4장 클래스, 객체, 인터페이스

신림프로그래머 최범균

- 클래스와 인터페이스
- 생성자와 프로퍼티
- 데이터 클래스
- 위임
- 오브젝트

인터페이스

인터페이스:

```
interface Clickable {
    fun click()
}
```

인터페이스 구현:

```
class Button : Clickable {
    override fun click() = println("I was clicked!")
}
```

- 클래스 이름 뒤에 세미콜론을 붙이고 상속할 클래스와 구현할 인터페이스 지정
  - 여러 인터페이스 구현 가능, 클래스는 한 개만 상속 가능
- override 수식어: 인터페이스나 상위 클래스의 메서드, 프로퍼티 재정의시 꼭 사용

디폴트 구현

디폴트 구현: 메서드 본문을 메서드 시그너처 뒤에 추가

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable")
}
```

동일 시그너처를 가진 디폴트 구현 상속시

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable")
}

interface Focusable {
    fun showOff() = println("I'm focusable!")
}
```

```
class Button : Clickable, Focusable {
    override fun click() = println("I was clicked")

    // 애매모호함을 없애기 위해 재정의함
    override fun showOff() {
        // super<타입>으로 사용할 상위 타입 지정
        super<Clickable>.showOff()
    }
}
```

- 코틀린의 디폴트 메서드는 자바의 정적 메서드로 구현 (코틀린은 자바 1.6과 호환)

open, final, abstract 수식어

- 클래스와 클래스의 멤버는 기본적으로 final
  - 상속을 허용하려면 클래스 앞에 open 수식어를 붙여야 함
  - 오버라이딩을 허용하고 싶은 메서드나 프로퍼티 앞에도 open 변경자를 붙여야 함
- 클래스를 abstract로 선언하면 추상 클래스
  - 추상 클래스는 인스턴스화할 수 없음
  - 추상 멤버는 항상 open임 (추상 멤버 앞에 open을 붙일 필요 없음)

클래스의 메서드는 기본으로 final

수식어	재정의	설명
final	오버라이딩 불가능	클래스 멤버의 기본 수식어
open	오버라이딩 가능	open을 명시해야 오버라이딩 가능
abstract	반드시 오버라이딩해야 함	추상 클래스의 멤버에만 붙일 수 있음, 구현이 있으면 안 됨
override	상위 클래스나 인스턴스의 멤버를 오버라이딩	오버라이딩하는 멤버는 기본으로 open. 오버라이딩을 금지하려면 final을 명시해야 함

가시성 수식어

수식어	클래스 멤버	최상위 선언
public	모든 곳에서 접근 가능	모든 곳에서 접근 가능
internal	같은 모듈에서만 접근 가능	같은 모듈에서만 접근 가능
protected	같은 클래스 및 하위 클래스 안에서만 접근 가능	(최상위 선언에 적용 불가)
private	같은 클래스 안에서만 접근 가능	같은 파일 안에서만 접근 가능

가시성 규칙

- 기반 타입 목록에 있는 타입은 자신보다 더 가시성이 같거나 넓어야 함
- 기반 타입의 제니릭 타입 파라미터에 있는 타입은 자신보다 가시성이 같거나 넓어야 함
- 메서드 시그너처에 사용된 모든 타입의 가시성은 메서드의 가시성과 같거나 넓어야 함

중첩 클래스

```
class Button : View {
    override fun getCurrentState(): State = ButtonState()
    override fun restoreState(state: State) { .... }

    class ButtonState : State { ... } // 자바의 정적 중첩 클래스에 대응
}
```

클래스 B 안에 정의된 클래스 A	자바에서는	코틀린에서는
중첩 클래스(바깥쪽 클래스에 대한 참조를 저장하지 않음)	static class A	class A
내부 클래스(바깥쪽 클래스에 대한 참조를 저장함)	class A	inner class A

내부 클래스에서 외부 클래스 참조: @클래스명

```
class Outer {
    inner class Inner {
        fun getOuterRef(): Outer = this@Outer // Outer에 대한 this
    }
}
```

sealed 클래스

sealed 클래스: 클래스 계층 정의 시 계층 확장 제한

- 같은 파일 안에서만 하위 클래스 선언 가능 (1.1부터)
- sealed 클래스의 내부 클래스로만 하위 클래스 선언 가능 (1.0)

sealed의 바로 하위 클래스가 한 파일에만 존재

- 다른 파일에 하위 클래스를 선언하면 컴파일 에러
- 단 sealed 클래스를 상속한 하위 클래스를 다른 파일에서 다시 상속하는 것은 가능

sealed 클래스와 when 식

sealed 클래스를 when 식과 함께 사용하면 유용

- when 식에 else 분기를 사용하지 않아도, 한 파일에 하위 타입이 몰려 있으므로 쉽게 분기에서 누락한 타입 확인 가능

```
sealed class Expr
class Num(val value: Int): Expr()
class Sum(val left: Expr, val right: Expr): Expr()
```

```
fun eval(e: Expr): Int =
        when(e) {
            is Num -> e.value
            is Sum -> eval(e.left) + eval(e.right)
        }
```

주 생성자와 초기화 블록

주 생성자(primary constructor)

- 클래스 이름 뒤에 오는 괄호로 둘러싸인 코드
  - 생성자 파라미터 지정
  - 생성자 파라미터로 초기화할 프로퍼티를 정의
- init 블록이나 프로퍼티 초기화 식에서만 주 생성자의 파라미터 참조 가능
- 생성자 파라미터에 디폴트 값 가능
  - 모든 파라미터에 디폴트 값 지정하면, 파라미터 없는 생성자를 만들어줌

```
// 클래스 이름 뒤의 constructor로 주 생성자 지정
class User constructor(_nickname: String) {
    val nickname: String
    init { // 초기화 블록
        nickname = _nickname
    }
}

// 프로퍼티를 생성자 파라미터로 초기화
// 별다른 애노테이션이나 가시성 수식어가 없다면
// constructor는 생략 가능
class User(_nickname: String) {
    val nickname = _nickname
}

// 파라미터로 프로퍼티를 바로 초기화
class User(val nickname: String)

// 디폴트 값
class User(val nickname: String,
           val isSubscribed: Boolean = true)
```

기반 클래스 생성자 호출

주 생성자에서 기반 클래스의 생성자를 호출해야 함

- 기반 클래스 이름 뒤에 괄호를 치고 생성자 인자 전달

```
open class User(val nickname: String, val isSubscribed: Boolean = true)

// User 클래스의 생성자 호출
class TwitterUser(nickname: String) : User(nickname)
```

디폴트 생성자

별도 생성자를 정의하지 않으면 컴파일러가 자동으로 아무 일도 하지 않는 인자 없는 디폴트 생성자를 만듦

```
open class Button // 인자 없는 디폴트 생성자를 만든다

class RadioButton: Button() // 기반 클래스의 인자 없는 생성자 호출
```

수식어와 constrcutor

생성자에 수식어를 붙이려면 constrcutor 키워드 필요

```
class Secretive private constructor() {}
```

부 생성자(secondary constructor)

여러 가지 방법으로 인스턴스를 초기화할 방법이 필요한 경우 부 생성자 사용

- 클래스 몸체에 constrcutor로 부 생성자 정의

```
open class View {
    constructor(ctx: Context) { ... }
    constructor(ctx: Context, attr: AttributeSet) { ... }
}

class MyButton: View {
    // 다른 부 생성자 호출
    constructor(ctx: Context): this(ctx, MY_STYLE) { ... }

    // 상위 클래스 생성자 호출
    constrcutor(ctx: Context, attr: AttributeSet): super(ctx, attr) {

    }
}
```

부 생성자와 주 생성자

주 생성자가 존재하면

- 부 생성자는 직접 또는 간접적(다른 부 생성자를 통해)으로 주 생성자를 호출해야 함

```
open class View (ctx: Context, attr: AttributeSet, type: Int) {
    constructor(ctx: Context): this(ctx, AttributeSet(), 0)
    constructor(ctx: Context, attr: AttributeSet): this(ctx, attr, 0)
}
```

주 생성자가 없다면

- 부 생성자는 반드시 상위 클래스를 초기화하거나 다른 생성자에 생성을 위임해야 함

인터페이스의 추상 프로퍼티와 구현

인터페이스에 추상 프로퍼티 선언 가능

```
interface User {
    val nickname: String // 추상 프로퍼티
}
```

인터페이스의 추상 프로퍼티는 지원 필드나 게터 등의 정보가 없음

- 인터페이스를 구현한 하위 클래스에서 상태 저장 위한 프로퍼티 등을 만들어야 함

```
class Privateuser(override val nickname: String) : User // 생성자의 프로퍼티로 구현
class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@') // 커스텀 게터
}
class FacebookUser(val accountId: Int) : User {
    override val nickname = getFBName(accountId) // 프로퍼티 초기화 식, 지원 필드에 초기화 식 결과 저장
}
```

인터페이스에서 게터와 세터 있는 프로퍼티 선언

```
interface User {
    val email: String // 추상 프로퍼티
    val nickname: String
        get() = email.substringBefore('@') // 지원 필드 없음
}
```

게터와 세터에서 지원 필드 접근

접근자 몸체에서 field라는 식별자로 지원 필드 접근

- 게터에서는 field 값을 읽을 수만 있음, 세터에서는 field 값을 읽거나 쓸 수 있음
- 게터나 세터에서 field에 접근하면, 지원 필드 생성
- field를 사용하지 않는 커스텀 접근자 구현 정의하면 지원 필드는 존재하지 않음

```
class User(val name: String) {
    val address: String = "unspecified"
      set(value: String) {
          println("changed $field -> $value")
          field = value
      }
}
```

접근자 가시성 변경

- 접근자 가시성은 기본적으로 프로퍼티 가시성과 같음
- get이나 set 앞에 가시성 수식어를 추가해 가시성 변경 가능

```
class LengthCounter {
    var counter: Int = 0
        private set

    fun addWord(word: String) {
        counter += word.length // set이 private
    }
}
```

데이터 클래스: equals(), hashCode(), toString()

```
data class Client(val name: String, val postalCode: Int)
```

- 다음 메서드를 자동으로 생성
  - 인스턴스 간 비교를 위한 equals
  - 해시 기반 컨테이너에서 키로 사용할 수 있는 hashCode
  - 각 필드를 선언 순서대로 표시하는 문자열 표현을 만들어주는 toString
- 주 생성자에 나열된 모든 프로퍼티를 고려해 equals()와 hashCode() 생성
  - 주 생성자 밖에 선언된 프로퍼티는 고려 대상이 아님에 주의

데이터 클래스: copy()

```
val lee = Client("이재성", 41225)
println(lee.copy(postalCode = 4000)) // name은 그대로 복사
```

- 객체 복사를 편하게 해주는 copy() 메서드 제공
  - 객체를 복사하면서 일부 프로퍼티를 바꿀 수 있게 해줌

위임

위임을 by 키워드로 손쉽게 구현

```
class CountingSet<T>(
    innerSet: MutableCollection<T> = HashSet<T>()
): MutableCollection<T> by innerSet {
    //  Collection<T> 타입에 대한 메서드 호출시 innerList에 위임

    var objectsAdded = 0

    // 필요한 메서드는 기본 위임 구현 대신 재정의 가능
    override fun add(element: T): Boolean {
        objectsAdded++
        return innerSet.add(element)
    }

    override fun addAll(c: Collection<T>): Boolean {
        objectsAdded += c.size
        return innerSet.addAll(c)
    }
}
```

객체 선언: 싱글톤

object 키워드로 객체 선언 시작

- 클래스 정의, 클래스의 인스턴스 생성, 변수에 인스턴스 저장을 한 문장으로 처리

```
object Payroll {
    val allEmployees = arraylistOf<Person>()

    fun calculateSalary() {
        for (person in allEmployees) {
            ...
        }
    }
}

Payroll.allEmployees.add(Person(...))
Payroll.calculateSalary() // 객체 선언 이름 뒤에 마침표로 메서드나 프로퍼티 접근
```

객체 선언

- 프로퍼티, 메서드, 초기화 블록 가능
- 생성자는 객체 선언에 사용할 수 없음 (주/부 생성자 모두)
- 클래스나 인터페이스 상속 가능
- 클래스 안에 객체 선언 가능

```
data class Person(val name: String) {
    // 클래스 안에 객체 선언
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
          p1.name.compareTo(p2.name)
    }
}

// Person.NameComparator로 접근
```

동반 객체(companion object)

- companion을 붙인 클래스 안에 정의된 객체
  - 동반 객체의 멤버를 사용하는 구문은 정적 메서드 호출이나 정적 필드 사용과 유사
  - 동반 객체는 자신을 둘러쌘 클래스의 모든 private 멤버에 접근 가능

```
class User private constructor(val nickname: String) {
    companion object { // 이름을 지정하지 않으면 동반 객체 이름은 Companion
        fun newSubscribingUser(email: String) =
            User(email.substringBefore('@')) // 바깥 클래스의 private 멤버 접근 가능
    }

    /*
    companion object Factory { // 이름붙인 동반 객체
        fun newSubscribingUser(email: String) = User(email.substringBefore('@'))
    }
    */
}

// 컴페니언 객체의 멤버를 정적 멤버처럼 접근
val user = User.newSubscribingUser("bob@gmail.com")
// val user2 = User.Factory.newSubscribingUser(...); // 동반 객체 이름 사용해서 접근
```

동반 객체와 타입 상속

- 동반 객체도 인터페이스 구현이나 클래스 확장 가능

```
interface JSONFactory<T> { ... }

class Person(vavl name: String) {
    companion object : JSONFactory<Person> { ... }
}
```

```
fun loadFromJSON<T>(factory: JSONFactory<T>): T { ... }

loadFromJSON(Person) // 동반객체 전달
```

동반 객체 확장

- 동반 객체에 대한 확장 함수도 가능

```
class Person(val firstName: String, val lastName: String) {
    companion object {}
}

fun Person.Companion.fromJSON(json: String): Person { ... }

val p = Person.fromJSON(json) // 동반 객체의 확장 함수 실행
```

객체 식

- object 키워드를 사용해서 익명 객체 정의

```
var clickCount = 0

window.addMouseListener(
    object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            // 객체 식이 포함된 함수의 변수에 접근 가능
            // 변수도 객체 식 안에서 사용할 수 있음
            clickCount++
        }
        ...
    }
)
```