5-2부 (객체지향 프로그래밍)

1. 클래스 위임

- 합성과 상속은 모두 새 클래스 안에 하위 객체를 심는다.
  합성에는 하위 객체가 명시적으로 존재
  상속에는 하위 객체가 암묵적으로 존재
- 클래스가 기존의 구현을 재사용하면서 동시에 인터페이스를 구현해야 하는 경우, **상속**과 **클래스 위임** 이라는 선택지가 있다.
- 클래스 위임은 상속과 합성의 중간 지점이다
  합성과 마찬가지로 새 클래스 안에 멤버 객체를 심고, 상속과 마찬가지로 심겨진 하위 객체의 인터페이스를 노출 시킨다. 게다가 새 클래스를 하위 객체의 타입으로 업캐스트할 수 있다. 코드를 재사용하기 위해 클래스 위임은 합성을 상속만큼 강하게 해준다.
- 언어의 지원이 없을 때 클래스 위임

```
interface Controls {
    fun up(velocity: Int): String
    fun down(velocity: Int): String
    fun left(velocity: Int): String
    fun right(velocity: Int): String
    fun forward(velocity: Int): String
    fun back(velocity: Int): String
    fun turboBoost(): String
}

// 제어 장치의 기능을 확장하거나 명령을 일부 조정하고 싶지만
// open 이 아니므로 상속할 수 없다.
class SpaceShipControls: Controls {
    override fun up(velocity: Int) = "up $velocity"
    override fun down(velocity: Int) = "down $velocity"
    override fun left(velocity: Int) = "left $velocity"
    override fun right(velocity: Int) = "right $velocity"
    override fun forward(velocity: Int) = "forward $velocity"
    override fun back(velocity: Int) = "back $velocity"
    override fun turboBoost() = "turboBoost"
}

class ExplicitControls: Controls {
    private val controls = SpaceShipControls()
    // 수동으로 위임 구현
    override fun up(velocity: Int) = controls.up(velocity)
    override fun down(velocity: Int) = controls.down(velocity)
```

```
    override fun left(velocity: Int) = controls.left(velocity)
    override fun right(velocity: Int) = controls.right(velocity)
    override fun forward(velocity: Int) = controls.forward(velocity)
    override fun back(velocity: Int) = controls.back(velocity)
    override fun turboBoost() = controls.turboBoost() + "... 슛!!!"
}
```

- 코틀린(언어)의 지원이 있을 때 클래스 위임

```
// 코틀린 클래스 위임 방법
interface AI
class A: AI

class B(val a: A) : AI by a
```

  - `by` 키워드를 사용한다.
    "이 코드를 클래스 B는 AI 인터페이스를 a 멤버 객체를 사용해(by) 구현" 로 읽는다.
  - 인터페이스에만 위임을 적용할 수 있다.
    당연히
    `by` 된 객체는 업캐스트 가능하다.
  - **위임자 객체 `a` 는 생성자로 지정한 프로퍼티이어야만 한다.**

    위임자 객체를 여러 개 두고, 각 `by` 를 여러 개 할 수 있다.
    (자바 인터페이스
    `implement` 와 같음)

```
class DelegatedControls(
    private val controls: SpaceShipControls = SpaceShipControls()
): Controls by controls {
    override fun turboBoost(): String =
        controls.turboBost() + "... 슛!!!"
}
```

  - 코틀린은 다중 클래스 상속을 허용하지 않지만, 클래스 위임을 통해 다중 클래스 상속을 흉내낼 수 있다.
    (with 합성)

- 합성 > 클래스 위임 > 상속 순으로 클래스 설계를 시도해라

2. 다운 캐스트

- 업캐스트 했던 객체의 구체적인 타입을 발견한다.
- 실행 시점에 일어남으로 **실행 시점 타입 식별**이라고도 한다.

```
interface Base {
    fun f()
}
```

```
class Derived1: Base {
    override fun f() {}
    fun g() = "g"
}

class Derived2: Base {
    override fun f() {}
    fun h() = "h"
}

fun main() {
    val b1: Base = Derived1() // 업캐스트
    b1.f()
    // b1.g() // 부모 클래스에 없음
    val b2: Base = Derived2() // 업캐스트
    b2.f()
    // b2.h() // 부모 클래스에 없음
}
```

  - 특정 객체를 부모 타입으로 업캐스트하면 컴파일러는 그 객체의 구체적인 타입을 알 수 없다.
    이 문제를 해결하려면 다운캐스트가 올바른지 보장하는 방법이 필요하다.

- **스마트 캐스트**

```
fun main() {
    val b1: Base = Derived1() // 업캐스트
    if (b1 is Derived1)
        b1.g() // 'is' 검사 영역 내부
    val b2: Base = Derived2() // 업캐스트
    if (b2 is Derived2)
        b2.h() // 'is' 검사 영역 내부
}
```

  - `is` 키워드를 사용해 어떤 객체가 특정 타입인지 검사한다.
  - `when` 키워드와 잘 맞는다.

```
fun what(b: Base): String =
    when (b) {
        is Derived1 -> b.g()
        is Derived2 -> b.h()
        else -> "else"
}
```

- **변경 가능한 참조**
  - 자동 다운 캐스트는 대상이 상수이어야만 제대로 동작한다.

```
class SmartCast1(val b: Base) {
    fun contact() {
```

```
        when (b) {
            is Derived1 -> b.g()
            is Derived2 -> b.h()
        }
    }
}

class SmartCast1(var b: Base) {
    fun contact() {
        when (val b = b) {
            is Derived1 -> b.g()
            is Derived2 -> b.h()
        }
    }
}
```

- `val` 을 사용해서 `b` 의 타입을 검사하는 시점과 다운캐스트한 타입으로 사용하는 시점 사이에 `b`의 값이 변하지 않도록 강제한다.
- `var` (가변 프로퍼티) 여도 그 사이 변하지 않는 걸 알 수 있지만, 동시성(다른 쓰레드에 접근 한다고 하였을 때) 문제일 때는 어떻게 될지 모름으로 `val` 로 변수 변화를 막아야 한다.

- **`as` 키워드**
  - 일반적인 타입을 구체적인 타입으로 강제 변환한다.

```
fun derived1Unsafe(b: Base) = (b as Derived1).g()

fun derived1Unsafe2(b: Base): String {
    // as 를 쓴 이후 b 는 Derived1 로 취급된다.
    b as Derived1
    b.g()
    return b.g() + b.g()
}
```

  - `as` 가 만약 실패한다면 `ClassCastException` 을 던진다.
    일반
    `as` 를 안전하지 않은 캐스트라 부른다.
  - `as?` 는 실패해도 예외를 던지지 않는 대신 `null` 을 반환한다.

```
fun derived1Safe(b: Base) = (b as? Derived1)?.g() ?: "it's base"
```

    - `null` 도 좋지는 않아서 엘비스 연산자(`?:`)를 사용한다.

- **리스트 원소의 타입 알아내기**

```
val group: List<Base> = listOf(
    Derived1(), Derived2(), Derived1()
)
```

```
fun main() {
    val derived1 = group
    // is 명시적으로 타입 변환을 하고 find 가 null 을 반환할 수도 있음으로
    // Derived1? 로 Null 이 될 수 있는 타입으로 변경해야한다.
        .find { it is Derived1 } as Derived1?
    // null 이 될 수 있는 타입으로 안전한 호출 연산자가 붙어야한다.
    derived1.g() eq "g"

    // fitler는 모든 반환 값의 원소가 Derived1 임에도 불구하고 List<Base> 해야하지만
    val derived1List1: List<Base> = group.filter { it is Derived1 }
    // filterIsInstance 는 대상 타입인 Derived1 만 줌으로 List<Derived1> 를 반환
    val derived1List2: List<Derived1> = group.filterIsInstance<Derived1>()
    derived1List1 eq derived1List2
}
```

3. 봉인된 클래스

- 클래스 계층을 제한하려면 부모 클래스를 `sealed` 로 선언하라

```
open class Transport

data class Train(
    val line:String
): Transport()

data class Bus(
    val number: String,
): Transport()

fun travel(transport: Transport) =
    when (transport) {
        is Train ->
            "Train ${transport.line}"
        is Bus ->
            "Bus ${transport.number}"
        // 다른 클래스에서 Transport 를 확장해서 하당 함수를 사용할 수도 있음으로 else 를
        else -> "transport is in limbo!"
}

sealed class Transport2

data class Train2(
    val line:String
): Transport2()

data class Bus2(
    val number: String,
```

```
): Transport2()

fun travel2(transport: Transport2) =
    when (transport) {
        is Train ->
            "Train ${transport.line}"
        is Bus ->
            "Bus ${transport.number}"
        // sealed 키워드를 사용하면 상속을 제한한 클래스로서 다른 클래스에서 상속할 수 없음
        // else 가 필요 없다.
}
```

- `sealed` 클래스를 직접 상속한 하위 클래스는 반드시 상위 클래스와 같은 패키지와 모듈 안에 있어야 한다.
- `sealed` 클래스는 기본적으로 하위 클래스가 모두 같은 파일 안에 정의되어야 한다는 제약이 가해진 `abstract` 클래스이다.
- `sealed` 클래스의 하위 클래스를 상속 받아 사용하는 클래스는 직접 `sealed` 를 상속하지 않아 상관없다.
- `sealed` 는 `interface` 에도 적용 가능하다.
- 하위 클래스 열거

  ```
  sealed class Top
  class Middle1: Top()
  open class Middle2: Top()
  class Bottom2: Middle2()

  fun main() {
      Top::class::sealedSubclasses.map { it.simpleName }
      eq "[Middle1, Middle2]"
  }
  ```

  - `sealedSubclasses` 는 봉인된 클래스의 모든 하위 클래스를 돌려준다.

    `sealedSubclasses` 는 리플렉션을 사용함으로 클래스를 동적으로 찾아낸다.
  - `simpleName` 는 클래스의 이름을 주는 프로퍼티이다.

4. 타입 검사

```
interface Shape {
    fun draw(): String
}

class Circle: Shape {
    override fun draw() = "Circle: Draw"
}

class Square: Shape {
```

```
    override fun draw() = "Square: Draw"
    fun rotate() = "Square: Rotate"
}
// 만약에 Shape 를 상속 받은 다른 클래스가 있다면 해당 함수는 추가적으로 작성을 해야한다.
fun turn(s: Shape) = when (s) {
    is Square -> s.rotate()
    else -> ""
}

fun main() {
    val shapes = listOf(Circle(), Square())
    shapes.map{ it.draw() } eq "[Circle: Draw, Square: Draw]"
    shapes.map{ turn(it) } eq "[, Square: Rotate]"
}
```

- `turn` 같은 함수를 **타입 검사 코딩**이라고 부르고, 안티패턴으로 간주된다.
  이 뜻은 시스템의 모든 타입을 검사한다는 뜻이다.
- 이런 안티패턴 해결책으로 `sealed` 키워드가 나온 것이다.
- 외부 함수에서 타입 검사하기
  (
  `sealed` 클래스를 상속한 클래스가 `sealed` 클래스이고 또 상속하는 클래스가 있는 경우 when 처리)

  ```
  fun TopSealClass.test() =
      when(this) {
          is MiddleSealClass1 -> "Middle 1"
          is MiddleSealClass2 -> "Middle 2"
      }
  }

  fun TopSealClass.test2() =
      when(this) {
          is MiddleSealClass1 -> when(this) {
              is BottomSealClass11 -> "Bottom 11"
              is BottomSealClass12 -> "Bottom 12"
          }
          is MiddleSealClass2 -> when(this) {
              is BottomSealClass21 -> "Bottom 21"
              is BottomSealClass22 -> "Bottom 22"
          }
      }
  }
  ```

  - `test2` (튼튼한 타입 검사 해법) 모든 하위 타입을 검사하도록 보장해야 한다.
  - 이런식으로 외부 함수에서 처리할 수도 있고, `test` 함수를 멤버 함수으로 두고 다형성으로 할 수도 있다.
    (코틀린은 선택지를 늘려줬다. (한 곳에서 모두 처리할 것인가? (외부함수), 각각의 곳에서 처리할 것인가? (멤버함수))

5. 내포된 클래스

```
class Airport(private val code: String) {
    open class Plane {
        // 내부 클래스는 외부 클래스의 private 프로퍼티에 접근할 수 있다.
        fun contact(airport: Airport) = "Contacting ${airport.code}"
    }
    private class PrivatePlane: Plane()
    fun privatePlane(): Plane() = PrivatePlane()
}

fun main() {
    val denver = Airport("DEN")
    // 내부 클래스를 객체로 만들 때 생성자 호출을 한정시켜야 한다.
    // import 패키지명.Airport.Plane 을 했다면 Plane 로 바로 호출할 수 있다.
    val plane = Airport.Plane()
    plane.contact(denver) eq "Contracting DEN"
    // 아래와 같이 할 수 없다.
    // val privatePlane = Airport.PrivatePlane()
    val frankfurt = Airport("FRA")
    plane = frankfurt.privatePlane()
    // 아래와 같이 할 수 없다.
    // val p = plane as PrivatePlane
    plane.contact(frnkfurt) eq "Contacting FRA"
}
```

- `PrivatePlane` 처럼 내포된 클래스가 `private` 일 수도 있다. `private` 로 정의했다는 말은 `PrivatePlane`을 `Airport` 밖에서는 절대로 볼 수 없다는 뜻이다. 따라서 `Airport` 밖에서 `PrivatePlane` 생성자를 호출할 수도 없다.
- `privatePlane()` 함수 처럼 멤버 함수가 `PrivatePlane` 을 반환한다면 결과를 `public` 타입으로 업캐스트해서 반환해야 하며(업캐스트가 싫으면 `private class`를 `public class` 로 바꿔야 한다.), **`Airport` 밖에서 이렇게 받은 `public` 타입의 객체 참조를 다시 `private` 타입으로 다운캐스트 할 수 없다.**
- **지역 클래스**
  - 함수 안에서 내포된 클래스를 지역 클래스라고 한다.

    ```
    fun localClasses() {
        open class Amphibian
        class Frog: Amphibian()
        val amphibian: Amphibian = Frog()
    }
    ```

  - 지역 클래스는 지원하지만 지역 인터페이스는 허용되지 않는다.
  - 지역 클래스로 open 클래스는 거의 정의하지 말아야 한다.

- `Amphibian` 과 `Frog` 는 `localClasses()` 밖에서 볼 수 없음으로, 이런 클래스를 함수가 반환할 수도 없다. 지역 클래스의 객체를 반환하려면 그 객체를 함수 밖에서 정의한 인터페이스나 클래스로 업캐스트 해야 한다.

```
interface Amphibian

fun createAmphibian(): Amphibian {
    class Flog: Amphibian
    return Frog()
}

fun main() {
    val amphibian = createAmphibian()
    // amphibian as Frog
}
```

  - `createAmphibian()` 밖에서는 여전히 `Frog` 를 볼 수 없음으로, `main()` 안에서 `amphibian` 을 `Frog` 로 업캐스트할 수는 없다. 다운캐스트를 시도하면 참조할 수 없음 이라는 컴파일 오류가 발생한다.

- **인터페이스에 포함된 클래스**

```
interface Item {
    val type: Type
    data class Type(val type: String)
}

class Bolt(type: String): Item {
    override val type = Item.Type(type)
}

fun main() {
    val items = listOf(
        Bolt("Slotted"), Bolt("Hex")
    )
    item.map(Item::type) eq "[Type(type=Slotted), Type(type=Hex)]"
}
```

  - Bolt 안에서는 val type 을 반드시 오버라이드 하고 Item.Type 이라는 한정시킨 클래스 이름을 써서 값을 대입해야 한다.

- **내포된 이넘(Enum)**
  - 인터페이스나 클래스에 이넘을 내포시킬 수 있다.
  - 이넘을 함수에 내포시킬수 없고, 이넘이 다른 클래스를 상속할 수도 없다.

```
class Ticket(
    val name: String,
    val seat: Seat = Coach
) {
    enum class Seat {
```

```
        Coach,
        Prmium,
        Business,
        First
    }
    ... 생략
}

interface Game {
    enum class State { Playing, Finished }
    enum class Mark { Blank, X, O }
}

class FillIt(val side: Int = 3, randomSeed: Int = 0): Game {
    private var state = Playing
    private val grid = MutableList(side * side) { Blank }
    private var play = X
    ... 생략
}
```

6. 객체

- 개인 의견) java 에서 class 안에서 static 변수와 함수만 두는 utility class 느낌이다.
- `object` 키워드를 사용해서 싱글턴 패턴으로 인스턴스를 하나만 만들어준다.
  (인스턴트 생성이 불가능하다, 정의하는 동시에 객체를 생성한다.)

```
object JustOne {
    val n = 2
    fun f() = n * 10
    fun g() = this.n * 20
}

fun main() {
    // val x = JustOne() // 오류
    JustOne.n eq 2
    JustOne.f() eq 20
    JustOne.g() eq 40
}
```

- `object`는 다른 클래스나 인터페이스를 상속할 수 있다.

```
open class Paint(val color: String) {
    open fun apply() = "Applying $color"
}

object Acylic: Paint("Blue") {
```

```
    override fun apply() = "Acrylic, ${super.apply()}"
}

interface PaintPreparation {
    fun prepare(): String
}

object Prepare: PaintPreparation {
    override fun prepare() = "Scrape"
}
```

- `object` 함수 안에 넣을 수는 없지만, 다른 `object` 나 `class` 안에 `object` 를 내포할 수 있다.

```
object Outer {
    object Nested {
        val a = "Outer.Nested.a"
    }
}

class HasObject {
    object Nested {
        val a = "HasObject.Nested.a"
    }
}
```

7. 내부 클래스

- 내포된 클래스와 비슷하지만, 내부 클래스의 객체는 자신을 둘러싼 클래스의 인스턴스에 대한 참조를 유지한다.

```
class Hotel(private val reception: String) {
    open inner class Room(val id: Int = 0) {
        // Rome 을 둘러싼 클래스의 'reception' 을 사용한다.
        fun callReception() = "Room $id Calling $reception"
        }
    }
    private inner class Closet: Room()
    fun closet(): Room = Closet()
}

fun main() {
    val nycHotel = Hotel("311")
    // 내부 클래스의 인스턴스를 생성하려면
    // 그 내부 클래스를 둘러싼 클래스의 인스턴스가 필요하다.
    val room = nycHotel.Room(319)
    room.callReception() eq "Room 319 Calling 311"
    val sfHotel = Hotel("0")
```

```
    val closet = sfHotel.closet()
    closet.callReception() eq "Room 0 Calling 0"
}
```

- `inner` 클래스는 자신을 둘러싼 클래스의 객체에 대한 암시적 링크를 갖고 있다.
- `Closet`은 내부 클래스 `Room`을 상속하기 때문에 `Closet`도 `Inner` 클래스여야 한다.
  (내포된 클래스는
  `inner` 클래스를 상속할 수 없다.)
- `Closet` 은 `private` 이므로 `Hotel`의 영역 안에서만 `Closet`을 볼 수 있다.
- `inner` 클래스의 객체는 자신과 연관된 외부 객체에 대한 참조를 유지한다. 따라서 `inner` 클래스의 객체를 생성하려면 외부 객체를 제공해야 한다.
- 코틀린은 `inner` `data` `class` 를 허용하지 않는다.

- **한정된 this**
  - `inner` 클래스에서 `this`는 `inner` 객체나 외부 객체를 가리킬 수 있다.
    이때, 한정된
    `this`는 `this` 뒤에 `@`를 붙이고 대상 클래스 이름을 덧붙인 것이다.

```
class Fruit { // @Fruit 라벨이 암시적으로 붙는다.
  fun changeColor(color: String) =
    "Fruit $color"
  fun absorbWater(amount: Int) {}
  inner class Seed { // @Seed 라벨이 암시적으로 붙는다.
    fun changeColor(color: String) =
      "Seed $color"
    fun germinate() {}
    fun whichThis() {
      // 디폴트로 (가장 안쪽의) 현재 클래스를 바라본다. (Seed)
      this.name eq "Seed"
      // 명확하게 하기 위해 default this를 한정 시킬 수 있다.
      this@Seed.name  eq "Seed"
      // name 이 Fruit 와 Seed에 다 있기 때문에 Fruit 를 명시해 접근해야 한다.
      this@Fruit.name  eq "Fruit"
      // 현재의 클래스의 내부 클래스에 @레이블을 써서 접근할 수는 없다.
      // this@DNA.name
    }
    inner class DNA { // @DNA 라벨이 암시적으로 붙는다.
      fun changeColor(color: String) {
        // changeColor(color) // 재귀
        this@Seed.changeColor(color)
        this@Fruit.changeColor(color)
      }
      fun plant() {
        // 한정을 시키지 않고 외부 클래스의 함수를 호출할 수 있다.
        germinate()
        absorbWater(10)
      }
```

```
      // 확장 함수
      fun Int.grow() { // @grow 라벨이 암시적으로 붙는다.
        // 디폴트는 Int.grow()로, 'Int'를 수신 객체로 받는다.
        this.name eq "Int"
        // @grow 한정은 없어도 된다.
        this@grow.name  eq "Int"
        // 여기서도 여전히 모든 프로퍼티에 접근할 수 있다.
        this@DNA.name  eq "DNA"
        this@Seed.name  eq "Seed"
        this@Fruit.name  eq "Fruit"
      }
      // 외부 클래스에 대한 확장 함수들
      fun Seed.plant() {}
      fun Fruit.plant() {}
      fun whichThis() {
        // 디폴트는 현재 클래스이다.
        this.name eq "DNA"
        // @DNA 한정은 없어도 된다.
        this@DNA.name  eq "DNA"
        // 다른 클래스 한정은 꼭 명시해야 한다.
        this@Seed.name  eq "Seed"
        this@Fruit.name  eq "Fruit"
      }
    }
  }
}

// 확장 함수
fun Fruit.grow(amount: Int) {
  absorbWater(amount)
  // Fruit 의 'changeColor()'를 호출한다.
  changeColor("Red") eq "Fruit Red"
}

// 내부 클래스를 확장한 함수
fun Fruit.Seed.grow(n: Int) {
  germinate()
  // Seed의 changeColor를 호출한다.
  changeColor("Green") eq "Seed Green"
}

// 내부 클래스의 확장 함
fun Fruit.Seed.DNA.grow(n: Int) = n.grow()

fun main() {
  val fruit = Fruit()
  fruit.grow(4)
  val seed = fruit.Seed()
```

```
  seed.grow(9)
  seed.whichThis()
  val dna = seed.DNA()
  dna.plant()
  dna.grow(5)
  dna.whichThis()
  dna.changeColor("Purple")
}
```

- `Fruit` 클래스 안에는 `inner` 클래스 `Seed` 가 들어 있다. `Seed` 클래스 안에는 다시 `inner` 클래스로 `DNA` 가 들어 있다.

- **내부 클래스 상속**
  - 내부 클래스는 다른 외부 클래스에 있는 내부 클래스를 상속할 수 있다.

```
open class Egg {
  private var yolk = Yolk()
  open inner class Yolk {
    init { trace("Egg.Yolk()") }
    open fun f() { trace("Egg.Yolk.f()") }
  }
  init { trace("New Egg()") }
  fun insertYolk(y: Yolk) { yolk = y }
  fun g() { yolk.f() }
}

class BigEgg : Egg() {
    // Egg.Yolk를 자신의 부모 클래스로 정의하고 f() 를 오버라이드 한다.
  inner class Yolk : Egg.Yolk() {
    init { trace("BigEgg.Yolk()") }
    override fun f() {
      trace("BigEgg.Yolk.f()")
    }
  }
  init { insertYolk(Yolk()) }
}

fun main() {
  BigEgg().g()
  trace eq """
    Egg.Yolk()
    New Egg()
    Egg.Yolk()
    BigEgg.Yolk()
    BigEgg.Yolk.f()
  """
}
```

- **지역 내부 클래스와 익명 내부 클래스**
  - 멤버 함수 안에 정의된 클래스를 **지역 내부 클래스** 라고 한다.
    이런 클래스를 객체 식이나 SAM 변환을 사용해 익명으로 생성할 수도 있다.

```
fun interface Pet {
  fun speak(): String
}

object CreatePet {
  fun home() = " home!"
  fun dog(): Pet {
    val say = "Bark"
    // 지역 내부 클래스
    class Dog : Pet {
      override fun speak() = say + home()
    }
    return Dog()
  }
  fun cat(): Pet {
    val emit = "Meow"
    // 익명 내부 클래스
    return object: Pet {
      override fun speak() = emit + home()
    }
  }
  fun hamster(): Pet {
    val squeak = "Squeak"
    // SAM 변환
    return Pet { squeak + home() }
  }
}

fun main() {
  CreatePet.dog().speak() eq "Bark home!"
  CreatePet.cat().speak() eq "Meow home!"
  CreatePet.hamster().speak() eq "Squeak home!"
}
```

  - 내부 클래스는 외부 클래스 객체에 대한 참조를 저장하므로 지역 내부 클래스도 자신을 둘러싼 클래스에 속한 객체의 모든 멤버에 접근할 수 있다.
- 사실 코틀린에서는...
  - 코틀린에서는 한 파일 안에 여러 최상위 클래스나 함수를 정의할 수 있다. 이로 인해 지역 클래스를 사용할 필요가 거의 없다. 따라서 지역 클래스는 아주 기본적이고 단순한 클래스만 사용해야 한다. 예를 들어 함수 내부에서 간단한 data 클래스를 정의해 쓰는 것은 합리적이고, 지역 클래스가 복잡해지면 이 클래스를 함수에서 꺼내 일반 클래스로 격상시켜야 한다.

> **내포된 클래스(Nested Class)**: 클래스 내에 또 다른 클래스를 정의할 수 있으며, 기본적으로 내포된 클래스는 바깥 클래스의 인스턴스에 접근할 수 없어요.
>
> **내부 클래스(Inner Class)**: 내부 클래스는 `inner` 키워드를 사용하여 정의되며, 이를 통해 바깥 클래스의 인스턴스에 접근할 수 있게 됩니다. 즉, 내부 클래스는 바깥 클래스의 인스턴스와 연결되어 있어요.
>
> **내포된 클래스**는 형태만 내부에 존재할 뿐 실질적으로는 내용을 서로 공유할 수 없는 별개의 클래스 이지만, **내부 클래스**는 외부 클래스 객체 안에서 사용되는 클래스 이므로 외부 클래스 객체이 속성이나 함수를 사용 할 수 있다.

8. 동반 객체 (companion object)

- 일반 클래스의 원소는 동반 객체의 원소에 접근할 수 있지만,
  동반 객체의 원소는 일반 클래스의 원소에 접근할 수 없다.

```
class WithCompanion {
    companion object {
        val i = 3
        fun f() = i * 3
    }
    // 클래스 멤버는 동반 객체의 원소에 접근할 수 있다.
    fun g() = i + f()
}

class WithNamed {
    // 동반 객체는 클래스당 하나만 허용된다.
    // 동반 객체에 이름 부여가 가능하다. 붙이지 않으면 Companion 이라는 이름을 default 로
    companion object Named {
        fun s() = "from Named"
    }
}

fun WithCompanion.Companion.h() = f() * i

fun main() {
    val wc = WithCompanion()
    wc.g() eq 12
    // 동반 객체의 멤버를 클래스 이름을 사용해 참조할 수 있다.
    WithCompanion.i eq 3
    WithCompanion.f() eq 9
    WithCompanion.h() eq 27
}
```

```
class WithObjectProperty {
    companion object {
        private var n: Int = 0 // 단 하나만 생성됨
    }
    fun increment() = ++n
}

fun main() {
    val a = WithObjectProperty();
    val b = WithObjectProperty();
    a.increment() eq 1
    b.increment() eq 2
    a.increment() eq 3
}
```

- 동반 객체 안에서 프로퍼티를 생성하면 해당 필드는 메모리상에 단 하나만 존재하게 되고, 동반 객체와 연관된 클래스의 인스턴스가 이 필드를 공유한다.

```
interface ZI {
  fun f(): String
  fun g(): String
}

open class ZIOpen : ZI {
  override fun f() = "ZIOpen.f()"
  override fun g() = "ZIOpen.g()"
}

class ZICompanion {
  companion object: ZIOpen()
  fun u() = trace("${f()} ${g()}")
}

class ZICompanionInheritance {
  companion object: ZIOpen() {
    override fun g() =
      "ZICompanionInheritance.g()"
    fun h() = "ZICompanionInheritance.h()"
  }
  fun u() = trace("${f()} ${g()} ${h()}")
}

class ZIClass {
  companion object: ZI {
    override fun f() = "ZIClass.f()"
    override fun g() = "ZIClass.g()"
  }
```

```
  fun u() = trace("${f()} ${g()}")
}

fun main() {
  ZIClass.f()
  ZIClass.g()
  ZIClass().u()
  ZICompanion.f()
  ZICompanion.g()
  ZICompanion().u()
  ZICompanionInheritance.f()
  ZICompanionInheritance.g()
  ZICompanionInheritance().u()
  trace eq """
    ZIClass.f() ZIClass.g()
    ZIOpen.f() ZIOpen.g()
    ZIOpen.f()
    ZICompanionInheritance.g()
    ZICompanionInheritance.h()
  """
}
```

- 동반 객체가 다른 곳에 정의한 클래스의 인스턴스 일 수 있다.
- `ZICompanion` 은 `ZIOpen` 객체를 동반 객체로 사용하고,

  `ZICompnionInheritance`는 `ZIOpen`을 확장하고 `Override` 하면서 `ZiOpen` 객체를 만들고,

  `ZIClass`는 동반 객체를 만들면서 인터페이스를 구현한다

```
class ZIClosed : ZI {
  override fun f() = "ZIClosed.f()"
  override fun g() = "ZIClosed.g()"
}

class ZIDelegation {
  companion object: ZI by ZIClosed()
  fun u() = trace("${f()} ${g()}")
}

class ZIDelegationInheritance {
  companion object: ZI by ZIClosed() {
    override fun g() =
      "ZIDelegationInheritance.g()"
    fun h() =
      "ZIDelegationInheritance.h()"
  }
  fun u() = trace("${f()} ${g()} ${h()}")
}
```

```
fun main() {
  ZIDelegation.f()
  ZIDelegation.g()
  ZIDelegation().u()
  ZIDelegationInheritance.f()
  ZIDelegationInheritance.g()
  ZIDelegationInheritance().u()
  trace eq """
    ZIClosed.f() ZIClosed.g()
    ZIClosed.f()
    ZIDelegationInheritance.g()
    ZIDelegationInheritance.h()
  """
}
```

- 확장 가능한 클래스가 아니라면 (`open` 이 아니면), 클래스 위임을 사용해 동반 객체가 해당 클래스를 활용할 수 있다.
- `ZIDelegationInheritance` 는 `open` 이 아닌 `ZIClosed` 클래스를 위임에 사용하고 이 위임을 오버라이드 하고 확장할 수 있다는 사실을 보여준다.
  위임은 인터페이스의 메서드를, 메서드 구현을 제공하는 인스턴스에 전달한다.

```
interface Extended: ZI {
  fun u(): String
}

class Extend : ZI by Companion, Extended {
  companion object: ZI {
    override fun f() = "Extend.f()"
    override fun g() = "Extend.g()"
  }
  override fun u() = "${f()} ${g()}"
}

private fun test(e: Extended): String {
  e.f()
  e.g()
  return e.u()
}

fun main() {
  test(Extend()) eq "Extend.f() Extend.g()"
}
```

- `Extend` 에서 `ZI` 에 해당하는 부분은 `Companion` 에서 구현이 되었음으로, 추가된 `u` 함수만 `override` 하면 된다.

```
class Numbered2
private constructor(private val id: Int) {
  override fun toString(): String = "#$id"
  companion object Factory {
    fun create(size: Int) =
      List(size) { Numbered2(it) }
  }
}

fun main() {
  Numbered2.create(0) eq "[]"
  Numbered2.create(5) eq
    "[#0, #1, #2, #3, #4]"
}
```

- 주된 사용법으로 **팩터리 메서드** 패턴이다.
- `Numbered2` 객체로 이뤄진 List 생성만 허용하고 개별 `Numbered2` 객체 생성은 막는다.
- 동반 객체의 생성자는 동반 객체를 둘러싼 클래스가 최초로 프로그램에 적재될 때 이뤄진다.

> **객체(Object)**: 객체 선언은 전체 클래스를 단일 싱글톤 객체로 선언하는 것을 의미해요. 코틀린에서는 `object` 키워드를 사용하여 객체를 선언할 수 있으며, 이는 클래스의 인스턴스를 단 하나만 생성하도록 보장합니다.
>
> **동반 객체(Companion Object)**: 클래스 내에 `companion object`를 사용하여 동반 객체를 정의할 수 있어요. 동반 객체는 클래스와 연관된 메서드와 속성을 포함할 수 있으며, 자바의 `static`과 유사한 기능을 제공하지만, 동반 객체는 클래스 내에 정의된 객체로서 메서드와 속성을 포함할 수 있다는 점에서 차이가 있어요