3️⃣

안드로이드를 위한 Kotlin 문법

태그

1. 코틀린의 틀링

- Java와 100% 상호 호환 → Java 코드를 완전히 대체 가능
- Java보다 문법이 간결
- 프로그램의 안정성을 높여줌
- var 또는 val 예약어를 통해 데이터 형식을 선언하지 않고 변수 선언 가능

2. 데이터 형식

분류	데이터 형식	설명
문자형	- Char - String	- 2byte를 사용하여 한글 또는 영문 1개만 입력 - 여러 글자의 문자열을 입력
정수형	- Byte - Short - Int - Long	- 1byte 사용하며 -128~+127까지 입력 - 2byte 사용하며 -32768~+32767까지 입력 - 4byte 사용하며 약 -21억~+21억까지 입력 - 8byte 사용하며 상당히 큰 정수까지 입력
실수형	- Float - Double	- 4btye 사용하며 실수를 입력 - 8btye 사용하며 실수를 입력 Float보다 정밀도가 높다
불리언형	- Boolean	true 또는 false 입력

3. 변수 선언 방식

- 암시적 선언

    - 변수의 데이터 형식을 지정하지 않고,
      대입되는 값에 따라 자동으로 변수의 데이터 형식이 지정
    - ex) var var1 : Int = 10 ⇒ var var1 = 10
    - 단, 초기화하지 않는 경우에는 데이터 형식을 반드시 명시
- var (variable)
    - 일반 변수를 선언할때 사용
    - 필요할 때마다 계속 다른 값 대입 가능
- val (value)
    - 변수 선언과 동시에 값을 대입하거나,
      초기화 없이 선언한 후에 한 번만 값을 대입 가능
    - 한 번 값을 대입하고 나면 값을 변경 X

```
var myVar : Int = 100
myVar = 200 // 정상
val myVal : Int = 100
myVal = 200 // 오류
```

4. 데이터 형식 변환

- 캐스팅 연산자 사용

```
var a : Int = "100".toInt()
var b : Double = "100.123".toDouble()
```

5. null 사용

- Kotlin은 기본적으로 변수에 null 값을 넣지 못함
    - 변수를 선언할 때 데이터 형식 뒤에 ?를 붙여야 null 대입 가능

```
var notNull : Int = null // 오류
var okNull : Int? = null // 정상
```

6. 변수가 null값이 아니라고 표시해야 하는 경우

- !! 로 나타냄

```
var ary = ArrayList<Int>(1) // 1개짜리 배열 리스트
ary!!.add(100) // 값 100을 추가
```

---

7. if 문

```
fun main() {
    var count : Int = 85
    if (count >= 90) {
        println("if문: 합격 (장학생)")
    } else if (count >= 60) {
        println("if문: 합격")
    } else {
        println("if문: 불합격")
    }
}
```

8. when 문

```
fun main() {
    var count : Int = 85
    var jumsu : Int = (count / 10) * 10
    when (jumsu) {
        100 -> println("when문: 합격(장학생)")
        90 -> println("when문: 합격(장학생)")
        80, 70, 60 -> println("when문: 합격")
        else -> println("when문: 불합격")
    }
}
```

```
var jumsu : Int = count
when (jumsu) {
    in 90 .. 100 -> println("when문: 합격(장학생)")
    in 60 .. 89 -> println("when문: 합격")
    else -> println("when문: 불합격")
}
```

9. 배열

- 여러개의 데이터를 하나의 변수에 저장하기 위해 사용

10. 일차원 배열 선언 형식

- var 배열명 = Array<데이터 형식>(개수, {초깃값})
- var 배열명 = Array<데이터 형식>(개수) {초깃값}

```
var one = Array<Int>(4, {0})
one[0] = 10
one[3] = 20
```

11. 배열을 선언하면서 바로 대입

`var three : IntArray = intArrayOf(1,2,3)`

12. 이차원 배열 선언 형식

- var 배열명 = Array<배열 데이터 형식>(행 개수, {배열 테이터 형식(열개수)})
- var 배열명 = Array<데이터 형식>(개수) {초깃값}

```
var two = Array<IntArray>(3, {intArray(4)})
two[0][0] = 100
two[2][3] = 200
```

13. ArrayList

```
var one = ArrayList<Int>(4)
var hap = 0
one.add(10)
one.add(20)
```

```
for (i in 0 until one.size) {
    hap = hap + one.get(i)
}
```

---

14. for 문, while 문, break, continue

- for (변수 in 시작..끝 step 증가량) { }
- indices - 배열의 개수만큼 변수에 대입하여 반복

```
var one : IntArray = intArrayOf(10,20,30,40)
for (i in one.indices) {
    println(one[i])
}

for (value in one) {
    println(value)
}

var two : Array<String> = arrayOf("하나", "둘", "셋")
for (i in 0..2 step 1) {
    println(two[i])
}

var k : Int = 0
while (k < two.size) {
    println(two[k])
    k++
}
```

---

15. 메소드, 변수

- 전역 변수, 지역변수

```
var myVar : Int = 100
fun main() {
    var myVar : Int = 0
    println(myVar) // 0

    var sum : Int = addFunction(10, 20)
    println(sum) // 130
}

fun addFunction(num1: Int, num2: Int) : Int {
    var hap : Int
```

```
    hap = num1 + num2 + myVar
    return hap
}
```

16. 예외(exception)

```
fun main() {
    var num1 : Int = 100
    var num2 : Int = 0
    try {
        println(num1/num2)
    } catch (e:ArithmeticException) {
        println("계산에 문제가 있습니다") //계산에 문제가 있습니다
    }
}
```

17. 연산자

연산자	설명
+, -, *, /, %	사칙 연산자 - %는 나머지값을 계산
+, -	부호 연산자 - 변수, 수, 식 앞에 붙임
=	대입 연산자 - 오른쪽을 왼쪽에 대입
++, —	1씩 증가 또는 감소
==, ===, !=, !==, <, >, <=, >=	비교 연산자 - 결과는 true, false
&&, ||, !	논리 연산자 - and, or, not
and, or, xor, inv()	비트 연산자 - 비트 단위로 and, or, exclusive, or, not
shr, shl	시프트 연산자 - 비트 단위로 왼쪽 또는 오른쪽 이동
+=, -=, *=, /=	복합 대입 연산자
toByte(), toShort(), toInt(), toLong() toFloat(), toDouble(), toChar()	데이터 형식을 갖에로 변환하는 함수

18. 클래스 정의와 인스턴스 생성

- 변수(필드)와 메소드로 구성
- 객체 지향 관점에서의 클래스
    - 실세계의 객체들이 가질수 있는 상태와 행동

```
class Car {
    var color : String = ""
    var speed : Int = 0

    fun upSpeed(value: Int) {
        if (speed+value >= 200)
            speed = 200
        else
            speed = speed + value
    }
}

fun main() {
    var myCar1 : Car = Car()
    myCar1.color = "빨강"
    myCar1.speed = 0

    myCar1.upSpeed(50)
    println("색상: " + myCar1.color + ", 속도: " + myCar1.speed)
}
```

    - 생성자 생성

```
 class Car {
    var color : String = ""
    var speed : Int = 0

    constructor(color: String, speed: Int) {
        this.color = color
        this.speed = speed
    }

    fun upSpeed(value: Int) {
        ...
    }
}

fun main() {
    var myCar1 : Car = Car("빨강", 0)
    ...
}
```

19. 메소드 오버로딩

- 한 클래스 내에서 메서드 이름은 같아도

  파라미터의 개수나 데이터 형식만 다르면 여러개 선언 가능

```
class Car {
    var color : String = ""
    var speed : Int = 0

    constructor(color: String, speed: Int) {
        this.color = color
        this.speed = speed
    }
    constructor(speed: Int) {
        this.speed = speed
    }
    constructor() {
    }
    ...
```

---

20. 정적 필드 (static field)

- 인스턴스를 생성하지 않고 클래스 자체에서 사용되는 변수
- companion object { } 안에 작성하여 정적 필드를 만듦

21. 정적 메소드 (static method)

- 메서드 또한 companion object { } 안에 작성
- 인스턴스를 생성하지 않고도 ‘클래스명.메소드명()’으로 호출하여 사용 가능

22. 상수 필드

- 정적 필드에 초깃값을 입력하고 const val로 선언
- 선언한 후에 값을 변경 X
- 클래스 안에 상수를 정의할 때사용

```
class Car {
    var color : String = ""
    var speed : Int = 0
```

```
    companion object {                    // 정적 필드
        var carCount : Int = 0
        const val MAXSPEED : Int = 200    // 상수 필드
        fun currentCarCount() : Int {     // 정적 메서드
            return carCount
        }
    }
    constructor(color: String, speed: Int) {
        this.color = color
        this.speed = speed
        carCount++
    }
    ...
}

fun main() {
    println(Car.carCount)              // 정적 필드
    println(Car.currentCarCount())     // 정적 메서드
    println(Car.MAXSPEED)              // 상수 필드
    println(Math.PI)
    println(Math.pow(3.0, 5.0))
```

23. 클래스 상속

```
open class Car {
    ...
    open fun upSpeed(value: Int) {
        if (speed+value >= 200)
            ...
}

class Automobile : Car {
    var seatNum : Int = 0

    constructor() {
    }
    fun countSeatNum() : Int {
        return seatNum
    }

    override fun upSpeed(value: Int) {
        if (speed+value >= 300)
            ...
    }
}

fun main() {
    var auto : Automobile = Automobile()
    auto.upSpeed(250)
}
```

- 슈퍼 클래스 (superclass, 부모클래스)
  - 자동차(Car) 클래스
- 서브 클래스 (subclass, 자식 클래스)
  - 승용차(Automobile), 트럭 클래스
  - 슈퍼 클래스의 모든 필드와 메소드를 상속 받음
- 메소드 오버라이딩 (overriding)
  - 슈퍼 클래스의 메소드를 무시하고 새로 정의
    - 승용차 클래스의 속도 올리기(upSpeed)

---

24. 추상(abstract) 클래스

- 인스턴스화를 금지하는 클래스
- 인스턴스화
  - 클래스로 인스턴스를 생성하는 것
  - ex) var auto : Automobile = Automobile()

25. 추상 메소드

- 본체가 없는 메소드
- 추상 메소드를 포함하는 클래스는 추상 클래스로 지정

26. 추상 클래스와 추상 메소드를 사용하는 목적

- 공통적으로 사용되는 기능을 추상 메소드로 선언 해놓고,
  - 추상 클래스를 상속받은 후에 추상 메소드를 오버라이딩해서 사용
- 구현한다 (implement)
  - 추상 메소드를 오버라이딩하는 것

```
abstract class Animal {
    var name : String = ""
```

```
    abstract fun move()
}

class Tiger : Animal() {
    var age : Int = 0
    override fun move() {
        println("네 발로 이동한다")
    }
}

fun main() {
    var tiger1 = Tiger() // Tiger tiger1 = new Tiger();
    tiger1.move() // 네 발로 이동한다
}
```

27. 다형성

- 서브클래스에서 생성한 인스턴스를 자신의 클래스 변수에 대입할 수 있음
- 하나의 변수에 여러 종류의 인스턴스를 대입

```
fun main() {
    var animal : Animal
    animal = Tiger() // Animal animal = new Tiger();
    animal.move()
}
```

---

28. 인터페이스(interface)

- 추상 클래스와 성격이 비슷
- interface 키워드를 사용하여 정의하고 내부에는 추상 메소드를 선언
- 상속받는다 X → 구현한다 O
- Kotlin은 다중 상속을 지원하지 않음
    - 대신 인터페이스를 사용하여 다중 상속과 비슷하게 작성

```
abstract class Animal {
    var name : String = ""
    abstract fun move()
}

interface iAnimal {
```

```
    abstract fun eat()
}

class iCat : iAnimal {
    override fun eat() {
        println("생선을 좋아한다")
    }
}

class iTiger : Animal(), iAnimal {
    override fun move() {
        println("네 발로 이동한다")
    }
    override fun eat() {
        println("멧돼지를 잡아먹는다")
    }
}

fun main() {
    var tiger = iTiger()
    tiger.move()
    tiger.eat()
}
```

---

29. 람다식 (lambda expression)

- 함수를 익명 함수(annoymous function) 형태로 간단히 표현한 것
- 한번 사용되고 재사용되지 않는 함수 만들 때 사용
- 코드가 간결해져서 가독성이 좋아짐

```
// 함수 표현방식1
fun addNumber (n1: Int, n2: Int) : Int {
    return n1 + n2
}

// 함수 표현방식2
fun addNumber (n1: Int, n2: Int) : Int = n1 + n2
// 3
fun addNumber (n1: Int, n2: Int) = n1 + n2

// 람다식 표현
val addNumber = { n1: Int, n2: Int -> n1 + n2 }
```

```
// 인자가 없는 익명 함수
fun main() {
    val greeting = fun() {
        println("Hello")
```

```
    }
    val result = greeting()
    print(result) // Hello
}

// 인자와 리턴값 있는 익명 함수
fun main() {
    val greeting2 = { name: String, age: Int ->
        "Hello, My name is $name. I am $age year old"
    }
    val result2 = greeting2("donyang", 20)
    println(result2) // Hello, My name is donyang. I am 20 year old
}
```

30. 람다식 특징

- { }로 감싸며 fun 예약어를 사용하지 않음
- { } 안 → 의 왼쪽은 파라미터, 오른쪽은 함수의 내용
- → 오른쪽 문장이 여러 개라면 세미클론(;)으로 구분
- 내용의 마지막 문장은 반환값(return)

```
버튼변수.setOnClickListener {
    // 버튼 클릭시 실행될 내용
}
```

---

31. 패키지

- 클래스와 인터페이스가 많아지면 관리하기 어렵다
  - 패키지 단위로 묶어서 관리함
  - package 패키지명

32. 제네릭스 (Generics)

- 데이터 형식의 안전성을 보장하는 데 사용
- ex) var strList = ArrayList<String>(4)

33. 문자열 비교

```
var str : String = "안녕하세요"
if (str.equals("안녕하세요")) {
    // 문자열이 같으면 수행될 문장
}
```

34. 날짜 형식

- DateFormat 클래스를 상속받은 SimpleData Format 사용
  - ‘연월일’이나 ‘시분초’와 같은 일반적인 표현이 가능

```
import java.text.DataFormat
import java.text.SimpleDateFormat
import java.util.*

fun main() {
    var now = Date()
    var sFormat : SimpleDateFormat

    sFormat = SimpleDateFormat("yyyyMMdd")
    sFormat = SimpleDateFormat("HH:mm:ss")
    println(sFormat.format(now)) // 23:15:21 형식
}
```