

Kotlin 클래스

박연종

20210463

park@duck.com

2024.10.16.

SSL SEMINAR



- 00 지난 시간
- 01 클래스
- 02 상속
- 03 속성
- 04 가시성 변경자
- 05 위임 패턴과 by 키워드

00

지난 시간

1급 객체 (First-class citizen_[,type,object,entity,value])

- 공정한 대우를 받는 집단의 구성원
- Christopher Strachey에 의해 고안되어 Robin Popplestone의 정의가 널리 퍼짐
- 모든 1급 객체는 함수의 실질적인 매개변수가 될 수 있다.
- 모든 1급 객체는 함수의 반환 값이 될 수 있다.
- 모든 1급 객체는 할당의 대상이 될 수 있다.
- 모든 1급 객체는 비교 연산을 적용할 수 있다.
- 함수가 1급 객체인 프로그래밍 언어는 Dart, Kotlin, Swift, Python, JavaScript, Rust 등이 있음
- Java에서는 람다식을 통해 메서드가 1급 객체로 다뤄짐

1급 객체 (First-class citizen_[,type,object,entity,value])

- 모든 1급 객체는 비교 연산을 적용할 수 있다.

```
1 fun hello() {  
2     println("Hello, World!")  
3 }  
4  
5 fun main() {  
6     val funcRef1 = ::hello  
7     val funcRef2 = ::hello  
8  
9     println(funcRef1 == funcRef2) // 출력: true (함수의 해시는 동일하기 때문)  
10    println(funcRef1 === funcRef2) // 출력: false (참조의 해시가 다르기 때문)  
11 }
```

1급 객체 (First-class citizen_[,type,object,entity,value])

- 모든 1급 객체는 비교 연산을 적용할 수 있다.

```
1 fun main() {  
2     ...val funcRef1 = { println("Hello, World!") }  
3     ...val funcRef2 = { println("Hello, World!") }  
4  
5     ...println(funcRef1 == funcRef2) // 출력: false (함수의 해시가 다르기 때문)  
6     ...println(funcRef1 === funcRef2) // 출력: false (참조의 해시가 다르기 때문)  
7 }
```

01 클래스

Classes



```
1 class Person { /* ... */ }
```


```
2
```

```
3 class Empty
```



```
1 class Person constructor(firstName: String) { /* ... */ }
2
3 class Person(firstName: String) { /* ... */ }
4
5 class InitOrderDemo(name: String) {
6     val firstProperty = "First property: $name"
7
8     init {
9         println("First initializer block that prints $name")
10    }
11
12    val secondProperty = "Second property: ${name.length}"
13
14    init {
15        println("Second initializer block that prints ${name.length}")
16    }
17 }
```

```
1 /* [실행 결과] (인자: "PARK, Yeonjong")
2 * First initializer block that prints PARK, Yeonjong
3 * firstProperty = PARK, Yeonjong, secondProperty = 14
4 * Second initializer block that prints 14
5 */
```



```
1 class Customer public @Inject constructor(name: String) { /* ... */ }
```



```
1 class Pet {  
2     ... constructor(owner: Person) { /* ... */ }  
3 }  
4  
5 class Person(val name: String = "PARK, Yeonjong") {  
6     ... init {  
7         ... println("Init block")  
8     }  
9     ...  
10    ... constructor(name: String, parent: Person) : this(name) { /* ... */ }  
11 }
```



```
1 val invoice = Invoice()↵  
2 ↵  
3 val customer = Customer("Joe Smith")
```

02

상속

Inheritance

```
public open class Any {
```

Indicates whether some other object is "equal to" this one. Implementations must fulfil the following requirements:

- Reflexive: for any non-null value `x`, `x.equals(x)` should return true.
- Symmetric: for any non-null values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- Transitive: for any non-null values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- Consistent: for any non-null values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in `equals` comparisons on the objects is modified.
- Never equal to null: for any non-null value `x`, `x.equals(null)` should return false.

Read more about equality [↗](#) in Kotlin.

```
public open operator fun equals(other: Any?): Boolean
```

Returns a hash code value for the object. The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified.
- If two objects are equal according to the `equals()` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

```
public open fun hashCode(): Int
```

Returns a string representation of the object.

```
public open fun toString(): String
```

```
}
```

- 자바의 Object 클래스에 대응됨
- 세 가지 메서드를 가지고 있음
- Object의 다른 메서드 원하면 캐스팅
- 자바 바이트 코드를 자바 코드로 변환하면
Object 타입으로 변환됨

상속 가능한 클래스로 변경하기

```
1 class Base(p: Int) {  
2  
3 class Derived(p: Int) : Base(p)
```

This type is final, so it cannot be inherited from

Make 'Base' 'open' Alt+Shift+Enter

More actions... Alt+Enter

```
public constructor Base(  
    val p: Int  
)
```

Base

learnKotlin

```
1 open class Base(p: Int) {  
2  
3 class Derived(p: Int) : Base(p)
```

```
1 open class Shape {  
2     ... open fun draw() { /* ... */ }  
3     ... fun fill() { /* ... */ }  
4 }  
5  
6 class Circle() : Shape() {  
7     ... override fun draw() { /* ... */ }  
8 }
```

'fill' in 'Shape' is final and cannot be overridden



Make Shape.fill open Alt+Shift+Enter

More actions... Alt+Enter

기반 클래스에서 val로 선언된 변수는 파생 클래스에서 var로 재정의할 수 있지만 그 역은 할 수 없음

```
1 interface Shape {  
2     ... val vertexCount: Int  
3 }  
4  
5 class Rectangle(override val vertexCount: Int = 4) : Shape  
6  
7 class Polygon : Shape {  
8     ... override var vertexCount: Int = 0  
9 }
```

내부 클래스에서 외부 클래스 호출

```
1 open class Rectangle {  
2     open fun draw() { println("Drawing a rectangle") }  
3     val borderColor: String get() = "black"  
4 }  
5  
6 class FilledRectangle: Rectangle() {  
7     override fun draw() {  
8         val filler = Filler()  
9         filler.drawAndFill()  
10    }  
11  
12    inner class Filler {  
13        fun fill() { println("Filling") }  
14        fun drawAndFill() {  
15            super@FilledRectangle.draw()  
16            fill()  
17            println("Drawn a filled rectangle with color: ${super@FilledRectangle.borderColor}")  
18        }  
19    }  
20 }
```

Java에서는 정규화된 this를 사용해 호출 가능
GilledRectangle.this.fill()

03 속성

Properties

변환된 자바 바이트 코드를 자바 코드로 변환하면 Java에서 자주 보던 설정자(Setter)와 획득자(Getter)를 확인 가능

```
1 var <propertyName>[: <PropertyType>] [= <property_initializer>] ↵  
2 ... [<getter>] ↵  
3 ... [<setter>]
```



```
1 var stringRepresentation: String {  
2     get() = this.toString()  
3     set(value) {  
4         setDataFromString(value)  
5     }  
}
```



```
1 var stringRepresentation: String {  
2     public get() = this.toString()  
3     protected @Inject set(value) {  
4         setDataFromString(value)  
5     }  
}
```

설정자의 매개변수 이름은 `value`가 아닌 다른 이름으로 설정할 수도 있음

```
1 var counter = 0
2 ...set(value) {
3     ... if (value ≥ 0)
4     ...     field = value
5     ...     // counter = value는 counter의 설정자를 반복 호출해 StackOverflow 예외가 발생
6 ... }
```

04

가시성 변경자

Visibility modifiers

최상위 수준 변수에 적용된 가시성 변경자

```
1 package foo
2
3 private fun foo() { ... } // 동일한 kt 파일 내에서만 사용할 수 있는 메서드
4
5 public var bar: Int = 5 // 모든 패키지에서 획득자로 획득 가능
6 ... private set ... // 동일한 kt 파일 내에서만 값을 지정 가능
7
8 internal val baz = 6 ... // 동일한 모듈 내에서만 값을 확인할 수 있음 (val이므로 설정자가 없음)
```


클래스 내 메서드에 가시성 변경자 지정

```
1 open class Outer {  
2     private val a = 1 // 동일 클래스에서 사용 가능  
3     protected open val b = 2 // 동일 클래스, 파생 클래스에서 사용 가능  
4     internal open val c = 3 // 같은 모듈 내에서 사용 가능  
5     // Java의 기본 접근 지정자는 동일 패키지 내 클래스에서만 사용 가능한 default  
6     // Kotlin의 기본 가시성 변경자는 public로 모든 클래스에서 사용 가능  
7     val d = 4  
8  
9     protected class Nested {  
10         public val e: Int = 5  
11     }  
12 }  
13  
14 class Subclass() {  
15     // 기본 클래스의 a는 Outer에서만 획득 가능  
16     // b, c, d는 파생 클래스 Subclass에서 획득할 수 있음  
17     // Nested, e는 파생 클래스 Subclass에서 획득할 수 있음  
18  
19     override val b = 5  
20     override val c = 7  
21 }  
22  
23 class Unrelated(o: Outer) {  
24     // o.a, o.b는 사용할 수 없음  
25     // o.c는 같은 모듈에서, o.d는 모든 곳에서 사용할 수 있음  
26     // Outer.Nested와 Nested::e는 사용할 수 없음  
27 }
```

가시성 변경자	최상위 층	클래스 층
public	모든 .kt 파일	모든 .kt 파일
internal	동일 모듈 내 모든 .kt 파일	동일 모듈 내 모든 .kt 파일
protected	-	해당 클래스, 파생 클래스
private	해당 .kt 파일	해당 클래스

05

위임 패턴과 by 키워드

- 상속이 슈퍼 클래스의 구현이 서브 클래스에 그대로 노출돼 캡슐화를 파괴한다는 시각이 있음
- 슈퍼 클래스가 변경되면 서브 클래스도 변경해야 함
- 상속의 깊이가 깊어질수록 기능 파악이 어려움
- A가 B의 기능을 사용하고 싶을 때,
A가 B의 기능을 직접 구현하지 않고, C에게 위임하여 구현하는 방법
- C를 통해 A가 B의 기능을 사용할 수 있음
- 위임을 통해 여러 객체가 상호작용하므로 코드가 복잡해지고 오류 분석에 애로사항이 발생할 수 있음
- 추가적인 클래스를 작성해야 하므로 메모리와 시간을 더 필요함

```
1 interface IWindow {  
2     fun getWidth(): Int  
3     fun getHeight(): Int  
4 }  
5  
6 open class TransparentWindow : IWindow {  
7     override fun getWidth(): Int {  
8         return 100  
9     }  
10  
11     override fun getHeight(): Int {  
12         return 150  
13     }  
14 }  
15  
16 class UI(window: IWindow) : IWindow {  
17     val mWindow: IWindow = window  
18  
19     override fun getWidth(): Int {  
20         return mWindow.getWidth()  
21     }  
22  
23     override fun getHeight(): Int {  
24         return mWindow.getHeight()  
25     }  
26 }  
27  
28 fun main() {  
29     val window: IWindow = TransparentWindow()  
30     val ui = UI(window)  
31     System.out.println("Width: ${ui.getWidth()}, height: ${ui.getHeight()}")  
32 }  
33  
34 // Width: 100, height: 150
```

```
1 interface IWindow {  
2     fun getWidth(): Int  
3     fun getHeight(): Int  
4 }  
5  
6 open class TransparentWindow : IWindow {  
7     override fun getWidth(): Int {  
8         return 100  
9     }  
10  
11     override fun getHeight(): Int {  
12         return 150  
13     }  
14 }  
15  
16 class UI(window: IWindow) : IWindow by window  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28 fun main() {  
29     val window: IWindow = TransparentWindow()  
30     val ui = UI(window)  
31     System.out.println("Width: ${ui.getWidth()}, height: ${ui.getHeight()}")  
32 }  
33  
34 // Width: 100, height: 150
```

감사합니다

Email / park@duck.com

Insta / [@yeonjong.park](https://www.instagram.com/yeonjong.park)

GitHub / [patulus](https://github.com/patulus)

