Chapter 03 코틀린 사용을 위한 기본 문법

1.1 코딩 준비하기 - 새프로젝트 생성

· Android Studio에서 새 프로젝트 생성
  - 프로젝트 유형: Empty Views Activity

1.2 로그의 활용

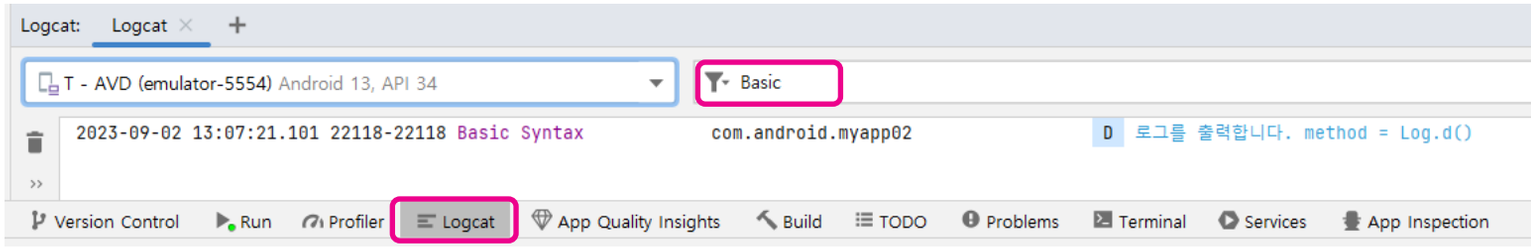
· mainActivity.kt 파일 수정
  - MainActivity 클래스의 onCreate 메소드의 마지막 행에 다음 내용 추가

Log.d("Basic Syntax", "로그를 출력합니다. method = Log.d('로그 제목', '로그 내용')"

```
1   package com.android.myapp02
2
3   import androidx.appcompat.app.AppCompatActivity
4   import android.os.Bundle
5   // Log 클래스를 사용하기 위한 import
6   import android.util.Log
7
8   class MainActivity : AppCompatActivity() {
9       override fun onCreate(savedInstanceState: Bundle?) {
10          super.onCreate(savedInstanceState)
11          setContentView(R.layout.activity_main)
12          // Log 출력을 위한 Log 객체의 [d]debug 메소드 호출
13          // 첫 번째 인자는 로그 제목, 두 번째 인자는 로그 내용
14          Log.d( tag: "Basic Syntax",  msg: "로그를 출력합니다. method = Log.d()")
15      }
16  }
```

1.2 로그의 활용(계속)

- 앱 실행
- Android Studio 아래에서 'Logcat' 탭 선택
- 필터 영역에 Log.d 메서드에서 첫 번째 인자로 입력한 태그(로그 제목)를 입력하고 엔터

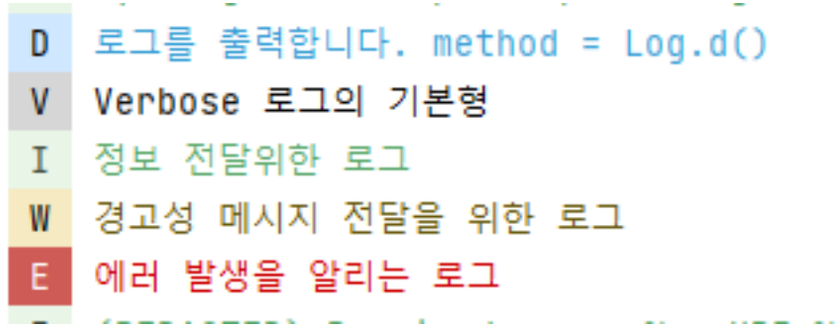


- Log 코딩할 때 코드의 흐름을 파악하기 위해 앱 외부에 출력하는 정보
  디버거 사용할 때는 Break Point 설정의 번거로움
- Logcat 출력되는 로그를 모아서 보여주는 도구
  태그 필터링 기능을 활용하여 원하는 로그만 추려서 볼 수 있음
- Log 메서드

1.2 로그의 활용(계속)

· Log 메서드

메서드(함수)	의미	설명
Log.v()	verbose	상세한 로그 내용을 출력하기 위해 사용
Log.d()	debug	개발자용으로 개발에 필요한 내용을 출력하기 위해 사용
Log.i()	information	정보성이 있는 일반적인 메시지를 출력하기 위해 사용
Log.w()	warning	에러는 아니지만 경고성 메시지를 출력하기 위해 사용
Log.e()	error	실제 에러 메시지를 출력하기 위해 사용



2.1 변수

· 변수 선언: var 키워드 사용

```
var name = "홍길동"
```

변수 선언과 초기화
변수의 type은 초기화하는 값에 따라 자동 결정

```
var name
name = "홍길동"
```

변수 선언과 초기화 분리
반드시 변수의 type 지정 필요

```
var age: int
age = 27
age = "장길산"  (?)
```

type을 명시한 변수 선언
초기화
다른 type으로 값 변경 불가능

2.2 데이터 타입

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

```
var shortValue: Byte = 127
var intValue = 127                    // Int 타입으로 처리
var doubleValue = 3.141592
var floatValue = 3.141592F
var someValue: Float = 3.141592654    The floating-point literal does not conform to the expected type Float
var someValue2: Int = 2147483749      The integer literal does not conform to the expected type Int
```

2.3 읽기 전용 변수 val

· 변수 선언: val 키워드 사용

```
val name = "홍길동"
name = "장길산"
```

Val cannot be reassigned

2.4 상수 const

· 상수 선언: const 키워드 사용

```
const val PI = 3.141592
```

· 상수 const는 컴파일 시간에 값이 결정 - val과 차이

**참고: 문자열 템플릿

· 예 var name = "홍길동"

```
Log.d("BasicSyntax", "이름은 $name 입니다.")
Log.d("BasicSyntax", "이름은 $name입니다.")
Log.d("BasicSyntax", "이름은 ${name}입니다.")
Log.d("BasicSyntax", "이름은 ${name + 1}입니다.")
```

Unresolved reference name입니다.

2.5 코틀린 코딩 규칙

· 클래스 이름
  - 클래스 이름은 대문자로 시작
  - camel case(어떤 의미의 단어를 합성, 각 단어의 첫 자는 대문자)
  - 클래스 내부에는 변수와 함수

· 함수명과 변수명
  - 함수명과 변수명은 소문자로 시작
  - camel case 예 onCreateActivity
  - snake case(밑줄 문자와 소문자만 사용) 예 on_create_activity

· 상수명
  - 대문자로만 작성

· 들여쓰기
  - 스페이스 또는 탭을 이용하여 일정 간격 띄어쓰기
  - 필수는 아니지만, 가장 중요한 규칙 - 가독성(Readability)
  - Settings
    [Editor} - [Code Style] - [Kotlin]에서 탭크기 설정

3.1 조건문 if

· 기본 if 문 형식 : 조건식의 결과(true/false)에 따라 true일 때 { } 블록 내부의 문장 실행

```
if ( 조건식 ) {
    문장
}
```

· 관계(비교) 연산자

연산자	의미	사용예	결과
>	왼쪽이 크면 true, 작으면 false	1 > 2	false
<	왼쪽이 작으면 true, 크면 false	1 < 2	true
>=	왼쪽이 크거나 같으면 true, 작으면 false	1 >= 2	false
<=	왼쪽이 작거나 같으면 true, 크면 false	1 <= 2	true
==	같으면 true, 다르면 false	1 == 2	false
!=	다르면 true, 같으면 false	1 != 2	true

· 논리 연산자v

연산자	의미	사용예	결과
&&	논리곱, 두 항이 모두 true일 때 true, 아니면 false	(2 > 1) && < (3 < 1)	false
||	논리합, 두 항 중 하나가 true일 때 true. 아니면 false	(2 > 1) || (3 < 1)	true
!	단항 부정 연산자, true이면 false, false이면 true	! (2 > 1)	false

3.1 조건문 if

· if ~ else 문 형식 : 조건식의 결과(true/false)에 따라 else 앞 뒤의 코드 영역을 선택 실행

```
if ( 조건식 ) {
        조건이 참일 때 실행되는 코드 영역
} else {
        조건이 거짓일 때 실행되는 코드 영역
}
```

· if ~ else if ~ else 문

· 변수에서 직접 if 문 사용하기

예

```
var a = 5
var b = 3
var bigger = if ( a > b )  a  else  b
```

· if 문의 마지막 값을 반환값으로 사용하기

예

```
var a = 5
        var b = 3
        var bigger = if ( a > b) {
                        var c = 30
                        a
                     }  else  {
                        b
                     }
```

3.2 조건문 when

· when 문의 기본 형식

```
when ( 변수 ) {
    값1 -> { 변수의 값이 값1과 같을 때 실행할 블록 }
    값2 -> { 변수의 값이 값2와 같을 때 실행할 블록 }
    ...
    else -> { 위에 나열된 값들과 일치하지 않을 때 실행할 블록 }
}

when ( 변수 ) {
    값1, 값2 -> { 변수의 값이 값1 또는 값2와 같을 때 실행할 블록 }
    값3 -> { 변수의 값이 값3과 같을 때 실행할 블록 }
    ...
    else -> { 위에 나열된 값들과 일치하지 않을 때 실행할 블록 }

when ( 변수 ) {
    in 값1..값2 -> { 변수의 값이 값1과 값2의 범위 내에 있을 때  실행할 블록 }
    in 값3..값4 -> { 변수의 값이 값3과 값4의 범위 내에 있을 때 실행할 블록 }
    ...
    else -> { 위에 나열된 값들과 일치하지 않을 때 실행할 블록 }
```

3.2 조건문 when(계속)

· when 문에서 변수가 없으면

"값" 위치에 비교할 식을 기술한다.

예

```
var currentValue = 6
when  {
        currentValue == 5 -> { currentValue의 값이 5일 때 실행할 블록 }
        currentValue > 5 ->  { currentValue의 값이 5보다 클때 실행할 블록 }
        ...
        else -> { 위에 나열된 값들과 일치하지 않을 때 실행할 블록 }
}
```

4.1 배열

· 배열 변수 선언

var 변수 = IntArray(개수)

예 var students = IntArrat(10)

· 배열 변수의 타입

CharArray,

ByteArray, ShortArray, IntArray, LongArray,

FloatArray, DoubleArray

BooleanArray

· 배열 변수의 선언과 배열 요소에 초기값 할당

var 변수 = Array(개수, {item->값})

예 var numArray = Array(5, { item -> 0 })

var stringArray = Array(5, { item -> "" }) // String 타입의 배열 선언

· 값으로 배열 공간 할당하기

예 var dayArray = arrayOf("Sun", "Mon", "Tue", "Web", "Thu", "Fri", "Sat")

4.1 배열(계속)

· 배열 요소에 값 할당하기

```
배열명[인덱스] = 값
배열명.set(인덱스, 값)
```

· 인덱스가 배열 범위를 벗어날 때

```
var intArray = IntArray(10)
intArray[10] = 100          // ArrayIndexOutOfBoundsException
```

· 배열 요소의 값 꺼내기

```
배열명[인덱스]
배열명.get(인덱스)
```

4.2 컬렉션

· 배열과 같이 여러 개의 자료를 넣을 수 있는 데이터 집단
· 동적 배열(Dynamic Array)
  - 컬렉션은 값을 넣는 대로 전체 크기가 달라질 수 있는 반면에 배열은 고정 크기
· 컬렉션의 종류
  - 리스트 mutableList
  - 맵 mutableMap
  - 셋 mutableSet

mutable vs. immutable
변경가능 변경불가능

· mutableList 생성하기 / 데이터 추가
  예 var mutableList = mutableListOf("MON", "TUE", "WEB") // 입력되는 값으로 데이터 타입 추정
  mutableList.add("THU")

· mutableList 데이터 사용 / 수정
  예 Log.d("check", "mutableList 컬렉션에서 0번째 값은 ${mutableList.get(0)}이다.")
  mutableList.set(0, "새로운값")

· mutableList 데이터 삭제
  예 mutableList.removeAt(0)

· Emyty mutableList 생성
  예 var emptyMutableList = mutableListOf<String>() // empty List는 반드시 데이터 type 지정

4.2 컬렉션(계속)

· mutableList의 크기 확인

예 Logd.d("check", "mutableList의 크기는 ${mutableList.size} 이다.")

· mutableSet 사용

- set은 중복을 허용하지 않음

예
```
var mutableSet = mutableSetOf("MON", "TUE")
var mutableSet2 = mutableSetOf<String>()        // empty set
mutableSet.add("TUE")   // 데이터가 set에 입력되지 않음(중복 허용 않기 때문)
mutableSet.add("WEB")
Log.d("check", "set에 있는 데이터는 ${mutableSet} 이다.")
mutableSet.remove("WEB")
Log.d("check", "set에 있는 데이터는 ${mutableSet} 이다.")
```

· mutableMap 생성과 데이터 추가

- map은 key와 value의 쌍으로 데이터 관리

예
```
var mutableMap = mutableMapOf<String, String>()
mutableMap.put("key1", "value1")
mutableMap.put("key2", "value2")
```

4.2 컬렉션(계속)

· 초기값을 갖는 mutableMap 생성과 맵 항목 값 사용하기

예 var mutableMap = mutableMapOf("키1" to "값1", "키2" to "값2")
Log.d("check", "map에서 키1의 값은 ${mutableMap.get("키1")}이다.")
// map에 없는 값을 사용하면 get 함수는 null을 반환

· mutableMap에서 데이터 삭제

예 mutableMap.remove("키1")

4.3 immutable 컬렉션

예 val MUTABLE_LIST = listOf("1", "2")
Log.d("check", "리스트에서 두 번째 값은 ${MUTABLE_LIST.get[1]}이다.")

5.1 반복문: for

· for 반복문 형식

- for(변수 in 시작값..종료값) { 실행 코드 } // 시작값부터 종료값까지 1씩 증가하면서 반복
- for(변수 in 시작값 until 종료값) { 실행 코드 } // 종료값은 포함하지 않고 반복
- for(변수 in 시작값..종료값 step 2) { 실행 코드 } // 시작값부터 종료값까지 2씩 증가하면서 반복
- for(변수 in 시작값 until 종료값 step 2) //
- for(변수 in 시작값..종료값 downto 1) // 시작값부터 종료값까지 1씩 감소하면서 반복
- for(변수 in <배열 또는 컬렉션>) // 배열 또는 컬렉션의 항목의 갯수만큼 반복

5.2 반복문: while

· while 반복문 형식

- while(조건식) {
    // 조건식의 결과가 true 반복 실행할 문장 블록
    // 이 부분에 조건식의 결과를 false를 만드는 부분이 있어야 함
  }

- do {
    // 조건식의 결과가 true 반복 실행할 문장 블록
    // 이 부분에 조건식의 결과를 false를 만드는 부분이 있어야 함
  } while(조건식)

· while과 do ~ while의 차이

- while은 조건식을 먼저 판단하기 때문에 한 번도 반복 범위를 실행하지 않을 수 있다.
- do ~ while은 반복 범위를 한 번은 실행한 뒤에 조건식을 판단한다.

5.3 반복문 제어: break, continue

· break

- break 문을 포함하는 반복문을 탈출한다.

예 for(index in 1..10) {
    Log.d("check", "현재 index는 ${index}이다.")
    if(index > 5) {
        break
    }
}
Log.d("breakcheck", "번복문을 종료한 뒤의 index는 ${index}이다.")

· continue

- 반복 범위 내에서 continue 이하의 문장 실행을 생략하고, 다음 조건식 판단으로 진행

예 for(except in 1..10) {
    if(except > 3 && except < 8) {
        continue
    }
    Log.d("continuecheck", "반복은 10회 모두 이루어진다. except의 값이 4, 5, 6, 7일 때는 생략")
}

6.1 함수 정의

· 함수 정의

- 함수는? 일련의 코드 블록을 모듈화 하는 방법
- 함수 정의 형식

```
fun 함수명(파라미터 이름: 타입): 반환 타입 {
    실행할 문장
    return 값
}
```

· 반환값과 입력값이 있는 함수

예
```
fun square(x: Int): Int {
    return x * x
}
```

· 반환값이 없고 입력값이 있는 함수

예
```
fun printSum(x: Int, y: Int) {
    Log.d("funccheck", "두 인수의 합계는 ${x} + ${y} = ${x + y}이다.")
}
```

· 반환값만 있는 함수

예
```
fun getPi(): Double {
    return 3.141592654
}
```

6.2 함수의 사용

· 반환값과 입력값이 있는 함수 호출

```
변수 = 함수명(파라미터)   // 함수를 호출하고 반환값을 변수에 저장
예 var squareResult = square(30)
   Log.d("funcheck", "함수 호출 결과는 ${squareResult}이다.")
```

· 반환값과 입력값이 있는 함수 호출

```
함수명(파라미터)
예 printSum(3, 5)
```

· 반환값이 있고, 입력값이 없는 함수 호출

```
예 var PI = getPI()
   Log.d("funccheck", "함수 호출 결과는 ${PI} 이다.")
```

6.2 함수의 파라미터

· 코틀린에서 함수 파라미터는 모두 읽기 전용(read only, val)
· 파라미터의 기본값 정의와 호출

예
```
fun newFunction(name: String, age: Int = 29, weight: Double = 65.5) {
    Log.d("funcheck", "파라미터 name의 값은 ${name}이다.")
    Log.d("funcheck", "파라미터 age의 값은 ${age}이다.")
    Log.d("funcheck", "파라미터 weight의 값은 ${weight}이다.")
}

// 호출
newFunction("Hello")        // 함수 호출에서 파라미터 age와 weight에는 값이 전달되지 않음
newFunction("Michael", weight = 67.5)
                            // 함수 호출에서 파라미터 age에는 값이 전달되지 않음
```

· 위치 파라미터와 이름 파라미터
- 위치 파라미터를 먼저
- 이름 파라미터에 값이 전달되지 않으면 함수 정의에 사용된 기본값을 함수 내에서 사용
- 이름 파라미터에 값을 전달하면 전달된 값을 함수 내에서 사용

7.1 클래스의 기본 구조

· 클래스의 기본 구조

```
class 클래스명 {
    var 변수                        // 멤버 변수
    func 함수 {                     // 멤버 함수
      함수에서 실행할 코드
    }
}
```

예
```
class MyString {
    var length: Int
    fun plus(other: Any) {
        // 코드
    }
    fun compareTo(other: String) {
        // 코드
    }
}
```

7.2 클래스 코드 작성하기

· 클래스 소코프 class scope

예
```
class MyString {
    // 클래스 스코프 class scope
}
```

· 프라이머리 생성자 primary constructor

예
```
class Person constructor(value: String) {
    // 코드
}
```

예
```
class Person(value: String) {      // constructor 키워드 생략
    init {     // 프라이머리 생성자 또는 기본 생성자를 사용할 때 적용
        Log.d("class", "생성자로부터 전달받은 value의 값은 ${value} 이다.")
    }
}
```

예
```
class Person(val value: String) {     // init 블록 생략하면 생성자 파라미터에 'val' 키워드 추가
    fun process() {                   // class scope에서 생성자 파라미터 사용 가능
        Log.d("class", "생성자로부터 전달받은 value의 값은 ${value} 이다.")
    }
}
```

7.2 클래스 코드 작성하기(계속)

· 세컨더리 생성자 secondary constructor

예 class Person {
```
    constructor (value: String) {
        Log.d("class", "생성자에서 전달 받은 값은 ${value}이다.")
    }
}
```

예 class Person {
```
    constructor(value: String) {     // 세컨더리 생성자 1
        Log.d("class", "생성자에서 전달 받은 값은 ${value}이다.")
    }
    constructir(value: Int) {          // 세컨더리 생성자 2
        Log.d("class", "생성자에서 전달 받은 값은 ${value}이다.")
    }
    constructir(value1: Int, value2: String) {          // 세컨더리 생성자 3
        Log.d("class", "생성자에서 전달 받은 값은 ${value1}, ${value2}이다.")
    }
}
```

· 하나 이상의 생성자가 필요할 때 세컨더리 생성자 사용

7.2 클래스 코드 작성하기(계속)

· 기본 생성자 default constructor

- 프라이머리 생성자를 생략하면, 파라미터가 없는 기본 생성자가 자동으로 적용

예
```
class Student {    // constructor가 없으면 기본 생성자 적용
    init {
        // 클래스 변수 초기화 코드
        // 프라이머리 생성자 또는 세컨더리 생성자
    }
}
```

7.3 클래스 사용

· 클래스의 인스턴스 생성

예
```
val kotlin = MyString()    // 기본 생성자를 이용한 객체 생성
Log.d("class", "객체 내부의 변수 length의 값은 ${kotlin.length}이다.)   // . 도트 연산자 사용
Log.d("class", "객체 내부의 함수 호출은 ${kotlin.compareTo(₩"문자열₩")} 형식이다.)  // 도트 연산자
```

7.4 오브젝트 object

· 오브젝트의 기본 구조

```
object 오브젝트명 {
    var 변수                        // 멤버 변수
    func 함수 {                     // 멤버 함수
      함수에서 실행할 코드
    }
}
```

예

```
object Pig {
    var name: String = "Pinky"
    fun printName() {
        Log.d("object", "Pig의 이름은 ${name}이다.")
    }
}
```

· 인스턴스 생성 없이 바로 사용 가능
· java에서 static 개념
· 앱 내에서 1개의 object만 선언 가능

7.4 오브젝트 object(계속)

· companion object
  - 클래스 내부에서 object를 사용
  - object의 멤버는 바로 사용 가능
  - object의 멤버가 아닌 class 멤버는 인스턴스 생성후 인스턴스를 통해 사용 가능

예 class Pig {
```
        companion object {
          var name: String = "Pinky"
          fun printName() {
              Log.d("class", "Pig의 이름은 ${name}이다.")
          }
        }  // companion object
        fun walk() {
          Log.d("class", "Pig 클래스의 walk 함수 호출되다.")
    }
    // Pig 클래스의 인스턴스 생성
    var myPig = Pig()
    Log.d("class", "companion object의 변수 name의 값은 ${Pig.name}이다.")
    Log.d("class", "companion object의 함수 호출 결과는 ${Pig.printName}이다.")
    Log.d("class", "myPig 객체 인스턴트의 walk 함수 호출 결과는 ${myPig.walk()}이다.")
```

7.5 data class

· 간단한 값의 저장 용도로 사용

예
```
data class UserData(val name: String, var age: Int)  // data class 정의
var userData = UserData("동양", 21)                  // data class 사용
userData.name = "컴퓨터"                  // 오류, name이 val로 선언되었기 때문에 immutable
userData.age = 18                         // age가 var로 선언되었기 때문에 mutable

Log.d("class", "UserData의 값은 ${userData.toString()}")  // userData의 값 확인
var newUserData = userData.copy()                         // userData 복사
```

예
```
data class UserData2(var name: String, var age: Int) {
    init {
        Log.d("dataclass", "초기화 실행")
    }
    fun process() {
        // 클래스와 동일하게 메서드 사용 가능
    }
}
// 사용
var userData2 = UserData2("동양", 21)      // 이 때 init 실행으로 문자열 출력
```

7.6 클래스의 상속과 확장

· 클래스 상속의 기본 형식

```
open class 부모클래스 {
        // 코드
}

class 자식클래스: 부모클래스 {
        // 코드
}
```

예

```
open class Activity {
    fun drawText() { ... }
    fun draw() { ... }
    fun showWindow() { ... }
}
```

```
class MainActivity: Activity {
    fun onCreate() {
        draw("새그림")
    }
}
```

부모클래스 Activity에서
상속받은 draw() 함수 사용

7.6 클래스의 상속과 확장(계속)

· 자식 클래스와 부모 클래스가 파라미터를 갖는 경우

예

```
open class Parent(value: String) {
    var value: String
    init {
        this.value = value
    }
    init {
        Log.d("inherit", "부모클래스 초기화, 변수 value의 값은 ${this.value}")
    }
}
```

```
class Child(value1: String, value2: Int): Parent(value1) {
    var valueCS: String
    var valueCI: Int
    init {
        this.valueCS = value1
        this.valueCI = value2
        Log.d("inherit", "자식클래스 초기화, 자식 클래스의 변수 valueCS의 값은 ${this.valueCS}")
        Log.d("inherit", "자식크랠스 초기화, 부모 클래스의 변수 value의 갑은 ${super.value}")
}
```

7.6 클래스의 상속과 확장(계속)

· 프로퍼티와 메서드 오버라이드 override

예

```
open class BaseClass {
    open var opened: String = "I am"
    open func openedFun() { ... }
    func notOpenedFunc() { ... }
}

class DerivedClass {
    override var opened: String = "You are"
    var anotherStr: String = "They are"
    override fun openedFun() { ... }
    fun anotherFun() { ... }
}
```

· 익스텐션: 클래스 확장

예 `fun DerivedClass.extFunc() { ... }`

```
// 클래스 DerivedClass에 extFunc() 함수를 실행할 때만 추가
// 클래스의 실제 내용이 변경되는 것은 아님
// DrivedClass: 확장할 클래스 이름
// extFunc(): 지정된 클래스에 실행 시간에만 추가하는 함수
```

7.7 설계도구

· 패키지 package
  - 기본 형식

```
package 메인디렉터리.서브디렉터리
class Sub { }
```

· 추상화 abstract / 인터페이스 interface
  - 클래스를 정의할 때 명확한 동작이 지정되지 않은 경우 메서드의 코드를 기술할 수 없다.
  - 메서드 이름만 갖는 클래스를 정의: abstrace, interface

예

```
abstract class Design {
    abstract fun drawText()
    abstract fun draw()
    fun showWindow() {
        // 코드
    }
}
```

```
class implements: Design {
    fun drawText() {
        // 구현 코드
    }
    fun draw() {
        // 구현 코드
    }
}
```

예

```
abstract class Animal {
    fun walk() {   // 명확한 행위
        Log.d("abstract", "걷는다."abstract fun drawText()
    }
    abstract fun move()   // 불명확한 행위
}
```

```
class Bird: Animal {
    override fun move() {
        Log.d("abstrace", "날아서 이동한다.")
    }
}
```

예

```
interface InterfaceKotlin {
    abstract var variable: String
    abstract fun get()
    abstract fun set()
}
```

```
class KotlinImpl: interfaceKotlin {
    override var variable: String = "init value"
    override fun get() {
        // 코드 구현
    }
    override fun set() {
        // 코드 구현
    }
}
```

예

```
interface Clickable {
    fun click()
    fun showOff() = "Log.d("interface", "기본으로 할당된 함수")
}
```

```
class Button: Clickable {
    override fun click() {
        Log.d("interface", "interface에서 받은 함수를 구현한 것")
    }
}
```

```
// interface를 구현한 class 사용
...
var btn = Button()
btn.click()          // intrface에서 받아 구현한 함수 호출
btn.showOff()    // interface에 정의된 함수 실행
...
```

7.7 설계도구(계속)

· 접근 제한자
  - 기존 객체 지향에서 접근 제한자의 기준으로 패키지
  - 함수형 언어인 코틀린에서는 패키지 대신 모듈

접근 제한자	제한 범위
private	다른 파일에서 접근할 수 없다.
internal	같은 모듈에 있는 파일만 접근할 수 있다.
protected	private와 같으나 상속 관계에서 자식 클래스가 접근할 수 있다.
public	제한 없이 모든 파일에서 접근할 수 있다.(기본)

예

예

```
open class Parent {
    private val privateVal = 1
    protected open val protectedVal = 2
    internal val internalVal = 3
    var defaultVal = 4
}
```

```
class Child: Parent {
    fun callVariables() {
        // priavateVal은 호출 불가
        Log.d("modifier", "Parent의 protected 변수값은 ${protectedVal} 이다.")
        Log.d("modifier", "Parent의 internal 변수 값은 ${internalVal} 이다.")
        Log.d("modifier", "Parent의 public 변수 값은 ${defaultVal} 이다.")
    }
}
```

예

```
open class Parent {
    private val privateVal = 1
    protected open val protectedVal = 2
    internal val internalVal = 3
    var defaultVal = 4
}
```

```
class Stranger {
    fun callVAriables() {
        var parent = Parent()
        // priavateVal은 호출 불가
        // protectedVal은 호출 불가
        Log.d("modifier", "Parent class의 internal 변수 값은 ${internalVal} 이다.")
        Log.d("modifier", "Parent class의 public 변수 값은 ${defaultVal} 이다.")
    }
}
```

7.7 설계도구(계속)

· 제네릭 generic
  - 입력되는 값의 타입을 자유롭게 사용하기 위한 설계 도구
  예 public interface MutableList<E> { // Element type을 E로 표현
      var list: Array<E>
      ....
    }

  예 var list: MutableList<E> = mutableListOf("월", "화", "수") // E는 String이 됨

8.0 null 값에 대한 안정적인 처리: Null Safety

- kotlin은 null 처리에 많은 공을 들인 언어
- null로 인해 프로그램(앱)이 멈출 수 있음

예
```
class One {
    fun print() {
        Log.d("null_safety", "can you call me")
    }
}

var one: One
if(1 > 2) {
    one = One()    // One 객체(인스턴스) 생성
}
one.print          // One 객체가 생성되어 있지 않아 실행 불가
                   // one은 null
                   // android의 경우 Android Studio에서 막아준다.
```

8.1 null 값 허용하기: ?

- kotlin에서 변수는 null이 입력되지 않는다.
- null 값을 허용하려면

```
예 var variable: String?          // 타입 뒤에 "?"를 붙이면 Nullable의 의미, 즉 null 저장 허용
   variable = null                // 변수 variable에 null 저장하기
   var notNullvar: String
   notNullvar = null              // Null can not be a value of a non-null type String
```

- 함수 파라미터에 null 허용하기

```
예 fun nullParameter(str: String?) {
       var length2 =str.length      // Only safe (?.) or non-null asserted (!!.) calls are
                                    // allowed on a nullable receiver of type String?
   }

   fun nullParameter(str: String?) {
     if(str != null) {              // 반드시 null이 아닐 때 동작할 수 있도록 if 문을 사용해야 한다.
        var length2 = str.length
     }
   }
```

8.1 null 값 허용하기: ? (계속)

- 함수의 리턴 타입에 null 허용하기

예
```
fun nullReturn(): String? {
    return null
}
```

8.2 안전한 호출: ?.

예
```
fun testSafeCall(str: String?): Int? {
    // str이 null이면 length를 체크하지 않고 null을 반환한다.
    var resultNull: Int? = str?.length
    return resultNull
}
```

8.3 null 값 대체하기: ?:

예
```
fun testElvisOperatorr(str: String?): Int {
    var resultNonNull: Int = str?.length:0
    return resultNonNull
}
```

9.1 지연 초기화 lateinit

- 객체(데이터)의 초기화가 늦게 이루어질 것임을 알려주 것

예
```
fun process() {
    lateinit var text: String
    // …
    var result1 = 30
    text = "Result : $result1"
    Log.d("lateinit", "변수 text의 값은 ${text}이다.")
    // …
    var result2 = 50
    text = "Result : ${result1 + result2}이다."
    Log.d("lateinit, "변수 text의 값은 ${text}이다.")
}
```

9.2 지연 초기화 lazy

예
```
fun lazyProcess() {
    lateinit var text: String
    val textLength: Int by lazy {
        text.length
    }
    // …
    text = "임의의 문자열 할당"
    Log.d("lateinit", "변수 text의 길이는 ${textLength}이다.")
}
```

10.1 run과 let으로 보는 스코프 함수

· run과 let은 자신의 함수 스코프(함수의 코드 블록) 안에서 호출한 대상을 this와 it로 대체해서 사용할 수 있다.

예
```
var list1 = mutableListOf("Scope", "Function")
list1.run {
    val listSize = size     // this.size를 사용한 것과 같다.
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}
```

예
```
var list2 = mutableListOf("Scope", "Function")
list2.let {
    val listSize = it.size     // it 대신에 target 사용 가능
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}

list2.let { target ->
    val listSize = target.size     // it 대신에 target 사용 가능
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}
```

10.2 this와 it으로 구분하기

- 스코프 함수에서 자신을 호출하는 대상을 this 또는 it로 대체해서 사용
- this로 사용되는 스코프 함수: run, apply, with

예
```
var list = mutableListOf("Scope", "Function")
list.apply {
    val listSize = size     // this.size를 사용한 것과 같다.
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
    Log.d("scope", "리스트의 첫 번째 항목은 ${this[0]}이다.")
}

with(list)
    val listSize = size     // this.size를 사용한 것과 같다.
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}
```

- 호출 대상이 null일 때는 with보다는 apply 또는 run을 사용하는 것이 효율적

10.2 this와 it으로 구분하기(계속)

- it 사용되는 스코프 함수: let, also

예
```
var list = mutableListOf("Scope", "Function")
list.let { target ->
    val listSize = target.size
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}

list.also {
    val listSize = it.size
    Log.d("scope", "리스트의 길이는 ${listSize}이다.")
}
```

10.3 반환값으로 구분하기

- 호출 대상인 this 자체를 반환하는 스코프 함수: apply. also
- apply, also를 실행하면 this 자체를 반환한다.

예
```
var list = mutableListOf("Scope", "Function")
var afterApply = list.apply {
  add("Apply")
  count()     // this의 element 개수를 반환하는 것이 아니라, this 자체를 반환
}
Log.d("scope", "list는 ${afterApply}")

var afterAlso = list.also {
  it.add("Also")
  it.count()     // it의 element 개수를 반환하는 것이 아니라, this 자체를 반환
}
Log.d("scope", "list는 ${afterAlso}")
```

10.3 반환값으로 구분하기(계속)

- 마지막 실행 코드의 결과를 반환하는 스코프 함수: let, run, with

예 var list = mutableListOf("Scope", "Function")

```
val lastCount = list.let {
    it.add("Let")
    it.count()
}
Log.d("scope", "let의 마지막 결과는 ${lastCount}")

val lastItem = list.run {
    add("Run")
    get(size - 1)
}
Log.d("scope", "run의 마지막 항목은 ${lastItem}")

val lastItemWith = with(list) {
    add("With")
    get(size - 1)
}
Log.d("scope", "with의 마지막 항목은 ${lastItemWith}")
```