

제어문

제어문 - if문

- If 문

- 기존 java와 같은 형태로 사용 가능함

```
if(a > b){  
    result = a  
}else{  
    result = b  
}
```

- 코틀린에서는 if문을 하나의 표현식으로 간주하며, 값을 반환함.
 - 따라서, 변수의 대입문에 식처럼 사용 가능

```
val result = if(a>b) a else b
```

```
val number = -10  
val result = if (number > 0)  
    "Positive number"  
else  
    "Negative number"
```

제어문 - if문

- If문

- 조건에 따라 여러 표현식을 실행해야 하는 경우, **블록의 마지막 값이 반환됨**

```
val a = -9
val b = -11
val max = if (a > b) {
    println("$a is larger than $b.")
    println("max variable holds value of a.")
    a
} else {
    println("$b is larger than $a.")
    println("max variable holds value of b.")
    b
}
```

제어문 - when문

- when 문 :

- Switch 구문에 대응하는 구문

```
val a = 12
```

```
val b = 5
```

```
println("Enter operator either +, -, * or /")
```

```
val op = readLine()
```

```
val result = when (op) {
```

```
    "+" -> a + b
```

```
    "-" -> a - b
```

```
    "*" -> a * b
```

```
    "/" -> a / b
```

```
    else -> "$op operator is invalid operator."
```

```
}
```

제어문 - when문

- when 문
 - Switch 구분에 대응하는 구문

```
val start = 0
val end = 100
val score = 60
when(score){
    in 90..end->println("우수함")
    50 -> println("평균임")
    in start..end -> println("범위에 있음")
    else -> println("범위를 벗어남")
}
```

제어문 - when문

- when 문
 - Switch 구분에 대응하는 구문

```
val x = 6
when(x){
    1 -> println("x==1")
    2,3 -> println("x==2 or x==3")
    in 4..7 -> println("4부터 7사이")
    !in 8..10 -> println("8부터 10사이가 아님")
    else ->{
        print("x는 1이나 2가 아님")
    }
}
```

제어문 - when문

- when 문
 - 변수에 대입 가능

```
val number = 1
val numStr = when(number % 2){
    0->"짝"
    else->"홀"
}
println(numStr)
```

- 함수의 반환 값으로 사용 가능

```
fun isEven(num:Int) : String = when(num%2){
    0->"짝"
    else->"홀"
}
println( isEven(100) )
```

```
fun isEven(num:Int) : String{
    return when(num%2){
        0->"짝"
        else->"홀"
    }
}
```

반환 타입 생략 가능

```
fun isEven(num:Int) = when(num%2){...}
```

제어문 - for문

- For 문
 - Foreach 구문과 비슷

```
val numbers = arrayOf(1,2,3,4,5)
for(num in numbers){
    println(num)
}
```

```
val numbers = arrayOf(1,2,3,4,5)
for(index in numbers.indices){
    println("number at $index is ${numbers[index]}")
}
```

```
val numbers = listOf(1,2,3,4,5)
for(index in numbers.indices){
    println("number at $index is ${numbers[index]}")
}
```

```
for(i in 1..3){}
for(i in 0..10 step 2){}
for(i in 10 downTo 0 step 2){}
for(i in 1 until 10){} // 1씩 증가
repeat(10){}
```


제어문 - while문

- While 과 do-while 문

```
var x=10
while(x>0){
    println(x)
    x--
}
```

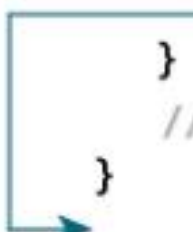
```
val items = listOf("사과", "바나나", "키위")
var index = 0
while(index < items.size){
    println("item at $index is ${items[index]}")
    index++
}
```

```
var x=10
do{
    println(x)
    x--
}while(x>0)
```

```
val items = listOf("사과", "바나나", "키위")
var index = 0
do{
    println("item at $index is ${items[index]}")
    index++
}while(index < items.size)
```

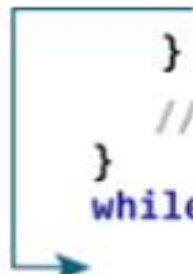
제어문 - Break

```
while (testExpression) {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
}
```




A flowchart illustrating the execution of a while loop with a break statement. It starts at the 'while (testExpression)' condition. If the condition is true, it enters the loop body, executing the code until it reaches the 'break' statement. A blue arrow then points from the 'break' statement to the closing brace of the while loop, indicating an immediate exit from the loop.

```
do {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
} while (testExpression)
```



A flowchart illustrating the execution of a do-while loop with a break statement. It starts by entering the loop body and executing the code until it reaches the 'break' statement. A blue arrow then points from the 'break' statement to the 'while (testExpression)' condition, indicating that the loop will check the condition and potentially exit.

```
for (iteration through iterator) {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
}
```

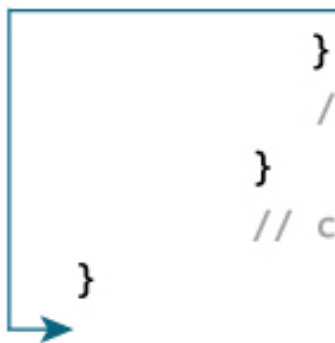


A flowchart illustrating the execution of a for loop with a break statement. It starts at the 'for (iteration through iterator)' condition. If the condition is true, it enters the loop body, executing the code until it reaches the 'break' statement. A blue arrow then points from the 'break' statement to the closing brace of the for loop, indicating an immediate exit from the loop.

제어문 - Break

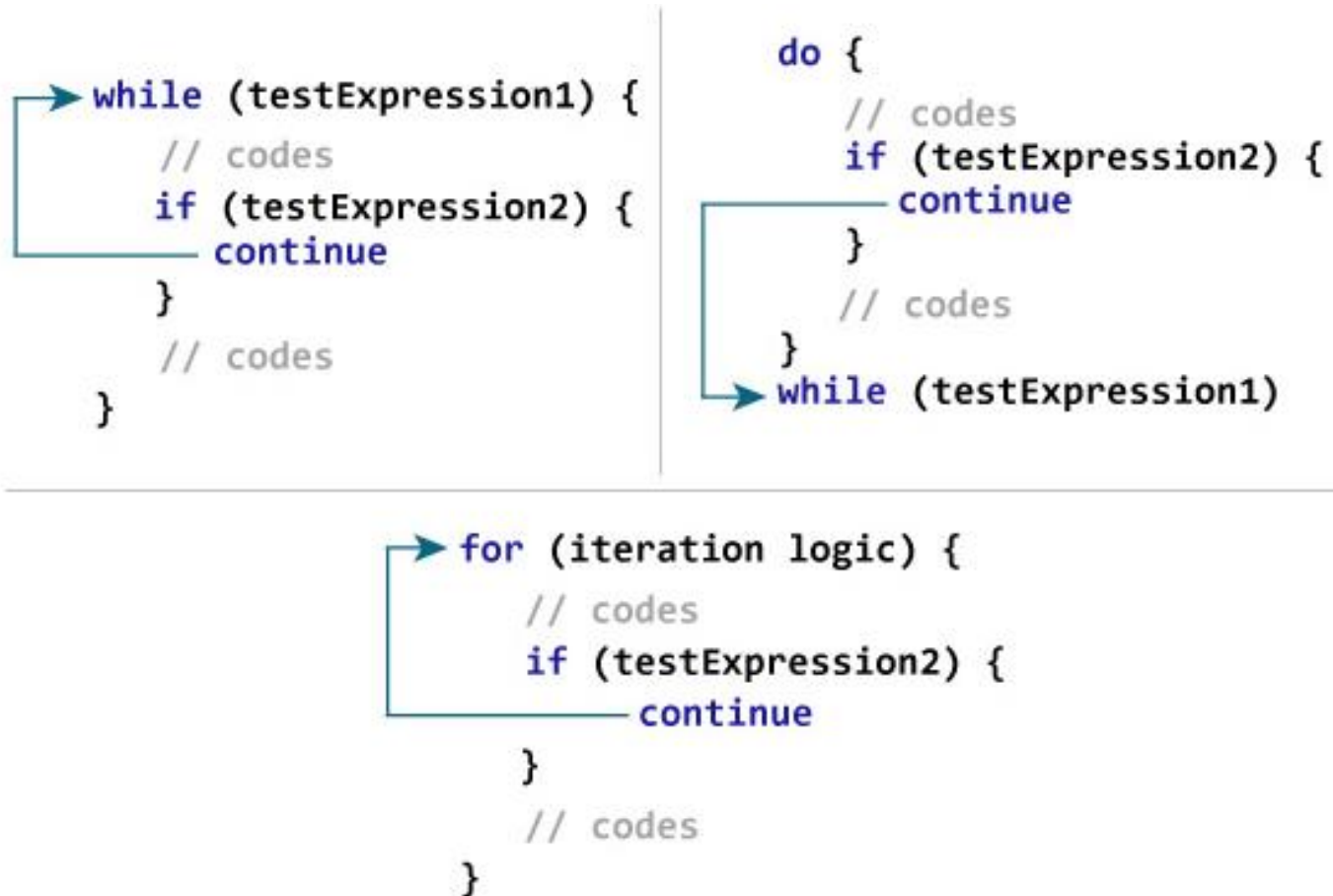
- Labeled break

```
test@ while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition to break) {  
            break@test  
        }  
        // codes  
    }  
    // codes  
}
```



```
first@ for (i in 1..4) {  
    second@ for (j in 1..2) {  
        println("i = $i; j = $j")  
        if (i == 2)  
            break@first  
    }  
    *공백있으면 안됨  
}
```

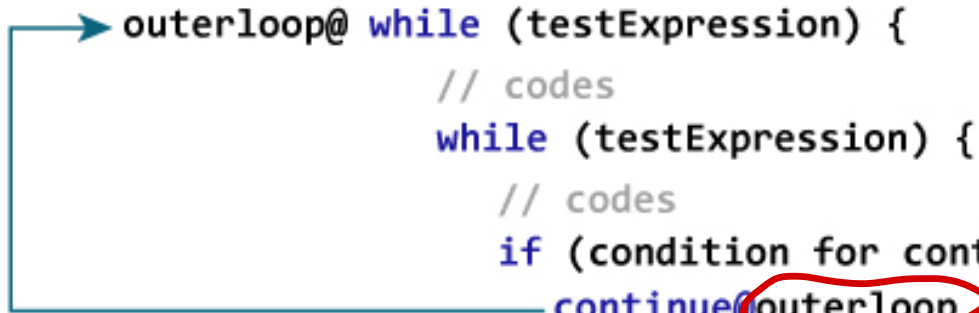
제어문 - Continue




제어문 - Continue

- Labeled Continue

```
→ outerloop@ while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition for continue) {  
            continue(outerloop)  
        }  
        // codes  
    }  
    // codes  
}
```



```
here@ for (i in 1..5) {  
    for (j in 1..4) {  
        if (i == 3 || j == 2)  
            continue@here  
        println("i = $i; j = $j")  
    }  
}
```



Functions

함수

```
fun 함수(인수1:자료형1, 인수2:자료형2,...):반환자료형 {  
}
```

*인자 생략 가능, 반환값이 없는 경우 Unit 및 생략가능

```
fun greet(str:String):Unit{  
    println(str)                greet("Hello World!")  
}
```

```
fun addNumbers(n1: Double, n2: Double): Int {  
    val sum = n1 + n2  
    val sumInteger = sum.toInt()  
    return sumInteger  
}
```

함수

- 간단한 값을 반환하는 경우 `{ }` 생략 가능


```
fun getName(firstName: String, lastName: String): String  
= "$firstName $lastName"
```

```
fun getName(firstName: String, lastName: String) =  
"$firstName $lastName"
```


디폴트 매개변수

- 매개변수의 값을 입력하지 않으면 디폴트 값을 가지게 됨


```
fun main(args: Array<String>) {  
    foo('x', 2)  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```



A diagram with two dotted arrows. One arrow starts from the string 'x' in the call `foo('x', 2)` and points to the parameter `letter` in the function definition `fun foo(letter: Char = 'a', number: Int = 15)`. The other arrow starts from the integer 2 and points to the parameter `number`.

letter = 'x'	number = 2
--------------	------------

```
fun main(args: Array<String>) {  
    foo('y')  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```



A diagram with one dotted arrow starting from the string 'y' in the call `foo('y')` and pointing to the parameter `letter` in the function definition `fun foo(letter: Char = 'a', number: Int = 15)`.

letter = 'y'	number = 15
--------------	-------------

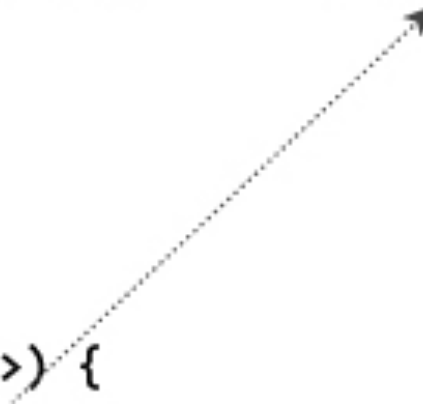
```
fun main(args: Array<String>) {  
    foo()  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```

letter = 'a'	number = 15
--------------	-------------

디폴트 매개변수

- 매개변수의 이름을 사용하면 매개변수의 위치에 상관없이 사용 가능함

```
fun displayBorder(character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main(args: Array<String>) {  
    displayBorder(length = 5)  
}
```



infix 함수

- 함수의 중위적 표현 가능

```
val a = true
val b = false
var result: Boolean
```

```
result = a or b // a.or(b)
println("result = $result")
```

```
result = a and b // a.and(b)
println("result = $result")
```

```
class Structure() {
    infix fun createPyramid(rows: Int){
        var k = 0
        for (i in 1..rows) {
            k = 0
            for (space in 1..rows-i) {
                print(" ")
            }
            while (k != 2*i-1) {
                print("* ") ++k
            }
            println()
        }
    }
}
```

```
val p = Structure()
p createPyramid 4 // p.createPyramid(4)
```

재귀함수(Recursive Function)

- 무한 루프에 빠지지 않도록 주의 해서 사용

```
fun main() {  
    val number = 4  
    val result: Long  
  
    result = factorial(number)  
    println("Factorial of $number = $result")  
}  
  
fun factorial(n: Int): Long {  
    return if (n == 1) n.toLong() else n*factorial(n-1)  
}
```

꼬리 재귀 함수(Tail Recursive Function)

- 기존의 재귀함수는 모든 재귀 호출이 완료될 때까지는 결과를 얻을 수 없었으나, **꼬리 재귀에서는 계산이 먼저 수행되고, 재귀 호출이 수행되는 구조**
 - 컴파일러가 **stackoverflow가 발생하지 않도록 효율적인 순환 기반의 버전으로 최적화**
 - 마지막으로 수행하는 구문이 자신을 호출하는 구문
 - 재귀호출 후 다른 코드가 있으면 사용할 수 없음

꼬리 재귀 함수(Tail Recursive Function)

```
val eps = 1E-15 // "good enough", could be 10^-15
```

```
tailrec fun findFixPoint(x: Double = 1.0): Double
```

```
    = if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

```
val eps = 1E-15 // "good enough", could be 10^-15
```

```
private fun findFixPoint(): Double {
```

```
    var x = 1.0
```

```
    while (true) {
```

```
        val y = Math.cos(x)
```

```
        if (Math.abs(x - y) < eps) return x
```

```
        x = Math.cos(x)
```

```
    }
```

```
}
```

람다 함수 (Lambda)

- Kotlin에서의 함수
 - 반환 자료형 생략 및 블록과 return 생략 가능

```
fun add(x:Int, y:Int): Int{  
    return x+y  
}  
println(add(2,5))
```

```
fun add(x:Int, y:Int) = x+y
```

- 람다함수 : 익명 함수를 간결하게 표현할 수 있는 방법

```
val add: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

```
val add = { x:Int, y:Int -> x + y }
```

```
println(add(10,20))
```

람다 함수 (Lambda)

- 코틀린의 람다식

- 형식 : { 매개변수 -> 함수내용 }
- 람다 함수는 항상 { }로 감싸서 표현
- 인수 목록을 나열하고 -> 이후에 본문 위치
- 인자는 ()로 감싸지 않음
- 인자는 형식 추론이 가능하므로 타입 생략 가능
- 함수 반환값은 함수 내용의 마지막 표현식

람다식 (Lambda)

- 코틀린 람다식

- 변수에 람다식을 저장하고, 변수를 일반 함수처럼 사용
 - 변수에 대입하지 않으면 이후 람다 함수를 사용할 수 없음
 - 람다 함수 뒤에 ()를 추가하여 함수 호출
 - Run() 함수에 대입해도 바로 함수가 호출되어 실행
 - 람다식이 유일한 인자일 경우 () 생략가능 함

```
{println("Hello")}()
```

```
run {println("World")}
```

SAM(Single Abstract Method) 변환

- 추상 메서드 하나를 인수로 사용할 때만 함수 인수 전달

Java

```
changeBtn.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        changeBtn.setBackgroundColor(Color.CYAN);  
    }  
});
```

Kotlin

```
changeBtn.setOnClickListener {  
    view -> changeBtn.setBackgroundColor(Color.CYAN)  
}
```

```
changeBtn.setOnClickListener {  
    _ -> changeBtn.setBackgroundColor(Color.CYAN)  
}
```

인수를 참조하지 않는
경우 _ 또는 생략

```
changeBtn.setOnClickListener{  
    changeBtn.setBackgroundColor(Color.CYAN)  
}
```

```
changeBtn.setOnClickListener {  
    it.visibility = View.INVISIBLE  
}
```

람다식의 인수가 하나인 경우는
it (view) 로 인수 접근

SAM(Single Abstract Method) 변환

- 추상 메서드 하나를 인수로 사용할 때만 함수 인수 전달

Java

```
changeBtn.setOnTouchListener(new View.OnTouchListener() {  
    @Override  
    public boolean onTouch(View v, MotionEvent event) {  
        changeBtn.setBackgroundColor(Color.CYAN);  
        return false;  
    }  
});
```

Kotlin

```
changeBtn.setOnTouchListener{  
    v, event ->  
        changeBtn.setBackgroundColor(Color.CYAN)  
        false  
}
```

Function Type

- 함수 타입의 변수 선언
 - 함수 타입 변수에서는 리턴을 쓰지 않고, 마지막 줄이 반환됨
- 형식

```
val functionType1 : ()->Unit
val functionType2 : (Int) -> Unit
val functionType3 : (Int, String) -> String
```

```
functionType1 = { println("greenjoa") }
functionType2 = { age -> println("나이는 $age") }
functionType3 = { age, name ->
    println("나이: $age, 이름: $name")
    "나이: $age, 이름: $name"
}
```

```
functionType1()
functionType2(20)
println(functionType3(20, "greenjoa"))
```

High-Order Function

- 함수의 인수로 함수나 람다식을 받거나 반환할 수 있는 함수

// 인수

```
fun highOderFunction1(func:()->Unit){  
    func()  
}
```

인자 x, 반환 x 함수

// 반환

```
fun highOderFunction2():()->Unit{  
    return { println("greenjoa") }  
}
```

인자 x. 반환 x 함수

// 인수 및 반환

```
fun highOderFunction3(func:()->Unit):()->Unit{  
    return func  
}
```

인자 x, 반환 x 함수 인자 x, 반환 x 함수

```
highOderFunction1 { println("hello") }
```

```
highOderFunction2()
```

```
highOderFunction3 { println("world") }()
```

확장 함수 (Extension function)

- 클래스에 새로운 함수를 추가
 - 기존 방식은 상속을 통해 새로운 클래스를 만들고, 함수 추가
 - 확장함수는 클래스 밖에서 정의된 클래스의 멤버 함수
 - 멤버 함수를 오버로딩한 경우 확장 함수가 호출됨
- 예) String의 처음과 마지막 문자 삭제 함수
→ String 클래스에 존재하지 않는 함수

```
fun String.removeFirstLastChar():String =  
    this.substring(1, this.length - 1)
```

Receiver type

Receiver type

"HelloWorld".removeFirstLastChar()

클래스

클래스

- 클래스 선언

```
class Person{  
  
}
```

- 객체 선언

- new 키워드는 사용하지 않음

```
val person = Person()
```

- 클래스 생성자

- Primary 생성자

- 매개변수들이 멤버 변수로 자동 추가됨

- Secondary 생성자

- 매개변수들이 멤버 변수로 추가되지 않음
 - 생성자 오버로딩의 개념으로 여러 개의 생성자 정의시 사용함
 - 반드시 Primary 생성자를 호출해야 함에 주의해야 함

클래스

- Primary 생성자
 - 빈 생성자를 생성하며, 코드를 포함할 수 없음
 - 매개변수는 자동으로 멤버 변수로 추가됨

```
class Person constructor(val name:String){  
}
```

```
class Person(val name:String){  
}
```

<java>

```
public class Person {  
    private final String name;  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

- 초기화 블록 활용한 초기화

```
class Person constructor(val name:String){  
    init{  
        println(name)  
    }  
}
```

복잡한 초기화 가능

클래스

- Secondary 생성자
 - 생성자 오버로딩의 기능
 - 매개 변수들이 멤버 변수로 추가되지 않음
 - 매개변수에 **val** / **var**를 사용할 수 없음

```
class Person {  
    var name:String?=null  
    constructor(name:String){  
        this.name = name  
    }  
  
    fun printName():Unit{  
        println(name)  
    }  
}
```

```
val person = Person("greenjoa")  
person.printName()
```

클래스

- Secondary 생성자
 - **Primary 생성자가 존재할 경우, 반드시 primary 생성자를 호출해야 함**, 호출 안 할 경우 오류 발생함

```
class Person (val name :String){  
    var addr:String?=null  
    constructor(name:String, addr:String) :this(name){  
        this.addr = addr  
    }  
    fun printName():Unit{  
        println(name)  
    }  
    fun printAddr():Unit{  
        println(addr)  
    }  
}
```

```
val person = Person("greenjoa", "Seoul")  
person.printAddr()
```

클래스

- Secondary 생성자

- 반드시 **primary** 생성자를 호출해야 함, 호출 안 할 경우 오류 발생
함

```
class Person (val name :String){  
    var addr:String?=null  
    var tel:String?=null  
    constructor(name:String, addr:String) :this(name){  
        this.addr = addr  
    }  
    constructor(name:String, addr:String, tel:String):this(name, addr){  
        this.tel = tel  
    }  
}
```

클래스

- Primary 생성자에게 매개변수를 정의
 - 생성자에서 수행할 내용 없으면 {} 생략 가능

```
class Person (val name:String, val addr:String, val tel:String){  
    constructor(name:String, addr:String) :this(name, addr, "")  
    constructor(name:String):this(name, "", "")  
    constructor():this("", "", "")  
    ...  
}
```

```
val person1 = Person("greenjoa", "Seoul", "010-1234-1234")  
val person2 = Person("greenjoa", "Seoul")  
val person3 = Person("greenjoa")  
val person4 = Person()
```

디폴트 매개변수

- 함수의 매개변수에 디폴트 값을 지정

```
class Person (val name:String="noInfo",  
              val addr:String="noInfo",  
              val tel:String="noInfo"){  
  
}
```

```
val person1 = Person("greenjoa", "Seoul", "010-1234-1234")  
val person2 = Person("greenjoa", "Seoul")  
val person3 = Person("greenjoa")  
val person4 = Person()
```

접근 제한자

- 4개의 접근 제한자 존재함
 - Public : 전체 공개, 생략하면 기본이 public
 - Private : 현재 파일 내에서만 사용 가능
 - 클래스 : 현재 클래스 or 인터페이스에서만 사용 가능
 - Protected : 해당 파일 내부에서만 사용 가능
 - 클래스 : private과 동일하지만, subclass에서 사용 가능
 - Internal : 같은 모듈 내에서만 사용 가능

```
private fun printName(name:String = "greenjoa"){  
    println(name)  
}
```

test1.kt

```
printName()
```

test2.kt **에러발생**

클래스의 상속

- 코틀린의 모든 기본 클래스는 상속이 불가능함
- 클래스 상속을 하려면 open 키워드를 추가해야 함
 - 단일 상속만 가능

```
open class Animal{  
}  
  
class Dog : Animal(){  
}
```

```
open class Animal(val name:String){  
}  
  
class Dog(name:String) : Animal(name){  
}  
  
class Dog : Animal {  
    constructor(name:String):super(name)  
}
```


추상 클래스

- 인스턴스화 할 수 없는 클래스
 - 추상 메소드는 상속을 통해 오버라이딩해 주어야 함
 - Open 키워드 생략

```
abstract class A{  
    abstract fun func()  
    fun func2(){  
    }  
}  
  
class B : A(){  
    override fun func() {  
        println("Hello")  
    }  
}
```

```
abstract class A{  
    abstract fun func()  
    open fun func2(){  
    }  
}  
  
class B : A(){  
    override fun func() {  
        println("Hello")  
    }  
  
    override fun func2() {  
        super.func2()  
        println("World")  
    }  
}
```

인터페이스

- 자바의 인터페이스와 동일한 기능을 수행함
 - 변수도 선언만 가능하며, 다중상속 가능

```
interface Runnable{  
    var type : Int  
    fun run()  
    fun fastRun() = println("빨리 달린다")  
}
```

```
class RunnableClass : Runnable{  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
}
```

```
val test = RunnableClass()  
test.fastRun()  
test.run()  
test.type = 50  
println(test.type)
```

```
class RunnableClass : Runnable{  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
    override fun fastRun() {  
        super.fastRun()  
        println("더 빨리 달린다")  
    }  
}
```

인터페이스

- 인터페이스와 클래스 다중 상속 가능하며, 순서 상관 없음

```
class RunnableClass : Runnable, A(){  
    override fun func() {  
        TODO("not implemented")  
    }  
  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
    override fun fastRun() {  
        println("더 빨리 달린다")  
    }  
}
```

인터페이스 와 클래스 상속

- 인터페이스와 클래스 다중 상속 가능하며, 순서 상관 없음

```
class Dog : Runnable, Eatable, Animal(){  
  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
    override fun fastRun() {  
        println("더 빨리 달린다")  
    }  
    override fun eat() {  
        println("먹는다")  
    }  
}
```

중첩 클래스

- 클래스 내에 선언된 정적인 클래스
 - Java에서의 중첩 정의된 정적 멤버 클래스의 개념과 동일
 - Outerclass의 일반 멤버 변수 및 함수 접근 불가
 - Outerclass의 객체 생성 필요 없음

```
class OuterClass{  
    var num1=10  
    class NestedClass{  
        var num2 = 20  
        fun something1(){  
            println(num2)  
        }  
        fun something2()=20  
    }  
}
```

static

```
val a =OuterClass.NestedClass()  
a.something1()  
val b= OuterClass. Nested Class().something2()
```

내부(Inner) 클래스

- Java 의 인스턴스 멤버 클래스 개념과 동일
 - Outerclass의 객체를 생성해야만 사용할 수 있는 중첩 클래스
 - Outerclass의 멤버 액세스 가능
 - Outclass 객체 생성 필요

```
class OuterClass{  
    var num=10  
    inner class InnerClass{  
        fun something1(){  
            num = 20  
            println(num)  
        }  
        fun something2()=20  
        fun getOuterReferences():OuterClass = this@OuterClass  
    }  
}
```

```
val a =OuterClass().InnerClass()  
a.something1()  
val b= OuterClass().InnerClass().something2()  
val c = a.getOuterReferences()  
println(c.num)
```

Companion Object

- 자바에서의 static 변수 및 메소드 기능이 필요한 경우 사용

```
class Person {  
    fun callMe() = println("I'm called.")  
}
```

```
val p1 = Person()  
p1.callMe()
```

```
class Person {  
    companion object {  
        fun callMe() = println("I'm called.")  
    }  
}  
  
Person.callMe()
```

Object 클래스

- Singleton 패턴의 객체 정의
 - 하나의 instance를 가지는 클래스 선언

```
object Test {  
  private var a: Int = 0  
  var b: Int = 1  
  fun makeMe12(): Int {  
    a = 12  
    return a  
  }  
}
```

```
val result = Test.makeMe12()  
println("b = ${Test.b}")  
println("result = $result")
```


Object 클래스

- 익명의 객체를 선언시에도 사용

- 인터페이스를 구현한 객체 생성시 사용

```
val obj = object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) {  
        // ...  
    }  
    override fun mouseEntered(e: MouseEvent) {  
        // ...  
    }  
}
```

Object 클래스

- 클래스의 함수를 오버라이딩한 익명 객체

```
open class Person(name: String, age: Int) {  
    init {  
        println("name: $name, age: $age")  
    }  
    fun eat() = println("Eating food.")  
    fun talk() = println("Talking with people.")  
    open fun pray() = println("Praying god.")  
}
```

```
val atheist = object : Person("greenjoa", 23) {  
    override fun pray() = println("I don't pray. I am an atheist.")  
}
```

```
atheist.eat()  
atheist.talk()  
atheist.pray()
```

Data 클래스

- 데이터를 저장하는 구조의 클래스
 - equals, hashCode, copy, toString, set, get, componentN 자동생성

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

```
val jack = User("jack", 29)
println("name = ${jack.name}")
println("age = ${jack.age}")
```

```
val u1 = User("John", 29)
val u2 = u1.copy(name = "Randy")
println("u1: name = ${u1.name}, age = ${u1.age}")
println("u2: name = ${u2.name}, age = ${u2.age}")
```

```
val u1 = User("John", 29)
val (name, age) = u1
println("name = $name")
println("age = $age")
```

객체 분리 선언
→ u1.component1()
→ u1.component2()

Enum 클래스

- 열거체의 기능을 수행하는 클래스
 - 클래스 안에 프로퍼티나 메소드 정의 가능
 - 프로퍼티와 메소드 사이에 ; 작성

```
enum class Color {  
    RED, YELLOW, GREEN, BLUE  
}
```

```
enum class Color(val r:Int, val g:Int, val b:Int){  
    RED(255,0,0),  
    YELLOW(255,255,0),  
    GREEN(0,255,0),  
    BLUE(0,0,255);  
    fun rgb() = (r*256 + g)*256+b  
}
```

```
println(Color.RED.r)  
println(Color.RED.ordinal) // index  
println(Color.RED.rgb())
```

```
fun getWarm(color: Color) = when(color){  
    Color.RED, Color.YELLOW -> "warm"  
    Color.BLUE -> "cold"  
    Color.GREEN -> "Neutral"  
}
```

Sealed 클래스

```
open class Expr
class Const(val value: Double) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()

fun eval(e: Expr): Double = when (e) {
    is Const -> e.value
    is Sum -> eval(e.right) + eval(e.left)
    else -> throw IllegalArgumentException("Unknown expression")
}
```

*else 일 경우 에러 발생

```
val obj1 :Expr = Const(10.2)
val obj2 :Expr = Sum(Const(10.2),Const(20.3))
println(eval(obj1))
println(eval(obj2))
```

- Subclass의 생성 가능성을 제한
 - When 표현식에서 모든 sealed 클래스의 서브클래스를 처리
 - Else문 필요없음
 - 자동 open 클래스

Sealed 클래스

```
sealed class Expr
class Const(val value: Double) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()
object NotANumber : Expr()

fun eval(e: Expr): Double = when (e) {
    is Const -> e.value
    is Sum -> eval(e.right) + eval(e.left)
    NotANumber -> java.lang.Double.NaN
}
```

- 주의 사항

- Sealed 클래스는 추상 클래스
- Sealed 클래스의 모든 서브 클래스는 sealed class가 선언된 같은 파일에 선언되어야 함

Collections

컬렉션 - List

- 리스트(List)
 - 같은 자료형의 데이터들을 순서대로 가짐
 - 중복 아이터를 가질 수 있고, 추가, 삭제, 수정이 용이함

- 리스트 생성

- 읽기 전용 리스트 생성 `listOf()` 메서드 사용

```
val list = ArrayList<String>()  
list.add("greenjoa")
```

```
val foods:List<String> = listOf("라면", "갈비", "밥")
```

```
val foods2 = listOf("라면", "갈비", "밥")
```

- 변경 가능한 리스트 생성 `mutableListOf()` 메서드 사용

```
val foods:MutableList<String> = mutableListOf("라면", "갈비", "밥")
```

```
val foods2 = mutableListOf("라면", "갈비", "밥")
```

```
foods.add("초밥")
```

```
foods.removeAt(0)
```

```
foods[1] = "부대찌개"
```

```
foods.set(1, "김치찌개")
```

```
val foods = mutableListOf<String>()
```


컬렉션 - Map

- 맵
 - 키와 값의 쌍으로 이루어진 자료구조
 - 키는 중복될 수 없는 자료구조

- 맵 생성

- 읽기 전용 맵

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val value : Int = map.getValue("a")
println(value)
for((k,v) in map){
    println("$k -> $v")
}
```

```
val map = HashMap<String, String>()
map.put("item1", "greenjoa")
```

- 변경 가능한 맵

```
val citiesMap = mutableMapOf("한국" to "서울", "일본" to "동경", "중국" to "북경")
citiesMap["한국"] = "서울특별시" // 덮어쓰기
citiesMap["미국"] = "워싱턴"    // 추가
for((k,v) in citiesMap){
    println("$k -> $v")
}
```

컬렉션 - Set

- 집합
 - 중복되지 않는 요소들로 구성된 자료구조
- 집합 생성
 - 읽기 전용 집합

```
val citySet = setOf("서울","수원","부산")  
println(citySet.size)  
println(citySet.contains("서울"))
```

- 변경 가능한 집합 생성

```
val citySet2 = mutableSetOf("서울","수원","부산")  
citySet2.add("안양")  
citySet2.add("안양")  
citySet2.add("수원")  
println(citySet2)  
println(citySet2.intersect(citySet))
```

수고하셨습니다.