Chapter02. 코틀린 기초

목차

- 함수, 변수, 클래스, 이넘, 프로퍼티를 선언하는 방법
- 코틀린 제어 구조
- 스마트 캐스트
- 예외 던지기와 예외 잡기

기본 요소: 함수와 변수

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

- 함수를 선언할 때 `fun` 키워드를 사용한다.
- 함수를 모든 코틀린 파일의 최상위 수준에 정의할 수 있으므로 클래스 안에 함수를 넣어야 할 필요는 없다.
- 최상위에 있는 `main` 함수를 애플리케이션의 진입점으로 지정할 수 있다.
    - `main`에 인자가 없어도 된다.
- 표준 자바 라이브러리 함수에 대해 더 간결한 구문의 래퍼 제공한다.
    - eg) `println`

파라미터와 반환값이 있는 함수

```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}
```

- 코틀린 함수의 기본 구조
- 코틀린의 if는 결과를 만드는 expression 이다.

> 💡 문(statement)과 식(expression)의 구분
>
> **식**은 값을 만들어 내며 다른 식의 하위 요소로 계산에 참여할 수 있다. **문**은 자신을 둘러싸고 있는 가장 안쪽 블록의 최상위 요소로 존재하며 아무런 값을 만들어내지 않는다. 자바에서는 모든 제어 구조가 문인 반면, 코틀린에서는 루프(for, while, do/while)를 제외한 대부분의 제어 구조가 식이다.

- 코틀린에서는 대입이 항상 문으로 취급된다.

식이 본문인 함수

```
fun max(a: Int, b: Int): Int = if(a > b) a else b
```

- 본문이 중괄호로 둘러싸인 함수를 블록 본문 함수(block body function)
  - 블록이 본문인 함수가 값을 반환한다면 반드시 반환 타입을 지정, return 문을 사용해 반환 값을 명시
- 등호와 식으로 이뤄진 함수를 식 본문 함수(expression body function)
  - 식 본문 함수의 경우 반환 타입을 적지 않아도 컴파일러가 결과 타입을 함수 반환 타입으로 정해준다.
  - 타입추론

변수 선언

- 변수 이름 뒤에 타입을 명시하거나 생략을 허용
- 타입을 지정하지 않으면 컴파일러가 초기화 식을 분석해서 초기화 식의 타입을 변수 타입으로 지정

- 초기화 식을 사용하지 않고 변수를 선언하려면 변수 타입을 반드시 명시
    - 초기화 식이 없다면 변수에 저장될 값에 대해 아무 정보가 없기 때문에 컴파일러가 타입을 추론할 수 없다.

읽기 전용 변수 혹은 재대입 가능 변수

- val: 읽기 전용 참조를 선언
    - 초기화시 단 한 번만 대입될 수 있다.
    - 자바의 final 키워드
    - 변수의 값을 바꿀 수 없더라도 해당 참조가 가리키는 객체의 내부 값은 변경될 수 있다.

```
val languages = numtableListOf("java")
languages.add("kotlin")
```

- var: 재대입 가능한 참조를 선언
    - 초기화 이후 다른 값 대입 가능
    - 자바의 일반 변수
    - 변수의 값을 변경할 수 있지만 변수의 타입은 고정된다.

```
var answer = 42
answer = "no answer" // ❌ Error: type mismatch 컴파일 오류 발생
```

기본적으로 val 키워드를 사용해 선언하는 방식을 사용하고 필요할 때에만 var로 변경하는 방식을 추천한다. 불변 객체를 부수 효과가 없는 함수와 조합해 사용하면 함수형 프로그래밍 스타일이 제공하는 이점을 살릴 수 있기 때문

문자열 템플릿

- 스크립트 언어와 비슷하게 변수 이름 앞에 $를 사용해서 문자열 안에 참조한는 것

```
fun main(args: Array<String>){
    val name = "Kotlin"
    println("hi ${name}")
    println("hello $name") // 입력 값이 하나일 때 괄호 생략가능
}
```

행동과 데이터 캡슐화: 클래스와 프로퍼티

자바 POJO 클래스

```
public class Person{
    private final String name;

    //생성자
    //게터
}
```

코틀린 클래스

```
class Person(val name: String)
```

- 코틀린의 기본 가시성은 public

프로퍼티

- 클래스의 기본 목적은 데이터를 캡슐화하기 위함
- 자바에서는 필드와 접근자를 묶어 프로퍼티로 정의
    - 프로퍼티 개념을 활용하는 프레임워크가 많다.
- 코틀린에서는 프로퍼티를 언어 기본 기능으로 제공하여 완전히 대신
    - 대부분의 프로퍼티에는 해당 프로퍼티의 값을 저장하기 위한 필드인 `backing field`가 있다.
    - 하지만 프로퍼티 값을 동적 계산할 수도 있다.

- 커스텀 게터

```
class Person(
    val name: String,
    var isStudent: Boolean
)
```

```
val person = Person("Bob", true)
print(person.name)
print(person.isStudent)

person.isStudent = false
```

커스텀 접근자

- 어떤 프로퍼티가 같은 객체 안의 다른 프로퍼티에서 계산되는 결과인 경우 커스텀 구현이 필요할 수 있다.

```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
        }
}
```

- isSquare 프로퍼티는 접근할 때 계산을 할 수 있는 on the go 프로퍼티
- isSquare 프로퍼티 vs isSquare() 메서드
  - 구현이나 성능 차이는 없다.
  - 가독성 차이
  - 클래스의 특성을 기술하고 싶다면 프로퍼티로, 클래스의 행동을 기술하고 싶다면 멤버 함수

코틀린 소스코드 구조: 디렉토리와 패키지

- 코틀린에서는 클래스 임포트와 함수 임포트에 차이가 없으며, 모든 선언을 import 키워드로 가져올 수 있다.
- 여러 클래스를 한 파일에 넣을 수 있고, 파일의 이름도 마음대로 정할 수 있다.
  - 디스크상의 어느 디렉터리에 소스코드 파일을 위치시키든 관계없다.
  - 따라서 원하는 대로 소스코드를 구성할 수 있다.
- 대부분의 경우 자바와 같이 패키지별로 디렉터리를 구성하는 편이 낫다.
  - 특히 자바와 코틀린을 함께 사용하는 프로젝트에서는 자바의 방식을 따르는게 중요하다.
  - 자바 클래스를 코틀린 클래스로 마이그레이션할 때 문제가 생길 수도 있다.
- 하지만 여러 클래스를 한 파일에 넣는 것을 주저해서는 안 된다.
  - 특히 각 클래스를 정의하는 소스코드 크기가 아주 작은 경우 더욱 그렇다.

선택 표현과 처리: enum, when

enum class 정의

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VILOET
}
```

- class 앞에 키워드로 enum 을 추가
- `enum` : 소프트 키워드
  - class 앞에서는 enum 클래스를 의미, class가 없다면 키워드가 아님
  - 따라서 다른 곳에서는 이름으로 사용할 수 있다.
    - class는 하드 키워드
- 일반적인 클래스와 마찬가지로 생성자와 프로퍼티를 선언

```
enum class Color(
    val r: Int,
    val g: Int,
    val b: Int
```

```
) {
    RED(255, 0, 0), ORANGE(255, 165, 0),
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
    INDIGO(75, 0, 130), VIOLET(238, 130, 238); // 마지막에 반드시 세미콜론을 ㅊ

    fun rgb() = (r * 256 + g) * 256 + b // enum 메서드 정의
```

- 메서드를 정의하는 경우 상수 목록과 메서드 정의사이에 세미콜론을 넣어야한다.

when으로 enum 클래스 다루기

- if와 마찬가지로 when도 값을 만들어내는 식이기 때문에 식이 본문인 함수에 when을 사용가능

```
fun getMnemonic(color: Color) = // 함수의 반환 값으로 when 식을 직접 사용한
    when (color) {
        Color.RED → "Richard"
        Color.ORANGE → "Of"
        Color.YELLOW → "York"
        Color.GREEN → "Gave"
        Color.BLUE → "Battle"
        Color.INDIGO → "In"
        Color.VIOLET → "Vain"
    }
```

- 분기에 break를 넣지 않아도 된다.

- when 식의 대상을 변수에 캡쳐

```
fun getWarmthFromSensor(): String {
    return when(val color = measureColor()) {
        Color.RED, Color.ORANGE, Color.YELLOW → "warm (red = ${color.r})
        Color.GREEN → "neutral (green = ${color.g})"
        Color.BLUE, Color.INDIGO, Color.VIOLET → "cold (blue = ${color.b})"
    }
}
```

- when 분기 조건에 임의 객체 사용

```
fun mix(c1: Color, c2: Color) =
        when (setOf(c1, c2)) {
            setOf(RED, YELLOW) -> ORANGE
            setOf(YELLOW, BLUE) -> GREEN
            setOf(BLUE, VIOLET) -> INDIGO
            else -> throw Exception("Dirty color")
        }
```

  - 객체 사이를 비교할 때 동등성을 사용한다.
  - 컴파일러가 모든 가능한 경로를 처리한다고 연역할 수 없다.
    - 디폴트 케이스를 제공해서 철저한 when 식으로 만들어야 한다.
  - when 식은 인자 값과 매치하는 조건 값을 찾을 때까지 각 분기를 검사

- 인자 없는 when 사용

```
fun mixOptimized(c1: Color, c2: Color) =
    when {
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE

        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN

        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO

        else -> throw Exception("Dirty color")
    }
```

  - 불필요한 Set 객체 생성을 막을 수 있다.
  - 가독성은 조금 떨어지지만 성능 향상

스마트 캐스트: 타입 검사와 타입 캐스트 조합

- (1 + 2) + 4와 같은 간단한 산술식을 계산하는 함수를 만들어보자.
  - 함수가 받을 산술식에서는 오직 두 수를 더하는 연산만 가능하다.
- 우선 식을 인코딩하는 방법을 생각해야 한다.
  - 식을 트리 구조로 저장하자. 노드는 합계(sum)나 수(num) 중 하나다.
  - Num은 항상 말단노드이지만, Sum은 자식이 둘 있는 중간노드로, 두 자식 노드는 덧셈의 두 인자이다.
- 식을 위한 Expr 인터페이스가 있고, Sum과 Num 클래스는 그 Expr 인터페이스를 구현한다.
  - Expr은 아무 메소드도 선언하지 않으며, 단지 여러 타입의 식 객체를 아우르는 공통 타입 역할만 수행한다.
  - 마커 인터페이스

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

- Sum은 Expr의 왼쪽과 오른쪽 인자에 대한 참조를 left와 right 프로퍼티로 저장한다.
  - 예제에서 left와 right는 각각 Num이나 Sum일 수 있다.
  - (1 + 2) + 4라는 식을 저장하면 `Sum( Sum( Num(1), Num(2) ), Num(4) )`라는 구조의 객체가 생긴다.
- Expr 인터페이스에는 두 가지 구현 클래스가 존재한다. 따라서 식을 평가하려면 두 가지 경우를 고려해야 한다.
  - 어떤 식이 수라면 해당하는 값을 반환한다.
  - 어떤 식이 합계라면 좌항과 우항 값을 재귀적으로로 계산한 다음에 두 값을 합한 값을 반환한다.
- 자바 스타일로 시작

```
fun eval(e: Expr): Int {
    if (e is Num) {
        val n = e as Num
        return n.value
    }
```

```
    if (e is Sum) {
        return eval(e.right) + eval(e.left)
    }
    throw IllegalArgumentException("Unknown expression")
}
```

- 코틀린 스타일

```
fun eval(e: Expr): Int =
    if (e is Num) {
        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }
```

  - 코틀린 `is` : 자바의 `instanceof`
    - 편의 기능 추가
    - 타입을 검사한 변수를 검사한 타입 변수처럼 사용 가능하다.
    - 실제로는 컴파일러가 타입을 대신 변환 (스마트 캐스트)
  - 스마트 캐스트
    - 타입을 검사한 다음에 그 값이 바뀔 수 없는 경우에만 작동
    - 예제에서는 변수가 val이거나 커스텀 접근자를 사용하지 않는 경우
  - 코틀린 `as` : 명시적인 형변환
  - 리팩토링: if를 when을 사용

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

```
fun eval(e: Expr): Int =
    when(e) {
        is Num -> { // 블록으로 사용할 수도 있다.
            println("hello $e.value")
            e.value
        }
        is Sum -> evalu(e.right) + eval(e.left)
        else -> throw IllegalArgumentException("Unknown expression")
    }
```

- 블럭이 본문인 함수에서는 명시적인 return이 필요하다.

대상 이터레이션: while과 for 루프

- while과 do-while 자바와 동일하므로 넘어가며, for는 자바의 for-each 루프에 해당하는 형태만 존재한다.

수에 대해 이터레이션: 범위와 순열

- 코틀린은 고전적인 for 루프가 없다.
  - `for(int i=0; i<10; i++)`
- 코틀린에서는 범위를 사용
- 범위 연산자 `rangeTo` 연산자가 존재한다
  - `..`
    - eg) `1..10`
  - 양끝을 포함하는 폐구간
  - 순열

```
for (i in 1..100) {
    /* 1 2 3 ... 100 */
}

// downTo, step, until 같은 확장함수를 사용하면 더 재밌어진다.
```

```
for (i in 100 downTo 1 step 2) { // 100 부터 1까지 +(2)씩
    /* 100 98 96 ... 2 */
}
```

```
public infix fun Int.downTo(to: Int): IntProgression {
    return IntProgression.fromClosedRange(this, to, -1)
}

public infix fun IntProgression.step(step: Int): IntProgression {
    checkStepIsPositive(step > 0, step)
    return IntProgression.fromClosedRange(first, last, if (this.step > 0) step else
}
```

- 끝값을 포함하지 않는 반폐구간: `for(x in 0..< size)`

맵에 대해 이터레이션

```
fun main() {
    val binaryReps = mutableMapOf<Char, String>() // 코틀린 가변 맵은 원소 이터
    for (char in 'A'..'F') {
        val binary = c.code.toString(radix = 2)
        binaryReps[char] = binary
    }

    for ((letter, binary) in binaryReps) {
        println("$letter = $binary")
    }
}
```

- (letter, binary): 구조 분해
- get, put 대신 `assignment` 사용

in으로 컬렉션이나 범위의 원소 검사

- `in` 연산자를 사용해 어떤 값이 범위에 속하는지 검사할 수 있다.

- `!in`을 사용하면 어떤 값이 범위에 속하지 않는지 검사할 수 있다.

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

println(isLetter('q')) → true
println(isNotDigit('x')) → true
```

- `c in 'a'..'z'`: a ≤ c && c ≤ z 로 변환된다.
- 범위는 문자에만 국한되지 않고, 비교가 가능한 클래스(Comparable 구현 클래스)라면 그 클래스의 인스턴스 객체를 사용해 범위를 만들 수 있다.
- 이렇게 만든 범위의 경우 그 범위 내의 모든 객체를 항상 이터레이션하지는 못한다.
  - 예를 들어 'Java'와 'Kotlin' 사이의 모든 문자열을 이터레이션할 수 있을까?
  - 그럴 수 없다. 하지만 in 연산자를 사용하면 값이 범위 안에 속하는지 결정할 수 있다.
- 컬렉션에도 in 연산을 사용할 수 있다.

코틀린에서 예외 던지고 잡아내기

- 자바나 다른 언어의 예외 처리와 비슷하다.
- 코틀린의 throw는 식이므로 다른 식에 포함될 수 있다.

```
val percentage = if(number in 0..100) number else throw IllegalArgumentE
```

  - if의 조건이 참인경우 percentage 변수는 number의 값으로 초기화
  - if의 조건이 거짓인경우 변수가 초기화되지 않는다.

try, catch, finally

```
fun readNumber(reader: BufferedReader): Int? {
    try {
```

```
        val line = reader.readLine()
        return Integer.parseInt(line)
    } catch (e: NumberFormatException) {
        return null
    } finally {
        reader.close()
    }
}

fun main() {
    val reader = BufferedReader(StringReader("239"))
    println(readNumber(reader))
}
```

- 코틀린에는 throws 절이 없다.
- 코틀린은 체크 예외와 언체크 예외를 구분하지 않는다.
    - 체크 예외를 의미 없이 예외를 다시 던지거나, 예외를 잡되 처리하지는 않고 그냥 무시하는 코드를 작성하는 경우가 잦기 때문

try를 식으로 사용

- try를 식으로 사용하면 중간 변수를 도입하느 것을 피함으로써 코드를 좀 더 간결하게 만들고, 더 쉽게 예외에 대비한 값을 대입하거나 try를 둘러싼 함수를 반환 시킬 수 있다.

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }

    println(number)
}

fun main() {
    val reader = BufferedReader(StringReader("not a number"))
```

```
    readNumber(reader)
}
```

- 아무것도 출력되지 않는다.

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        null
    }

    println(number)
}

fun main() {
    val reader = BufferedReader(StringReader("not a number"))
    readNumber(reader)
}
```

- null을 출력한다.

요약

- 함수를 정의할 때 fun 키워드를 사용한다.
  - val과 var는 각각 읽기 전용 변수와 변경 가능한 변수를 선언할 때 쓰인다.
- val 참조는 읽기 전용이지만 val 참조가 가리키는 객체의 내부 상태는 여전히 변경 가능할 수 있다.
- 문자열 템플릿을 사용하면 간결하게 문자열을 연결할 수 있다.
  - 변수 이름 앞에 `$`를 붙이거나, 식을 `${식}`처럼 둘러싸면 변수나 식의 값을 문자열 안에 넣을 수 있다.
- 코틀린에서는 클래스를 아주 간결하게 표현할 수 있다.
- 다른 언어에도 있는 if는 코틀린에서 식이며, 값을 만들어낸다.

- 코틀린 when은 자바의 switch와 비슷하지만 더 강력하다.
- 어떤 변수의 타입을 검사하고 나면 굳이 그 변수를 캐스팅하지 않아도 검사한 타입의 변수처럼 사용할 수 있다.
    - 컴파일러가 스마트 캐스트를 활용해 자동으로 타입을 바꿔준다.
- for, while, do-while 루프는 자바가 제공하는 같은 키워드의 기능과 비슷하다.
    - 하지만 코틀린의 for는 자바의 for보다 더 편리하다.
    - 특히 맵을 이터레이션하거나 이터레이션하면서 컬렉션의 원소와 인덱스를 함께 사용해야 하는 경우 코틀린의 for가 더 편리하다.
- 1..5와 같은 식은 범위를 만든다.
    - 범위와 수열은 코틀린에서 같은 문법과 추상화를 for 루프에서 사용하게 해주고, 어떤 값이 범위 안에 들어있거나 들어있지 않은지 검사하기 위해서 `in`이나 `!in`을 함께 사용할 수 있다.
- 코틀린 예외 처리는 자바와 비슷하다.
    - 하지만 코틀린에서는 함수가 던질 수 있는 예외를 선언하지 않아도 된다.