name: String) call...")  
    }  
    constructor(name: String, count: Int) {  
        println("constructor(name: String, count: Int) call...")  
    }  
}  
  
fun main() {  
    val user1 = User("kkang")  
    val user2 = User("kkang", 10)  
}
```

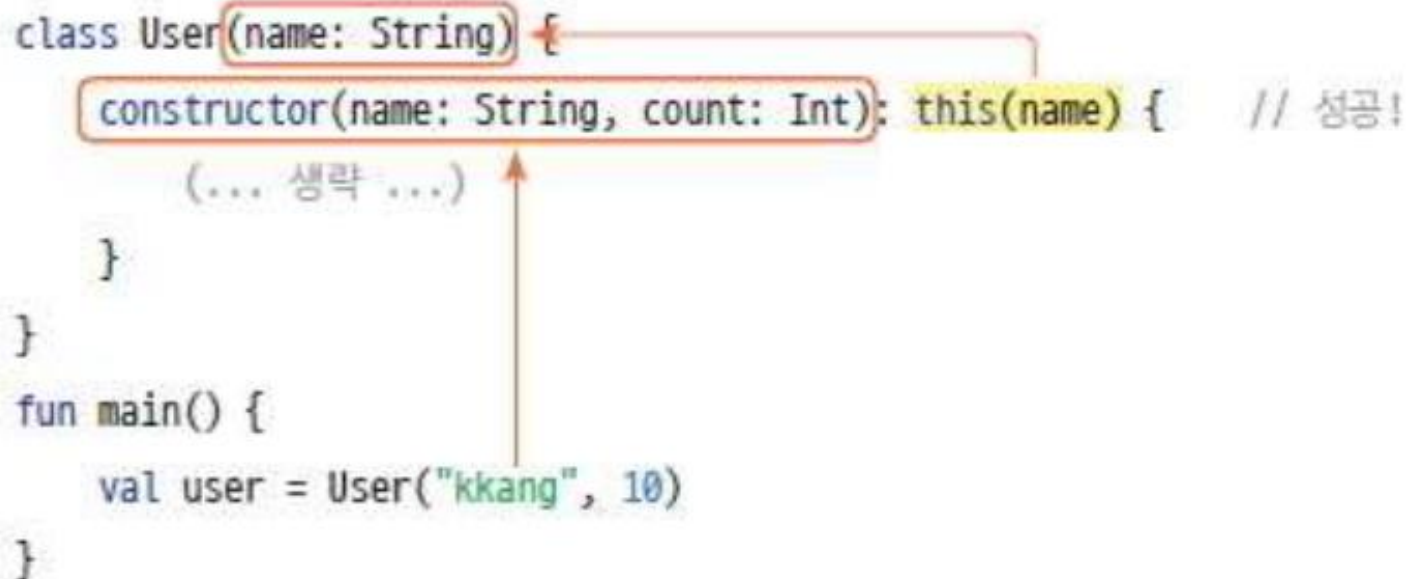
4. 객체지향

■ 클래스와 생성자 : 보조 생성자

- 보조 생성자에 주 생성자 연결
 - 보조 생성자로 객체를 생성할 때 클래스 내에 주 생성자가 있다면 this() 구문을 이용해 주 생성자를 호출해야 함.

보조 생성자에서 주생성자 호출

```
class User(name: String) {  
    constructor(name: String, count: Int): this(name) { // 성공!  
        (... 생략 ...)  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
}
```

A diagram with a red box around the primary constructor signature 'class User(name: String) {' and another red box around the secondary constructor signature 'constructor(name: String, count: Int): this(name) {'. A red arrow points from the 'this(name)' part of the secondary constructor to the primary constructor signature. Another red arrow points from the 'this(name)' part to the 'name' parameter in the secondary constructor's parameter list. A yellow highlight is under 'this(name)'. A comment '// 성공!' is at the end of the secondary constructor line. The text '(... 생략 ...)' is in Korean and means 'omitted'.

4. 객체지향

■ 클래스와 생성자 : 보조 생성자

보조생성자가 여럿일 때 생성자 연결

```
class User(name: String) {  
    constructor(name: String, count: Int): this(name) {  
        (... 생략 ...)  
    }  
    constructor(name: String, count: Int, email: String): this(name, count) {  
        (... 생략 ...)  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10, "a@a.com")  
}
```


4. 객체지향

■ 상속 : 상속과 생성자

■ 상속과 생성자

- 코틀린에서 어떤 클래스를 상속받으려면 선언부에 콜론(:)과 함께 상속받을 클래스 이름을 입력
- 코틀린의 클래스는 기본적으로 다른 클래스가 상속할 수 없음.
- 다른 클래스에서 상속할 수 있게 선언하려면 open 키워드를 사용
- 상위 클래스를 상속받은 하위 클래스의 생성자에서는 상위 클래스의 생성자를 호출해야 함.

매개변수가 있는 상위 클래스의 생성자 호출

```
open class Super(name: String) {  
}  
class Sub(name: String): Super(name) {  
}
```

하위 클래스에 보조 생성자만 있는 경우 상위 클래스의 생성자 호출

```
open class Super(name: String) {  
}  
class Sub: Super {  
    constructor(name: String): super(name) {  
    }  
}
```

4. 객체지향

■ 상속 : 오버라이딩-재정의

- 상속이 주는 최고의 이점은 상위 클래스에 정의된 멤버(변수, 함수)를 하위 클래스에서 자신의 멤버처럼 사용할 수 있다는 것
- 오버라이딩 : 상위 클래스에 선언된 변수나 함수를 같은 이름으로 하위 클래스에서 다시 선언하는 것

```
open class Super {  
    var superData = 10  
    fun superFun() {  
        println("i am superFun : $superData")  
    }  
}  
  
class Sub: Super()  
fun main() {  
    val obj = Sub()  
    obj.superData = 20  
    obj.superFun()  
}
```

```
open class Super {  
    open var someData = 10  
    open fun someFun() {  
        println("i am super class function : $someData")  
    }  
}  
  
class Sub: Super() {  
    override var someData = 20  
    override fun someFun() {  
        println("i am sub class function : $someData")  
    }  
}  
  
fun main() {  
    val obj = Sub()  
    obj.someFun()  
}
```

▶ 실행 결과

i am sub class function : 20

4. 객체지향

■ 상속 : 접근 제한자

- 접근 제한자란 클래스의 멤버를 외부의 어느 범위까지 이용하게 할 것인지를 결정하는 키워드

| 접근 제한자 | 최상위에서 이용 | 클래스 멤버에서 이용 |
|-----------|--------------|---------------------|
| public | 모든 파일에서 가능 | 모든 클래스에서 가능 |
| internal | 같은 모듈 내에서 가능 | 같은 모듈 내에서 가능 |
| protected | 사용 불가 | 상속 관계의 하위 클래스에서만 가능 |
| private | 파일 내부에서만 이용 | 클래스 내부에서만 이용 |

4. 객체지향

■ 상속 : 접근 제한자 사용 예

```
open class Super {  
    var publicData = 10  
    protected var protectedData = 20  
    private var privateData = 30  
}  
  
class Sub: Super() {  
    fun subFun() {  
        publicData++    // 성공!  
        protectedData++ // 성공!  
        privateData++   // 오류!  
    }  
}
```

```
fun main() {  
    val obj = Super()  
    obj.publicData++    // 성공!  
    obj.protectedData++ // 오류!  
    obj.privateData++   // 오류!  
}
```

4. 객체지향

- 코틀린 클래스의 종류 : 데이터 클래스

- 데이터 클래스, 오브젝트 클래스, 컴패니언 클래스

- 데이터 클래스

- 데이터 클래스는 data 키워드로 선언
- 데이터 클래스는 VO 클래스를 편리하게 이용할 수 있는 방법 제공
- 데이터 클래스 선언 예

```
class NonDataClass(val name: String, val email: String, val age: Int)
```

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

4. 객체지향

■ 코틀린 클래스의 종류 : 데이터 클래스

- 객체의 데이터를 비교하는 equals() 함수

데이터 클래스 객체 생성 예

```
fun main() {  
    val non1 = NonDataClass("kkang", "a@a.com", 10)  
    val non2 = NonDataClass("kkang", "a@a.com", 10)  
  
    val data1 = DataClass("kkang", "a@a.com", 10)  
    val data2 = DataClass("kkang", "a@a.com", 10)  
}
```

객체의 데이터를 비교하는 equals()함수

```
println("non data class equals : ${non1.equals(non2)}")  
println("data class equals : ${data1.equals(data2)}")
```

▶ 실행 결과

```
non data class equals : false  
data class equals : true
```

4. 객체지향

■ 코틀린 클래스의 종류 : 데이터 클래스

- equals() 함수는 주 생성자에 선언한 멤버 변수의 데이터만 비교 대상으로 함

데이터 클래스의 equals() 함수

```
data class DataClass(val name: String, val email: String, val age: Int) {  
    lateinit var address: String  
    constructor(name: String, email: String, age: Int, address: String):  
        this(name, email, age) {  
            this.address = address  
        }  
}  
  
fun main() {  
    val obj1 = DataClass("kkang", "a@a.com", 10, "seoul")  
    val obj2 = DataClass("kkang", "a@a.com", 10, "busan")  
    println("obj1.equals(obj2) : ${obj1.equals(obj2)}")  
}
```

▶ 실행 결과

obj1.equals(obj2) : true

4. 객체지향

■ 코틀린 클래스의 종류 : 데이터 클래스

- 객체의 데이터를 반환하는 toString() 함수
 - 데이터 클래스를 사용하면서 객체가 가지는 값을 확인해야 할 때 이용

```
fun main() {  
    class NonDataClass(val name: String, val email: String, val age: Int)  
    data class DataClass(val name: String, val email: String, val age: Int)  
    val non = NonDataClass("kkang", "a@a.com", 10)  
    val data = DataClass("kkang", "a@a.com", 10)  
    println("non data class toString : ${non.toString()}")  
    println("data class toString : ${data.toString()}")  
}
```

▶ 실행 결과

```
non data class toString : com.example.test4.ch2.Test2Kt$main$NonDataClass@61bbe9ba  
data class toString : DataClass(name=kkang, email=a@a.com, age=10)
```


4. 객체지향

■ 코틀린 클래스의 종류 : 오브젝트 클래스

- 오브젝트 클래스는 익명 클래스를 만들 목적으로 사용
- 선언과 동시에 객체를 생성한다는 의미에서 object라는 키워드를 사용
- 오브젝트 클래스의 타입은 object 뒤에 콜론(:)을 입력하고 그 뒤에 클래스의 상위 또는 인터페이스를 입력

오브젝트 클래스 사용 예

```
val obj = object {  
    var data = 10  
    fun some() {  
        println("data : $data")  
    }  
}  
  
fun main() {  
    obj.data = 20    // 오류!  
    obj.some()      // 오류!  
}
```

타입을 지정한 오브젝트 클래스

```
open class Super {  
    open var data = 10  
    open fun some() {  
        println("i am super some() : $data")  
    }  
}  
  
val obj = object: Super() {  
    override var data = 20  
    override fun some() {  
        println("i am object some() : $data")  
    }  
}
```

```
fun main() {  
    obj.data = 30    // 성공!  
    obj.some()      // 성공!  
}
```

▶ 실행 결과

```
i am object some() : 30
```

4. 객체지향

■ 코틀린 클래스의 종류 : 컴패니언 클래스

- 컴패니언 클래스는 멤버 변수나 함수를 클래스 이름으로 접근하고자 할 때 사용
- companion이라는 키워드로 선언

컴패니언 클래스 멤버 접근

```
class MyClass {  
    companion object {  
        var data = 10  
        fun some() {  
            println(data)  
        }  
    }  
}
```

```
fun main() {  
    MyClass.data = 20    // 성공!  
    MyClass.some()      // 성공!  
}
```

5. 람다 함수와 고차함수

■ 람다 함수

- 람다 함수는 익명 함수 정의 기법
- 람다 함수 선언과 호출
 - 람다 함수는 fun 키워드를 이용하지 않으며 함수 이름이 없음
 - 람다 함수는 {}로 표현.
 - {} 안에 화살표(->)가 있으며 화살표 왼쪽은 매개변수, 오른쪽은 함수 본문
 - 함수의 반환값은 함수 본문의 마지막 표현식임.

• 함수 선언 형식

```
fun 함수명(매개변수) { 함수 본문 }
```

• 람다 함수 선언 형식

```
{ 매개변수 -> 함수 본문 }
```

• 일반 함수 선언

```
fun sum(no1: Int, no2: Int): Int {  
    return no1 + no2  
}
```

• 람다 함수 선언

```
val sum = {no1: Int, no2: Int -> no1 + no2}
```

5. 람다 함수와 고차함수

■ 람다 함수

- 매개변수 없는 람다 함수
 - 화살표 왼쪽이 매개변수를 정의하는 부분인데 매개변수가 없을 경우 비워 두거나 화살표까지 생략 가능

• 매개변수가 없는 람다 함수

```
{-> println("function call")}
```

• 화살표를 생략한 람다 함수

```
{println("function call")}
```

■ 매개변수가 1개인 람다 함수

- 람다 함수의 매개변수가 1개일 때는 매개변수를 선언하지 않아도 it 키워드로 매개변수를 이용할 수 있음

• 매개변수가 1개인 람다 함수

```
fun main() {  
    val some = {no: Int -> println(no)}  
    some(10)  
}
```

▶ 실행 결과

10

• 매개변수가 1개인 람다 함수에 it 키워드 사용

```
fun main() {  
    val some: (Int) -> Unit = {println(it)}  
    some(10)  
}
```

▶ 실행 결과

10

5. 람다 함수와 고차함수

■ 람다 함수

- 람다 함수의 반환
 - 람다 함수에서는 return 문을 사용할 수 없습니다.
 - 람다 함수의 반환값은 본문에서 마지막 줄의 실행 결과

• 람다 함수에서 return 문 사용 오류

```
val some = {no1: Int, no2: Int -> return no1 * no2} // 오류!
```

• 람다 함수의 반환문

```
fun main() {  
    val some = {no1: Int, no2: Int ->  
        println("in lambda function")  
        no1 * no2  
    }  
    println("result : ${some(10, 20)}")  
}
```

▶ 실행 결과

```
in lambda function  
result : 200
```

5. 람다 함수와 고차함수

■ 함수 타입과 고차 함수

- 함수 타입 선언
 - 함수 타입이란 함수를 선언할 때 나타내는 매개변수와 반환 타입을 의미

• 일반 함수 선언

```
fun some(no1: Int, no2: Int): Int {  
    return no1 + no2  
}
```

• 함수 타입을 이용해 함수를 변수에 대입

```
val some: (Int, Int) -> Int = { no1: Int, no2: Int -> no1 + no2 }
```

함수 타입 함수 내용

5. 람다 함수와 고차함수

■ 함수 타입과 고차 함수

- 타입 별칭 — typealias
 - typealias는 타입의 별칭을 선언하는 키워드

타입 별칭 선언과 사용

```
typealias MyInt = Int
fun main() {
    val data1: Int = 10
    val data2: MyInt = 10
}
```

함수 타입 별칭

```
typealias MyFunType = (Int, Int) -> Boolean

fun main() {
    val someFun: MyFunType = {no1: Int, no2: Int ->
        no1 > no2
    }
    println(someFun(10, 20))
    println(someFun(20, 10))
}
```

5. 람다 함수와 고차함수

■ 함수 타입과 고차 함수

- 매개변수 타입 생략
 - 매개변수의 타입을 유추할 수 있다면 타입 선언을 생략할 수 있음

매개변수 타입을 생략한 함수

```
typealias MyFunType = (Int, Int) -> Boolean  
val someFun: MyFunType = {no1, no2 ->  
    no1 > no2  
}
```

매개변수 타입 선언을 생략한 함수

```
val someFun: (Int, Int) -> Boolean = {no1, no2 ->  
    no1 > no2  
}
```

변수 선언 시 타입을 생략

```
val someFun = {no1: Int, no2: Int ->  
    no1 > no2  
}
```


5. 람다 함수와 고차함수

■ 함수 타입과 고차 함수

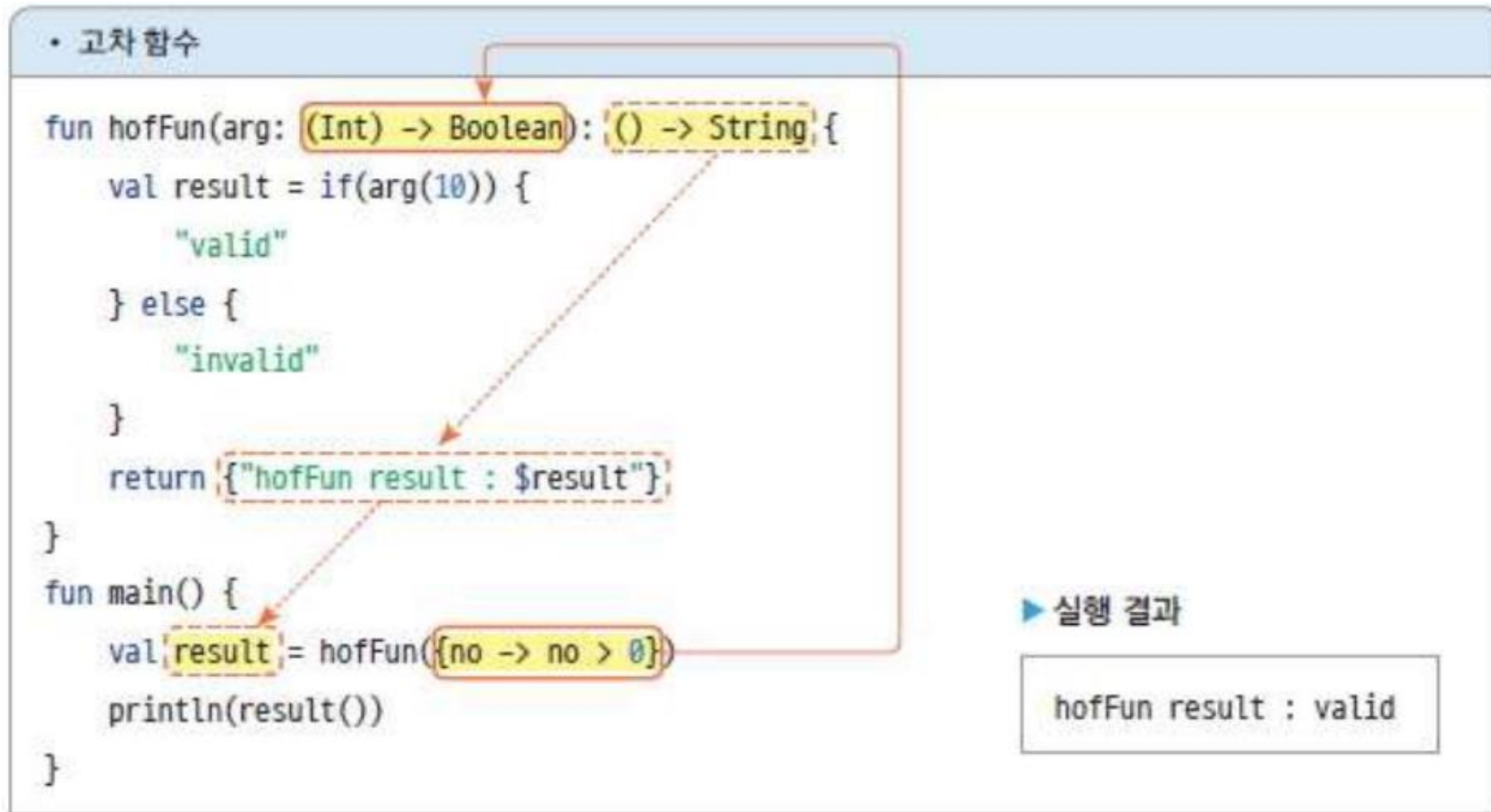
- 고차함수 : 고차 함수란 함수를 매개변수로 전달받거나 반환하는 함수를 의미

• 고차 함수

```
fun hofFun(arg: (Int) -> Boolean): () -> String {  
    val result = if(arg(10)) {  
        "valid"  
    } else {  
        "invalid"  
    }  
    return {"hofFun result : $result"}  
}  
  
fun main() {  
    val result = hofFun({no -> no > 0})  
    println(result())  
}
```

▶ 실행 결과

hofFun result : valid



6. 널 안정성

■ 널 안전성이란?

- 널(null)이란 객체가 선언되었지만 초기화되지 않은 상태를 의미
- 널인 상태의 객체를 이용하면 널 포인트 예외(NullPointerException)가 발생
- 널 안정성이란 널 포인트 예외가 발생하지 않도록 코드를 작성하는 것

• 널 안전성을 개발자가 고려한 코드

```
fun main() {  
    var data: String? = null  
    val length = if (data == null) {  
        0  
    } else {  
        data.length  
    }  
    println("data length : $length")  
}
```

▶ 실행 결과

data length : 0

6. 널 안정성

■ 널 안정성이란?

- 프로그래밍 언어가 널 안전성을 지원한다는 것은 객체가 널인 상황에서 널 포인터 예외가 발생하지 않도록 연산자를 비롯해 여러 기법을 제공한다는 의미

• 코틀린이 제공하는 널 안전성 연산자를 이용한 코드

```
fun main() {  
    var data: String? = null  
    println("data length : ${data?.length ?: 0}")  
}
```

▶ 실행 결과

data length : 0

6. 널 안정성

■ 널 안전성 연산자

- 널 허용 — ? 연산자 : 코틀린에서는 변수 타입을 널 허용과 널 불허로 구분
 - 널 허용으로 선언한 변수의 멤버에 접근할 때는 반드시 ?. 연산자를 이용해야 함.

• 널 포인트 예외 오류

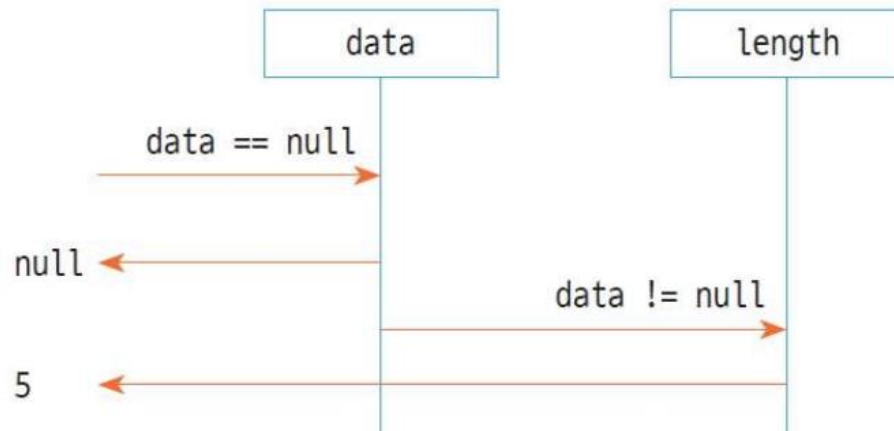
```
var data: String? = "kkang"  
var length = data.length // 오류!
```

• 널 안전성 호출 연산자 사용

```
var data: String? = "kkang"  
var length = data?.length // 성공!
```

• 널 허용과 널 불허

```
var data1: String = "kkang"  
data1 = null // 오류!  
  
var data2: String? = "kkang"  
data2 = null // 성공!
```



6. 널 안정성

■ 엘비스 — ?: 연산자

- 널일 때 대입해야 하는 값이나 실행해야 하는 구문이 있는 경우 이용

• 엘비스 연산자 사용

```
fun main() {  
    var data: String? = "kkang"  
    println("data = $data : ${data?.length ?: -1}")  
    data = null  
    println("data = $data : ${data?.length ?: -1}")  
}
```

▶ 실행 결과

```
data = kkang : 5  
data = null : -1
```

6. 널 안정성

■ 예외 발생 — !! 연산자

- 객체가 널일 때 예외를 일으키는 연산자

• 예외 발생 연산자

```
fun some(data: String?): Int {  
    return data!!.length  
}  
  
fun main() {  
    println(some("kkang"))  
    println(some(null))  
}
```

▶ 실행 결과

```
5  
Exception in thread "main" java.lang.NullPointerException
```