3부

1. 확장 함수

```
fun String.singleQuote() = "'$this'"
fun String.doubleQuote() = "\"$this\""

fun main() {
    "Hi".singleQuote() eq "'Hi'"
    "Hi".doubleQuote() eq "\"Hi\""
}
```

- **확장함수** - 코틀린은 기존 클래스에 멤버 함수를 추가하는 것이 가능하다.
  ex)
  `String.singleQuote(), String.doubleQuote()`
- 확장할 대상 타입은 **수신 객체 타입**이라고 한다.
  ex)
  `String`
- 다른 패키지에 사용하려면 import 를 해줘야 한다.
- 클래스 내부에서 this 를 생략했던 것 처럼 확장함수 안에서도 this 를 생략할 수 있다.

2. 이름 붙은 인자와 디폴트 인자

```
// 이름 붙은 인자.
fun color(red: Int, green: Int, blue: Int) = "$(red, $green, $blue)"

// 디폴트 인자
fun colorDefault (red: Int = 0, green: Int = 0, blue: Int = 0) = "$(red, $gre

fun main() {
    // args에 이름을 붙이지 않았다.
    color (1, 2, 3) eq "(1, 2, 3)"
    // args에 이름을 붙였다.
    color (
        red = 76,
        green = 89,
        blue = 0
    ) eq "(76, 89, 0)"
    // args 에 이름을 일부 붙였다.
    color (52, 34, blue = 0) eq "(52, 34, 0)"
    // args에 이름을 모두 붙인다면 args 위치를 변경해도 된다.
    // 주의 일단 인자 순서를 변경하면 나머지 부분에서도 이름 붙은 인자를 사용해야 한다.
```

```
    color (blue = 0, red = 99, green = 52) eq "(99, 52, 0)"

    color(139) eq "(139, 0, 0)"
    color(blue = 139) eq "(0, 0, 139)"
    color(red = 128, blue = 139) eq "(128, 0, 139)"

    val list = listOf(1,2,3)
    // 이 내용보단
    list.joinToString(".", "", "!") eq "1.2.3!"
    // 이렇게 쓰는게 좋다. args 가 뭘 의미하는지 모를 것 임으로.
    list.joinToString(separator = ".", postfix = "!") eq "1.2.3!"
}
```

- 코틀린은 인자의 이름을 붙일 수 있어 위치 인자와 이름 인자와 섞어서 사용 가능하다.
- default 값을 지원한다.
  (java 에서는 default 값을 지원하는 것 처럼 보이기 위해 오버로딩을 사용한다)

3. 오버로딩

```
class My {
    fun foo() = 0
}

// 시그니처가 중복되는 확장 함수는 호출해도 의미 없다.
fun My.foo() = 1
// 다른 파라미터 목록을 제공함으로써 멤버 함수를 확장 함수로 오버로딩 가능하다.
fun My.foo(i: Int) = i + 2

// 디폴트 인자를 흉내 내기 위해 확장 함수를 사용하면 안 된다.
// 나쁜 예시 f1
fun f1(n: Int) = n + 300
fun f1() = f(0)

// 디폴트 인자를 사용해 한 함수로 바꾸자
// 좋은 예시 f2
fun f2(n: Int = 0) = n = 300

fun main() {
    My().foo() eq 0
    My().foo(1) eq 3

    f1() eq 300
    f2() eq 300
}
```

- **함수 시그니처**는 함수 이름, 파라미터 목록, 반환 타입으로 이뤄진다.
- 오버로딩으로 default 인자를 흉내내지 말자. (java 스타일)

- 오버로딩으로 '같은 주제를 다르게 변경하다' 를 사용하자. (ex - 파라미터 타입)

4. when 식

```
class Coordinates {
    var x: Int = 0
        set(value) {
            trace("x gets $value")
            field = value
        }
    var y: Int = 0
        set(value) {
            trace("y gets $value")
            field = value
        }
    override fun toString() = "($x, $y)"
}

fun processInput(inputs: List<String>) {
    val coordinates = Coordinates()
    for (input in inputs) {
        when (input) {
            "up", "u" -> coordinates.y--
            "down", "d" -> coordinates.y++
            "left", "l" -> coordinates.x--
            "right", "r" -> {
                trace("Moving right")
                coordinates.x++
            }
            "nowhere" -> {}
            "exit" -> return
            else -> trace("bad input: $input")
        }
    }
}

fun main() {
    processInput(listOf("up", "d", "nowhere", "left", "right", "exit", "r"))
    trace """
    y gets -1
    y gets 0
    x gets -1
    Moving right
    x gets 0
"""
}
```

- when 으로 콤마를 써서 한 가지에 여러 값을 나열해도 된다.
- 여러 동작을 수행하는 경우 중괄호를 사용한다.
- 아무 것도 하지 않음은 빈 중괄호를 사용한다.
- else 반드시 들어가야 한다. (컴파일 에러남)
  모든 경우에 대해서 when 은 처리해야 함으로 맨 마지막에 설정해야한다.
- if 문과 기능이 겹치지만, 더 유연한 것은 when 이다.

4. enum(열거타입)

```
// ver1
enum class Level {
    Overflow, High, Medium, Low, Empty
}

fun main() {
    Level.Medium eq "Medium"
}

//ver2
//import
import enumerations.Size.*

enum class Size {
    Tiny, Small, Medium, Large, Huge, Gigantic
}

fun main() {
    Gigantic eq "Gigantic"
    Size.values().toList() eq listOf(Tiny, Small, Medium, Large, Huge, Gigant
    Tiny.ordinal eq 0
    Huge.ordinal eq 4
}

import enumration.Level
import enumration.Level.*

enum class Direction(val notation: String)
    North("N"), South("S"),
    East("E"), West("W")
    val opposite: Direction
        get() = when (this) {
            North -> South
            South -> North
            West -> East
            East -> West
            // 중요! 여기서는 모든 enum 값을 처리 하고 있음으로 else 를 작성하지 않아도
        }
```

```
}

fun main() {
    North.notation eq "N"
    North.opposite eq South
}
```

- enum 을 만들면 enum에 해당하는 문자열을 돌려주는 toString()이 생성된다.
- import 를 사용하면 enum 이름을 한정시키지 않아도 된다.
- values() 를 사용해 이넘의 값을 이터레이션할 수 있다.
- enum 안에서 맨 처음 정의된 상수에 0이라는 ordinal 값이 지정된다.
  그 다음부터 순서대로 1씩 증가된 ordinal 값이 각 상수에 부여된다.
- enum 은 인스턴스 개수가 미리 정해져 있고 클래스 본문 안에 이 모든 인스턴스가 나열되어 있는 특별한 종류의 클래스다. 이 점을 제외하면 enum 은 일반 클래스와 똑같이 동작한다.

5. 데이터 클래스

```
class Person(val name: String)

data class Contact(
    val name: String,
    val number: String
)

fun main() {
    Person("Cleo") neq Person("Cleo")
    val contact = Contact("Miffy", "1-234-567890")
    contact eq Contact("Miffy", "1-234-567890")
    val copyContact = contact.copy(number = "2-345-678901")
    copyContact eq Contact("Miffy", "2-345-678901")
}
```

- 특징
  - `toString` 을 override 하지 않아도 data 클래스는 객체를 더 읽기 쉽고 보기 좋은 형식으로 표현
  - `equals` 가 자동으로 생성되어, 생성자 파라미터에 열겨된 모든 프로퍼티가 같은지 검사하는 식이된다. (타당한 동등성 검사 제공)
  - `copy()` 함수를 제공한다.
    copy 파라미터 이름은 생성자 파라미터의 이름과 동일하다.
    모든 인자에는 각 프로퍼티의 현재 값이 디폴트 인자로 지정된다.
  - `hashCode()` 해시 함수를 자동으로 생성해준다.

6. 구조 분해 선언

```
data class Computation(
    val data: Int,
```

```
    val info: String
)

fun main() {
    val (num, str) = Pair(14, "hi")
    num eq 14
    str eq "hi"

    val (v1, v2) = Computation(10, "str")
    v1 eq 10
    v2 eq "str"

    // map 의 경우 구조 분해 선언을 지원한다. (key, value)
    val map = mapof(1 to "one", 2 to "two")
    for ((key, value) in map) { ... }

    // list 의 경우 withIndex() 라는 함수로 구조 분해 선언을 지원한다
    val list = listOf("o","t")
    for ((index, value) in list.withIndex()) { ... }
}
```

- `val (a, b, c) = 여러 값이 들어있는 값`
- 표준 라이브러리 `Pair, Triple` 을 사용하면 first, second, third 등 멤버 변수를 사용할 수 있다.
  이때
  `(a, b)` 식으로 하면 안에 있는 값을 a, b 에 할당 가능하다.
- data 클래스를 사용하면 자동으로 구조 분해 선언을 지원한다.
  data 클래스의 인스턴스를 구조 분해할 때는 data 클래스 생성자에 각 프로퍼티가 나열된 순서대로 값이 대입된다.
- 구조 분해 선언은 지역 `var, val` 에서만 적용 가능하며, 클래스 프로퍼티를 정의할 때는 사용 불가

7. null 이 될 수 있는 타입

```
fun main() {
    val s1 = "abc"
    // 컴파일 오류
    // val s2: String = null

    // null 이 될 수 있는 정의
    val s3: String? = null
    val s4: String? = s1

    // 컴파일 오류
    // val s5: String = s4

    // 컴파일 오류 (?. 나 !!. call 을 사용하라고 한다.)
    // s4.length
    // 아래는 가능하다.
```

```
    if (s4 != null) s4.length
}
```

- 코틀린에서는 null 이 될 수 있는 타입을 단순히 **역참조(dereference)** 할 수 없다.
- 코틀린에서는 null 이 될 수 있는 타입의 멤버를 참조하는 경우 오류를 발생한다.
  명시적으로 if 검사를 하고 나면 코틀린이 null 이 될 수 있는 객체를 참조하도록 허용한다.
  아래 8. 에서 보다 좋은 방법을 제시해준다.

8. 안전한 호출과 엘비스 연산자

```
fun main() {
    val s: String? = "abc"

    // 아래 두 row 는 같은 의미를 뜻 한다.
    if (s != null) s.length else null
    // 안전한 호출
    s?.length

    // 엘비스 연산자
    val s1: String? = "abc"
    (s1 ?: "---") eq "abc"
    val s2: String? = null
    (s1 ?: "---") eq "---"

    // 보통의 사용처
    val length = s?.length ?: 0
}
```

- 안전한 호출(`.?`) 은 수신 객체가 null 이 아닐 때만 연산을 수행한다.
- 엘비스 연산자(`?:`) 은 null 이라면 그 다음 식이 그 값이 된다.
- 보통 엘비스 연산자를 안전한 호출 다음에 사용한다.
  안전한 호출이 null 수신 객체에 대해 만들어내는 null 대신 default 값을 제공하기 때문

9. Null 아닌 단언

```
fun main() {
    var x: String? = "abc"
    x!! eq "abc"
    x = null
    capture {
        val s: String = x!!
    } eq "NullPointerException"
}
```

- **null 아님 단언,** null 이 될 수 없다고 주장하기 위해 느낌표 두 개(`!!`) 를 쓴다.
- 일반적으로 `!!` 를 그냥 쓰는 경우는 없다. 역참조를 함께 쓴다.

10. 확장 함수와 Null이 될 수 있는 타입

```
fun main() {
    val s1: String? = null
    s1.isNullOrEmpty() eq true
    s1.isNullOrBlank() eq true

    val s2 = ""
    s2.isNullOrEmpty() eq true
    s2.isNullOrBlank() eq true

    val s3: String = "\t\n"
    s3.isNullOrEmpty() eq false
    s3.isNullOrBlank() eq true
}
```

- null 이 될 수 있는 타입을 확장(함수) 할 때는 조심해야 한다.

11. 제네릭스 소개

```
data class Automobile(val brand: String)

class GenericHolder<T>(
    private val value: T
) {
    fun getValue(): T = value
}

fun main() {
    val h1 = GenericHolder(Automobile("Ford"))
    val a: Automobile = h1.getValue()
    a eq "Automobile(brand=Ford)"
    val h2 = GenericHolder(Automobile(1))
    val i: Int = h2.getValue()
    i eq 1
    val h3 = GenericHolder("Str")
    val s: String = h3.getValue()
    h3 eq "Str"
}
```

- java 와 별 다른 점은 없다.

12. 확장 프로퍼티

```
// 형식
// 확장 함수
fun ReceiverType.extensionFunction() { ... }
// 확장 프로퍼티
```

```
val ReceiverType.extensionProperty: PropType
    get() { ... }

val String.indices: IntRange
    get() = 0 until length

val <T> List<T>.firstOrNull: T?
    get() = if (isEmpty()) null else this[0]

fun main() {
    "abc".indices eq 0..2

    listOf(1, 2, 3).firstOrNull eq 1
    listOf<String>().firstOrNull eq null
}
```

- 확장 프로퍼티에는 커스텀 게터가 필요하다.
  확장 프로퍼티에 접근할 때마다 프로퍼티 값이 계산된다.

13. break 와 continue

```
// 레이블
fun main() {
    val strings = mutableListOf<String>()
    outer@ for (c in 'a'..'e') {
        for (i in 1..9) {
            if (i ==5) continue@outer
            if ("c$i" == "c3") break@outer
            strings.add("$c$i")
        }
    }
    strings eq listOf("a1", "a2", "a3", "a4", "b1", "b2", "b3", "b4", "c1", "
}
```

- java 와 다른 것은 **레이블** 이라는 게 추가되었다.
- `레이블@` 이런식으로 사용하고 continue, break 뒤에 붙이면 해당 레이블 루프로 이동한다.