

세상의 속도를
따라잡고 싶다면

**Do
it!**

깡샘의
안드로이드
앱 프로그래밍
with **코틀린**

이지스퍼블리싱(주)

04

코틀린 객체지향 프로그래밍



04-1 클래스와 생성자

04-2 클래스를 재사용하는 상속

04-3 코틀린의 클래스 종류

04-1 클래스와 생성자

클래스 선언

- 클래스는 class 키워드로 선언
- 클래스의 본문에 입력하는 내용이 없다면 {}를 생략
- 클래스의 멤버는 생성자, 변수, 함수, 클래스로 구성
- 생성자는 constructor라는 키워드로 선언하는 함수

```
• 클래스의 멤버

class User {
    var name = "kkang"
    constructor(name: String) {
        this.name = name
    }
    fun someFun() {
        println("name : $name")
    }
    class SomeClass { }
}
```

04-1 클래스와 생성자

- 객체를 생성해 사용하며 객체로 클래스의 멤버에 접근
- 객체를 생성할 때 new 키워드를 사용하지 않습니다.

• 객체 생성과 멤버 접근

```
val user = User("kim")  
user.someFun()
```

04-1 클래스와 생성자

주 생성자

- 생성자를 주 생성자와 보조 생성자로 구분
- 주 생성자는 constructor 키워드로 클래스 선언부에 선언합니다.
- 주 생성자 선언은 필수는 아니며 한 클래스에 하나만 가능합니다.
- constructor 키워드는 생략할 수 있습니다.

• 주 생성자 선언

```
class User constructor() {  
}
```

• constructor 키워드 생략 예

```
class User() {  
}
```

• 매개변수가 없는 주 생성자 자동 선언

```
class User {  
}
```

04-1 클래스와 생성자

- 주 생성자의 본문 — init 영역
 - init 키워드를 이용해 주 생성자의 본문을 구현할 수 있습니다.
 - init 키워드로 지정한 영역은 객체를 생성할 때 자동으로 실행

• init 키워드로 주 생성자의 본문 지정

```
class User(name: String, count: Int) {  
    init {  
        println("i am init...")  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
}
```

▶ 실행 결과

i am init...

04-1 클래스와 생성자

- 생성자의 매개변수를 클래스의 멤버 변수로 선언하는 방법
 - 생성자의 매개변수는 기본적으로 생성자에서만 사용할 수 있는 지역 변수

• 생성자의 매개변수를 init 영역에서 사용하는 예

```
class User(name: String, count: Int) {  
    init {  
        println("name : $name, count : $count")    // 성공!  
    }  
    fun someFun() {  
        println("name : $name, count : $count")    // 오류!  
    }  
}
```

04-1 클래스와 생성자

- 매개변수를 var나 val 키워드로 선언하면 클래스의 멤버 변수

• 생성자의 매개변수를 클래스의 멤버 변수로 선언하는 방법

```
class User(val name: String, val count: Int) {  
    fun someFun() {  
        println("name : $name, count : $count")    // 성공!  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
    user.someFun()  
}
```

▶ 실행 결과

name : kkang, count : 10

04-1 클래스와 생성자

보조 생성자

- 보조 생성자는 클래스의 본문에 constructor 키워드로 선언하는 함수
- 여러 개를 추가할 수 있습니다.

• 보조 생성자 선언

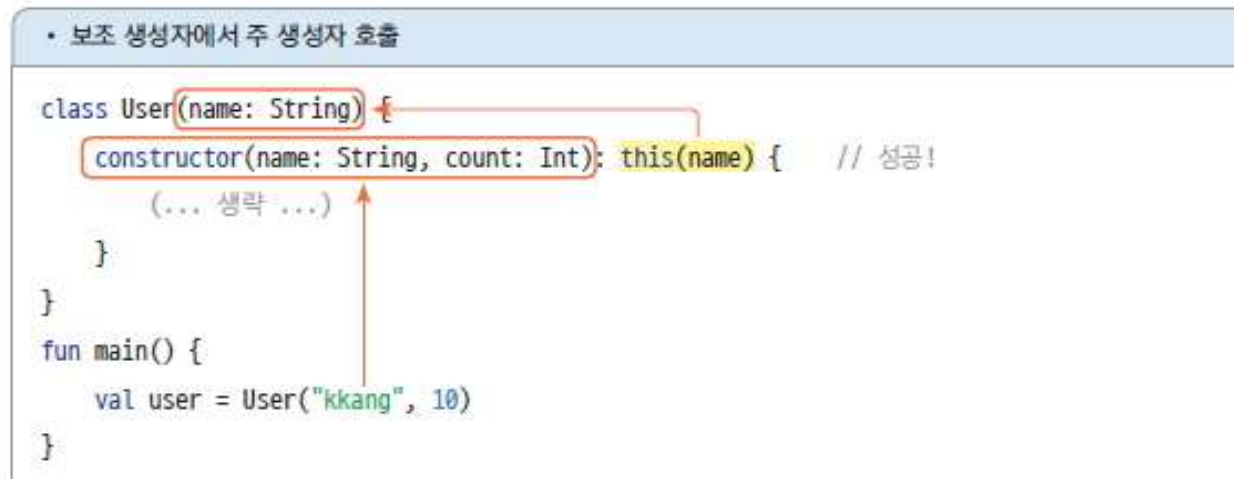
```
class User {  
    constructor(name: String) {  
        println("constructor(name: String) call...")  
    }  
    constructor(name: String, count: Int) {  
        println("constructor(name: String, count: Int) call...")  
    }  
}  
  
fun main() {  
    val user1 = User("kkang")  
    val user2 = User("kkang", 10)  
}
```

04-1 클래스와 생성자

- 보조 생성자에 주 생성자 연결
 - 보조 생성자로 객체를 생성할 때 클래스 내에 주 생성자가 있다면 `this()` 구문을 이용해 주 생성자를 호출해야 합니다.

• 보조 생성자에서 주 생성자 호출

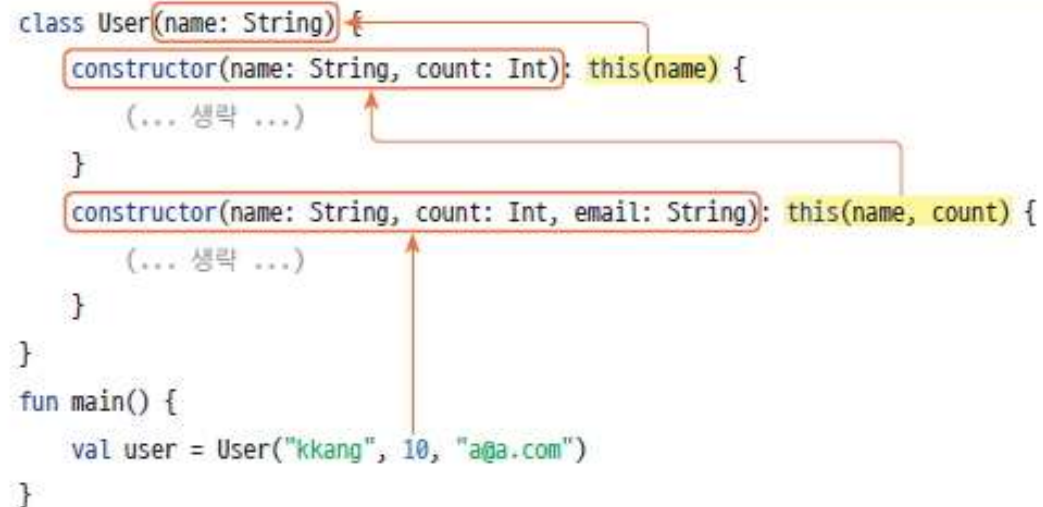
```
class User(name: String) {  
    constructor(name: String, count: Int): this(name) { // 성공!  
        (... 생략 ...)  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10)  
}
```

A diagram illustrating the relationship between the primary constructor and the auxiliary constructor. A red box highlights the primary constructor `class User(name: String) {`. Another red box highlights the auxiliary constructor `constructor(name: String, count: Int): this(name) {`. A red arrow points from the `this(name)` call in the auxiliary constructor to the opening curly brace of the primary constructor. A brown arrow points from the `val user = User("kkang", 10)` line in the `main` function to the auxiliary constructor.

04-1 클래스와 생성자

• 보조 생성자가 여럿일 때 생성자 연결

```
class User(name: String) {  
    constructor(name: String, count: Int): this(name) {  
        (... 생략 ...)  
    }  
    constructor(name: String, count: Int, email: String): this(name, count) {  
        (... 생략 ...)  
    }  
}  
  
fun main() {  
    val user = User("kkang", 10, "a@a.com")  
}
```



04-2 클래스를 재사용하는 상속

상속과 생성자

- 코틀린에서 어떤 클래스를 상속받으려면 선언부에 콜론(:)과 함께 상속받을 클래스 이름을 입력
- 코틀린의 클래스는 기본적으로 다른 클래스가 상속할 수 없습니다.
- 다른 클래스에서 상속할 수 있게 선언하려면 open 키워드를 사용
- 상위 클래스를 상속받은 하위 클래스의 생성자에서는 상위 클래스의 생성자를 호출해야 합니다.

• 매개변수가 있는 상위 클래스의 생성자 호출

```
open class Super(name: String) {  
}  
class Sub(name: String): Super(name) {  
}
```

• 하위 클래스에 보조 생성자만 있는 경우 상위 클래스의 생성자 호출

```
open class Super(name: String) {  
}  
class Sub: Super {  
    constructor(name: String): super(name) {  
    }  
}
```

04-2 클래스를 재사용하는 상속

오버라이딩 - 재정의

- 상속이 주는 최고의 이점은 상위 클래스에 정의된 멤버(변수, 함수)를 하위 클래스에서 자신의 멤버처럼 사용할 수 있다는 것

• 상속 관계인 두 클래스

```
open class Super {  
    var superData = 10  
    fun superFun() {  
        println("i am superFun : $superData")  
    }  
}  
class Sub: Super()  
fun main() {  
    val obj = Sub()  
    obj.superData = 20  
    obj.superFun()  
}
```

04-2 클래스를 재사용하는 상속

- 상위 클래스에 선언된 변수나 함수를 같은 이름으로 하위 클래스에서 다시 선언하는 것을 오버라이딩이라고 합니다.

• 오버라이딩 예

```
open class Super {  
    open var someData = 10  
    open fun someFun() {  
        println("i am super class function : $someData")  
    }  
}  
  
class Sub: Super() {  
    override var someData = 20  
    override fun someFun() {  
        println("i am sub class function : $someData")  
    }  
}  
  
fun main() {  
    val obj = Sub()  
    obj.someFun()  
}
```

▶ 실행 결과

i am sub class function : 20

04-2 클래스를 재사용하는 상속

접근 제한자

- 접근 제한자란 클래스의 멤버를 외부의 어느 범위까지 이용하게 할 것인지를 결정하는 키워드

접근 제한자	최상위에서 이용	클래스 멤버에서 이용
public	모든 파일에서 가능	모든 클래스에서 가능
internal	같은 모듈 내에서 가능	같은 모듈 내에서 가능
protected	사용 불가	상속 관계의 하위 클래스에서만 가능
private	파일 내부에서만 이용	클래스 내부에서만 이용

04-2 클래스를 재사용하는 상속

• 접근 제한자 사용 예

```
open class Super {  
    var publicData = 10  
    protected var protectedData = 20  
    private var privateData = 30  
}  
  
class Sub: Super() {  
    fun subFun() {  
        publicData++    // 성공!  
        protectedData++ // 성공!  
        privateData++   // 오류!  
    }  
}  
  
fun main() {  
    val obj = Super()  
    obj.publicData++    // 성공!  
    obj.protectedData++ // 오류!  
    obj.privateData++   // 오류!  
}
```


04-3 코틀린의 클래스 종류

데이터 클래스

- 데이터 클래스는 data 키워드로 선언
- 데이터 클래스는 VO 클래스를 편리하게 이용할 수 있는 방법 제공

• 데이터 클래스 선언

```
class NonDataClass(val name: String, val email: String, val age: Int)
```

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

04-3 코틀린의 클래스 종류

- 객체의 데이터를 비교하는 equals() 함수

• 데이터 클래스 객체 생성

```
fun main() {  
    val non1 = NonDataClass("kkang", "a@a.com", 10)  
    val non2 = NonDataClass("kkang", "a@a.com", 10)  
  
    val data1 = DataClass("kkang", "a@a.com", 10)  
    val data2 = DataClass("kkang", "a@a.com", 10)  
}
```

• 객체의 데이터를 비교하는 equals() 함수

```
println("non data class equals : ${non1.equals(non2)}")  
println("data class equals : ${data1.equals(data2)}")
```

▶ 실행 결과

```
non data class equals : false  
data class equals : true
```

04-3 코틀린의 클래스 종류

- equals() 함수는 주 생성자에 선언한 멤버 변수의 데이터만 비교 대상으로 삼습니다.

• 데이터 클래스의 equals() 함수

```
data class DataClass(val name: String, val email: String, val age: Int) {  
    lateinit var address: String  
    constructor(name: String, email: String, age: Int, address: String):  
        this(name, email, age) {  
            this.address = address  
        }  
}  
  
fun main() {  
    val obj1 = DataClass("kkang", "a@a.com", 10, "seoul")  
    val obj2 = DataClass("kkang", "a@a.com", 10, "busan")  
    println("obj1.equals(obj2) : ${obj1.equals(obj2)}")  
}
```

▶ 실행 결과

```
obj1.equals(obj2) : true
```

04-3 코틀린의 클래스 종류

- 객체의 데이터를 반환하는 toString() 함수
 - 데이터 클래스를 사용하면서 객체가 가지는 값을 확인해야 할 때 이용

• 데이터 클래스의 toString() 함수

```
fun main() {  
    class NonDataClass(val name: String, val email: String, val age: Int)  
    data class DataClass(val name: String, val email: String, val age: Int)  
    val non = NonDataClass("kkang", "a@a.com", 10)  
    val data = DataClass("kkang", "a@a.com", 10)  
    println("non data class toString : ${non.toString()}")  
    println("data class toString : ${data.toString()}")  
}
```

▶ 실행 결과

```
non data class toString : com.example.test4.ch2.Test2Kt$main$NonDataClass@61bbe9ba  
data class toString : DataClass(name=kkang, email=a@a.com, age=10)
```

04-3 코틀린의 클래스 종류

오브젝트 클래스

- 오브젝트 클래스는 익명 클래스를 만들 목적으로 사용
- 선언과 동시에 객체를 생성한다는 의미에서 object라는 키워드를 사용

```
• 오브젝트 클래스 사용 예

val obj = object {
    var data = 10
    fun some() {
        println("data : $data")
    }
}

fun main() {
    obj.data = 20    // 오류!
    obj.some()      // 오류!
}
```

04-3 코틀린의 클래스 종류

- 오브젝트 클래스의 타입은 object 뒤에 콜론(:)을 입력하고 그 뒤에 클래스의 상위 또는 인터페이스를 입력합니다.

```
• 타입을 지정한 오브젝트 클래스

open class Super {
    open var data = 10
    open fun some() {
        println("i am super some() : $data")
    }
}

val obj = object: Super() {
    override var data = 20
    override fun some() {
        println("i am object some() : $data")
    }
}

fun main() {
    obj.data = 30    // 성공!
    obj.some()      // 성공!
}
```

▶ 실행 결과

```
i am object some() : 30
```

04-3 코틀린의 클래스 종류

컴패니언 클래스

- 컴패니언 클래스는 멤버 변수나 함수를 클래스 이름으로 접근하고자 할 때 사용
- companion이라는 키워드로 선언

• 컴패니언 클래스의 멤버 접근

```
class MyClass {  
    companion object {  
        var data = 10  
        fun some() {  
            println(data)  
        }  
    }  
}  
  
fun main() {  
    MyClass.data = 20    // 성공!  
    MyClass.some()      // 성공!  
}
```



감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.



윌리엄 셰익스피어
William Shakespeare