

SHOUT · OUR · PASSION · TOGETHER

ODo IT SOPT O

# Kotlin 문법

SHOUT OUR PASSION TOGETHER  
SOPT

# 01 Kotlin 연습 방법

# 02 Kotlin 문법

○

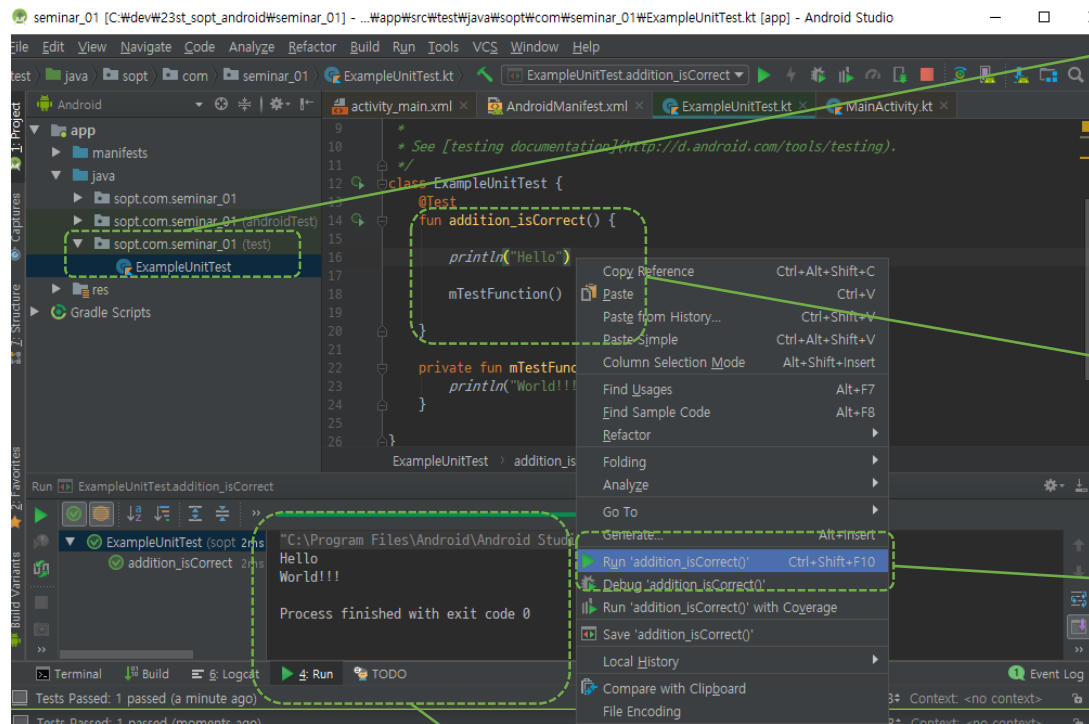
01

○

# Kotlin 연습 방법

## 01 Kotlin 연습 방법

### 코틀린 문법을 테스트해보기 위해서는?



1. ExampleUnitTest 여기서!  
(폴더 명 옆 “test”인 폴더 입니다!)

2. 이곳에서 코딩해주세요!

3. 실행은  
마우스 오른쪽 클릭 후  
Run 'addition\_isCorrect()' 클릭!

4. 결과 확인은 이곳에!



02



# Kotlin 문법

**모든 문법을 다룰 수 없으므로  
세미나때 자주 보일 문법 위주로 작성하였습니다!**

너무 기본적인 프로그래밍 문법은 제외하겠습니다!

# 상수 / 변수 정의

```
//val
val finalValue : String = "나를 바꿀 수 없어!"
finalValue = "안바뀌네 ㅋㅋ"

//var
var mutableValue : String = "나를 바꿀 수 있어!"
mutableValue = "바뀐다!!!"
```

### val (상수)

1. 초기화 되면 값을 바꿀 수 없다. (Only Read)
2. 값이 바뀌지 않을 때 사용

### var (변수)

1. 값을 바꿀 수 있다. (Read + Write)
2. 값이 바뀌지 않을 때 사용

## 상수 / 변수 타입

같다

```
//문자열
var s : String = "문자"

//정수
var i : Int = 365 // 32bit
var l : Long = 123L // 64bit

//실수
var f : Float = 0.1f // 32bit
var d : Double = 0.3 // 64bit

//논리
var b : Boolean = true
b = false
```

```
//문자열
var s = "문자"

//정수
var i = 365
var l = 123L

//실수
var f = 0.1f
var d = 0.3


//논리
var b = true
```

Kotlin에서는 타입을 지정해주지 않고 초기화 시,  
초기화 값에 따라 타입이 자동으로 지정됩니다



## 상수 / 변수 더 알아보기

```
var imDouble : Double = 180.33 // 실수 형 변수  
var imInt : Int // 정수 형 변수  
imInt = imDouble // 정수 형에 실수 형 대입 불가! error  
imInt = imDouble.toInt() // 실수 형을 .toInt() 메소드를 통해 정수
```



toInt()	Int
toString()	String
toByte()	Byte
toChar()	Char
toDouble()	Double
toFloat()	Float
toLong()	Long
toShort()	Short

Kotlin에서 지원하는 인코딩 메소드들

## Class – 주 생성자

- 파라미터(name, age, partName)를 받음과 동시에 프로퍼티 생성 가능하다.

(즉, java처럼 this.name = name 같은 작업을 하지 않아도 인스턴스 상수/변수로 사용가능 하다.)

```
class Person(val name : String, val age : Int?, var partName: String) {  
    fun printInfo(){  
        println("이름: $name, 나이: $age, 파트: $partName")  
    }  
}
```

- 파라미터(name, age, partName)를 초기화도 가능하며, 기본적으로 접근 지정자는 public

(접근 지정자는 본인 사용 목적에 따라 자율 지정)

```
class Person(val name : String = "홍길동", val age : Int?, var partName: String = "Android") {  
    fun printInfo(){  
        println("이름: $name, 나이: $age, 파트: $partName")  
    }  
}
```

# Class – 인스턴스 생성 방법

- 인스턴스 생성 시 java처럼 new를 쓰지 않는다.
- 문장 끝에 세미콜론(;)을 쓰지 않는다. (써도 무방하나 의미 없다.)
- 프로퍼티(name, partName)들은 다음과 같이 참조 가능

```
val p1 : Person = Person( name: "남윤환", age: 28, partName: "Android")  
println("p1의 이름은 ${p1.name}")  
println("p1의 파트는 " + p1.partName)  
p1.printInfo()
```

## Class – 그 외 생성자 (그냥 참조만 하세요!)

```
class Person1{
    val name : String
    private val age : Int?
    var partName : String
    constructor(name: String, age: Int, partName: String){
        this.name = name
        this.age = age
        this.partName = partName
    }
}

class Person2(name : String, age: Int, partName: String){
    val name : String
    private val age : Int?
    var partName : String
    init {
        this.name = name
        this.age = age
        this.partName = partName
    }
}

class Person3(name : String, age: Int, partName: String){
    val name : String = name
    private val age : Int? = age
    var partName : String = partName
}
```

### - constructor(){}

부 생성자 형태, 부 생성자를 통해 파라미터 종류를 다르게 하여 여러 개의 생성자를 만들 수 있다.

### - init{}

객체가 생성될 때 가장 먼저 실행될 것을 구현하는 곳.

세미나때 딱히 사용하지 않을 문법입니다.

하지만 구글링을 통해 코드 분석 시 이러한 문법이 보일 수 있으니 참조하라는 의미에서 넣어봤습니다.

## Array<> - 배열

- 다음과 같은 방식으로 배열을 만들며 여기서 참조할 것은,
- 제네릭 지정 시, kotlin에서 Any라는 것은 java에서 object과 비슷하다는 점과,
- 타입이 다른 값들을 모두 넣을 수 있다는 점입니다.

```
val imStringArray : Array<String> = arrayOf("안드로이드", "IOS", "서버", "디자인", "기획")  
val imIntArray : Array<Int> = arrayOf(1,2,3,4,5,6,7,8,9)  
val imAnyArray : Array<Any> = arrayOf("안드로이드", 3, 0.4f, 33.33, true)
```

- 앞서 말했 듯, kotlin에서는 초기화 값이 주어진다면 타입을 굳이 써주지 않아도 자동 매칭됩니다.

```
val imAnyArray = arrayOf("안드로이드", 3, 0.4f, 33.33, true)
```

Collection은 주로 ArrayList를 사용할 예정입니다!

# Collection – ArrayList

```
// ArrayList 생성과 동시에 초기화
val myArrayList : ArrayList<String> = ArrayList<String>()
//데이터 넣기
myArrayList.add("android")
myArrayList.add("ios")
myArrayList.add("server")
myArrayList.add("design")
myArrayList.add("기획")

//3번째 인덱스 참조
println(myArrayList[3])// design

//ArrayList에 들어있는 데이터들 처음부터 쪽~ 참조,
//암시적인 인자(ArrayList에 든 값들)에 대해 인자 명을 따로 정해주지 않으면 기본적으로 it
myArrayList.forEach { it: String
    print("$it ")
} // android ios server design 기획

//인자를 str이라 이름으로 지정해서 사용가능, but 그냥 인자가 하나면 it을 쓰자!
myArrayList.forEach { str ->
    print("$str ")
} // android ios server design 기획

//ArrayList에 들어있는 데이터 필터로 거르고, 정렬 후 쪽~ 참조
myArrayList.filter { it.length > 4 } // 문자열 길이가 5이상인 것들만 남김
    .sortedByDescending { it } // 내림차순으로 정렬(여기선 z->a 순으로 정렬되겠죠!?)
    .forEach { print("$it ") } // 위 조건에 따라 데이터 쪽~ 참조 server design android
```

- 언제 쓰는지?  
서버에서 통신을 통해 데이터를 받아올 때 등..
- forEach는 kotlin에서 제공하는 기능으로, list를 처음부터 쪽~ 훑어주는 기능을 합니다! 자주 애용해요.
- filter는 가끔 쓰이므로 세미나때 차차 다시 설명하도록 하겠습니다.

## 조건문 - if

```
val number : Int = 365
//조건문
if (number < 0) { //false 이므로 넘어감
    print("음수")
} else if (number < 100){ //false 이므로 넘어감
    print("0 <= number < 100")
} else if (number < 1000) { //true 이므로 해당 블록 코드 수행!
    print("100 <= number < 1000")
} else {
    print("1000이상")
}
// 출력 결과 : 100 <= number < 1000
```

# 조건문 - when

- java에서 switch가 있다면 kotlin에는 더욱 기능이 확장된 강력한 when이 있습니다.

```
val target : Int = 100
when (target) {
    0, 1 -> println("0과 1중 하나다")
    in 0..10 -> println("${target}는 0~10사이에 있다.")
    !in 100..1000 -> println("${target}는 100~1000사이에 없다.") // !: ~가 아니면
    100 -> println("100이다.")
    else -> println("위 조건에 만족하는게 없다.")
}
```

- 가장 큰 차이점이 있다면, switch와 달리 순차적으로 조건을 비교하다가 첫번째로 조건이 맞는 곳에서 조건에 맞는 명령을 수행하고 when 블록 밖으로 나간다는 점



# Null 허용

- kotlin은 애초부터 null을 허용하지 않습니다!
- 그러므로 null이 발생할 수 있는 곳이 있다면 타입 뒤 **?**를 붙여서 null을 허용해주도록 합니다.

```
val str : String? = null
if (str.isNullOrEmpty()) {
    println("it is null")
} else {
    println(str!!.length)
}
```

## Null 허용

- 만약 ?가 붙지 않은 객체에 null이 들어오는 경우, java에서는 치명적인 널 포인트 에러가 발생하여 앱이 터집니다!
- 하지만 kotlin에서는 null을 피할 다양한 방법이 있습니다. 아래의 경우는 null임에도 불구하고 터지지 않고 실행되는 kotlin의 null처리 중 하나이고, 세미나를 통해 null을 피하는 방법을 차차 학습해 보도록 하겠습니다.

```
11  */
12  class ExampleUnitTest {
13      @Test
14      fun addition_isCorrect() {
15          var nullableString : String? = null
16          println(nullableString?.length)
17      }
18  }
19
20
21
22
```

ExampleUnitTest > addition\_isCorrect()

1 test passed

"C:\Program Files\Android\Android Studio\jre\bin\java" ...  
null

우리의 똑똑한 안드로이드 스튜디오는  
null을 피할 수 있도록 컴파일 시키기도 전에  
빨간 줄로 컴파일을 막아버립니다!  
(코딩하면서 null관련해서 스스로 찾아내고 알려줘요!)

하지만 서버 통신할 때,  
null 에러로 앱이 터지는 경우가 많은데 이것은  
세미나에서 저의 노하우로 알려드리겠습니다~!

## 반복문 – for/while

```
for (i in 0..10){  
    print("$i ")  
} //0 1 2 3 4 5 6 7 8 9 10  
  
for (i in 0..10 step 2){  
    print("$i ")  
} // 0 2 4 6 8 10  
  
for (i in 10 downTo 0 step 3){  
    print("$i ")  
} // 10 7 4 1
```

- do while은 일단 블록 명령 수행 후 조건을 맞춰봅니다!

```
var y = 0  
do {  
    y--  
    print(y)  
} while (y > 0)  
//-1
```

- while은 선 조건을 맞춰보죠!

```
var y = 0  
while (y > 0){  
    y--  
    print(y)  
}
```

# 반복문 – for with ArrayList

```
val numberArrayList : ArrayList<Int> = arrayListOf(1,2,3,4,5,6,7,8,9,10)
var sum : Int = 0
for (value in numberArrayList){
    println("$value 더하는 중...")
    sum += value // sum = sum + value
}
println("결과는? $sum")
```

- 결과 차이를 알고 싶다면 직접 코딩해보기!

```
val numberArrayList : ArrayList<Int> = arrayListOf(1,2,3,4,5,6,7,8,9,10)
var sum : Int = 0
for ((index, value) in numberArrayList.withIndex()){
    println("[ $index ] $value 더하는 중...")
    sum += value // sum = sum + value
}
println("결과는? $sum")
```

## 함수 - fun

- **public fun** 함수이름(파라미터) : 리턴 타입 { 로직 }
- 접근지정자를 명시하지 않는다면 기본적으로 public 입니다.
- java에서 void의 역할을 kotlin에서는 **Unit**이 하는데, 굳이 리턴 타입이 없다면 명시해주지 않아도 됩니다!

```
fun printString() : Unit {  
    println("안녕!!!")  
}
```

```
fun printString() {  
    println("안녕!!!")  
}
```

```
fun addString(first : String, second : String) : String {  
    return first + second  
}
```

```
printString() //안녕!!!
```

```
val addStr = addString("Hello ", "world!!!")  
println(addStr) //Hello world!!!
```

# data class

- kotlin의 매우 큰 장점인 data class!
- java와 다르게 getter, setter를 만들지 않아도 되기 때문에 한 줄로 끝낼 수 있습니다.
- (자동으로 메소드 equals(), toString(), copy() 등을 생성해줍니다!)
- 서버 통신할 때 데이터를 받아올 틀을 만들어야 하는데, 시간을 엄청나게 단축 할 수 있어요~!

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

# 그 외 Kotlin만의 문법 - let

- null인지 아닌지 모를 때 사용하면 유용한 문법, 즉 null처리 시 유용한 문법
- 아래 코드처럼 사용하며, 예시의 nullableString의 경우 null이므로 let안의 블록 로직은 수행되지 않습니다!

```
var nullableString : String? = null
nullableString?.let { it: String
    println("널이 아닙니다.") //널이 아닐 경우 실행되는 블록입니다.
}
```

# 그 외 Kotlin만의 문법 - run

- 이미 생성된 객체에 대해 재 접근 시 불필요한 코드를 줄여주는 문법
- 아래 코드처럼 사용하며, “**dataList.**”이라는 **반복적인 코드**를 run문법을 통해 줄일 수 있습니다.
- 이와 비슷한 문법으로 “apply”가 있는데, 이것은 객체를 생성할 때 동시에 값을 초기화 시킬 시 사용합니다.  
(세미나를 통해 설명 추가하겠습니다.)

```
dataList.add(Person( name: "홍길금", age: 21))
dataList.add(Person( name: "홍길은", age: 22))
dataList.add(Person( name: "홍길동", age: 23))
dataList.forEach { it: Person
    println(it.name)
}
```

```
dataList.run { this: ArrayList<Person>
    add(Person( name: "홍길금", age: 21))
    add(Person( name: "홍길은", age: 22))
    add(Person( name: "홍길동", age: 23))
    forEach { it: Person
        println(it.name)
    }
}
```



### 그 외 Kotlin만의 문법 - lateinit

- “**변수(var)**”를 나중에 초기화 시킬 경우 사용하는 문법
- 아래 코드처럼 사용하며, 추후 꼭! 초기화 시켜줘야 합니다! 주로 클래스 블록 내 인스턴스 변수를 만들 때 사용하며 앞으로 세미나때 자주 보일 예정이므로 추후 설명을 추가하겠습니다!

```
lateinit var lateinitWord : String
```

### 그 외 Kotlin만의 문법 - lazy

- “**상수(val)**”를 호출 할 때 초기화 되는 문법
- 아래 코드처럼 사용하며, 사용하지 않으면 불필요한 인스턴스를 생성하지 않습니다.
- 이 문법 또 한 앞으로 세미나 때 자주 보일 예정이므로 추후 설명을 추가하겠습니다!

```
private val dataList : ArrayList<Person> by lazy {  
    ArrayList<Person>()  
}
```

S H O U T · O U R · P A S S I O N · T O G E T H E R

ODo IT SOPT O

THANK U

PPT 디자인 한승미 세미나 자료 남윤환

SHOUT OUR PASSION TOGETHER  
SOPT