2장 키워드

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

P.60 Hello, World! 예제 출력

- 함수의 선언은 *fun* 키워드를 사용한다.
- 파라미터 이름 : 파라미터의 타입 - 이름 뒤에 타입
- 함수를 최상위 수준에 정의 가능 -꼭 클래스 안에 함수를 넣어야 할 필요가 없다. + 배열 처리를 위한 문법이 따로 존재하지 않음.
- 표준 자바 라이브러리를 간결하게 사용할 수 있게 감싼 wrapper를 제공
  System.out.println 대신 => *println* 사용
- 끝에 세미콜론(;)을 사용하지 않아도 된다.

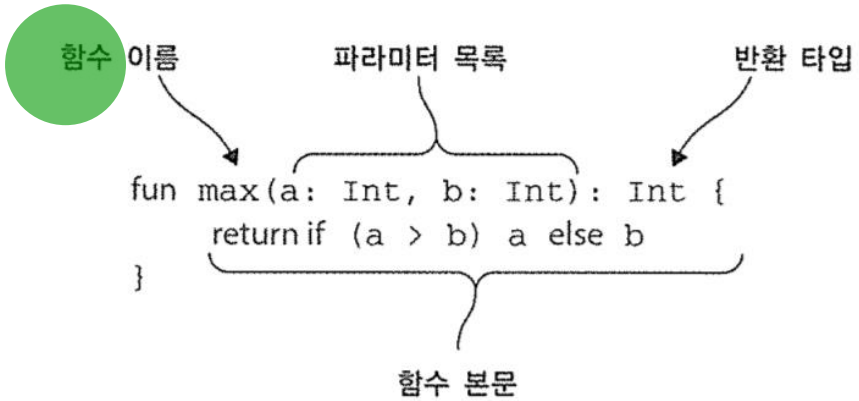


그림 2.1 코틀린 함수 정의

식이 본문인 함수의 경우 사용자가 굳이 반환 타입을 명시하지 않아도 컴파일러가 타입 추론을 해준다.

statement(문)과 expression(식)의 구분

코틀린에서의 if는 expression(식)이다. 식은 값을 만들어내며 다른 식의 하위 요소로 계산에 참여할 수 있다. 자바와 달리 대입식과 비교식을 잘못 바꿔 써서 버그가 생기는 경우가 없다.

- 블록이 본문인 함수: {}로 둘러싸인 함수
- 식이 본문인 함수: 등호와 식으로 이루어진 함수

변경 가능한 변수와 변경 불가능한 변수

변수 선언 시 사용하는 키워드는 다음과 같은 2가지가 있다.

- val(값을 뜻하는 value에서 따옴) – 변경 불가능한immutable 참조를 저장하는 변수다. val로 선언된 변수는 일단 초기화하고 나면 재대입이 불가능하다. 자바로 말하자면 final 변수에 해당한다.
- var(변수를 뜻하는 variable에서 따옴[4]) – 변경 가능한mutable 참조다. 이런 변수의 값은 바뀔 수 있다. 자바의 일반 변수에 해당한다.

초기화 식을 사용하지 않고 변수를 선언하려면 변수 타입을 반드시 명시해야 한다.
컴파일러의 타입추론
val 참조 자체는 불변일지라도 그 참조가 가리키는 객체의 내부 값은 변경될 수 있다는 사실을 기억. ()

기본적으로 모든 변수를 *val*(불변 변수)로 선언하고, 나중에 꼭 필요할 경우에만 *var* 로 변경하는 것이 좋다

➡ 변경 불가능한 참조를 side-effect가 없는 함수와 조합한다면 함수형 코드에 가까워질 수 있기 때문이다.

val 변수는 블록을 실행할 때 정확히 한 번만 초기화 되어야 한다. 이를 컴파일러가 확신할 수 있다면 조건에 따라 다른 여러 값으로 초기화할 수도 있다. (p.66)

문자열 템플릿 - 변수를 문자열 안에서 사용할 때

➡ $ 사용

* 한글을 사용할 경우 변수 이름을 {} 로 감싼다.

문자열 템플릿 자바의 (“Hello, “ + name + “!”) 과 동일

존재하지 않는 변수를 문자열 템플릿 안에서 사용하면 컴파일 오류 발생

특수문자 사용 시

- println(“$x”)

복잡한 식 사용 시 중괄호 사용

- println(“Hello, ${args[0]}!”)
- println(“Hello, ${if (args.size > 0) args[0] else “someone”}!”)

컴파일된 코드는 StringBuilder를 사용하고 문자열 상수와 변수의 값을 append로 문자열 빌더 뒤에 추가함

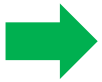
간단한 클래스 만들기 - Person (p.71 - 리스트 2.5)

자바를 코틀린으로 변환한 결과 – public 가시성 변경자가 사라짐 확인
이런 유형의 클래스를 값 객체(value object)

```
/* 자바 */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```



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

```
/* 자바 */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

```
>>> val person = Person("Bob", true)
>>> println(person.name)
Bob
>>> println(person.isMarried)
true
```

new 키워드를 사용하지 않고 생성자를 호출한다.

프로퍼티 이름을 직접 사용해도 코틀린이 자동으로 게터를 호출해준다.

프로퍼티

```kotlin
class Person {
    val name: String,   // 읽기 전용 프로퍼티 - 코틀린은 (비공개)필드와 (공개) 게터를 만들어낸다.
    var isMarried: Boolean  // 쓸 수 있는 프로퍼티 - 코틀린은 (비공개)필드, (공개) 게터, (공개) 세터를 만들어낸다.
}
```

```java
/* 자바 */
Person person = new Person("Bob", true);
person.getName();
person.isMarried();
person.setMarried(false);

/* 코틀린 */
val person = Person("Bob", true)
person.name
person.isMarried
person.isMarried = false
```

2.2.2. 커스텀 접근자

코틀린에서는 자바와 다르게 여러 클래스를 한 파일에 넣을 수 있으며, 파일의 이름도 마음대로 정할 수 있다. 코틀린에서는 자바에서 파일명과 클래스명이 다르면 오류가 발생하는 일이 생기지 않는다.

```
package ch02 // 패키지 선언

import java.util.Random // 외부 라이브러리 클래스 임포트

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

fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}
```

Package 사용 (p.76)

- 파일의 모든 선언 [클래스, 함수, 프로퍼티 등] 이 해당 패키지에 속함
- 다른 패키지에서 사용하려면 import 키워드로 사용할 선언을 임포트

코틀린- 디스크상의 어느 디렉토리에 소스코드 파일을 위치시키든 관계없다.
하지만 대부분의 경우 자바와 같이 패키지 별로 디렉토리를 구성하는 것이 좋다
java -〉 Kotlin 마이그레이션 고려

```
package geometry.shapes                          <-- 패키지 선언

import java.util.Random                          <-- 표준 자바 라이브러리 클래스를 임포트한다

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

fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}
```

```
package geometry.example

import geometry.shapes.createRandomRectangle     <-- 이름으로 함수 임포트하기

fun main(args: Array<String>) {
    println(createRandomRectangle().isSquare)    <-- "true"가 아주 드물게 출력된다.
}
```

Java

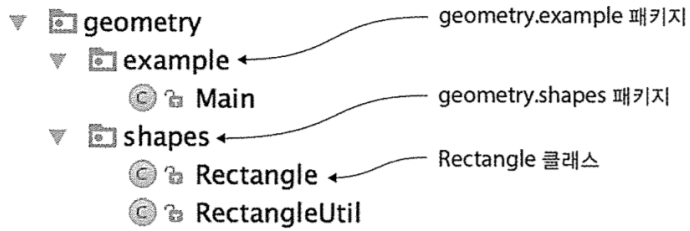


그림 2.2 자바에서는 디렉터리 구조가 패키지 구조를 그대로 따라야 한다.

Kotlin

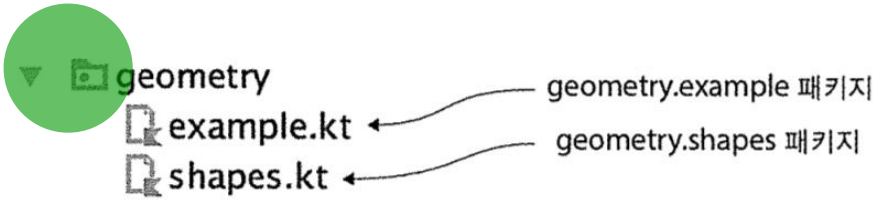


그림 2.3 패키지 구조와 디렉터리 구조가 맞아 떨어질 필요는 없다.

enum, when

enum 클래스 안에도 프로퍼티나 메소드를 정의할 수 있다.

이때, enum 클래스 안에 메소드를 정의하는 경우 반드시 enum 상수 목록과 메소드 정의 사이에 세미콜론을 넣어야 한다.

```
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);

    fun rgb() = (r * 256 + g) * 256 + b
}
>>> println(Color.BLUE.rgb())
255
```

상수의 프로퍼티를 정의한다.

각 상수를 생성할 때 그에 대한 프로퍼티 값을 지정한다.

여기 반드시 세미콜론을 사용해야 한다

enum 클래스 안에서 메소드를 정의한다.

enum

프로퍼티와 메소드가 있는 enum 클래스 선언하기

```
enum class Color(
    val r:Int, val g: Int, val b: Int   //상수의 프로퍼티 정의
) {
    RED(255, 0, 0), ORANGE(255, 165, 0), YELLOW(255, 255, 0);

    fun rgb() = (r * 256 + g) * 256 + b
}

Color.RED.rgb()
```

when을 사용해 올바른 enum 값 찾기

자바의 switch

```
fun getMnemonic(color: Color) =
    when (color) {
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
    }

    getMnemonic(Color.RED)
```

- 한 when 분기 안에 여러 값 사용하기

```
fun getMnemonic(color: Color) =
    when (color) {
        Color.RED, Color.ORANGE -> "warm"
        Color.YELLOW -> "Of"
    }

    getMnemonic(Color.RED)
```

- **when과 임의의 객체를 함께 사용**

코틀린 when의 분기 조건은 임의의 객체를 허용

```
fun mix(c1: Color, c2: Color) =
    when (setOf(c1, c2)) {  //when 식의 인자로 아무 객체나 사용 가능
        setOf(RED, YELLOW) -> ORANGE
        setOf(YELLOW, BLUE) -> GREEN
        else -> throw Exception("Dirty color")
    }

    mix(BLUE, YELLOW)
```

인자 없는 when 사용

- 2.3.3 예제의 비효율점 매번 Set 인스턴스를 생성
  가비지 객체 늘어남
- 인자 없는 when 사용 시 불필요한 객체 생성 막을 수 있음
  그러나 가독성은 떨어짐

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

        else -> throw Exception("Dirty color")
    }
    mixOptiomized(BLUE, YELLOW)
```

- **스마트 캐스트**

```
interface Expr

class Num(val value: Int) : Expr

class Sum(val left: Expr, val right: Expr) : Expr

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

    if (e is Sum) {
        return eval(e.left) + eval(e.right)
    }

    throw IllegalArgumentException("Unknown expression")
}

fun main(args: Array<String>) {
    val sum = eval(Sum(Sum(Num(1), Num(2)), Num(4)))
    println("Sum: $sum")
}
```

코틀린에서는 *is*를 사용해 변수 타입을 체크한다.
마치 자바의 instanceof와 비슷하다.
자바에서는 변수 타입을 체크하고 명시적으로 변수 타입으로 캐스팅해야 한다.
그렇지 않으면 그 변수 타입에 속한 멤버 접근을 할 수 없다. 타입 체크, 캐스트를 각각 수행해야 한다.

Expr
: value 라는 프로퍼티만 존재하는 단순한 클래스. 이 타입의 객체라면 어떤 것이나 Sum 연산의 인자 가 될 수 있다. 따라서 Num이나 다른 Sum이 인자로 올 수 있다.

2.3.5 스마트 캐스트 : 타입 검사와 타입 캐스트를 조합

- 스마트 캐스팅 - 변수를 원하는 타입으로 캐스팅하지 않아도 원하는 타입으로 사용할 수 있는 것
  컴파일러가 캐스팅 수행
- 원하는 타입으로 명시적으로 타입 캐스팅하려면 as 키워드 사용
  val n = e as Num

리팩토링: if를 when으로 변경

코틀린의 if가 값을 만들어내기 때문에 자바와 달린 3항 연산자가 따로 없다. (p.86)

```
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")
    }
```

if 분기에 식이 하나밖에 없다면 중괄호 생략 가능

if 분기에 블록을 사용하는 경우 그 블록의 마지막 식이 그 분기의 결과 값이다.

if 중첩 대신 when 사용하기

```
fun eval(e: Expr): Int =
    when(e) {
        is Num ->
            e.value
        is Sum ->
            eval(e.right) + eval(e.left)    //스마트 캐스티 사용 됨
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

실행 예제

```
fun evalWithLogging(e:Expr):Int=
    when(e){
        is Num->{
            println("num: ${e.value}")
            e.value//결과
        }
        is Sum->{
            val left= evalWithLogging(e.left)
            val right=evalWithLogging(e.right)
            println("sum:$left+$right")
            left+right//결과
        }
        else->
            throwException("UnknownExp")
    }
println (evalWithLogging (Sum(Sum(Num(1), Num(2)), Num(4))))
```

while 루프

코틀린에서 사용되는 while, do-while 루프이다. 자바와 동일하다.

```
while (조건) {
    ...
}

do {
    ...
} while (조건)
```

수에 대한 이터레이션

코틀린에는 자바의 for 루프(초깃값, 증가 값, 최종 값)을 사용하는 형태는 존재하지 않는다.
범위를 통해 반복하는 for 루프문만 존재한다.

범위(CloseRange 인터페이스): 두 값으로 이루어진 구간 -> .. 연산자
수열(Progression): 범위에 속한 값을 일정한 순서로 이터레이션

예시

1 rangeTo 10 step 2 또는 1..10 step 2: 1~10 까지 2씩 증가하며 이터레이션
100 downTo 1 step 2: 100부터 1로 줄어들며 2씩 감소하며 이터레이션
0 until 10: 0부터 10까지 이터레이션(단, 10은 미포함)

```
for(i in 1..100) {
    print(fizzBuzz(i))
}
```

```
for(i in 1..100 downTo 1 step 2) {
    print(fizzBuzz(i))
}

98 94 92...
```

2.4.3 맵에 대한 이터레이션

맵을 초기화하고 이터레이션하기

```
val binaryReps = TreeMap<Char, String>()
for (c in 'A'...'F') {
    val binary = Integer.toBinaryString(c.toInt())
    binaryReps[c] = binary
}

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

아스키 코드를 2진 표현으로 바꾼다.

C를 키로 C의 2진 표현을 맵에 넣는다.

letter = 키
binary = 2진 표현

2.4.4 in으로 컬렉션이나 범위의 원소 검사

in을 통해 값이 범위에 속하는지 검사하기
in은 contains와 동일

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

```
// Comparable 구현 클래스
println("Kotlin" in "Java".."Scala")
println("Kotlin" in setOf("Java", "Scala", "Kotlin")) // 결과: true
```

2.5 코틀린의 예외 처리

자바와 비슷하다.

자바와 마찬가지로 try 사용하기

자바나 다른 언어의 예외 처리와 비슷하다.
즉, 함수 실행 중 오류가 발생하면 예외를 던질(throw) 수 있고 함수를 호출하는 쪽에서는 그 예외를 잡아 처리(catch)할 수 있다.
예외에 대해 처리를 하지 않은 경우 함수 호출 스택을 거슬러 올라가면서 예외를 처리하는 부분이 나올 때까지 예외를 다시 던진다(rethrow).

2.5 코틀린의 예외 처리

자바와 비슷하다.

자바와 마찬가지로 try 사용하기 - 올바른 수가 아닐 경우 null 반환하는 예제

```
fun readNumber(reader: BufferedReader): Int? {  //함수가 던질 수 있는 예외를 명시할 필요가 없다.
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
    }
    catch (e: NumberFormatException) {
        return null
    }
    finally {
        reader.close()
    }
}
```

2.5 코틀린의 예외 처리

2.5.2 try를 식으로 사용

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

if와는 달리 반드시 {}로 둘러싸야 한다.
내부에 여러 문장이 있을 경우 마지막 식의 값이 전체 결과 값