inno SOPT

# JAVA로 알아보는 코틀린 [기본편]

# Contents

22기 SOPT

Java로 알아보는 코틀린[기본편]

SHOUT OUR PASSION TOGETHER
SOPT

# 01 기본 변수 타입

```java
int android1 = 0;
double android2 = 1.0;
char android3 = 'a';
String android4 = "hello";
boolean android5 = true;
```

```kotlin
//별도의 타입 선언 없이 값 할당 가능
var android1 = 0
var android2 = 1.0
var android3 = 'a'
var android4 = "hello"
var android5 = true
```

**변수 선언 할 때의 키워드는** 'var'!!(variation)

```
android1 is Int
android2 is Double
android3 is Char
android4 is String
android5 is Boolean
```

**별도의 타입 지정을 하지 않았을 시에는 각 데이터의 기본 타입으로 지정**

```java
int android1 = 0;
double android2 = 1.0;
char android3 = 'a';
String android4 = "hello";
boolean android5 = true;
```

```
//물론 다음과 같이 타입 지정도 가능
var android1 : Int = 0
var android2 : Double = 1.0
var android3 : Char = 'a'
var android4 : String = "hello"
var android5 : Boolean = true
```

**물론 타입 지정도 가능**!!

**별도의 타입 지정을 하지 않았을 시에는 각 데이터의 기본 타입으로 지정**
(**정수** = Int, **소수** = Double, **문자열** = String,
**문자** = Char, **논리** = Boolean)

```
android1 = 2;
android1 = 3.0;


//자바는 소수타입에 정수타입 대입이 가능
android2 = 2;
android2 = 3.0;
```

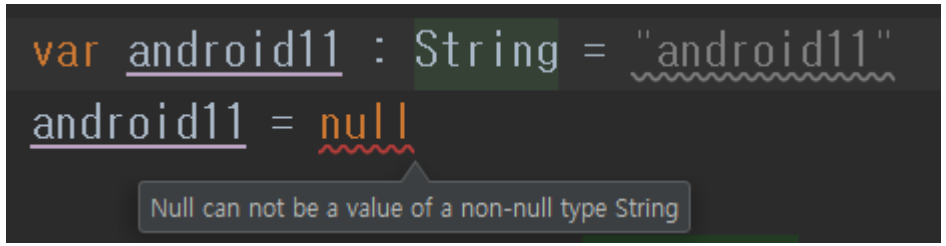
```
//자신의 타입과 어긋나는 데이터를 할당하면 물론 오류가 뜬다.
android1 = 2
android1 = 3.0

//자바와는 달리 소수 타입에 정수를 넣는 것 역시 불가능!
android2 = 2
android2 = 3.0
```

```
android1 = 2;
android2 = (int)3.0;
```

```
//그래서 캐스팅을 한 후에 대입해야 한다.
android1 = 2
android1 = 3.0.toInt()

android2 = 2.toDouble()
android2 = 3.0
```




**코틀린은 일반적으로** null**값을 가질 수 없다**.

```
var android11 : String? = "android11"
android11 = null
//null값을 가질 가능성이 있는 변수에 대해 무작정 null을 대입하면 오류
//이럴 때 쓰는 연산자가 '?'다
```

**주석 내용을 보충하자면 프로그램을 실행시켰을 때** null**값이 생길 가능성이 있는 변수에 대해서는** '?'**를 꼭 붙여주어야 한다**.

```
Object android6;
android6 = 2;
android6 = 3.0;
android6 = 'a';
android6 = "bananaMilk";
android6 = true;
```

```
//Any는 코틀린의 최상위 타입이다.
// 그래서 어떠한 값도 대입 가능하다.
var android6 : Any

android6 = 2
android6 = 3.0
android6 = 'a'
android6 = "bananMilk"
android6 = true
```

```
//자바는 void부터 타입으로 쓸 수 없다
void android7;
```

```
//Unit은 Void와 비슷한 타입니다.
//즉 어떠한 타입도 받을 수 없다.
var android7 : Unit

android7 = 2
android7 = 3.0
android7 = 'a'
android7 = "bananaMilk"
android7 = true
```

```java
//일반적인 자바의 배열 선언 방법
Object[] array1 = {1, "YG", true};
int[] array2 = {10, 20, 30};
double[] array3 = {10.0, 20.0, 30.0};
```

```kotlin
//코틀린에서 배열은 arrayOf로 생성 가능
//별도의 타입을 지정하지 않았을 때는 Any타입으로 인식하여 어떠한 데이터도 넣을 수 있음
var array1 = arrayOf(1, "YG", true)

//타입 지정도 물론 가능. 이때는 arrayOf<T> 형태를 갖게 됨.
var array2 = arrayOf<Int>(10, 20, 30)
var array3 = arrayOf<Double>(10.0, 20.0, 30.0)

//arrayOf<T> 외에도 아래와 같이 typepArrayOf로도 쓸 수 있다.
var array4 = intArrayOf(10, 20, 30)
var array5 = doubleArrayOf(10.0, 20.0, 30.0)
```

```
booleanArrayOf(vararg … BooleanArray
byteArrayOf(vararg elemen… ByteArray
charArrayOf(vararg elemen… CharArray
doubleArrayOf(vararg el… DoubleArray
floatArrayOf(vararg elem… FloatArray
intArrayOf(vararg elements… IntArray
longArrayOf(vararg elemen… LongArray
shortArrayOf(vararg elem… ShortArray
```

다양한
typeArrayOf 함수들

```java
ArrayList<Integer> array6 = new ArrayList<>();
ArrayList<String> array7 = new ArrayList<>();

array6.add(1);
array6.add((2);

array7.add("OB");
array7.add("YB");
```

```kotlin
//여러 데이터 대입을 위한 방법은 배열 외에도 ArrayList가 존재
var array6 = ArrayList<Int>()
var array7 = ArrayList<String>()

array6.add(1)
array6.add(2)

array7.add("OB")
array7.add("YB")
```

자바와 코틀린의 변수를 취급하는 데 있어 가장 큰 차이는 코틀린을 변수도 일종의 클래스로 취급한다. 즉 자바의 Integer, Double 등과 같은 취급을 한다.

# 02 함수

```java
if(android6 == "android"){
    System.out.println("The best part!!");
}
else{
    System.out.println("sopt");
}
```

```java
if(android6 == "android" && android3 == 'a'){
    System.out.println("Best android!!");
}

if(android6 == "android" || android3 == 'a'){
    System.out.println("Hew likes bananaMilk!!");
}
```

```java
//if-else의 경우 타 언어와 유사하다 못해 똑같다
//()안에 조건을 넣어주고 true면 {}안을 실행하고 false면 else문을 실행
if(android6 == "android"){
    println("The best part!!")
}
else{
    println("sopt")
}
```

```java
//타 언어의 &&는 and로 ||는 or로 표현 가능
//타 언어에 비해 명시적이다.
if((android6 == "android") and (android3 == 'a')){
    println("Best android!!")
}
if((android6 == "android") or (android3 == 'a')){
    println("He lieks bananaMilk!!")
}
```

```
void switchTest(Object obj){
    switch(obj){
        //자바는 switch에서 Object가 올 수 없다.
    }
}
```

```
switch(android1){
    case 0:
        break;
    case 2:
        break;
    case 3:
        break;
    default:
        break;
}
```

```
fun whenTest(obj : Any?){
    when(obj){
        1->println("One")
        "Hello" -> println("Greeting")
        is Long -> println("Long")
        !is String -> println("Not a String")
        else -> println("Unknown")
    }
}
```

**타 언어의** switch**와 유사한 구문.**
switch**처럼** case-break**로 구분하지 않고** ->**와줄 바꿈으로 구분**

*is**는 타입 구분에 쓰이는 연산자**

```java
for(int i = 1; i<10; i++){
    System.out.println(i);
}

for(int i = 10; i>1; i--){
    System.out.println(i);
}
```

```java
for(int i = 1; i<10; i+=2){
    System.out.println(i);
}

for(int i = 10; i>1; i-=2){
    System.out.println(i);
}
```

```
for (i in 1..10){
    //1에서 10까지 i가 1씩 증가하면서 반복
    println(i)
}
for(i in 10 downTo 1){
    //10에서 1까지 i가 1씩 감소하면서 반복
    println(i)
}
```

```
for(i in 1..10 step 2){
    //1에서 10까지 i가 2씩 증가하면서 반복
    println(i)
}


for(i in 10 downTo 1 step 2){
    //10에서 1까지 i가 2씩 감소하면서 반복
    println(i)
}
```

Step**을 통해 증감 크기 조절 가능**

```java
String[] chairmanList = {"유동현", "조수현", "최다에", "유현영"};
String[] partLeaderList = {"안다혜", "이영규", "이상은", "이혜진",
"오강훈", "박찬은"};


for (String name : chairmanList) {
    System.out.println(name);
}

for(String name : partLeaderList){
    System.out.println(name);
}
```

```kotlin
//콜렉션타입을 이용해 반복문을 사용할 수도 있다.
var chairmanList : Array<String> = arrayOf("유동현", "조수현", "최다에", "유현영")
var partLeaderList : Array<String> = arrayOf("안다혜", "이영규", "이상은",
        "이혜진", "오강훈", "박찬은")
var soptLeader = chairmanList + partLeaderList

for(name in chairmanList){
    println(name)//회장단 이름 출력
}
for(name in partLeaderList){
    println(name)//파트장 이름 출력
}
for(name in soptLeader){
    println(name)//전체 임원진 이름 출력
}
```

for외에도 while이 있는데 타 언어와 동일하므로 여기서는 생략

```java
int functionBase1(int a, int b){
    return a+b;
}

int functionBase4(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

```kotlin
fun functionBase1(a : Int, b : Int) : Int{
    return a+b
}

fun functionBase2(a : Int, b : Int) : Int = a+b

fun functionBase3(a : Int, b : Int) = a+b

fun functionBase4(a : Int, b : Int) : Int {
    if(a>b)
        return a
    else
        return b
}

fun functionBase5(a : Int, b : Int) = if(a>b) a else b
```

**기본 형은 다음과 같다.**

fun **함수명**(**인자명** : **타입**...) : **리턴타입**{}

```java
int functionBase1(int a, int b){
    return a+b;
}

int functionBase4(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

```kotlin
fun functionBase1(a : Int, b : Int) : Int{
    return a+b
}

fun functionBase2(a : Int, b : Int) : Int = a+b

fun functionBase3(a : Int, b : Int) = a+b

fun functionBase4(a : Int, b : Int) : Int {
    if(a>b)
        return a
    else
        return b
}

fun functionBase5(a : Int, b : Int) = if(a>b) a else b
```

Base2,3**과 같이 함수 내의 연산이 간단하다면 일반 변수에 대입하듯이** = **사용 가능**

3**처럼 타입을 지정하지 않는다면** var a = 5**와 같은 원리로 타입이 지정된다.**

```java
int functionBase1(int a, int b){
    return a+b;
}

int functionBase4(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

```kotlin
fun functionBase1(a : Int, b : Int) : Int{
    return a+b
}

fun functionBase2(a : Int, b : Int) : Int = a+b

fun functionBase3(a : Int, b : Int) = a+b

fun functionBase4(a : Int, b : Int) : Int {
    if(a>b)
        return a
    else
        return b
}

fun functionBase5(a : Int, b : Int) = if(a>b) a else b
```

Base5도 2,3과 **마찬가지로** Base4**의 내용을 압축할 수 있다.**

```
Object functionbBase6(int a, int b){
    return null;
}
```

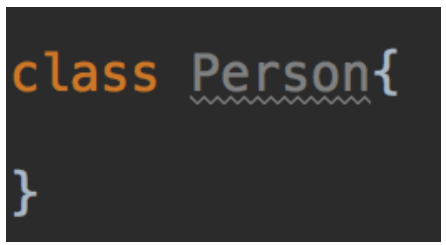
**자바는 객체 타입일 경우 얼마든지** null **리턴이 가능**

```
fun functionBase6(a : Int, b:Int) : Int?{
    return null
}
```

**혹시** null**을 리턴할 수 있는 함수는 반환타입에 ?를 반드시 넣어줘야 한다**.

03

# 클래스





**가장 기본적인 형태의 클래스**

class **클래스명**{}

**일반적으로 클래스명은 '대문자 ' 로 시작**

**기본 생성자도 존재하지 않는 형태**

```
class Person{
    //자바는 생성자 상속 같은 건 없다.
    //새로운 생성자를 만들 땐 그냥 이름(인자..) 해주면 된다

    Person(){

    }

    Person(String name, int age){

    }

}
```

```
class Person(){

}
```

**코틀린은 함수명 옆에 ()를 통해 '기본 생성자'를 만들 수 있다.**
**위는 인자 없는 생성자를 기본 생성자로 갖는 클래스**

```
class Person(){
    constructor(name : String, age : Int) : this()
}
```

**생성자를 새롭게 만들 때는** 'constructor' **키워드를 쓰게 된다.**
**단 이 경우 기본 생성자가 있다면 꼭 그것을 상속받아야 한다.**

```
class Person(name : String){
    constructor(name : String, age : Int) : this(name)
}
```

**위 경우는 인자가** String**형인** name**을 갖는 기본생성자를 갖고 있고** String**형과** Int**형의 기타 생성자를 갖는 경우다. 이 경우** name**은 중간 배개변수 없이 바로 쓸 수 있다.**

```java
class Person{

    String name;

    Person(){
        this.name = "loveAndroid";
    }
}
```

**자바는 생성자에서 생성자가 아닌 변수에 대한 초기화를 해줄 수 있다.**

```
class Person{
    var name : String
    init {
        this.name = "loveAndroid"
    }
}
```

**생성자의 인자는 아닌데 인스턴스 생성 시 초기화를 하고픈 변수가 있을 때는** init{}**을 사용한다.**

**이때 {}안에 변수 값을 넣어주면 해당 클래스 내에서 그 값이 들어간 변수를 사용할 수 있다.**

**위의 경우는** Person **클래스를 통해 객체가 만들어졌다면** name **에** "loveAndroid"**라는 문자열이 들어가게 된다.**

```java
class Android{

    String memberName;
    int memberAge;

    Android(String memberName, int memberAge){
        this.memberName = memberName;
        this.memberAge = memberAge;
    }

    void printMemberInfo(){
        System.out.println("name is " + memberName);
        System.out.println("age is + " + memberAge);
    }
}
```

```java
class NewAndroid extends Android{

    NewAndroid(String memberName, int memberAge) {
        super(memberName, memberAge);
    }

    @Override
    void printMemberInfo(){
        System.out.println("-------YB-------")
        System.out.println("name is " + memberName);
        System.out.println("age is + " + memberAge);
    }
}
```

**코틀린에서 상속은 타 언어와의 공통점은 자식 클래스의 생성자는 부모 클래스의 생성자를 갖고 있어야 한다는 것이다.**

**차이점은 부모 클래스는 꼭 open되어 있어야 하고 만약 메소드 오버라이드를 할 시에는 그 메소드에도 open을 해주어야 한다. 또한 상속할 때 extends가 아닌 : 을 쓴다**

```kotlin
open class Android(memberName : String, memberAge : Int){
    var memberName = memberName
    var memberAge = memberAge

    open fun printMemberInfo(){
        println("name is " + memberName)
        println("age is + " + memberAge)
    }
}

class NewAndroid(memberName: String, memberAge: Int) : Android(memberName,memberAge) {
    override fun printMemberInfo(){
        println("------YB------")
        println("name is " + memberName)
        println("age is " + memberAge)
    }
}
```

```java
class Android{

    String memberName;
    int memberAge;

    Android(String memberName, int memberAge){
        this.memberName = memberName;
        this.memberAge = memberAge;
    }

    void printMemberInfo(){
        System.out.println("name is " + memberName);
        System.out.println("age is + " + memberAge);
    }
}
```

```java
class NewAndroid extends Android{

    NewAndroid(String memberName, int memberAge) {
        super(memberName, memberAge);
    }

    @Override
    void printMemberInfo(){
        System.out.println("-------YB-------")
        System.out.println("name is " + memberName);
        System.out.println("age is + " + memberAge);
    }
}
```

**그런 거 없다..**

**상속은 크게 두 가지 형태로 할 수 있다.**
**기본 생성자 직접 상속과 기타 생성자 생성 및 상속 방식이 있는데**
**아래의 구조를 더 권장.**

```kotlin
class NewAndroid(memberName: String, memberAge: Int) : Android(memberName,memberAge) {
    override fun printMemberInfo(){
        println("------YB------")
        println("name is " + memberName)
        println("age is " + memberAge)
    }
}

class OldAndroid : Android{
    constructor(memberName: String, memberAge: Int):super(memberName, memberAge)

    override fun printMemberInfo() {
        println("------OB------")
        println("name is " + memberName)
        println("age is " + memberAge)
    }
}
```

```java
class Android{

    int count;
    String leader;
    boolean check;
    Android(int count, String leader, boolean check){
        this.count = count;
        this.leader = leader;
        this.check = check;
    }
    public void setCount(int count){
        this.count = count;
    }

    public void setLeader(String count){
        this.leader = leader;
    }

    public void setCheck(boolean check){
        this.check = check;
    }

    public int getCount(){
        return this.count;
    }

    public String getLeader(){
        return this.leader;
    }

    public boolean getCheck(){
        return this.check;
    }
}
```

```kotlin
data class Android(
        var count : Int,
        var leader : String,
        var check : Boolean
)
```

**순수히 '데이터(변수)'만을 담는 클래스**

**일반 클래스와 달리 {}가 아니라 '()'로 감싸져 있다.**

Json **구조와 흡사하여 통신할 때 많이 보게 될 클래스**

**놀랍게도 오른쪽 코틀린 코드는 왼쪽 자바 코드와 동일한 역할을 한다. 세상..**

```java
class Sopt{

    //자바는 static 변수를 지정해주어야 한다
    static String android = "android";

    void sayAndroid(){
        System.out.println(android);
    }
}
```

```
object Sopt{
    var android : String = "android"
    fun sayAndroid(){
        println(this.android)
    }
}
```

**코틀린에서는** static**키워드가 없다**.

**하지만** Object**클래스를 쓴다면** static**처럼 사용할 수 있다**.

```java
System.out.println(Sopt.android);
```

```
println(Sopt.android)
```

**객체화를 하지 않아도 클래스명을 통해 바로 접근 가능**

04

# 마치며

# 大 더하기 더하기

**다음은 이 친구와 함께...**

inno SOPT

# 끄읕ㅅ~ㅅ