

04

코틀린 객체지향 프로그래밍



04-1 클래스와 생성자

04-2 클래스를 재사용하는 상속

04-3 코틀린의 클래스 종류

04-1 객체지향 개념의 동의어들

- 객체지향 개념사이 유사한 어휘를 찾아보기
- 코틀린에서 사용하는 용어 그 밖에 용어

| | |
|-----------------|---|
| 클래스 (Class) | 분류, 범주 |
| 프로퍼티 (Property) | 속성 (Attribute), 변수 (Variable), 필드 (Field), 데이터 (Data) |
| 메서드 (Method) | 함수 (Function), 동작 (Operation), 행동 (Behavior) |
| 객체 (Object) | 인스턴스 (Instance) |

- 자바에서 사용하는 필드는 코틀린에서 프로퍼티로 부른다.

04-1 클래스

- 클래스 다이어그램



클래스명

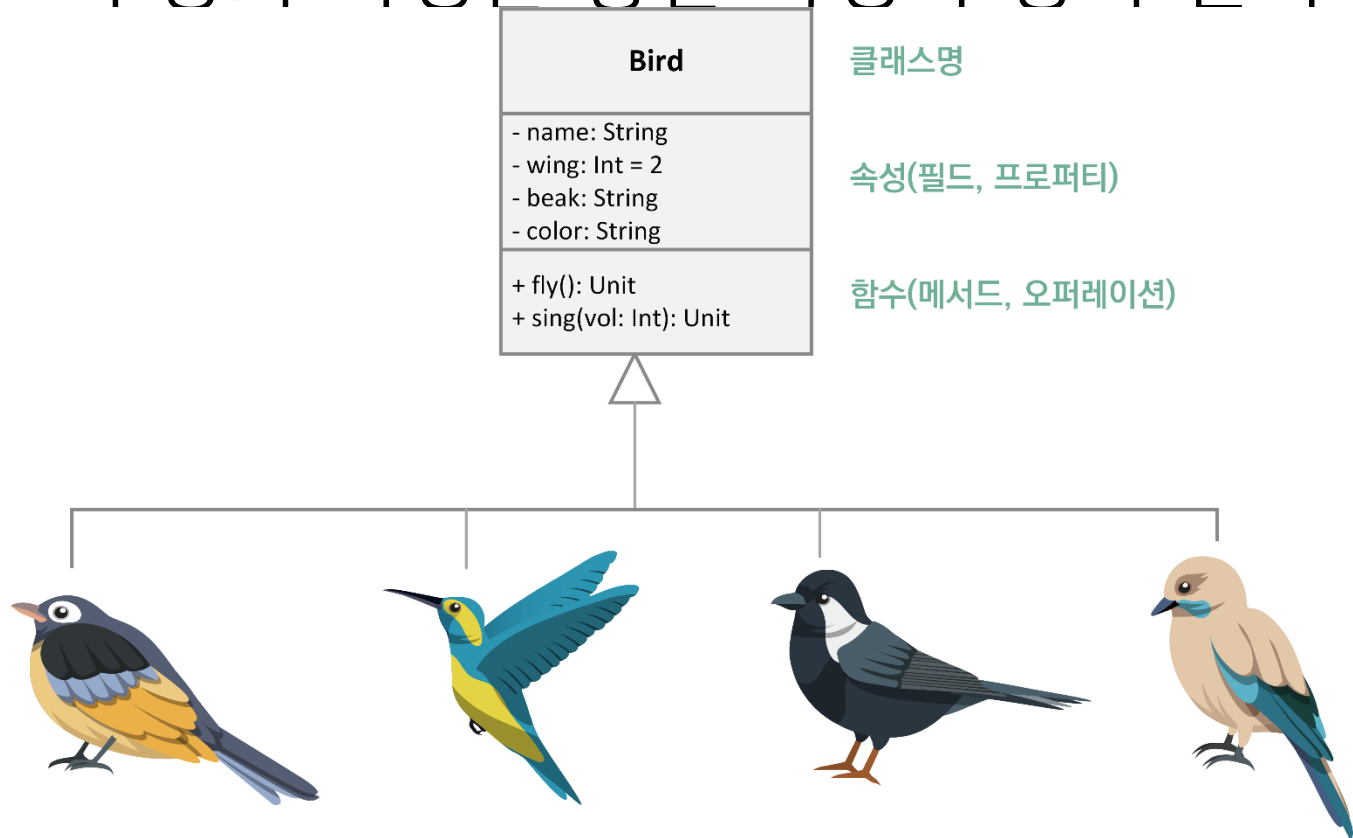
속성(필드, 프로퍼티)

함수(메서드, 오퍼레이션)

!석과 개념 구현에 용이

04-1 클래스

- 추상화 과정을 통한 속성과 동작 골라내기



04-1 클래스

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

```
class Bird { } // 내용이 비어있는 클래스 선언
class Bird // 중괄호는 생략 가능
```

```
class Bird {
    // 프로퍼티...
    // 메서드...
}
```

04-1 Bird 클래스 만들어 보기

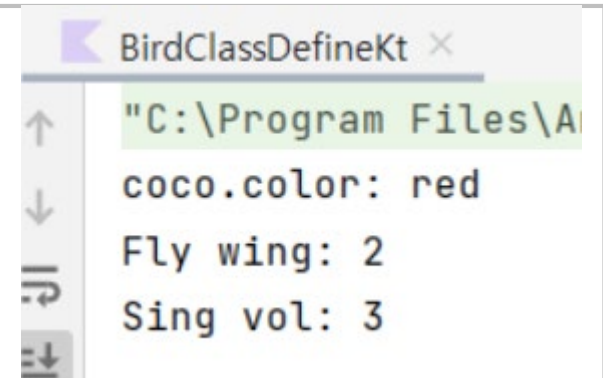
- BirdClassDefine.kt

- 객체를 생성해 사용하며 객체로 클래스의 멤버에 접근
- 객체 생성할 때 new 키워드 사용하지 않습니다

```
class Bird { // ① 클래스의 정의
    // ② 프로퍼티들(속성)
    var name: String = "mybird"
    var wing: Int = 2
    var beak: String = "short"
    var color: String = "blue"
    // ③ 메서드들(함수)
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol") }

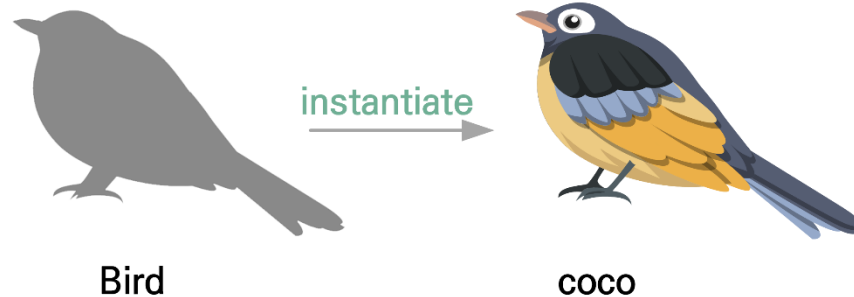
fun main() {
    val coco = Bird() // ④ 클래스의 생성자를 통한 객체의 생성
    coco.color = "red" // ⑤ 객체의 프로퍼티에 값 할당

    println("coco.color: ${coco.color}") // ⑥ 객체의 멤버 프로퍼티 읽기
    coco.fly() // ⑦ 객체의 멤버 메서드의 사용
    coco.sing(3) }
```



04-1 객체의 정의

- 객체(Object)
 - Bird 클래스란 일종의 선언일 뿐 실제 메모리에 존재해 실행되고 있는 것이 아님
 - 객체(Object)는 물리적인 메모리 영역에서 실행되고 있는 클래스의 실체
 - 따라서 클래스로부터 객체를 생성해 냄
 - 구체화 또는 인스턴스화(instantiate)되었다고 이야기할 수 있다.
 - 메모리에 올라 부름



```
val coco = Bird() // Bird로부터 만들어진 객체 coco
```

04-1 생성자

- 생성자(Constructor)란
 - 클래스를 통해 객체가 만들어질 때 기본적으로
 - 객체 생성 시 필요한 값을 인자로 설정할 수

생성자로 인해 특별한 함수인 constructor(

```
class 클래스명 constructor(필요한 매개변수들..) { // 주 생성자의 위치
....
constructor(필요한 매개변수들..) { // 부 생성자의 위치
    // 프로퍼티의 초기화
}
[constructor(필요한 매개변수들..) { ... }] // 추가 부 생성자
...
}
```

• 주 생성자 선언

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

• constructor 키워드 생략 예

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

• 매개변수가 없는 주 생성자 자동 선언

```
class User {
}
```


04-1 생성자의 정의

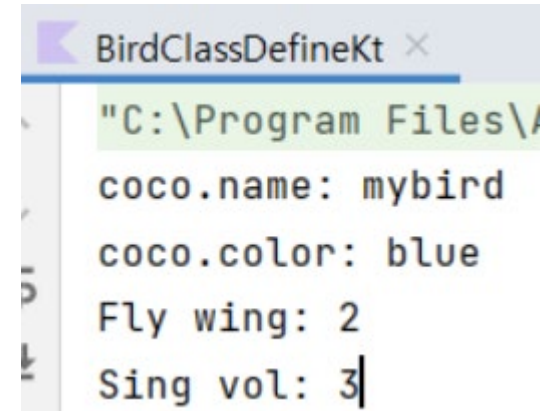
- 주 생성자(Primary Constructor)
 - 클래스명과 함께 기술되며 보통의 경우 constructor 키워드를 생략할 수 있다.
- 부 생성자(Secondary Constructor)
 - 클래스 본문에 기술되며 하나 이상의 부 생성자를 정의할 수 있다.

04-1 부 생성자를 사용하는 Bird 클래스

- BirdSecondaryConstructor.kt

```
class Bird {  
    // ① 프로퍼티들 - 선언만 함  
    var name: String  
    var wing: Int  
    var beak: String  
    var color: String  
    // ② 부 생성자 - 매개변수를 통해 초기화할 프로퍼티에 지정  
    constructor(name: String, wing: Int, beak: String, color: String) {  
        this.name = name // ③ this.wing는 선언된 현재 클래스의 프로퍼티를 나타냄  
        this.wing = wing  
        this.beak = beak  
        this.color = color  
    }  
    // 메서드들  
    fun fly() = println("Fly wing: $wing")  
    fun sing(vol: Int) = println("Sing vol: $vol")  
}
```

```
fun main() {  
    val coco = Bird("mybird", 2, "short", "blue") // ④ 생성자의 인자로 객체 생성과 동시에 초기화  
    println("coco.color: ${coco.color}") // ⑥ 객체의 멤버 프로퍼티 읽기  
    coco.fly() // ⑦ 객체의 멤버 메서드의 사용  
    coco.sing(3)  
}
```



```
BirdClassDefineKt ×  
"C:\Program Files\  
coco.name: mybird  
coco.color: blue  
Fly wing: 2  
Sing vol: 3
```

04-1 객체 생성 시 생성자로부터 일어나는 일

- BirdSecondaryConstructor.kt

```
class Bird {  
    // 프로퍼티들  
    var name: String  
    var wing: Int  
    var beak: String  
    var color: String  
  
    // 부 생성자  
    constructor(name: String, wing: Int, beak: String, color: String) {  
        this.name = name  
        this.wing = wing  
        this.beak = beak  
        this.color = color  
    }  
    ...  
}  
  
fun main(args: Array<String>) {  
    val coco = Bird("mybird", 2, "short", 2, "blue")  
    ...  
}
```

1

2

3

04-1 this 키워드를 생략하고 하는 경우

- 생성자의 매개변수와 프로퍼티의 이름을 다르게 구성

```
class Bird {  
    // ① 프로퍼티들 - 선언만 함  
    var name: String  
    var wing: Int  
    var beak: String  
    var color: String  
    // ② 부 생성자 - 매개변수를 통해 초기화할 프로퍼티에 지정  
    constructor(_name: String, _wing: Int, _beak: String, _color: String) {  
        name = _name // _를 매개변수에 사용하고 프로퍼티에 this를 생략할 수 있음  
        wing = _wing  
        beak = _beak  
        color = _color  
    }  
    // 메서드들  
    fun fly() = println("Fly wing: $wing")  
    fun sing(vol: Int) = println("Sing vol: $vol")  
}
```

→ 지양 함 (this 사용선호)

```
fun main() {  
    val coco = Bird("mybird", 2, "short", "blue") // ④ 생성자의 인자로 객체 생성과 동시에 초기화  
    println("coco.color: ${coco.color}") // ⑥ 객체의 멤버 프로퍼티 읽기  
    coco.fly() // ⑦ 객체의 멤버 메서드의 사용  
    coco.sing(3)  
}
```

04-1 부 생성자를 여러 개 포함한 클래스

- 클래스에 부 생성자를 하나 혹은 그 이상 포함할 수 있다.

```
class 클래스명 {  
    constructor(매개변수[,매개변수...]) {  
        // 코드  
    }  
  
    constructor(매개변수[,매개변수...]) {  
        // 코드  
    }  
    ...  
}
```

- 매개변수는 서로 달라야 함

04-1 여러 개의 부 생성자 지정

// 주 생성자가 없고 여러 개의 보조 생성자를 가진 클래스

```
class Bird5 {  
    // 프로퍼티들  
    var name: String  
    var wing: Int  
    var beak: String  
    var color: String  
    // 첫 번째 부 생성자  
    constructor(name: String, wing: Int, beak: String, color: String) {  
        this.name = name  
        this.wing = wing  
        this.beak = beak  
        this.color = color  
    }  
    // 두 번째 부 생성자  
    constructor(_name: String, _beak: String) {  
        name = _name  
        wing = 2  
        beak = _beak  
        color = "grey"  
    }  
    // 메서드들  
    fun fly() = println("Fly wing: $wing")  
    fun sing(vol: Int) = println("Sing vol: $vol")  
}
```

코난조깅기법

```
fun main() {  
    val bird1 = Bird5("mybird", 2, "short", "blue") // 첫번째 부 생성자 호출  
    val bird2 = Bird5("mybird2", "long") // 두번째 부 생성자 호출  
    println("bird1.color: ${bird1.color}") // ⑥ 객체의 멤버 프로퍼티 읽기  
    println("bird2.color: ${bird2.color}")  
    bird1.fly() // ⑦ 객체의 멤버 메서드의 사용  
    bird2.fly()  
    bird1.sing(3)  
    bird2.sing(0)  
}
```

BirdClassTwoKt x

"C:\Program Files\And

bird1.color: blue

bird2.color: grey

Fly wing: 2

Fly wing: 2

Sing vol: 3

Sing vol: 0

04-1 주 생성자

- 클래스명과 함께 생성자 정의

```
// 주 생성자 선언
class Bird constructor(_name: String, _wing: Int, _beak: String, _color: String) {
    // 프로퍼티
    var name: String = _name
    var wing: Int = _wing
    var beak: String = _beak
    var color: String = _color

    // 메서드
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol")
}
...
```

04-1 주 생성자

- 클래스명과 함께 생성자 정의 - constructor 키워드 생략

```
...
// 주 생성자 선언
class Bird (_name: String, _wing: Int, _beak: String, _color: String) {
    // 프로퍼티
    var name: String = _name
    var wing: Int = _wing
    var beak: String = _beak
    var color: String = _color

    // 메서드
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol")
}

fun main() {
    val coco = Bird6("Youbird", 2, "long", "red") // 기본값이 있는 것은 생략하고
    없는 것만 전달 가능
    println("coco.name: ${coco.name}, coco.wing ${coco.wing}")
    println("coco.color: ${coco.color}, coco.beak ${coco.beak}")
}
```

- 기시형시시시시시 에포데이션 표기식 배틀 형식 형식 기성

04-1 주 생성자

- 클래스명과 함께 생성자 정의 - 프로퍼티가

```
...
// 주 생성자 선언
class Bird7(var name: String, var wing: Int, var beak: String, var color: String) {
    // 프로퍼티 - 위에 var 혹은 val로 선언하므로서 프로퍼티가 이미 포함됨

    // 메서드
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol")
}

fun main() {
    val coco = Bird7("bird", 11, "long", "orange") // 기본값이 있는 것은 생략하고 없는 것만 전달 가능
    println("coco.name: ${coco.name}, coco.wing ${coco.wing}")
    println("coco.color: ${coco.color}, coco.beak ${coco.beak}")
}
```

BirdPrimaryInitKt x

```
"C:\Program Files\Android\Android St
coco.name: bird, coco.wing 11
coco.color: orange, coco.beak long
```

04-1 주 생성자의 초기화 블

• 초기화 블록이 포함된 주 생성자 - BirdPrimary

```
...
// 주 생성자 선언
class Bird6(var name: String, var wing: Int, var beak: String, var color: String) {

    // ① 초기화 블록
    init {
        println("-----초기화 블록 시작-----")
        println("이름은 $name, 부리는 $beak")
        this.sing(3)
        println("----- 초기화 블록 끝 -----")
    }

    // 메서드
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol")
}

fun main() {
    val coco = Bird6("Youbird", 2, "long", "red") // 기본값이 있는 것은 생략하고 없는 것만 전달 가능
    println("coco.name: ${coco.name}, coco.wing ${coco.wing}")
    println("coco.color: ${coco.color}, coco.beak ${coco.beak}")
}
```

초기화 블록에는 간단한 코드가 허용된다.

```
BirdPrimaryInitKt x
"C:\Program Files\Android\Android S
-----초기화 블록 시작-----
이름은 Youbird, 부리는 long
Sing vol: 3
----- 초기화 블록 끝 -----
coco.name: Youbird, coco.wing 2
coco.color: red, coco.beak long

Process finished with exit code 0
```

04-1 프로퍼티의 기본값

- 생성시 필요한 기본값을 지정할 수 있다.

// 프로퍼티의 기본값 지정

```
class Bird10(var name: String = "NONAME", var wing: Int = 2, var beak: String, var color: String) {  
    // 메서드  
    fun fly() = println("Fly wing: $wing")  
    fun sing(vol: Int) = println("Sing vol: $vol")  
}
```

```
fun main() {
```

```
    val coco = Bird10(beak = "long", color = "red") // 기본값이 있는 것은 생략하고 없는 것만 전달 가능
```

```
    println("coco.name: ${coco.name}, coco.wing ${coco.wing}")  
    println("coco.color: ${coco.color}, coco.beak ${coco.beak}")  
    coco.fly()
```

```
}
```

BirdPrimaryInitKt x

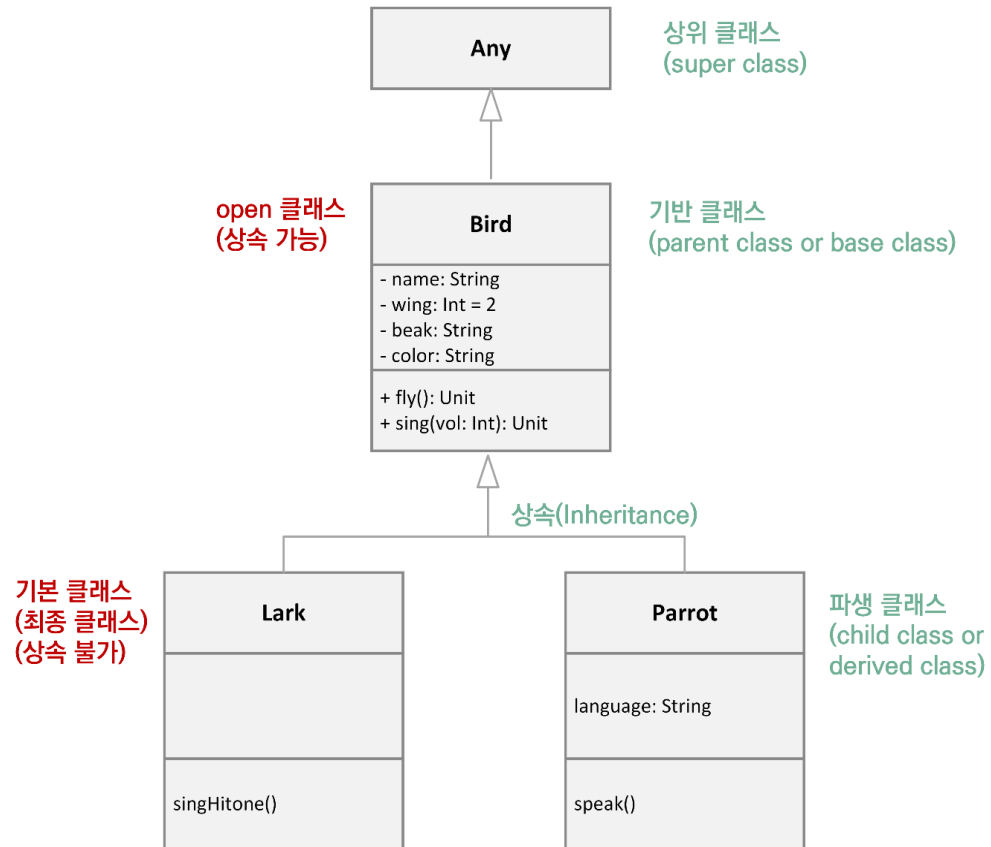
```
"C:\Program Files\Android\Android  
coco.name: NONAME, coco.wing 2  
coco.color: red, coco.beak long  
Fly wing: 2
```

04-1 상속과 클래스의 계층

- 상속(inheritance)
 - 자식 클래스를 만들 때 상위 클래스(부모 클래스)의 속성과 기능을 물려받아 계승
 - 상위(부모) 클래스의 프로퍼티와 메서드가 자식에 적용됨

04-1 상속의 예

- 종달새(Lark)와 앵무새(Parrot) 클래스



04-1 상속 가능한 클래스와 하위 클래스 선언

- open 키워드를 통한 선언

```
open class 기반 클래스명 { // open으로 파생 가능 (다른 클래스가 상속 가능한 상태가 됨)
    ...
}
class 파생 클래스명 : 기반 클래스명() { // 기반 클래스로부터 상속, 최종 클래스로 파생 불가
    ...
}
```

- 코틀린의 모든 클래스는 묵시적으로 Any로부터 상속

04-1 파생 클래스 만들어 보기 - BirdChildClasses.kt (1/2)

```
...
// ① 상속 가능한 클래스를 위해 open 사용
open class Bird(var name: String, var wing: Int, var beak: String, var color: String) {
    // 메서드
    fun fly() = println("Fly wing: $wing")
    fun sing(vol: Int) = println("Sing vol: $vol")
}
// ② 주 생성자를 사용하는 상속
class Lark(name: String, wing: Int, beak: String, color: String) : Bird(name, wing, beak, color) {
    fun singHitone() = println("Happy Song!") // 새로 추가된 메서드
}
// ③ 부 생성자를 사용하는 상속
class Parrot : Bird {
    val language: String

    constructor(name: String, wing: Int, beak: String, color: String,
                language: String) : super(name, wing, beak, color) {
        this.language = language // 새로 추가된 프로퍼티
    }
    fun speak() = println("Speak! $language")
}
...
```

04-1 파생 클래스 만들어 보기 - BirdChildClasses.kt (2/2)

```
...
fun main() {

    val coco = Bird("mybird", 2, "short", "blue")
    val lark = Lark("mylark", 2, "long", "brown")
    val parrot = Parrot("myparrot", 2, "short", "multiple", "korean") // 프로퍼티가 추가됨

    println("Coco: ${coco.name}, ${coco.wing}, ${coco.beak}, ${coco.color}")
    println("Lark: ${lark.name}, ${lark.wing}, ${lark.beak}, ${lark.color}")
    println("Parrot: ${parrot.name}, ${parrot.wing}, ${parrot.beak}, ${parrot.color}, ${parrot.language}")

    lark.singHitone() // 새로 추가된 메서드가 사용 가능
    parrot.speak()
    lark.fly()
}
```

```
BirdChildClassesKt x
"C:\Program Files\Android\Android Studio\jre\bi
Coco: mybird, 2, short, blue
Lark: mylark, 2, long, brown
Parrot: myparrot, 2, short, multiple, korean
Happy Song!
Speak! korean
Fly wing: 2
```

- 하위 클래스는 상위 클래스의 메서드나 프로퍼티를 그대로 상속하면서 상위 클래스에는 없는 자신만의 프로퍼티나 메서드를 확장

04-1 다형성

- 다형성(polymorphism)이란
 - 같은 이름을 사용하지만 구현 내용이 다르거나 매개변수가 달라서 하나의 이름으로 다양한 기능을 수행할 수 있는 개념

04-1 다형성

- ☆• 오버라이딩(overriding)
 - 기능을 완전히 다르게 바꾸어 재설계
 - 누르다 → 행위 → push()
 - push()는 '확인' 혹은 '취소' 용도로 서로 다른 기능을 수행 할 수 있음
- ☆• 오버로딩(overloading)
 - 기능은 같지만 인자를 다르게 하여 여러 경우를 처리
 - 출력한다 → 행위 → print()
 - print(123), print("Hello") 인자는 다르지만 출력의 기능은 동일함

04-1 오버로딩의 예

- 일반 함수에서의 오버로딩

```
fun add(x: Int, y: Int): Int { // 정수 자료형 매개변수 2개를 더함
    return x + y
}
```

```
fun add(x: Double, y: Double): Double { // 실수 자료형 매개변수 2개를 더함
    return x + y
}
```

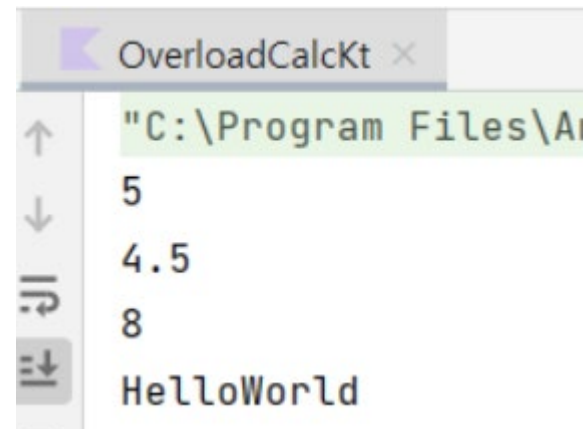
```
fun add(x: Int, y: Int, z: Int): Int { // 정수 자료형 매개변수 3개를 더함
    return x + y + z
}
```

04-1 덧셈 동작의 오버로딩 -

OverloadCalc.kt

- 클래스 메서드의 오버로딩

```
fun main() {  
    val calc = Calc()  
    println(calc.add(3,2))  
    println(calc.add(3.2, 1.3))  
    println(calc.add(3, 3, 2))  
    println(calc.add("Hello", "World"))  
}  
  
class Calc {  
    // 다양한 매개변수로 오버로딩된 메서드들  
    fun add(x: Int, y: Int): Int = x + y  
    fun add(x: Double, y: Double): Double = x + y  
    fun add(x: Int, y: Int, z: Int): Int = x + y + z  
    fun add(x: String, y: String): String = x + y  
}
```



04-1 오버라이딩

- 개념 재정리
 - 오버라이드(override)란 사전적 의미로 '(기존의 작업을) 중단하다', '뒤엎다' 등으로 해석
 - 상위 클래스의 메서드의 내용을 **완전히 새로 만들어 다른 기능을 하도록 정의**
 - 오버라이딩하기 위해 부모 클래스에서는 **open** 키워드, 자식 클래스에서는 **override** 키워드를 각각 이용
 - 메서드 및 프로퍼티등에 사용할 수 있다.

04-1 오버라이딩의 예

• 메서드 오버라이딩의 예

```
open class Bird { // 여기의 open은 상속 기능을 나타냄
```

```
...
```

```
    fun fly() { ... } // ① 최종 메서드로 오버라이딩 불가
```

```
    open fun sing() {...} // ② sing() 메서드는 하위 클래스에서 오버라이딩 가능
```

↳ 상속가능 (= 오버라이딩 가능)

```
}
```

```
class Lark() : Bird() { // ③ 하위 클래스
```

```
    fun fly() { /* 재정의 */ } // 에러! 상위 메서드에 "open키워드가 없어" 오버라이딩 불가
```

```
    override fun sing() { /* 구현부를 새롭게 재정의 */ } // ④ 구현부를 새롭게 작성
```

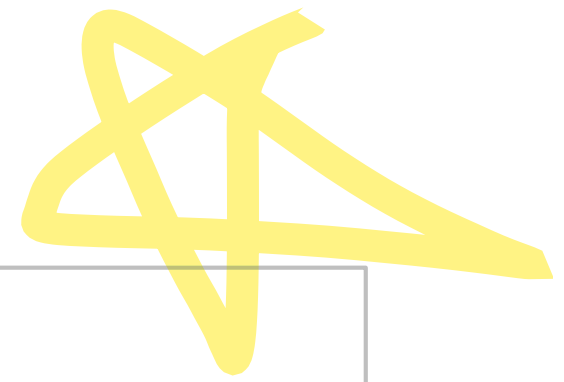
```
}
```

04-1 오버라이딩 금지

- 파생 클래스에서 오버라이딩을 금지할 때

```
open class Lark() : Bird() {  
    final override fun sing() { /* 구현부를 새롭게 재정의 */ } // 하위 클래스에서 재정의 금지  
}
```

04-1 메서드를 오버라이딩 하기 -



// 상속 가능한 클래스를 위해 open 사용

```
open class Bird(var name: String, var wing: Int, var beak: String, var color: String) {
```

// 메서드

```
fun fly() = println("Fly wing: $wing")
```

```
open fun sing(vol: Int) = println("Sing vol: $vol") // 오버라이딩 가능한 메서드
```

```
}
```

```
class Parrot(name: String,
```

```
    wing: Int = 2,
```

```
    beak: String,
```

```
    color: String, // 마지막 인자만 var로 선언되어 프로퍼티가 추가됨
```

```
    var language: String = "natural") : Bird(name, wing, beak, color) {
```

```
    fun speak() = println("Speak! $language") // Parrot에 추가된 메서드
```

```
    override fun sing(vol: Int) { // 오버라이딩된 메서드
```

```
        println("I'm a parrot! The volume level is $vol")
```

```
        speak() // 달라진 내용!
```

```
    }
```

```
}
```

```
fun main() {
```

```
    val parrot = Parrot(name = "myparrot", beak = "short", color = "multiple")
```

```
    parrot.language = "English"
```

```
    println("Parrot: ${parrot.name}, ${parrot.wing}, ${parrot.beak}, ${parrot.color}, ${parrot.language}")
```

```
    parrot.sing(5) // 달라진 메서드 실행 가능
```

```
}
```

BirdOverrideExKt x

"C:\Program Files\Android\Android Studio\jre\bin\java.exe"

Parrot: myparrot, 2, short, multiple, English

I'm a parrot! The volume level is 5

Speak! English

04-1 super와 this

- 현재 클래스에서 참조의 기능
 - 상위 클래스는 super 키워드로 현재 클래스는 this 키워드로 참조가 가능

super

this

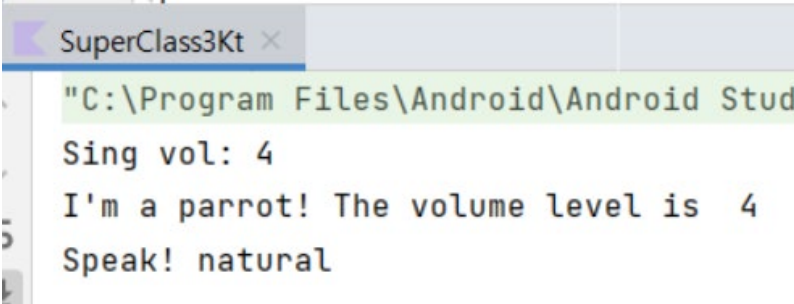
```
super.프로퍼티명 // 상위 클래스의 프로퍼티 참조  
super.메서드명( ) // 상위 클래스의 메서드 참조  
super( ) // 상위 클래스의 생성자의 참조
```

```
this.프로퍼티명 // 현재 클래스의 프로퍼티 참조  
this.메서드명( ) // 현재 클래스의 메서드 참조  
this( ) // 현재 클래스의 생성자의 참조
```

04-1 super로 상위 참조

- 상위 클래스의 메서드 실행

```
open class Bird100(var name: String, var wing: Int, var beak: String , var color: String ) {  
    fun fly() = println("Fly wing: $wing")  
    open fun sing(vol: Int) = println("Sing vol: $vol")  
}  
class Parrot100(name: String, wing: Int = 2, beak: String, color: String,  
    var language: String = "natural") : Bird100(name, wing, beak, color) {  
    fun speak() = println("Speak! $language")  
    override fun sing(vol: Int) { // ① 부모의 내용과 새로 구현된 내용을 가짐  
        super.sing(vol) // 상위 클래스의 sing()을 먼저 수행  
        println("I'm a parrot! The volume level is $vol")  
        speak()  
    }  
}  
fun main(){  
    val parrotmo = Parrot100("leemiso", 4, "long", "orange")  
    parrotmo.sing(4)  
}
```



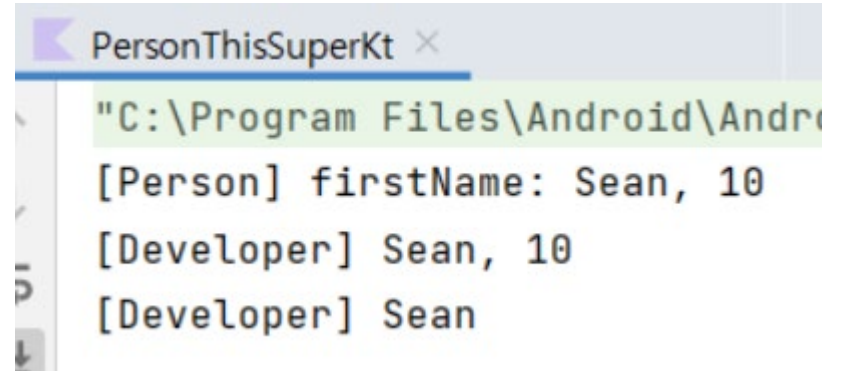
```
SuperClass3Kt x  
"C:\Program Files\Android\Android Stud  
Sing vol: 4  
I'm a parrot! The volume level is 4  
Speak! natural
```

04-1 this와 super를 사용하는 부 생성자

- PersonThisSuper.kt

```
open class Person {
    constructor(firstName: String) {
        println("[Person] firstName: $firstName")
    }
    constructor(firstName: String, age: Int) { // ③
        println("[Person] firstName: $firstName, $age")
    }
}
class Developer: Person {

    constructor(firstName: String): this(firstName, 10) { // ①
        println("[Developer] $firstName")
    }
    constructor(firstName: String, age: Int): super(firstName, age) { // ②
        println("[Developer] $firstName, $age")
    }
}
fun main() {
    val sean = Developer("Sean")
}
```



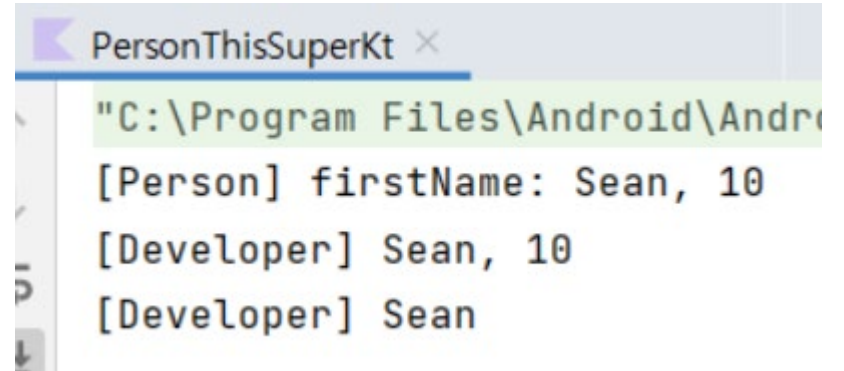
```
PersonThisSuperKt x
"C:\Program Files\Android\Andro
[Person] firstName: Sean, 10
[Developer] Sean, 10
[Developer] Sean
```

04-1 this와 super를 사용하는 부 생성자

- PersonThisSuper.kt

```
open class Person {
    constructor(firstName: String) {
        println("[Person] firstName: $firstName")
    }
    constructor(firstName: String, age: Int) { // ③
        println("[Person] firstName: $firstName, $age")
    }
}
class Developer: Person {

    constructor(firstName: String): this(firstName, 10) { // ①
        println("[Developer] $firstName")
    }
    constructor(firstName: String, age: Int): super(firstName, age) { // ②
        println("[Developer] $firstName, $age")
    }
}
fun main() {
    val sean = Developer("Sean")
}
```



```
PersonThisSuperKt x
"C:\Program Files\Android\Andro
[Person] firstName: Sean, 10
[Developer] Sean, 10
[Developer] Sean
```

04-1 PersonThisSuper.kt 호출 순서

```
open class Person {  
    constructor(firstName: String) {  
        println("[Person] firstName: $firstName")  
    }  
    constructor(firstName: String, age: Int) { // (3)  
        println("[Person] firstName: $firstName, $age")  
    }  
}  
  
class Developer: Person {  
    constructor(firstName: String): this(firstName, 10) { // (1)  
        println("[Developer] $firstName")  
    }  
    constructor(firstName: String, age: Int): super(firstName, age) { // (2)  
        println("[Developer] $firstName, $age")  
    }  
}  
  
fun main() {  
    val sean = Developer("Sean")  
}
```

The diagram illustrates the execution order of constructors in the provided Kotlin code. The sequence is as follows:

1. The `Developer` constructor is called from the `main` function.
2. The body of the `Developer` constructor (the `println` statement) is executed.
3. The `Person` constructor is called from the `Developer` constructor.
4. The body of the `Person` constructor (the `println` statement) is executed.
5. The execution returns from the `Person` constructor to the `Developer` constructor.
6. The execution returns from the `Developer` constructor to the `main` function.

04-1 주 생성자와 부 생성자 함께 사용하기 - PersonPriSeconRef.kt

```
class Person(firstName: String,
              out: Unit = println("[Primary Constructor] Parameter")) { // ② 주 생성자

    val fName = println("[Property] Person fName: $firstName") // ③ 프로퍼티 할당

    init {
        println("[init] Person init block") // ④ 초기화 블록
    }

    // ① 보조 생성자
    constructor(firstName: String, age: Int,
                 out: Unit = println("[Secondary Constructor] Parameter")): this(firstName) {
        println("[Secondary Constructor] Body: $firstName, $age") // ⑤ 부 생성자 본문
    }
}

fun main() {

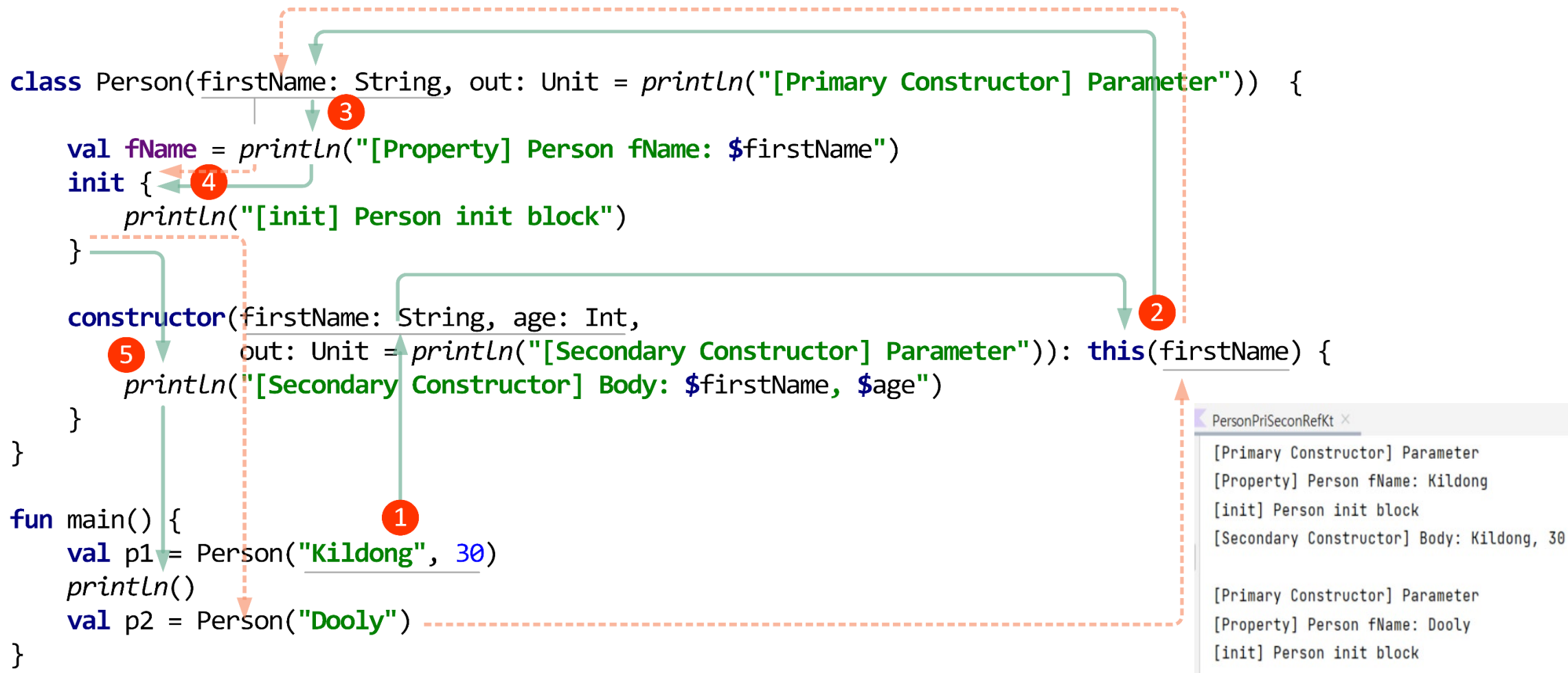
    val p1 = Person("Kildong", 30) // ①→② 호출, ③→④→⑤ 실행
    println()
    val p2 = Person("Dooly") // ② 호출, ③→④ 실행
}
```

PersonPriSeconRefKt ×

```
[Primary Constructor] Parameter
[Property] Person fName: Kildong
[init] Person init block
[Secondary Constructor] Body: Kildong, 30
```

```
[Primary Constructor] Parameter
[Property] Person fName: Dooly
[init] Person init block
```

04-1 PersonPriSeconRef.kt 호출 순서



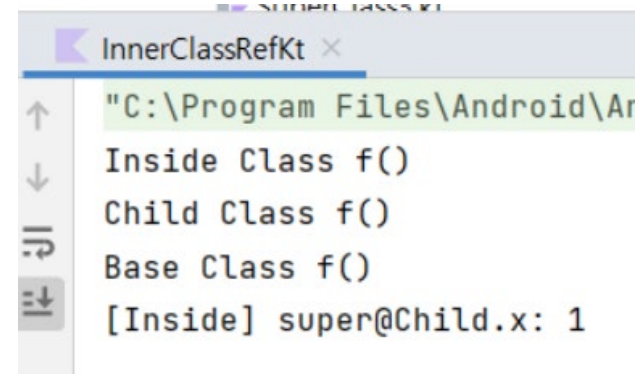
04-1 바깥 클래스 호출하기

- 엷(@) 기호의 이용
 - 이너 클래스에서 바깥 클래스의 상위 클래스를 호출하려면 super 키워드와 함께 엷(@) 기호 옆에 바깥 클래스명을 작성해 호출

04-1 이너 클래스에서 바깥 클래스 접근하기 - InnerClassRef.kt

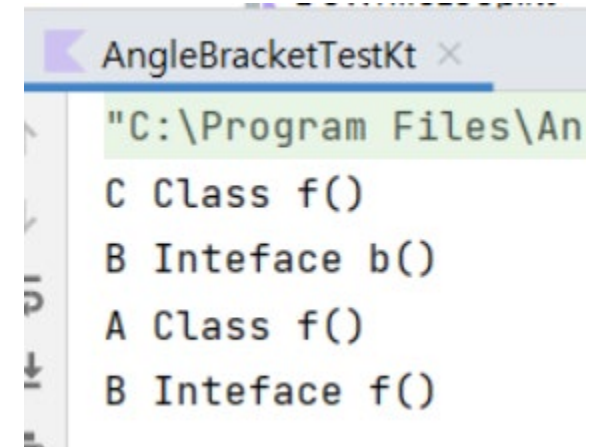
```
open class Base {
    open val x: Int = 1
    open fun f() = println("Base Class f()")
}
class Child : Base() {
    override val x: Int = super.x + 1
    override fun f() = println("Child Class f()")

    inner class Inside {
        fun f() = println("Inside Class f()")
        fun test() {
            f() // ① 현재 이너 클래스의 f() 접근
            Child().f() // ② 바로 바깥 클래스 f()의 접근
            super@Child.f() // ③ Child의 상위 클래스인 Base 클래스의 f() 접근
            println("[Inside] super@Child.x: ${super@Child.x}") // ④ Base의 x 접근
        }
    }
}
fun main() {
    val c1 = Child()
    c1.Inside().test() // 이너 클래스 Inside의 메서드 test() 실행
}
```



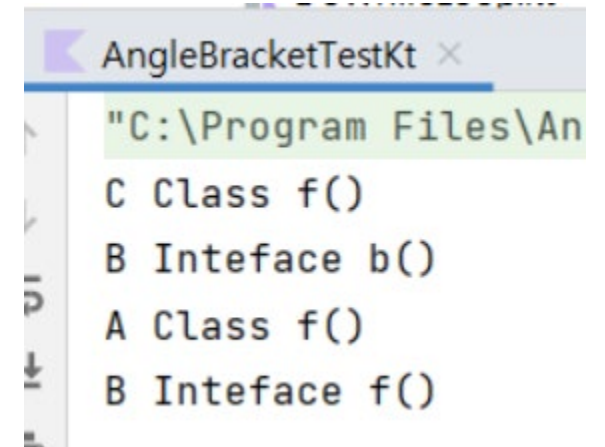
04-1 앵글브라켓을 사용한 이름 중복 해결 - AngleBracketTest.kt

```
open class A {  
    open fun f() = println("A Class f()")  
    fun a() = println("A Class a()")  
}  
interface B {  
    fun f() = println("B Inteface f()") // 인터페이스는 기본적으로 open임  
    fun b() = println("B Inteface b()")  
}  
class C : A(), B { // ① 콤마(,)를 사용해 클래스와 인터페이스를 지정  
    // 컴파일되려면 f()가 오버라이딩되어야 함  
    override fun f() = println("C Class f()")  
    fun test() {  
        f() // ② 현재 클래스의 f()  
        b() // ③ 인터페이스 B의 b()  
        super<A>.f() // ④ A 클래스의 f()  
        super<B>.f() // ⑤ B 클래스의 f()  
    }  
}  
fun main() {  
    val c = C()  
    c.test()  
}
```

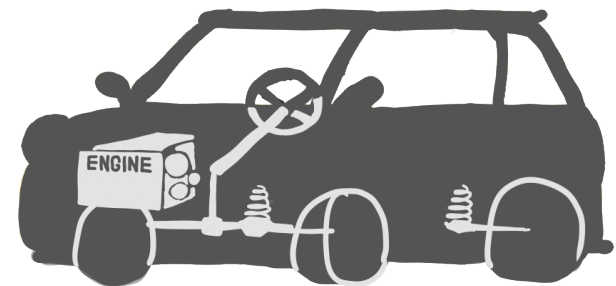


04-1 앵글브라켓을 사용한 이름 중복 해결 - AngleBracketTest.kt

```
open class A {  
    open fun f() = println("A Class f()")  
    fun a() = println("A Class a()")  
}  
interface B {  
    fun f() = println("B Inteface f()") // 인터페이스는 기본적으로 open임  
    fun b() = println("B Inteface b()")  
}  
class C : A(), B { // ① 콤마(,)를 사용해 클래스와 인터페이스를 지정  
    // 컴파일되려면 f()가 오버라이딩되어야 함  
    override fun f() = println("C Class f()")  
    fun test() {  
        f() // ② 현재 클래스의 f()  
        b() // ③ 인터페이스 B의 b()  
        super<A>.f() // ④ A 클래스의 f()  
        super<B>.f() // ⑤ B 클래스의 f()  
    }  
}  
fun main() {  
    val c = C()  
    c.test()  
}
```



04-1 정보 은닉 캡슐화



- 캡슐화(encapsulation)

- 클래스를 작성할 때 외부에서 숨겨야 하는 속성이나 기능

- 가시성 지시자(visibility modifiers)를 통해 외부 접근 범위를 결정할 수 있음

- `private` 이 지시자가 붙은 요소는 이 클래스에서 접근할 수 없음

| 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 |

04-1 정보 은닉 캡슐화

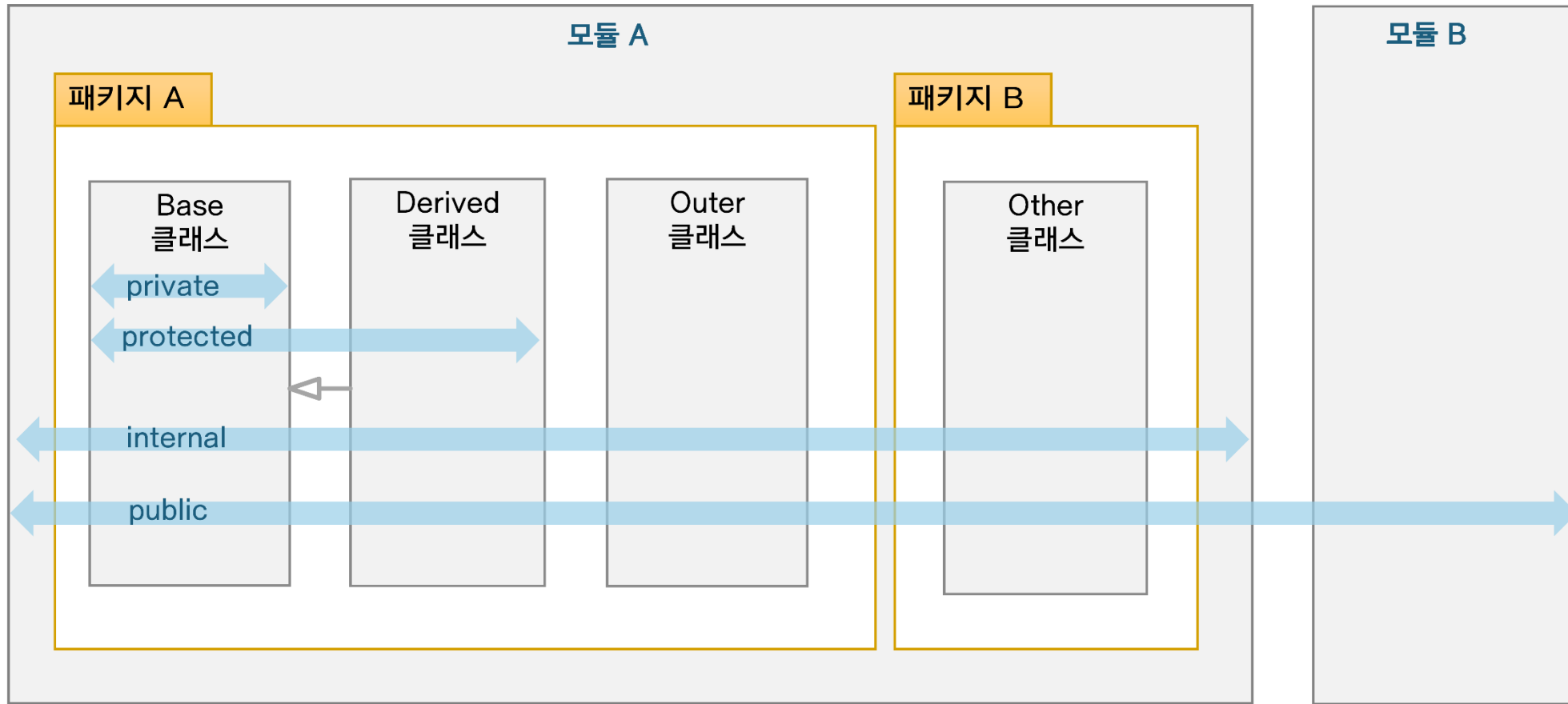
• 가시성 지시자이 서어 위치

[가시성 지시자] <val | var> 전역 변수명

[가시성 지시자] fun 함수명() { ... }

[가시성 지시자] [특정키워드] class 클래스명 [가시성 지시자] constructor(매개변수들) {
 [가시성 지시자] constructor() { ... }
 [가시성 지시자] 프로퍼티들
 [가시성 지시자] 메서드들
}

04-1 가시성 지시자의 공개 범위



04-1 private 가시성 테스트하기 - PrivateTest.kt

```
private class PrivateClass {  
    private var i = 1  
    private fun privateFunc() {  
        i += 1 // 접근 허용  
    }  
    fun access() {  
        privateFunc() // 접근 허용  
    }  
}  
class OtherClass {  
    val opc = PrivateClass() // 불가 - 프로퍼티 opc는 private이 되어 함  
    fun test() {  
        val pc = PrivateClass() // 생성 가능  
    }  
}  
fun main() {  
    val pc = PrivateClass() // 생성 가능  
    pc.i // 접근 불가  
    pc.privateFunc() // 접근 불가  
}  
fun TopFunction() {  
    val tpc = PrivateClass() // 객체 생성 가능  
}
```

2 $\frac{1}{2}$ 2 = 7

04-1 protected 가시성 테스트 - ProtectedTest.kt

```
open class Base { // 최상위 선언 클래스에는 protected를 사용할 수 없음
    protected var i = 1
    protected fun protectedFunc() {
        i += 1 // 접근 허용
    }
    fun access() {
        protectedFunc() // 접근 허용
    }
    protected class Nested // 내부 클래스에는 지시자 허용
}
class Derived : Base() {
    fun test(base: Base): Int {
        protectedFunc() // Base 클래스의 메서드 접근 가능
        return i // Base 클래스의 프로퍼티 접근 가능
    }
}
fun main() {
    val base = Base() // 생성 가능
    base.i // 접근 불가
    base.protectedFunc() // 접근 불가
    base.access() // 접근 가능
}
```


04-1 protected 가시성 테스트 - ProtectedTest.kt

```
open class Base { // 최상위 선언 클래스에는 protected를 사용할 수 없음
    protected var i = 1
    protected fun protectedFunc() {
        i += 1 // 접근 허용
    }
    fun access() {
        protectedFunc() // 접근 허용
    }
    protected class Nested // 내부 클래스에는 지시자 허용
}
class Derived : Base() {
    fun test(base: Base): Int {
        protectedFunc() // Base 클래스의 메서드 접근 가능
        return i // Base 클래스의 프로퍼티 접근 가능
    }
}
fun main() {
    val base = Derived() // 생성 가능
    base.i // 접근 불가
    base.protectedFunc() // 접근 불가
    base.access() // 접근 가능
}
```

04-1 internal 가시성 테스트하기 - InternalTest.kt

```
internal class InternalClass {
    internal var i = 1
    internal fun icFunc() {
        i += 1 // 접근 허용
    }
    fun access() {
        icFunc() // 접근 허용
    }
}
class Other {
    internal val ic = InternalClass() // 프로퍼티 지정시 internal로 맞춰야 한다.
    fun test() {
        ic.i // 접근 허용
        ic.icFunc() // 접근 허용
    }
}
fun main() {
    val mic = InternalClass() // 생성 가능
    mic.i // 접근 허용
    mic.icFunc() // 접근 허용
}
```

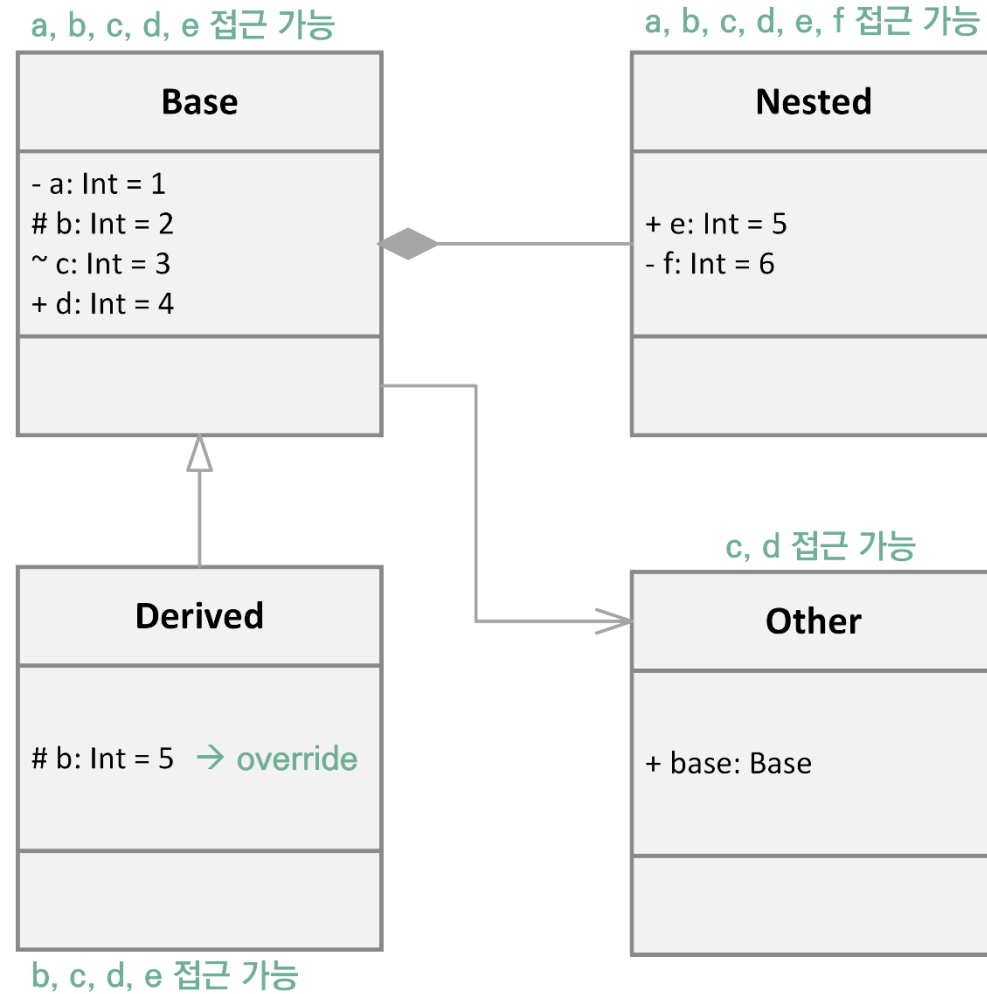
04-1 internal 가시성 테스트하기 - InternalTest.kt

```
internal class InternalClass {
    internal var i = 1
    internal fun icFunc() {
        i += 1 // 접근 허용
        println(i)
    }
    fun access() {
        icFunc() // 접근 허용
    } }
class Other {
    internal val ic = InternalClass() // 프로퍼티 지정시 internal로 맞춰야 한다.
    fun test() {
        ic.i // 접근 허용
        ic.icFunc() // 접근 허용
    } }
fun main() {
    val mic = InternalClass() // 생성 가능
    mic.i // 접근 허용
    mic.icFunc() // 접근 허용
}

val otheric = InternalClass()

println(otheric.i)
otheric.icFunc()
```

04-1 가시성 지시자와 클래스의 관계



04-1 가시성 지시자와 클래스의 관계

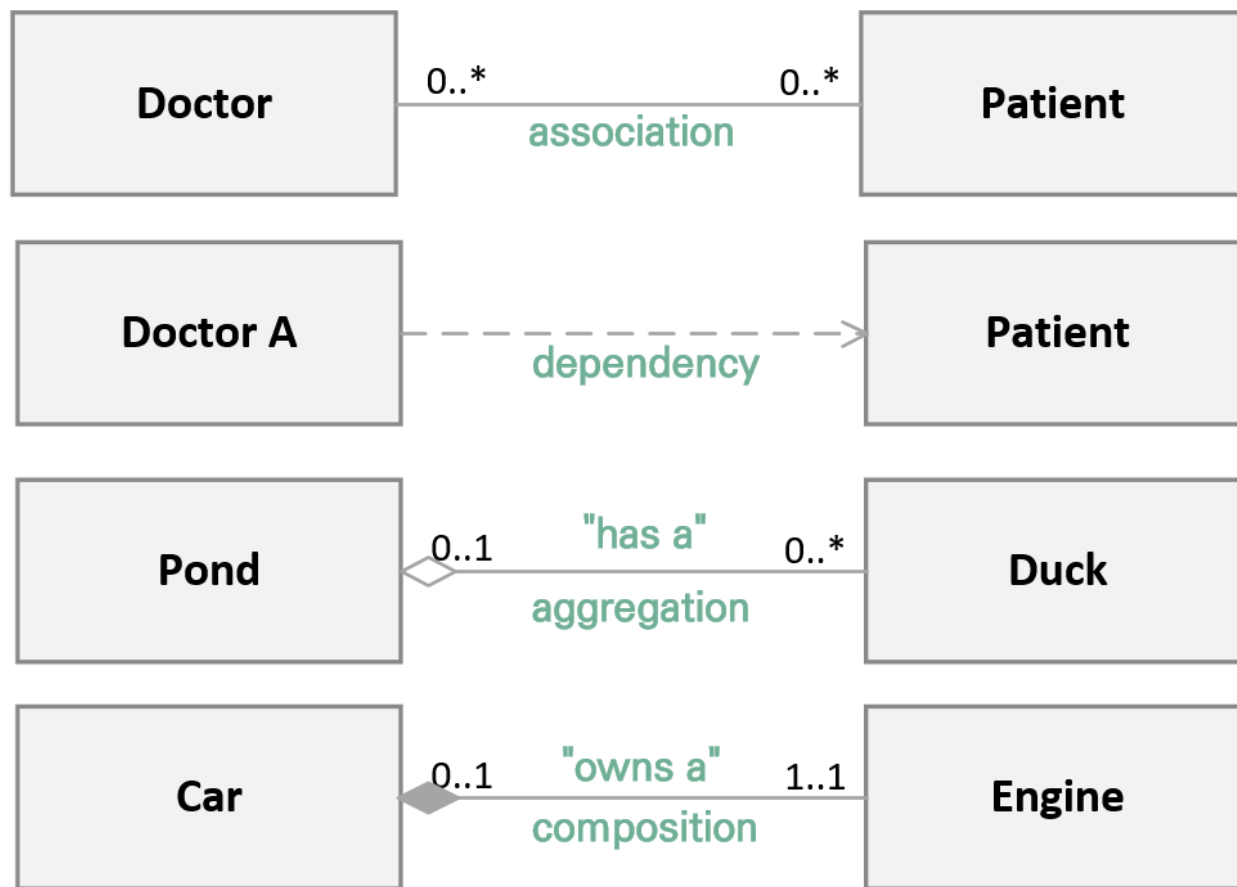
```
open class Base {  
    // 이 클래스에서는 a, b, c, d, e 접근 가능  
    private val a = 1  
    protected open val b = 2  
    internal val c = 3  
    val d = 4 // 가시성 지시자 기본값은 public  
    protected class Nested {  
        // 이 클래스에서는 a, b, c, d, e, f 접근 가능  
        public val e: Int = 5 // public 생략 가능  
        private val f: Int = 6  
    }  
}  
class Derived : Base() {  
    // 이 클래스에서는 b, c, d, e 접근 가능  
    // a 는 접근 불가  
    override val b = 5 // Base의 'b' 는 오버라이딩됨 - 상위와 같은 protected 지시자  
}  
class Other(base: Base) {  
    // base.a, base.b는 접근 불가  
    // base.c와 base.d는 접근 가능(같은 모듈 안에 있으므로)  
    // Base.Nested는 접근 불가, Nested::e 역시 접근 불가  
}
```

04-1 클래스와 관계

- 일반적인 실세계의 관계
 - 서로 관계를 맺고 서로 메시지를 주고받으며, 필요한 경우 서로의 관계를 이용
 - 자동차와 엔진처럼 종속적인 관계
 - 아버지와 아들처럼 상속의 관계

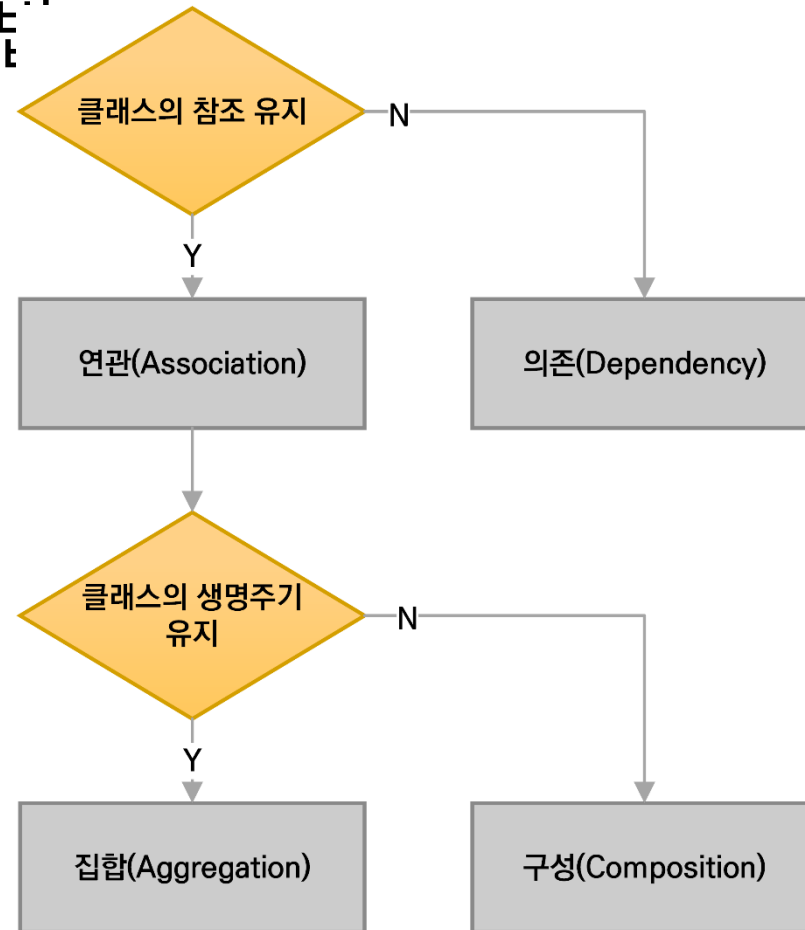
04-1 클래스 혹은 객체 간의 관계

- 관계(relationship)
 - 연관(association)
 - 의존(dependency)
 - 집합(aggregation)
 - 구성(composition)



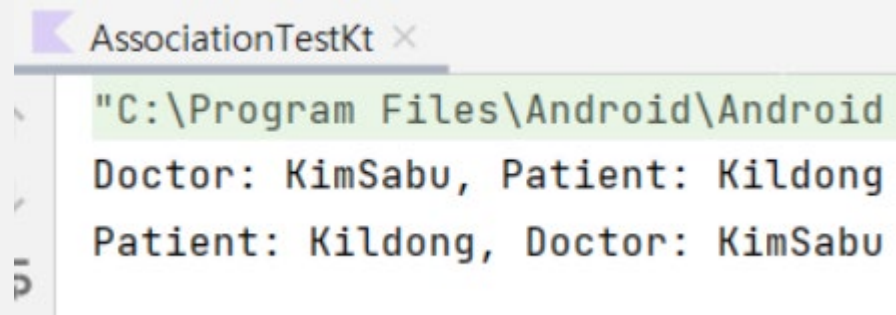
04-1 클래스 혹은 객체 간의 관계

- 관계의 판별 방법



04-1 연관 관계 나타내기 - AssociationTest.kt

```
class Patient(val name: String) {  
  
    fun doctorList(d: Doctor) { // 인자로 참조  
        println("Patient: $name, Doctor: ${d.name}")  
    }  
}  
  
class Doctor(val name: String) {  
  
    fun patientList(p: Patient) { // 인자로 참조  
        println("Doctor: $name, Patient: ${p.name}")  
    }  
}  
  
fun main() {  
    val doc1 = Doctor("KimSabu") // 객체가 따로 생성된다  
    val patient1 = Patient("Kildong")  
    doc1.patientList(patient1)  
    patient1.doctorList(doc1)  
}
```

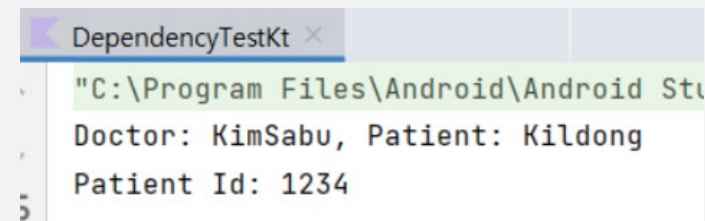


AssociationTestKt ×

```
"C:\Program Files\Android\Android  
Doctor: KimSabu, Patient: Kildong  
Patient: Kildong, Doctor: KimSabu
```

04-1 의존 관계 나타내기 - DependencyTest.kt

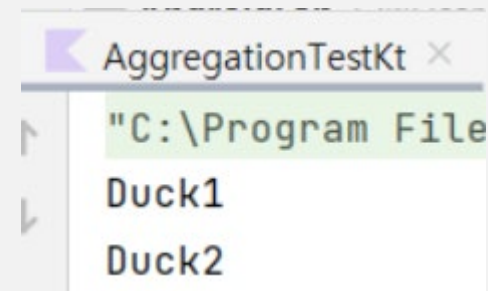
```
class Patient(val name: String, var id: Int) {  
  
    fun doctorList(d: Doctor) {  
        println("Patient: $name, Doctor: ${d.name}")  
    }  
}  
  
class Doctor(val name: String, val p: Patient) {  
  
    val customerId: Int = p.id  
  
    fun patientList() {  
        println("Doctor: $name, Patient: ${p.name}")  
        println("Patient Id: $customerId")  
    }  
}  
  
fun main() {  
    val patient1 = Patient("Kildong", 1234)  
    val doc1 = Doctor("KimSabu", patient1)  
    doc1.patientList()  
}
```



```
DependencyTestKt x  
"C:\Program Files\Android\Android Stu  
Doctor: KimSabu, Patient: Kildong  
Patient Id: 1234
```

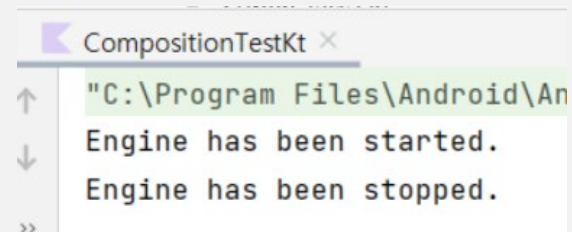
04-1 연못의 오리들로 집합 관계 - AggregationTest.kt

```
// 여러 마리의 오리를 위한 List 매개변수
class Pond(_name: String, _members: MutableList<Duck>) {
    val name: String = _name
    val members: MutableList<Duck> = _members
    constructor(_name: String): this(_name, mutableListOf<Duck>())
}
class Duck(val name: String)
fun main() {
    // 두 개체는 서로 생명주기에 영향을 주지 않는다.
    val pond = Pond("myFavorite")
    val duck1 = Duck("Duck1")
    val duck2 = Duck("Duck2")
    // 연못에 오리를 추가 - 연못에 오리가 집합한다
    pond.members.add(duck1)
    pond.members.add(duck2)
    // 연못에 있는 오리들
    for (duck in pond.members) {
        println(duck.name)
    }
}
```

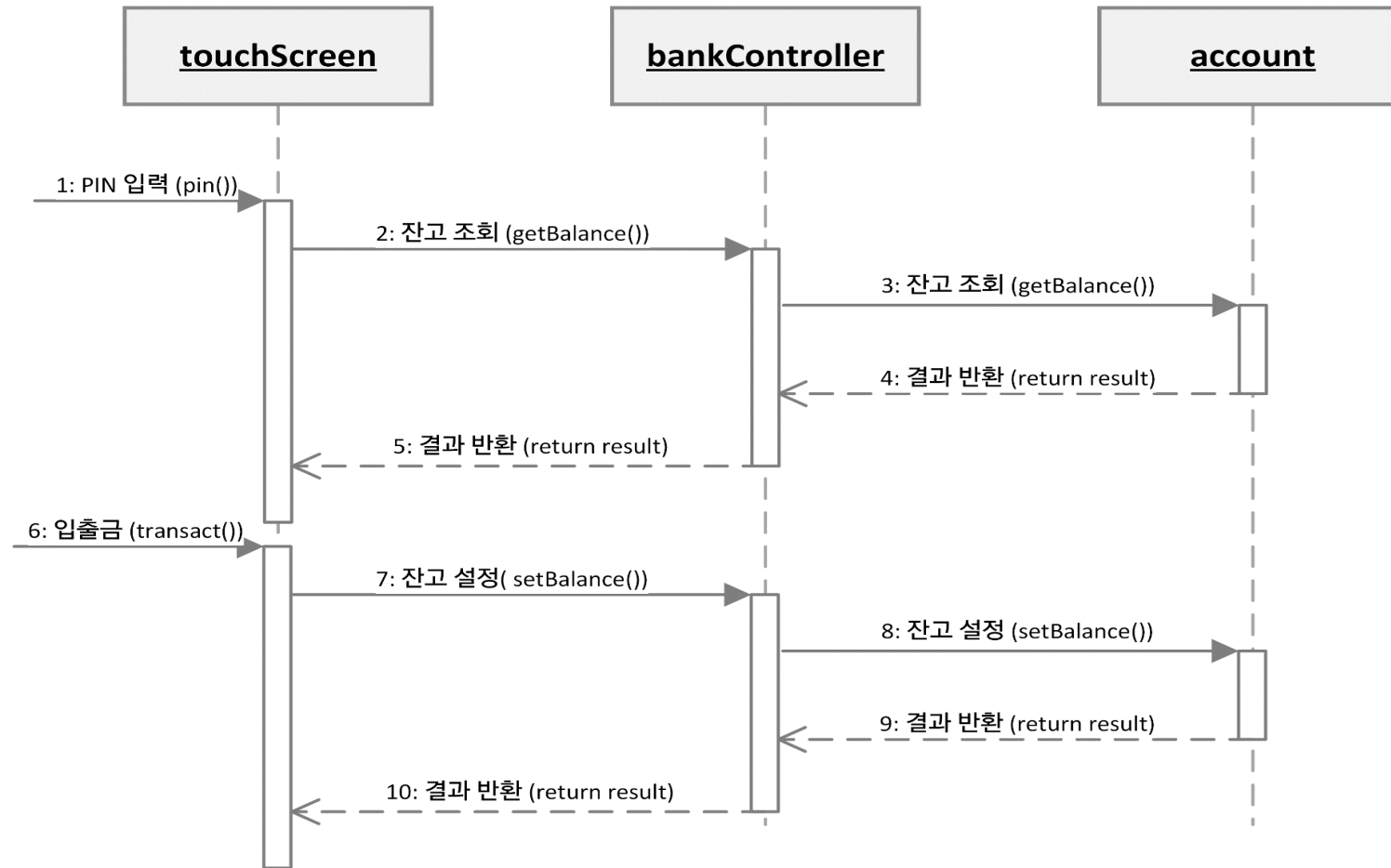


04-1 구성 관계 나타내기 - CompositionTest.kt

```
class Car(val name: String, val power: String) {  
    private var engine = Engine(power) // Engine 클래스 객체는 Car에 의존적  
  
    fun startEngine() = engine.start()  
    fun stopEngine() = engine.stop()  
}  
  
class Engine(power: String) {  
    fun start() = println("Engine has been started.")  
    fun stop() = println("Engine has been stopped.")  
}  
  
fun main() {  
    val car = Car("tico", "100hp")  
    car.startEngine()  
    car.stopEngine()  
}
```



04-1 객체 간의 메시지 전달

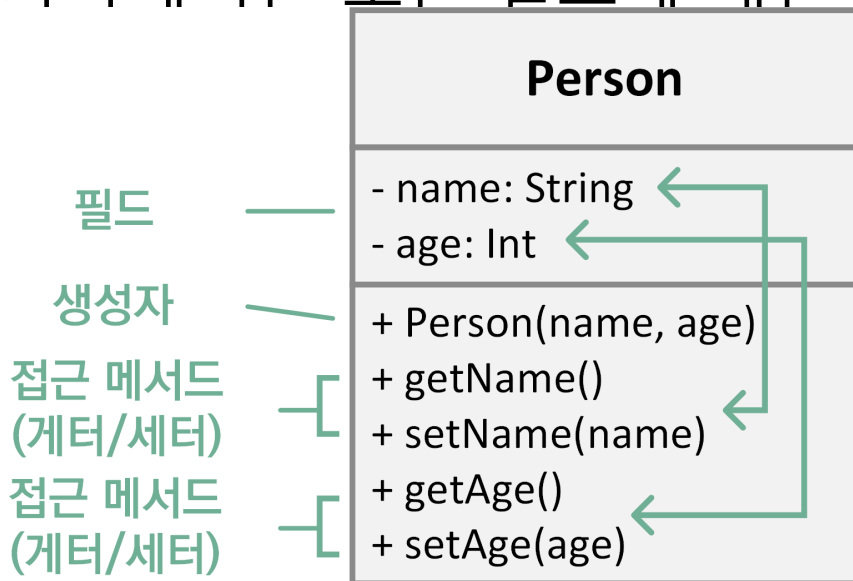


04-1 프로퍼티의 접근

- 자바의 필드(Fields)
 - 단순한 변수 선언만 가지기 때문에 접근을 위한 메서드를 따로 만들어야 함
- 코틀린의 프로퍼티(Properties)
 - 변수 선언과 기본적인 접근 메서드를 모두 가지고 있음
 - 따로 접근 메서드를 만들지 않아도 내부적으로 생성하게 됨

04-1 자바에서 필드를 사용할 때

- 게터(Getter)와 세터(Setter)의 구성
 - 게터와 세터를 합쳐 접근 메서드(Access methods)라고 함
 - 자바에서는 모든 필드에 대한 접근 메서드를 만들어야 하는 수고를 해야



04-1 자바의 Person 클래스와 접근 메서드 - PersonTest.java

```
class Person {  
    // 멤버 필드  
    private String name;  
    private int age;  
    // 생성자  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // 게터와 세터  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
    ...  
}
```


04-1 게터와 세터가 작동하는 방식

- 접근 매서드는 생략 (내부적으로 생성됨)

```
// 주 생성자에 3개의 매개변수 정의
class User(_id: Int, _name: String, _age: Int) {
    // 프로퍼티들
    val id: Int = _id           // 불변 (읽기 전용)
    var name: String = _name    // 변경 가능
    var age: Int = _age         // 변경 가능
}
```

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

- 좀 더 간략화 하면

04-1 코틀린에서 게터와 세터가 작동하는 방식

- 게터와 세터의 동작

```
fun main() {  
    val user = User(1, "Sean", 30)
```

// 게터에 의한 값 획득

```
val name = user.name
```

String name = user.getName(); // Getter

// 세터에 의한 값 지정

```
user.age = 41
```

user.setAge(41); // Setter

```
println("name: $name, ${user.age}")
```

```
}
```

04-1 기본 게터와 세터 직접 지정

- 게터와 세터가 포함되는 프로퍼티 선언 구조

```
var 프로퍼티이름[: 프로퍼티자료형] [= 프로퍼티 초기화]  
    [get() { 게터 본문 } ]  
    [set(value) {세터 본문}]
```

```
val 프로퍼티이름[: 프로퍼티자료형] [= 프로퍼티 초기화]  
    [get() { 게터 본문 } ]
```

- 불변형인 val은 게터만 설정 가능

04-1 기본 게터와 세터 지정 - NormalGetterSetter.kt

```
class User(_id: Int, _name: String, _age: Int) {  
    // 프로퍼티  
    val id: Int = _id  
        get() = field  
  
    var name: String = _name  
        get() = field  
        set(value) {  
            field = value  
        }  
  
    var age: Int = _age  
        get() = field  
        set(value) {  
            field = value  
        }  
}  
  
fun main() {  
    val user1 = User(1, "Kildong", 30)  
    // user1.id = 2 // 에러! val 프로퍼티는 값 변경 불가  
    user1.age = 35 // 세터 동작  
    println("user1.age = ${user1.age}") // 게터 동작  
}
```

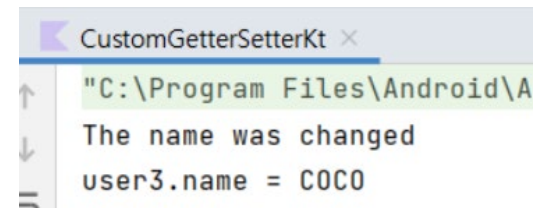
04-1 게터 세터에서 사용하는 특수 변수

- value: 세터의 매개변수로 외부로부터 값을 가져옴
 - 외부의 값을 받을 변수가 되므로 value 대신에 어떤 이름이든지 상관 없음
- field: 프로퍼티를 참조하는 변수로 보조 필드(backing field)로 불림
 - 프로퍼티를 대신할 임시 필드로 만일 프로퍼티를 직접 사용하면 게터나 세터가 무한 호출되는 재귀에 빠짐

04-1 커스텀 게터와 세터 사용하기 - CustomGetterSetter.kt

// 커스텀 게터와 세터의 사용

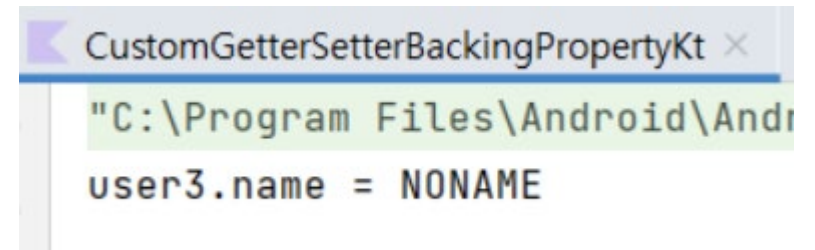
```
class User(_id: Int, _name: String, _age: Int) {  
    val id: Int = _id  
    var name: String = _name  
        set(value) {  
            println("The name was changed")  
            field = value.toUpperCase() // ① 받은 인자를 대문자로 변경해 프로퍼티에 할당  
        }  
  
    var age: Int = _age  
}  
  
fun main() {  
    val user1 = User(1, "kildong", 35)  
    user1.name = "coco" // ② 여기서 사용자 고유의 출력 코드가 실행된다.  
    println("user3.name = ${user1.name}")  
}
```



04-1 임시적인 보조 프로퍼티 - CustomGetterSetterBackingProperty.kt

```
class User(_id: Int, _name: String, _age: Int) {
    val id: Int = _id
    private var tempName: String? = null
    var name: String = _name
    get() {
        if (tempName == null) tempName = "NONAME"
        return tempName ?: throw AssertionError("Asserted by others")
    }
    var age: Int = _age
}

fun main() {
    val user1 = User(1, "kildong", 35)
    user1.name = ""
    println("user3.name = ${user1.name}")
}
```



04-1 프로퍼티의 오버라이딩 사용하기 - PropertyOverride.kt

```
open class First {
    open val x: Int = 0 // ① 오버라이딩 가능
    get() {
        println("First x")
        return field
    }
    val y: Int = 0 // ② open 키워드가 없으면 final 프로퍼티임
}
class Second : First() {
    override val x: Int = 0 // ③ 부모와 구현이 다름
    get() {
        println("Second x")
        return field + 3
    }
    // override val y: Int = 0 // ④ 에러! 오버라이딩 불가
}
fun main() {
    val second = Second()
    println(second.x) // ⑤ 오버라이딩된 두번째 클래스 객체의 x
    println(second.y) // 부모로 부터 상속 받은 값
}
```


04-1 지연 초기화가 필요한 이유

- 변수나 객체의 값은 생성시 초기화 필요
 - 클래스에서는 기본적으로 선언하는 프로퍼티 자료형들은 null을 가질 수 없음
 - 하지만, 객체의 정보가 나중에 나타나는 경우 나중에 초기화 할 수 있는 방법 필요
 - 지연 초기화를 위해 lateinit과 lazy 키워드 사용

04-1 lateinit를 사용한 지연 초기화

- 의존성이 있는 초기화나 unit 테스트를 위한 코드를 작성 시
 - 예) Car클래스의 초기화 부분에 Engine 클래스와 의존성을 가지는 경우 Engine 객체가 생성되지 않으면 완전하게 초기화 할 수 없는 경우
 - 예) 단위(Unit) 테스트를 위해 임시적으로 객체를 생성 시켜야 하는 경우
- 프로퍼티 지연 초기화
 - 클래스를 선언할 때 프로퍼티 선언은 null을 허용하지 않는다.
 - 하지만, 지연 초기화를 위한 lateinit 키워드를 사용하면 프로퍼티에 값이 바로 할당되지 않아도 됨

04-1 lateinit를 사용한 지연 초기화

- lateinit의 제한
 - var로 선언된 프로퍼티만 가능
 - 프로퍼티에 대한 게터와 세터를 사용할 수 없음

04-1 lateinit을 이용해 늦은 초기화하기 - LateinitTest.kt

```
class Person {  
    lateinit var name: String // ① 늦은 초기화를 위한 선언  
  
    fun test() {  
        if(!::name.isInitialized) { // ② 프로퍼티의 초기화 여부 판단  
            println("not initialized")  
        } else {  
            println("initialized")  
        }  
    }  
}  
  
fun main() {  
    val kildong = Person()  
    kildong.test()  
    kildong.name = "Kildong" // ③ 이 시점에서 초기화됨(지연 초기화)  
    kildong.test()  
    println("name = ${kildong.name}")  
}
```

04-1 객체 지연 초기화

- 객체 생성 시 lateinit을 통한 지연 초기화 가능

```
data class Person(var name:String, var age:Int)

lateinit var person1: Person // 객체 생성의 지연 초기화

fun main() {
    person1 = Person("Kildong",30) // 생성자 호출 시점에서 초기화됨
    print(person1.name + " is " + person1.age.toString())
}
```

04-1 lazy를 사용한 지연 초기화

- lazy를 통한 지연 초기화 특징_val만 가능
 - 호출 시점에 by lazy {...} 정의에 의해 블록 부분의 초기화를 진행한다.
 - 불변의 변수 선언인 val에서만 사용 가능하다.(읽기 전용)
 - val이므로 값을 다시 변경할 수 없다.

04-1 by lazy로 선언된 프로퍼티 지연 초기화 하기

- ByLazyTest.kt

```
class LazyTest {  
    init {  
        println("init block") // ②  
    }  
    val subject by lazy {  
        println("lazy initialized") // ⑥  
        "Kotlin Programming" // ⑦ lazy 반환값  
    }  
    fun flow() {  
        println("not initialized") // ④  
        println("subject one: $subject") // ⑤ 최초 초기화 시점!  
        println("subject two: $subject") // ⑧ 이미 초기화된 값 사용  
    }  
}  
  
fun main() {  
    val test = LazyTest() // ①  
    test.flow() // ③  
}
```

ByLazyTestKt x

"C:\Program Files\Android\Android S

init block
not initialized
lazy initialized
subject one: Kotlin Programming
subject two: Kotlin Programming

04-1 by lazy로 선언된 객체 지연 초기화

• BvLazvObi.kt

```
class Person(val name: String, val age: Int)
fun main() {
    var isPersonInstantiated: Boolean = false // ① 초기화 확인 용도

    val person : Person by lazy { // ② lazy를 사용한 person 객체의 지연 초기화
        isPersonInstantiated = true
        Person("Kim", 23) // ③ 이 부분이 Lazy 객체로 반환 됨
    }

    val personDelegate = lazy { Person("Hong", 40) } // ④ 위임 변수를 사용한 초기화

    println("person Init: $isPersonInstantiated")
    println("personDelegate Init: ${personDelegate.isInitialized()}")

    println("person.name = ${person.name}") // ⑤ 이 시점에서 초기화
    println("personDelegate.value.name = ${personDelegate.value.name}") // ⑥ 이 시점에서 초기화

    println("person Init: $isPersonInstantiated")
    println("personDelegate Init: ${personDelegate.isInitialized()}")
}
```

ByLazyObjKt ×

```
"C:\Program Files\Android\Android S
person Init: false
personDelegate Init: false
person.name = Kim
personDelegate.value.name = Hong
person Init: true
personDelegate Init: true
```


04-1 by lazy의 모드

- 3가지 모드 지정 가능
 - SYNCHRONIZED 락을 사용해 단일 스레드만이 사용하는 것을 보장(기본값)
 - PUBLICATION 여러 군데서 호출될 수 있으나 처음 초기화된 후 반환값의 사요

```
private val model by lazy(mode = LazyThreadSafetyMode.NONE) {  
    Injector.app().transactionsModel() // 이 코드는 단일 스레드의 사용이 보장될 때  
}
```

(값의 일관성을 보장할 수 없음)

04-1 by를 이용한 위임

- 위임(delegation)
 - 하나의 클래스가 다른 클래스에 위임하도록 선언
 - 위임된 클래스가 가지는 멤버를 참조없이 호출

< val|var|class> 프로퍼티 혹은 클래스 이름: 자료형 by 위임자

04-1 클래스의 위임

- 다른 클래스의 멤버를 사용하도록 위임

```
interface Animal {  
    fun eat() { ... }  
    ...  
}  
class Cat : Animal { }  
val cat = Cat()  
class Robot : Animal by cat // Animal의 정의된 Cat의 모든 멤버를 Robot에 위임함
```

- cat은 Animal 자료형의 private 멤버로 Robot 클래스 내에 저장
- Cat에서 구현된 모든 Animal의 메소드는 정적 메소드로 생성
- 따라서, Animal에 대한 명시적인 참조를 사용하지 않고도 eat()을 바로 호출

04-1 위임을 사용하는 이유?

- 코틀린의 기본 라이브러리는 open되지 않은 최종 클래스
 - 표준 라이브러리의 무분별한 상속의 복잡한 문제들을 방지
 - 단, 상속이나 직접 클래스의 기능 확장을 하기 어렵다.
- 위임을 사용하면?
 - 위임을 통해 상속과 비슷하게 최종 클래스의 모든 기능을 사용하면서 동시에 기능을 추가 확장 구현할 수 있다.

04-1 클래스의 위임 사용하기

- DelegatedClass.kt

```
interface Car {  
    fun go(): String  
}  
class VanImpl(val power: String): Car {  
    override fun go() = "는 짐을 적재하며 $power 마력을 가집니다."  
}  
class SportImpl(val power: String): Car {  
    override fun go() = "는 경주용에 사용되며 $power 마력을 가집니다."  
}  
class CarModel(val model: String, impl: Car): Car by impl {  
    fun carInfo() {  
        println("$model ${go()}") // ① 참조 없이 각 인터페이스 구현 클래스의 go를 접근  
    }  
}  
fun main() {  
    val myDamas = CarModel("Damas 2010", VanImpl("100마력"))  
    val my350z = CarModel("350Z 2008", SportImpl("350마력"))  
  
    myDamas.carInfo() // ② carInfo에 대한 다형성을 나타냄  
    my350z.carInfo()  
}
```

DelegatedClassKt ×

```
"C:\Program Files\Android\Android Studio\jre\bin\j  
Damas 2010 는 짐을 적재하며 100마력 마력을 가집니다.  
350Z 2008 는 경주용에 사용되며 350마력 마력을 가집니다.
```

04-1 프로퍼티 위임과 by lazy 다시 보기

- by lazy {...} 도 위임
 - 사용된 프로퍼티는 람다식 함수에 전달되어(위임되어) 함수에 의해 사용
- 동작 분석
 - 1. lazy 람다식 함수는 람다를 전달받아 저장한 Lazy<T> 인스턴스를 반환한다.
 - 2. 최초 프로퍼티의 게터 실행은 lazy에 넘겨진 람다식 함수를 실행하고 결과를 기록한다.
 - 3. 이후 프로퍼티의 게터 실행은 이미 초기화되어 기록된 값을 반환한다.

04-1 observable과 vetoable의 위임

- observable
 - 프로퍼티를 감시하고 있다가 특정 코드의 로직에서 변경이 일어날 때 호출
- vetoable
 - 감시보다는 수여한다는 의미로 반환값에 따라 프로퍼티 변경을 허용하거나 취소

04-1 observable의 간단한 사용의 예

- DelegatedProperty.kt

```
import kotlin.properties.Delegates
```

```
class User {
```

```
    // observable은 값의 변화를 감시하는 일종의 콜백 루틴
```

```
    var name: String by Delegates.observable("NONAME") { // ① 프로퍼티를 위임  
        prop, old, new -> // ② 람다식 매개변수로 프로퍼티, 기존값, 새로운 값  
        println("$old -> $new") // ③ 이부분은 이벤트가 발생할 때만 실행됨  
    }
```

```
}
```

```
fun main() {
```

```
    val user = User()
```

```
    user.name = "Kildong" // ④ 값이 변경되는 시점에서 첫 이벤트 발생
```

```
    user.name = "Dooly" // ⑤ 값이 변경되는 시점에서 두번째 이벤트 발생
```

```
}
```

CustomGetterSetterKt ×

The name was changed
user3.name = COCO

04-1 vetoable을 사용한 최대값

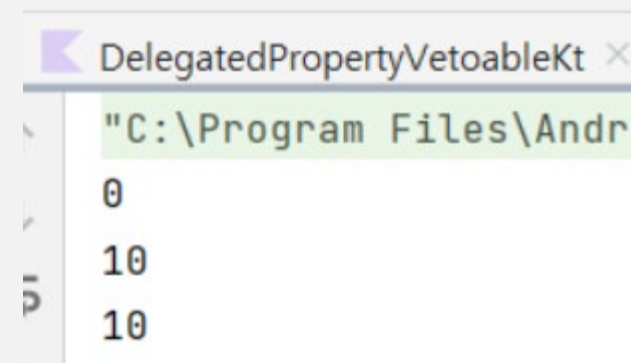
- DelegatedPropertyVetoable.kt

```
import kotlin.properties.Delegates

fun main() {
    var max: Int by Delegates.vetoable(0) { // ① 초기값은 0
        prop, old, new ->
        new > old // ② 조건에 맞지 않으면 거부권 행사
    }

    println(max) // 0
    max = 10
    println(max) // 10

    // 여기서는 기존값이 새 값보다 크므로 false
    // 따라서 5를 재할당하지 않는다.
    max = 5
    println(max) // 10
}
```



04-1 정적 변수와 컴페니언 객체

- 사용 범위에 따른 분류
 - 지역(local), 전역(global)
- 보통 클래스는 동적으로 객체를 생성하는데 정적으로 고정하는 방법은?
 - 동적인 초기화 없이 사용할 수 있는 개념으로 자바에서는 static 변수 또는 객체
 - 코틀린에서는 이것을 **컴페니언 객체(Companion object)**로 사용
 - 프로그램 실행 시 고정적으로 가지는 메모리로 객체 생성 없이 사용
 - 단, 자주 사용되지 않는 변수나 객체를 만들면 메모리 낭비

04-1 컴패니언 객체 사용해 보기

- CompanionObjectTest.kt

```
class Person {  
    var id: Int = 0  
    var name: String = "Youngdeok"  
    companion object {  
        var language: String = "Korean"  
        fun work() {  
            println("working...")  
        }  
    }  
}  
  
fun main() {  
    println(Person.language) // 인스턴스를 생성하지 않고 기본값 사용  
    Person.language = "English" // 기본값 변경 가능  
    println(Person.language) // 변경된 내용 출력  
    Person.work() // 메서드 실행  
    // println(Person.name) // name은 companion object가 아니므로 에러 }  
}
```



컴패니언 객체는 실제 객체의 싱글톤 (singleton)으로 정의됨

```
CompanionObjectTestKt x  
"C:\Program Files\A  
Korean  
English  
working...
```

04-1 코틀린에서 자바의 static 멤버의 사용

```
// 자바의 Customer 클래스
public class Customer {
    public static final String LEVEL = "BASIC"; // static 필드
    public static void login() { // static 메서드
        System.out.println("Login...");
    }
}
```

Customer.java

```
// 코틀린에서 자바의 static 접근
fun main() {
    println(Customer.LEVEL)
    Customer.login()
}
```

CustomerAccess.kt

04-1 자바에서 코틀린 컴패니언 객체 사용

- @JvmStatic
 - 자바에서는 코틀린의 컴패니언 객체를 접근하기 위해 @JvmStatic 애노테이션(annotation) 표기법을 사용

```
class KCustomer {
    companion object {
        const val LEVEL = "INTERMEDIATE"
        @JvmStatic fun login() = println("Login...") // 애노테이션 표기 사용
    }
}
```

KCustomer.kt

```
public class KCustomerAccess {
    public static void main(String[] args) {
        // 코틀린 클래스의 컴패니언 객체를 접근
        System.out.println(KCustomer.LEVEL);
        KCustomer.login(); // 애노테이션을 사용할 때 접근 방법
        KCustomer.Companion.login(); // 위와 동일한 결과로 애노테이션을 사용하지 않을 때 접근 방법
    }
}
```

KCustomerAccess.java

04-1 최상위 함수 정리

- 최상위 함수(top-level function)
 - 클래스 없이 만들었던 최상위 함수들은 객체 생성 없이도 어디에서든 실행
 - 패키지 레벨 함수(package-level function)라고도 함
 - 최상위 함수는 결국 자바에서 static final로 선언된 함수임
- 자바에서 코틀린의 최상위 함수 접근
 - 코틀린의 최상위 함수는 클래스가 없으나 자바와 연동시 내부적으로 파일명에 Kt 접미사가 붙은 클래스를 자동 생성하게 된다.
 - 자동 변환되는 클래스명을 명시적으로 지정하고자 하는 경우 @file:JvmName("ClassName")을 코드 상단에 명시한다.

04-1 자바에서 코틀린의 최상위 함수 접근

```
// 패키지 레벨 함수 혹은 최상위 함수
fun packageLevelFunc() {
    println("Package-Level Function")
}

fun main() {
    packageLevelFunc()
}
```

PackageLevelFunction.kt

```
public class PackageLevelAccess {
    public static void main(String[] args) {

        PackageLevelFunctionKt.packageLevelFunc();
    }
}
```

PackageLevelAccess.java

04-1 자바에서 코틀린의 최상위 함수 접근 - 이름 명시

```
@file:JvmName("PKLevel")
```

PackageLevelFunction.kt

```
// 패키지 레벨 함수 혹은 최상위 함수
fun packageLevelFunc() {
    println("Package-Level Function")
}

fun main() {
    packageLevelFunc()
}
```

```
public class PackageLevelAccess {
    public static void main(String[] args) {

        PKLevel.packageLevelFunc();
    }
}
```

PackageLevelAccess.java

04-1 object와 싱글톤

- 상속할 수 없는 클래스에서 내용이 변경된 객체를 생성할 때
 - 자바의 경우 익명 내부 클래스를 사용해 새로운 클래스 선언
 - 코틀린에서는 object 표현식이나 object 선언으로 이 경우를 좀 더 쉽게 처리

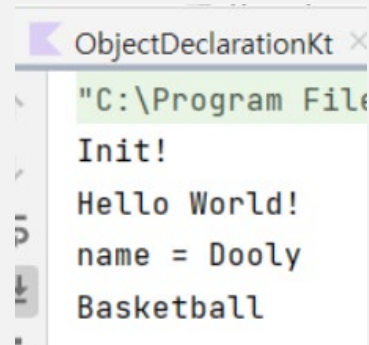
04-1 object 선언과 컴패니언 객체의 비교

• ObjectDeclaration.kt

```
// (1) object 키워드를 사용한 방식
object OCustomer {
    var name = "Kildong"
    fun greeting() = println("Hello World!")
    val HOBBY = Hobby("Basketball")
    init {
        println("Init!")
    }
}

...
class Hobby(val name: String)
fun main() {

    OCustomer.greeting() // 객체의 접근 시점
    OCustomer.name = "Dooly"
    println("name = ${OCustomer.name}")
    println(OCustomer.HOBBY.name)
    ...
}
```



```
ObjectDeclarationKt x
"C:\Program File
Init!
Hello World!
name = Dooly
Basketball
```



object 선언 방식은 접근 시점에 객체가 생성.
그렇기 때문에 생성자 호출을 하지 않으므로
object 선언에는 주/부 생성자를 사용할 수 없다.

자바에서는 OCustomer.INSTANCE.getName();
와 같이 접근해야 한다.

04-1 object 표현식

- object 표현식을 사용할 때
 - object 선언과 달리 이름이 없으며 싱글톤이 아님
 - 따라서 object 표현식이 사용될 때마다 새로운 인스턴스가 생성
 - 이름이 없는 익명 내부 클래스로 불리는 형태를 object 표현식으로 만들 수 있다.

04-1 object 표현식 사용해 보기

- ObjectExpressionSuperMan.kt

```
open class Superman() {  
    fun work() = println("Taking photos")  
    fun talk() = println("Talking with people.")  
    open fun fly() = println("Flying in the air.")  
}
```

```
fun main() {  
    val pretendedMan = object: Superman() { // (1) object 표현식으로 fly()구현의 재정의  
        override fun fly() = println("I'm not a real superman. I can't fly!")  
    }  
    pretendedMan.work()  
    pretendedMan.talk()  
    pretendedMan.fly()  
}
```



하위 클래스를 만들지 않고도 새로운 구현인 fly()를 포함할 수 있음

ObjectExpressionSuperManKt x

```
"C:\Program Files\Android\Android Studio  
Taking photos  
Talking with people.  
I'm not a real superman. I can't fly!"
```