둘째마당 – 코틀린 객체지향 프로그래밍

05 클래스와 객체
06 프로퍼티와 접근 메서드
07 다양한 클래스와 인터페이스

Copyright Note

- Book: Do it! 코틀린 프로그래밍 / 이지스퍼블리싱
- Author: 황영덕 (Youngdeok Hwang; sean.ydhwang@gmail.com)
- Update Date: 10-July-2019
- Issue Date: 25-Nov-2017
- Slide Revision #: rev02
- Homepage: acaroom.net
- Distributor: 이지스퍼블리싱 (담당: 박현규)

05장 – 클래스와 객체

05 클래스와 객체

05-1 클래스와 객체의 정의

05-2 생성자

05-3 상속과 다형성

05-4 super와 this의 참조

05-5 정보 은닉 캡슐화

05-6 클래스와 관계

객체 지향 프로그래밍

- OOP; Object-Oriented Programming
  - 프로그램의 구조를 객체 간 상호작용으로서 표현하는 프로그래밍 방식
  - 절차적 프로그래밍의 한계를 극복하고자 나온 언어의 한 가지 방법론
  - 객체와 관계를 표현하고 이를 통해 확장과 재사용이 용이

- 코틀린에서는 OOP를 지원

객체지향의 기본 용어

- 객체지향 개념상의 용어들
  - 추상화(abstraction)
  - 인스턴스(instance)
  - 상속(inheritance)
  - 다형성(polymorphism)
  - 캡슐화(encapsulation)
  - 메시지 전송(message sending)
  - 연관(association)

클래스와 추상화

- 클래스(Class)란
  - 분류, 계층, 등급 등의 우리말 뜻
  - 특정 대상을 분류하고 특징(속성)과 동작 활동(함수) 내용을 기록

- 추상화(Abstraction)
  - 목표로 하는 것에 대해 필요한 만큼 속성과 동작을 정의하는 과정

객체지향 개념의 동의어들

- 객체지향 개념상의 용어가 언어마다 약간씩 다르다.

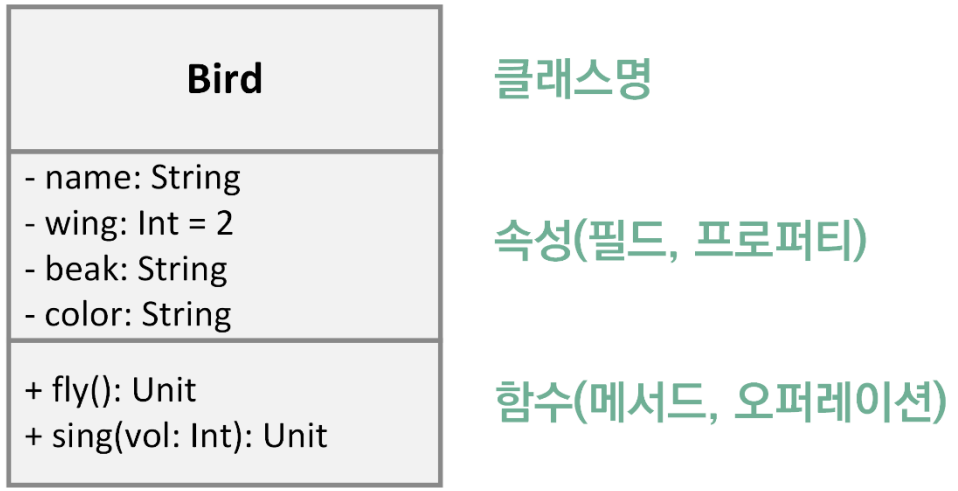
코틀린에서 사용하는 용어	그 밖에 용어
클래스(Class)	분류, 범주
프로퍼티(Property)	속성(Attribute), 변수(Variable), 필드(Field), 데이터(Data)
메서드(Method)	함수(Function), 동작(Operation), 행동(Behavior)
객체(Object)	인스턴스(Instance)

  - 자바에서 사용하는 필드는 코틀린에서 프로퍼티로 부른다.

클래스

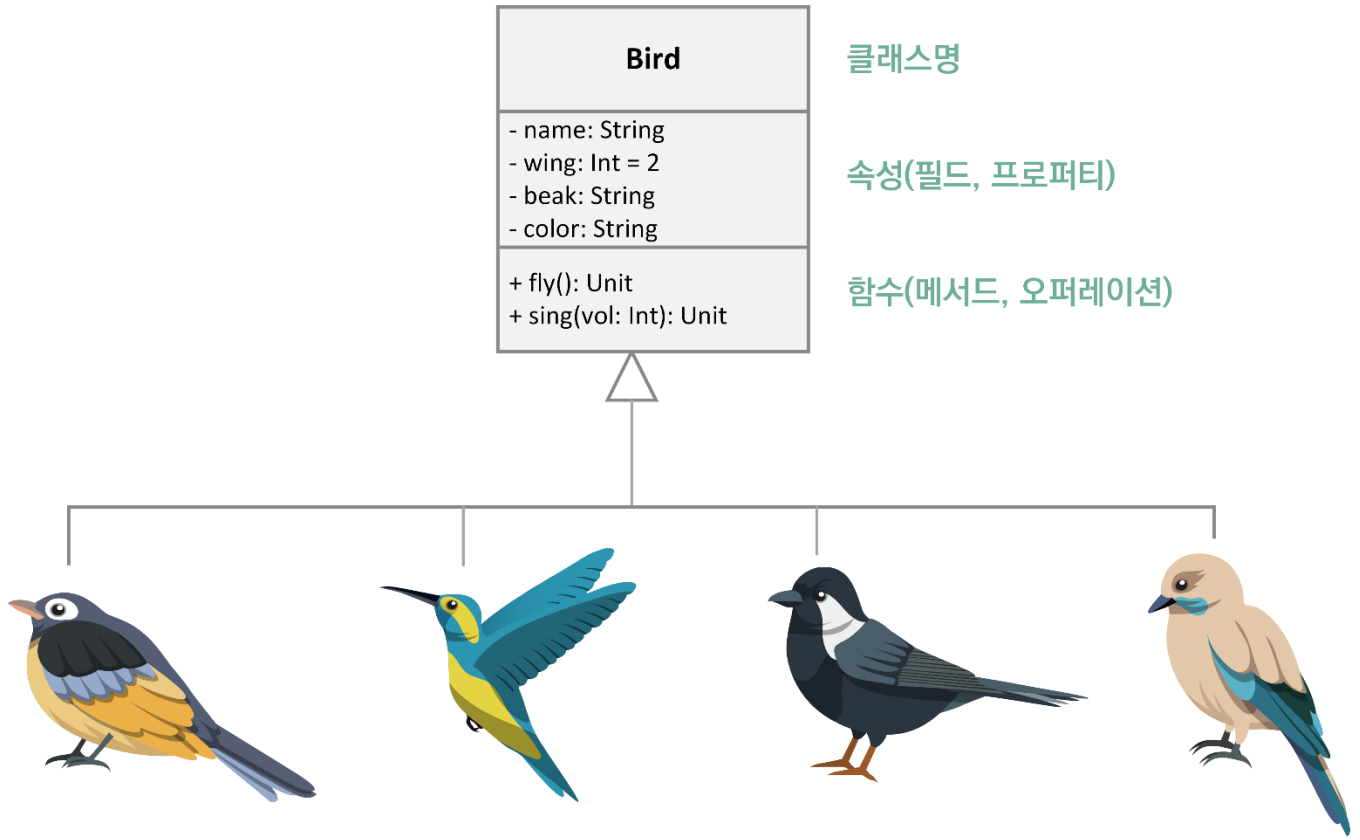
클래스 다이어그램

- 클래스를 시각적으로 나타내 분석과 개념 구현에 용이



클래스

- 추상화 과정을 통한 속성과 동작 골라내기



클래스의 선언

- 빈 형태의 클래스 선언

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

- 클래스내에 프로퍼티와 메서드가 정의된 경우

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

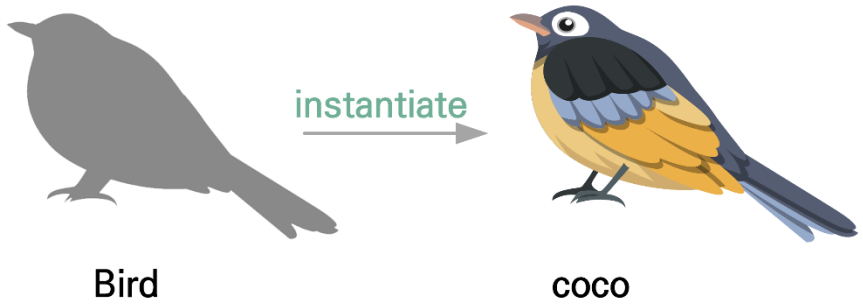
코딩해 보세요! Bird 클래스 만들어 보기

- BirdClassDefine.kt

```
...
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 = "blue"  // 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)
}
```

객체와 인스턴스 정리

- 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 올라간 객체를 인스턴스(instance)라고도 부름




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

05 클래스와 객체

05-1 클래스와 객체의 정의

05-2 생성자

05-3 상속과 다형성

05-4 super와 this의 참조

05-5 정보 은닉 캡슐화

05-6 클래스와 관계

생성자

- 생성자(Constructor)란
  - 클래스를 통해 객체가 만들어질 때 기본적으로 호출되는 함수
  - 객체 생성 시 필요한 값을 인자로 설정할 수 있게 한다.
  - 생성자를 위해 특별한 함수인 constructor()를 정의

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

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.

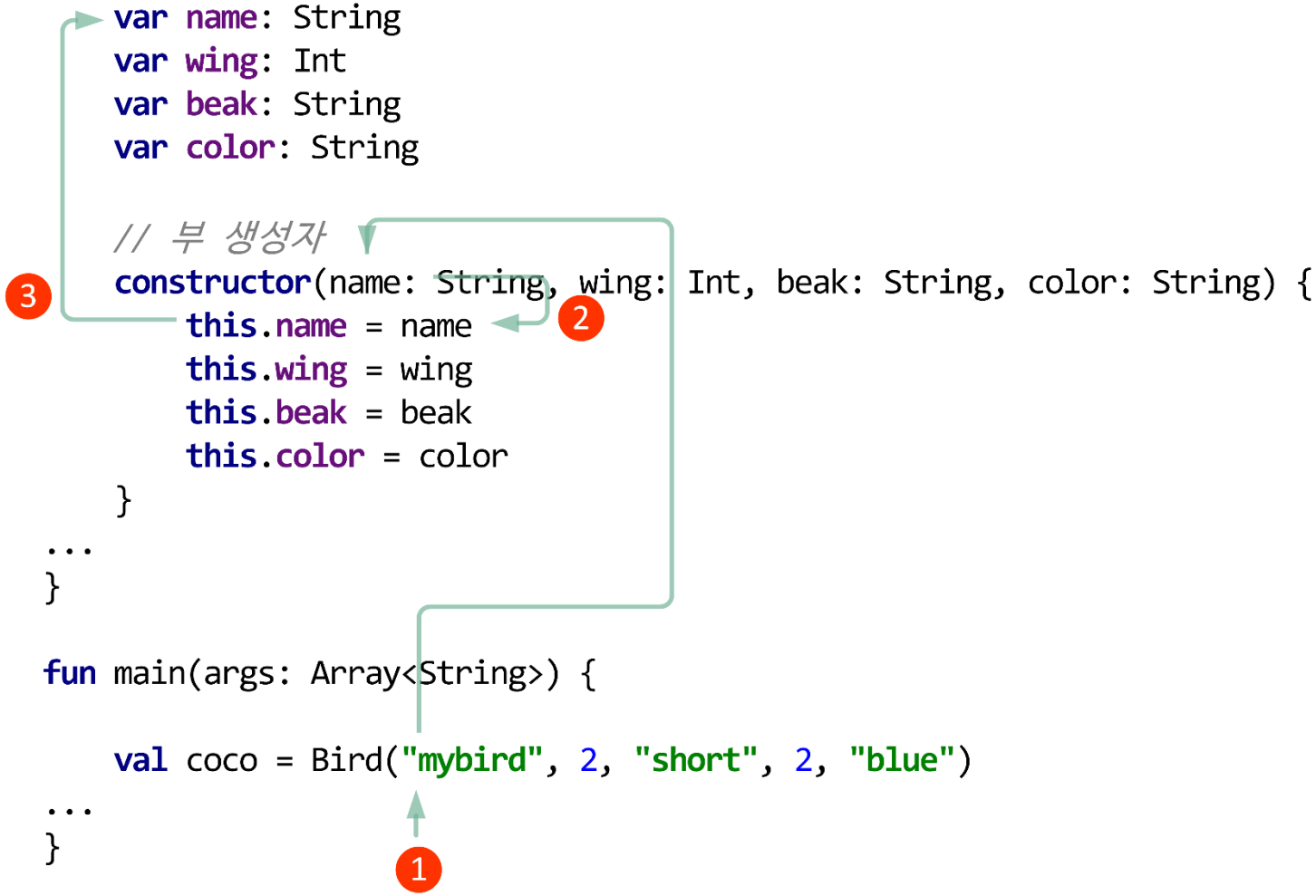
코딩해 보세요! 부 생성자를 사용하는 Bird 클래스

- 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
 ...
}
```

객체 생성 시 생성자로부터 일어나는 일

```
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")
...
}
```



this 키워드를 생략하고 하는 경우

- 생성자의 매개변수와 프로퍼티의 이름을 다르게 구성

```
...
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
}
...
```

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

여러 개의 부 생성자 지정의 예

```
// 주 생성자가 없고 여러 개의 보조 생성자를 가진 클래스
class Bird {
    // 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) {
        name = _name
        wing = _wing
        beak = _beak
        color = _color
    }

    // 두 번째 부 생성자
    constructor(_name: String, _beak: String) {
        name = _name
        wing = 2
        beak = _beak
        color = "grey"
    }
    ...
}
```

여러 개의 부 생성자 지정의 예

- 객체 생성 시 생성자의 인자에 따라 부 생성자 선택

```
val bird1 = Bird("mybird", 2, "short", "blue") // 첫번째 부 생성자 호출
val bird2 = Bird("mybird2", "long") // 두번째 부 생성자 호출
```

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")
}
...
```

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")
}
...
```

  - 가시성지시자나 애노테이션 표기가 없을 경우 생략 가능

주 생성자

- 클래스명과 함께 생성자 정의 - 프로퍼티가 포함된 주 생성자

```
...
// 주 생성자 선언
class Bird(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")
}
...
```

주 생성자의 초기화 블록

- 초기화 블록이 포함된 주 생성자 - BirdPrimaryInit.kt

```
...
// 주 생성자 선언
class Bird(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")
}
...
```

초기화 블록에는 간단한 코드가 허용된다.

프로퍼티의 기본값 지정

- 생성시 필요한 기본값을 지정할 수 있다.

```
// 프로퍼티의 기본값 지정
class Bird(var name: String = "NONAME", var wing: Int = 2, var beak: String, var color: String) {
    ...
}

fun main() {

    val coco = Bird(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}")
}
```

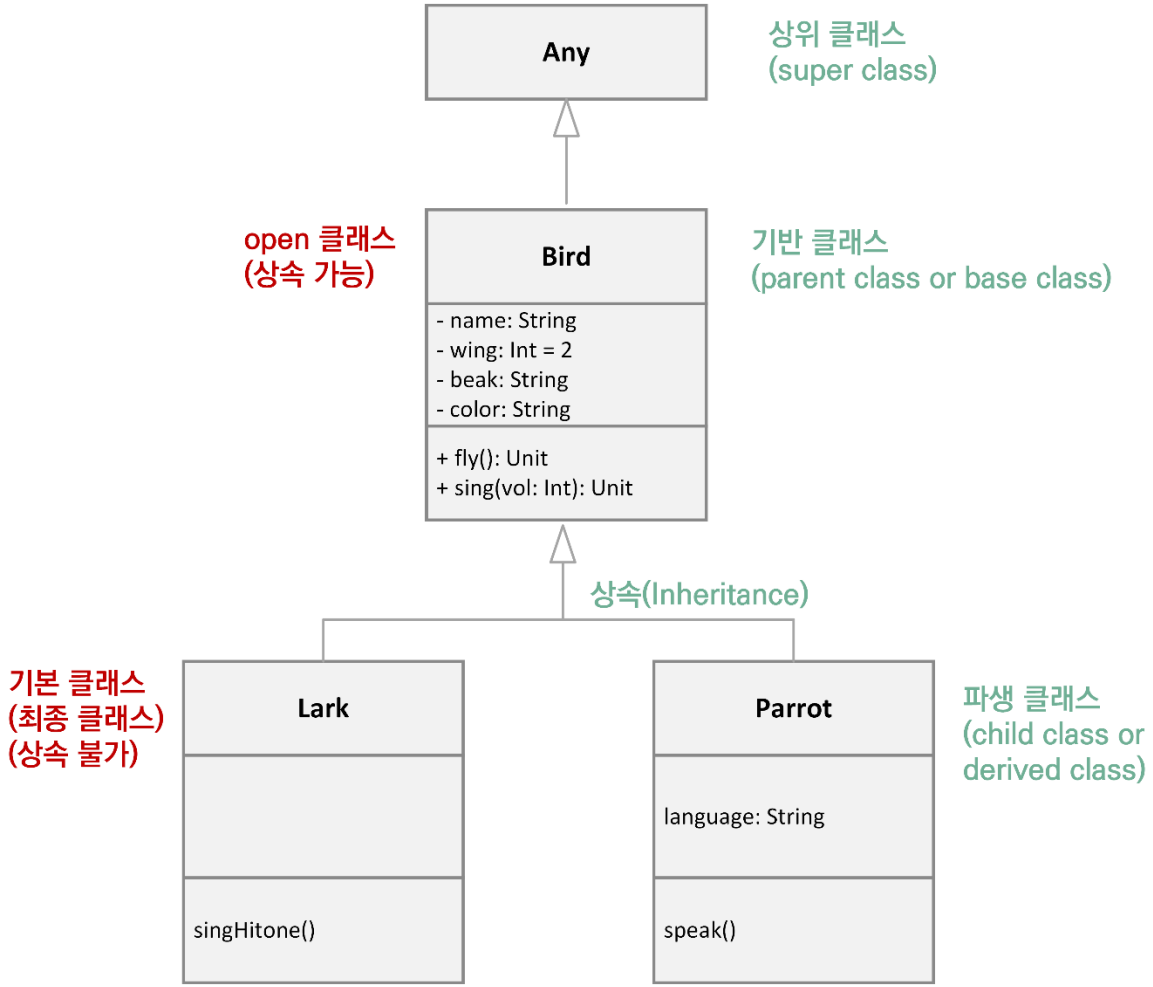
05 클래스와 객체

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

상속의 예

- 종달새(Lark)와 앵무새(Parrot) 클래스



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

코딩해 보세요! 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")
}
...
```

코딩해 보세요! 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()
}
```

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

다형성

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

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

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
}
```

코딩해 보세요! 덧셈 동작의 오버로딩 - OverloadCalc.kt

- 클래스 메서드의 오버로딩

```
package chap05.section3

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
}
```

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.

오버라이딩의 예

- 메서드 오버라이딩의 예

```
open class Bird { // 여기의 open은 상속 가능을 나타냄
...
    fun fly() { ... } // ① 최종 메서드로 오버라이딩 불가
    open fun sing() {...} // ② sing() 메서드는 하위 클래스에서 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
}
```

오버라이딩 금지

- 파생 클래스에서 오버라이딩을 금지할 때

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

코딩해 보세요! 메서드를 오버라이딩 하기 - BirdOverrideEx.kt

```
// 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
}
```

05 클래스와 객체

super와 this

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

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

super로 상위 참조

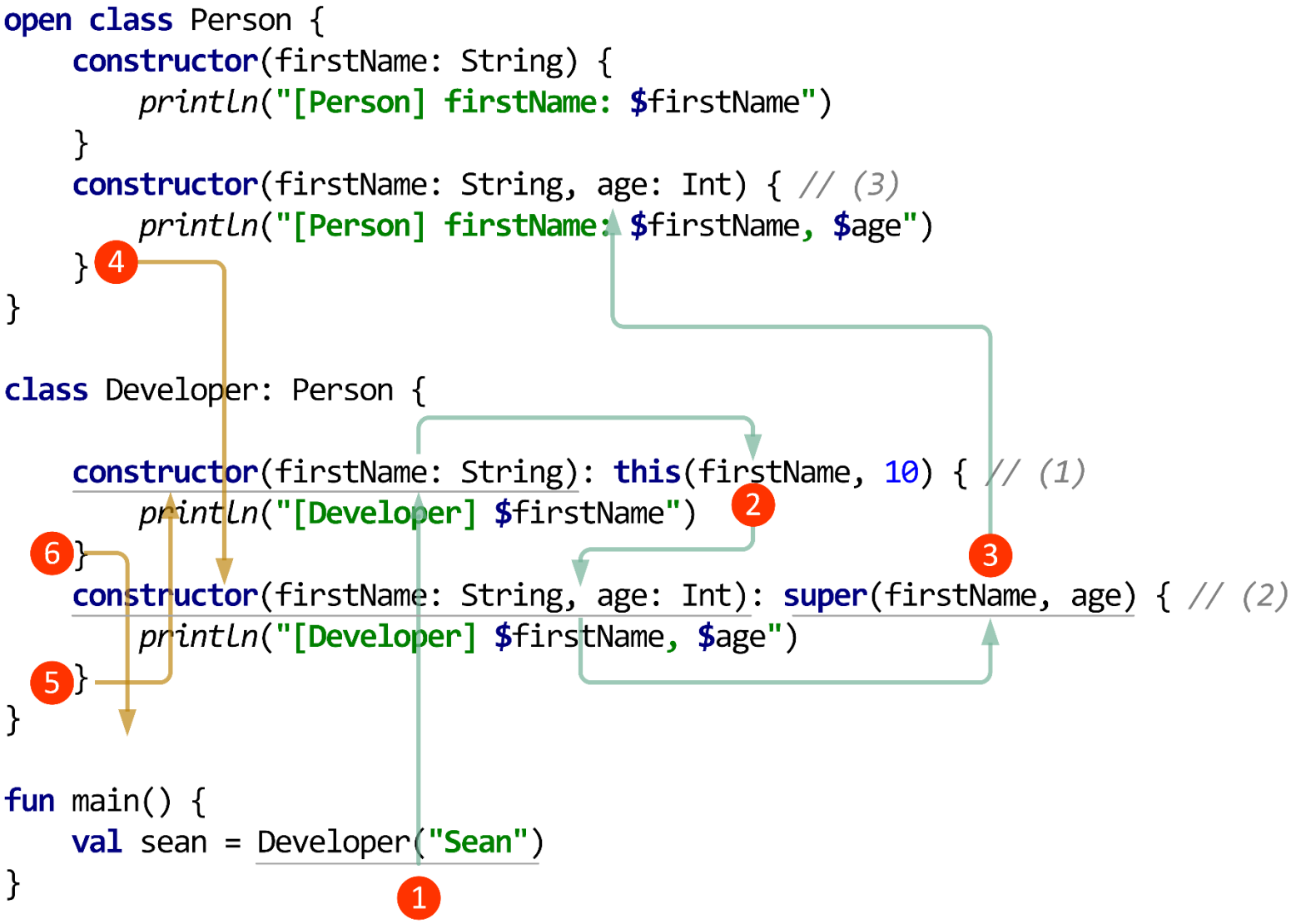
- 상위 클래스의 메서드 실행

```
open class Bird(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 Parrot(name: String, wing: Int = 2, beak: String, color: String,
             var language: String = "natural") : Bird(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()
    }
}
```

코딩해 보세요! this와 super를 사용하는 부 생성자 - PersonThisSuper.kt

```
package chap05.section4.personthis

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")
}
```

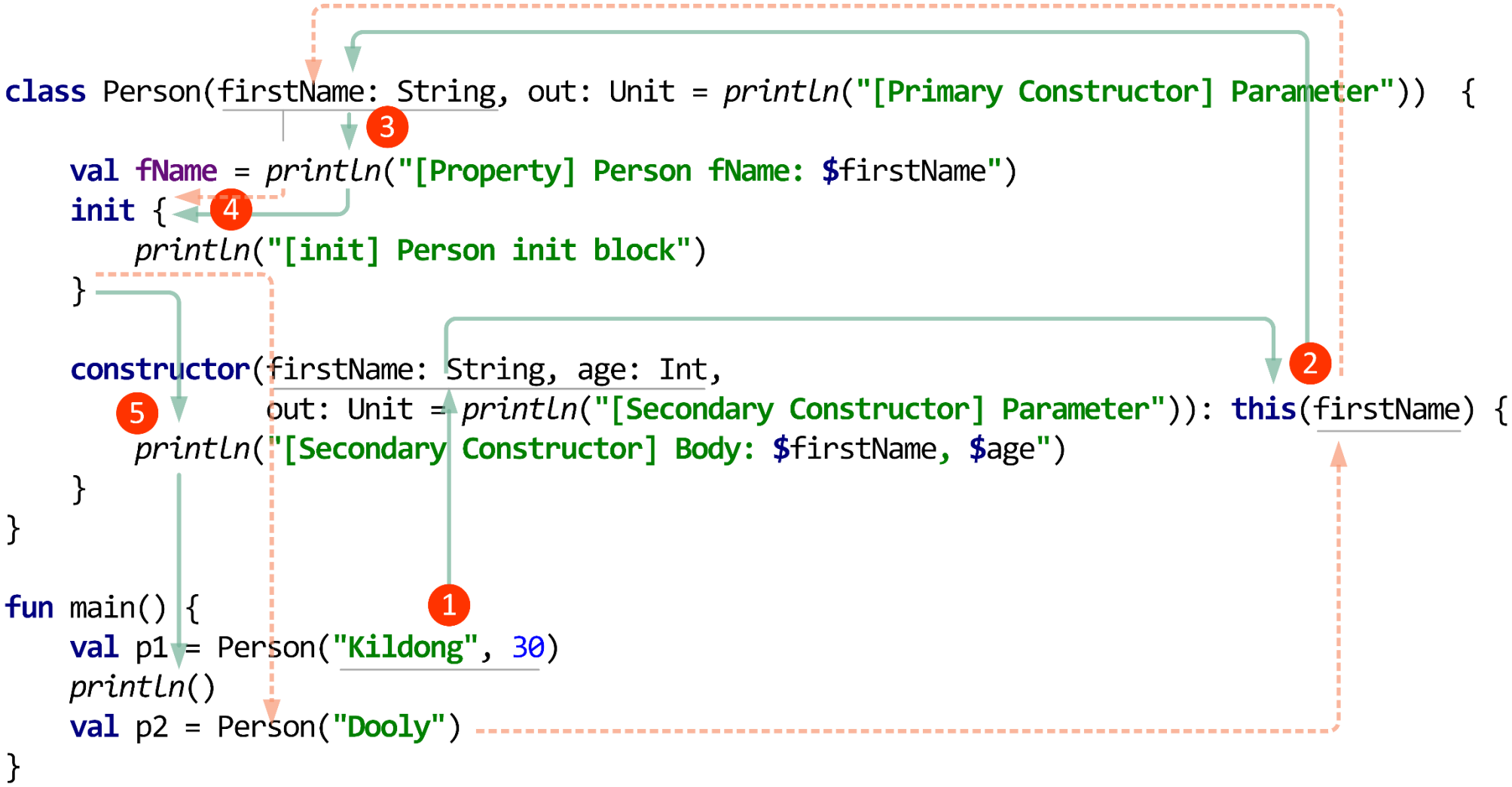
PersonThisSuper.kt 호출 순서



코딩해 보세요! 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
}
```

PersonPriSeconRef.kt 호출 순서



바깥 클래스 호출하기

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

코딩해 보세요! 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
}
```

코딩해 보세요! 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()
}
```

05 클래스와 객체

정보 은닉 캡슐화

- 캡슐화(encapsulation)
  - 클래스를 작성할 때 외부에서 숨겨야 하는 속성이나 기능

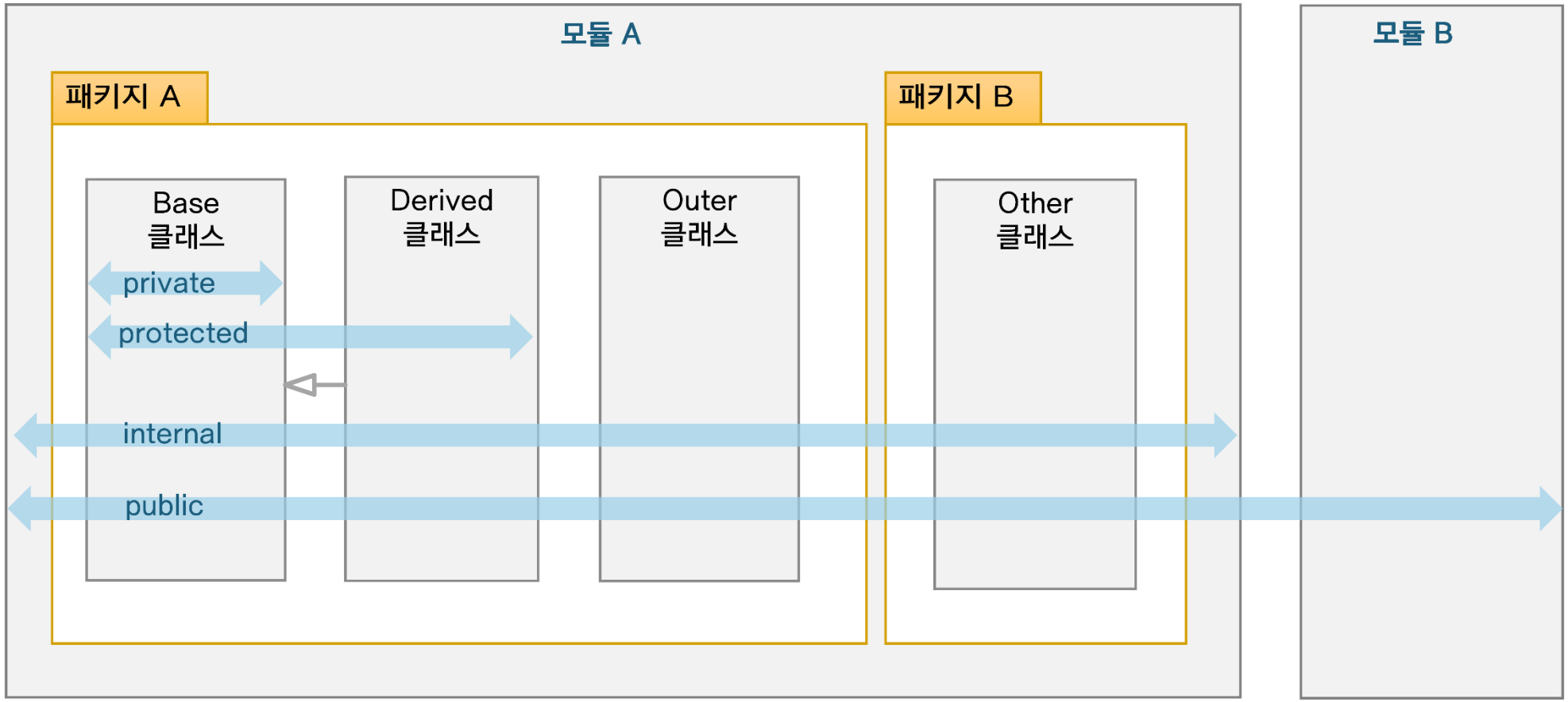
  - 가시성 지시자(visibility modifiers)를 통해 외부 접근 범위를 결정할 수 있음
    - private: 이 지시자가 붙은 요소는 외부에서 접근할 수 없음
    - public: 이 요소는 어디서든 접근이 가능 (기본값)
    - protected: 외부에서 접근할 수 없으나 하위 상속 요소에서는 가능
    - internal: 같은 정의의 모듈 내부에서는 접근이 가능

정보 은닉 캡슐화

- 가시성 지시자의 선언 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
}
```

가시성 지시자의 공개 범위



코딩해 보세요! 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 되야 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
}
```

코딩해 보세요! 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
}
```

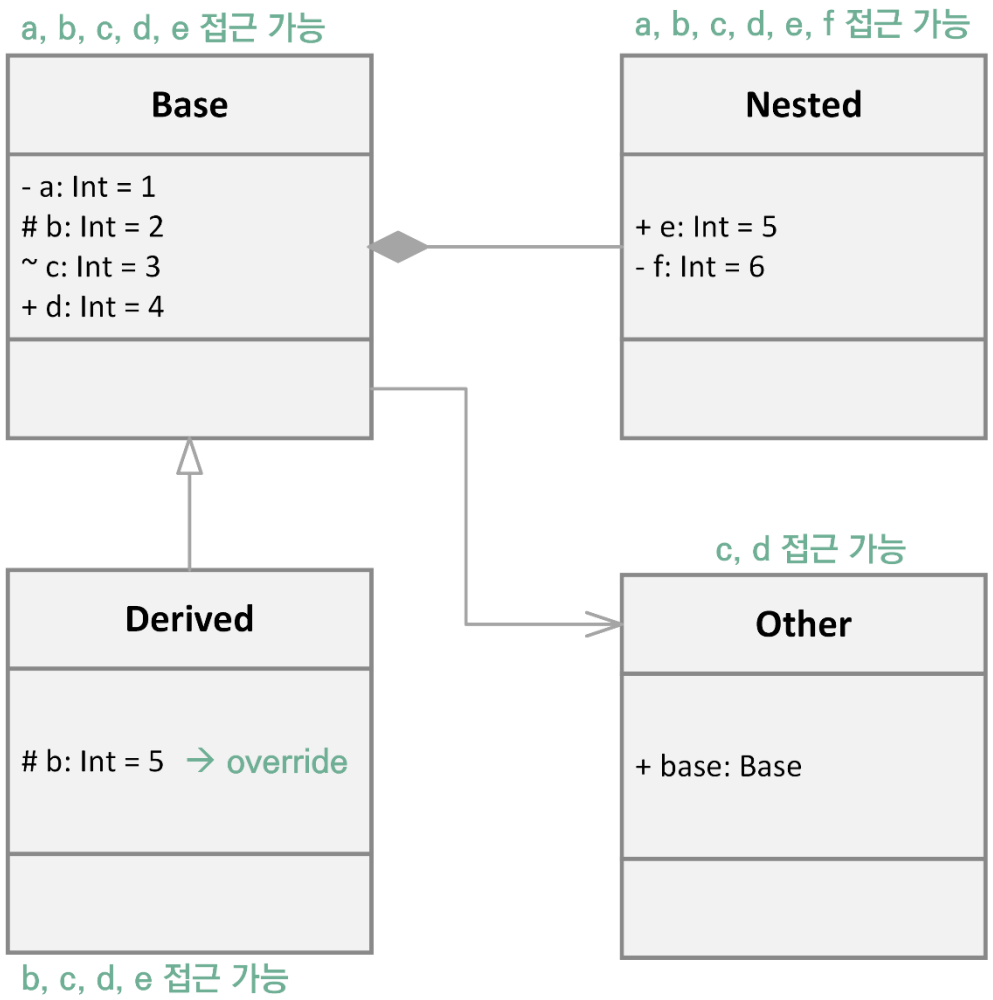
코딩해 보세요! internal 가시성 테스트하기 - InternalTest.kt

```kotlin
package chap05.section5.internal

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
}
```

코딩해 보세요! **internal** 가시성 테스트 - InternalTestOtherFile.kt

```
package chap05.section5.internal

fun main() {

    val otheric = InternalClass()

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

가시성 지시자와 클래스의 관계



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
}
```

05 클래스와 객체

05-1 클래스와 객체의 정의

05-2 생성자

05-3 상속과 다형성

05-4 super와 this의 참조

05-5 정보 은닉 캡슐화

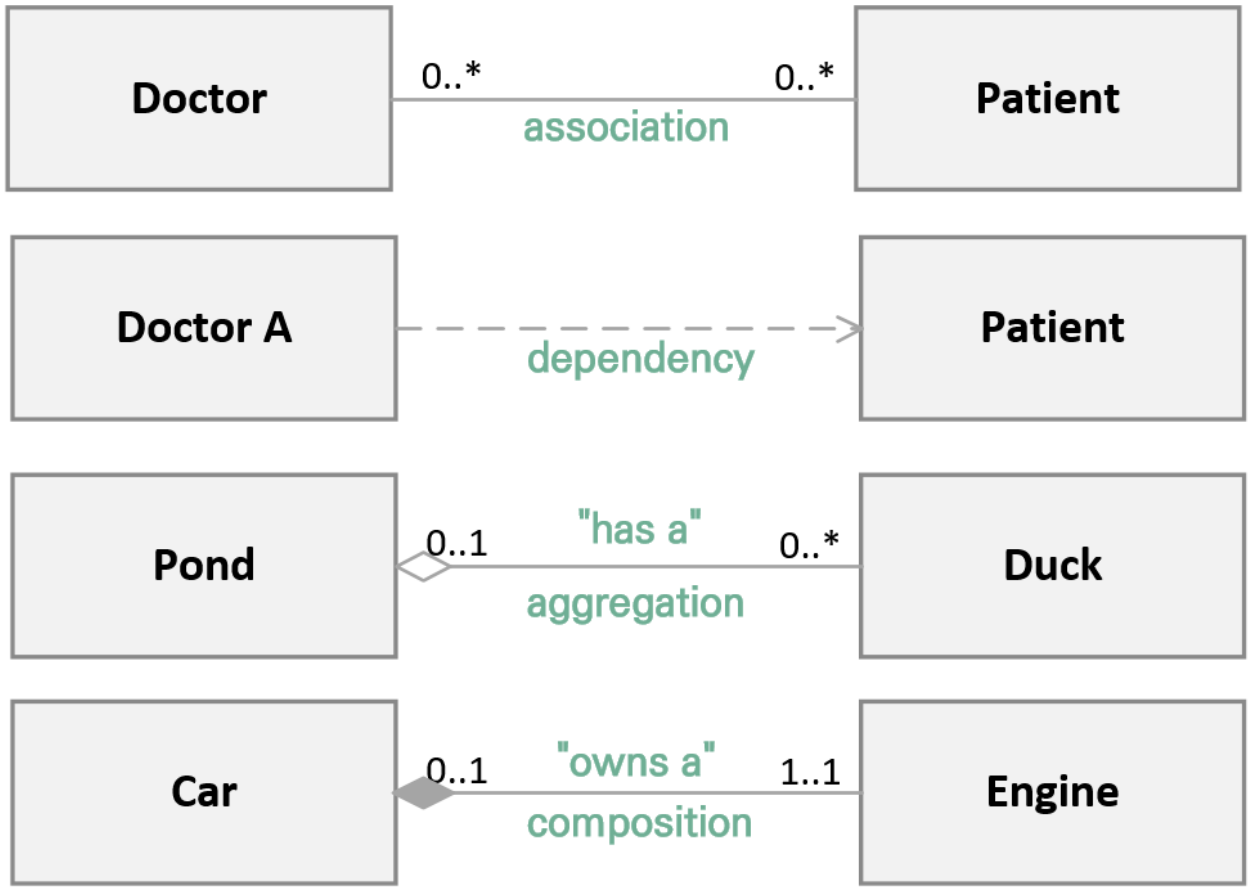
05-6 클래스와 관계

클래스와 관계

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

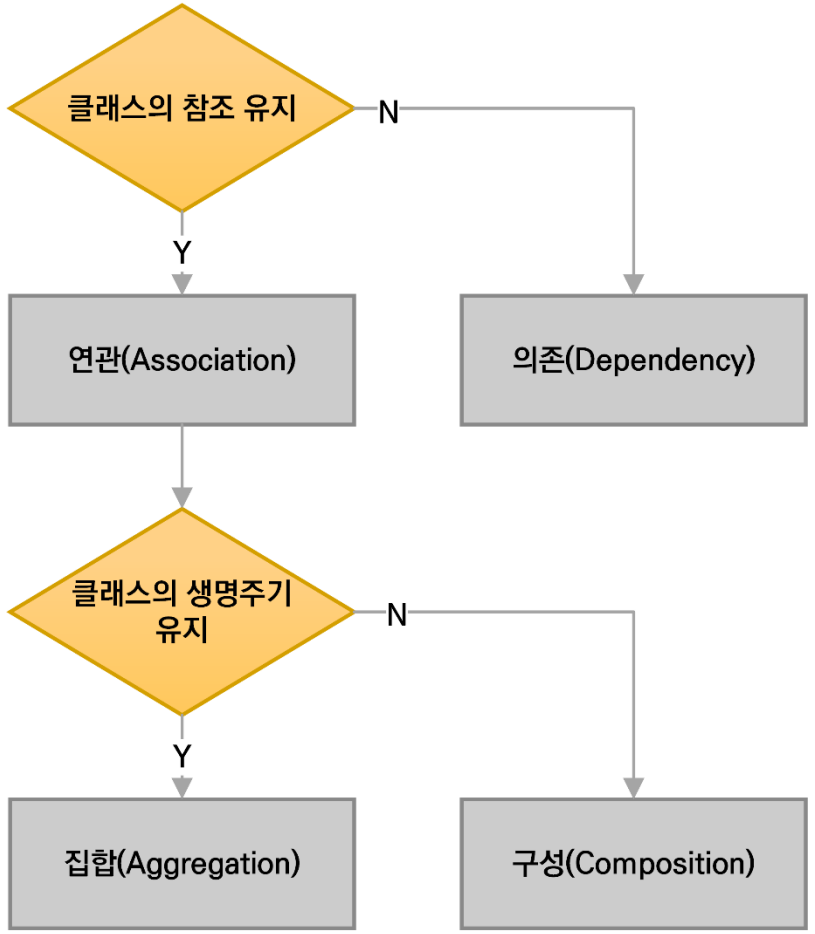
클래스 혹은 객체 간의 관계

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



클래스 혹은 객체 간의 관계

- 관계의 판별 방법



코딩해 보세요! 연관 관계 나타내기 - AssociationTest.kt

```
package chap05.section6.association

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)
}
```

코딩해 보세요! 의존 관계 나타내기 - DependencyTest.kt

```
package chap05.section6.dependency

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()
}
```

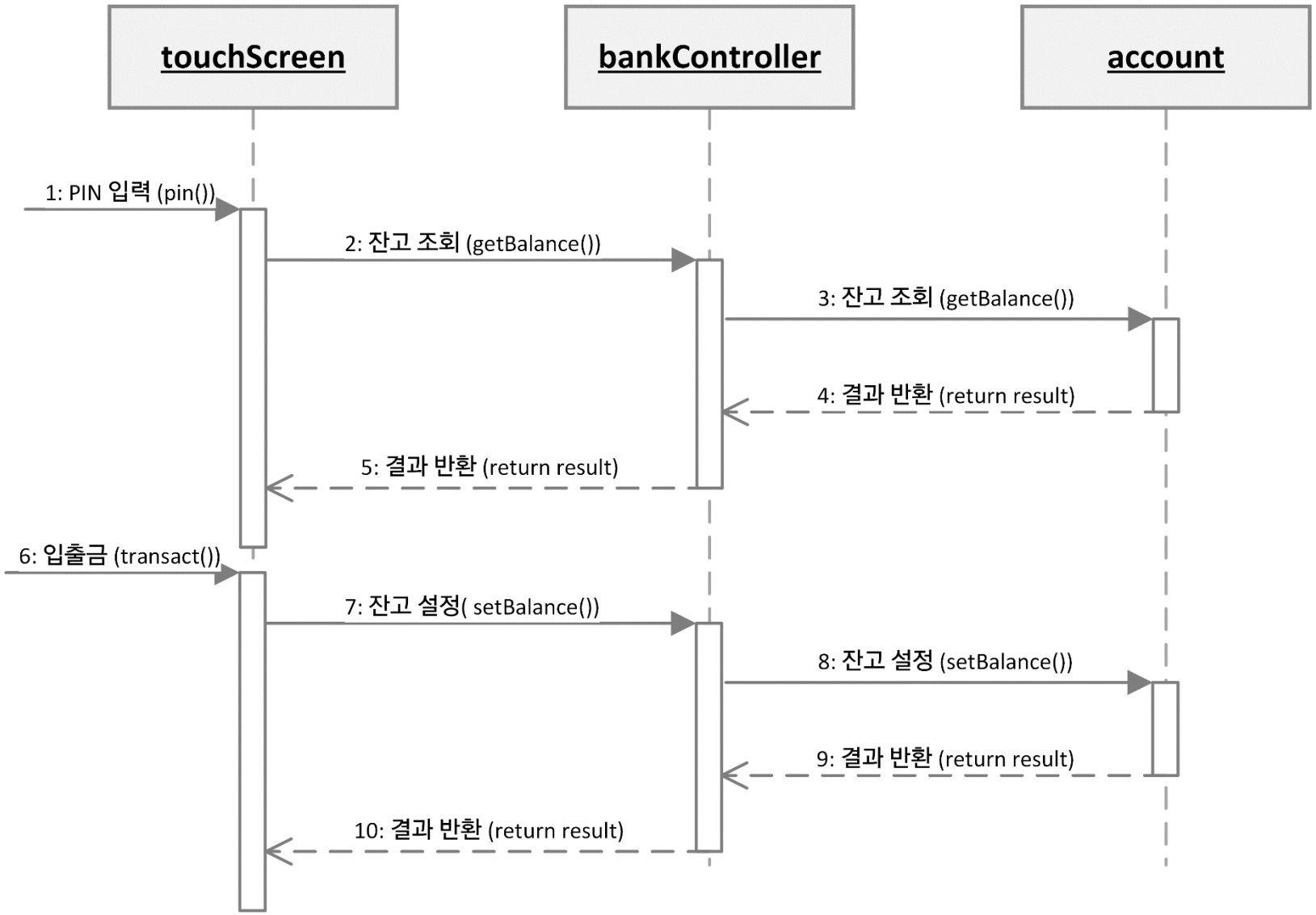
코딩해 보세요! 연못의 오리들로 집합 관계 - AggregationTest.kt

```kotlin
package chap05.section6

//  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)
    }
}
```

코딩해 보세요! 구성 관계 나타내기 - CompositionTest.kt

```
package chap05.section6.composition

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()
}
```

객체 간의 메시지 전달


touchScreen
bankController
account
1: PIN 입력 (pin())
2: 잔고 조회 (getBalance())
3: 잔고 조회 (getBalance())
4: 결과 반환 (return result)
5: 결과 반환 (return result)
6: 입출금 (transact())
7: 잔고 설정(setBalance())
8: 잔고 설정 (setBalance())
9: 결과 반환 (return result)
10: 결과 반환 (return result)