3장 코틀린

구글(Google)은 2017년 5월 18일 열린 Google I/O 행사에서 안드로이드 앱 개발 공식 언어로 코틀린(kotlin)을 추가했습니다. 코틀린은 러시아 발틱(Baltic)해 인근에 있는 섬의 이름이라고 합니다. 최근 등장하는 프로그래밍 언어의 이름은 개발자의 고향(Dalvik)에서 가져오거나, 개발자가 재미있게 봤던 드라마(python) 제목에서 가져온 것처럼, 개발자의 의중이 많이 반영되고 있습니다.

구글이 개발 언어를 자바에서 코틀린으로 바꾸기로 한 결정적 이유는 오라클(Oracle)사와의 분쟁 때문입니다. 자바 언어의 소유권을 인수한 오라클은 자사의 소스 코드 11,500줄을 구글이 안드로이드 플랫폼에 불법으로 사용했다고 소송을 제기했습니다. 소프트웨어 소스 코드와 관련된 오라클과 구글의 법적 소송은 세계적 관심을 끌었습니다. 결국 2021년 4월 5일 미국 대법원은 6:2 판결로 구글의 손을 들어줬습니다. 구글이 코드를 임의로 도용한 것이 아니라 안드로이드 플랫폼의 사용자 인터페이스를 위해 재구현(re-implement)한 것이라고 판결했습니다.

오라클이 자바를 인수하면서 유료화 정책을 전개하는 데 반발해 공개 소프트웨어를 주장하는 쪽에서 Open JDK를 만들어 배포하면서, JDK(java development kit)는 오라클 JDK, Open JDK(Zulu JDK), 코틀린 등 3개로 나뉘었습니다.

최근 IT 기업 규모가 비대해지면서 소프트웨어 코드 저작권을 주장하는 사례가 빈번하게 발생하고 있습니다. 또 플랫폼 기업이 서비스를 제공하며 수집한 사용자 데이터도 유료화하면서, 데이터마저 제대로 활용하기 힘든 현상이 현실화하고 있습니다.

어쨌든 구글은 안드로이드 스튜디오 3.0부터 코틀린을 기본 지원하고 있습니다. 스튜디오 이전 버전에서도 플러그-인(plug-in)만 설치하면 코틀린을 사용할 수 있었습니다.

코틀린의 공식 웹 사이트는 https://kotlinlang.org입니다. 코틀린은 Jet Brains에서 개발하였습니다. 코틀린은 2011년 최초 공개되었고, 2016년 정식 버전(1.0)이 출시되었습니다. 자바와 코틀린은 100% 호환됩니다.

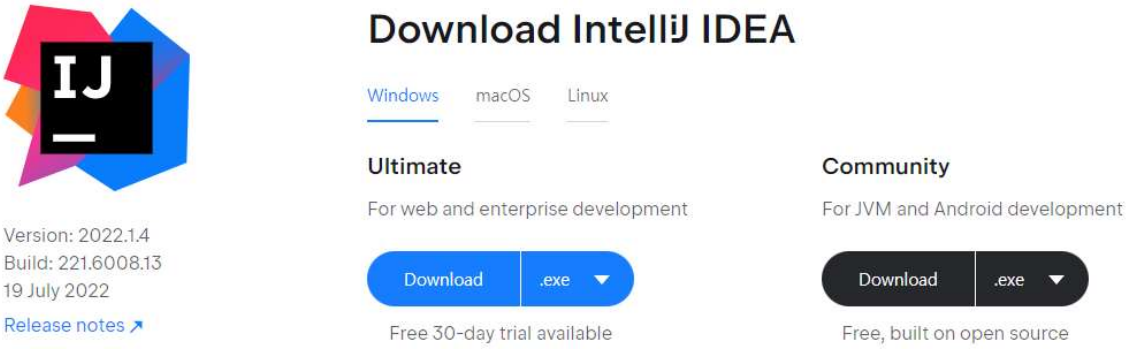
코틀린의 특징은 정적타입, 간략한 코드, 다중 패러다임(paradigm) 언어 등 3가지로 요약할 수 있습니다. 프로그램 구성 요소의 타입(type)을 실행 시간이 아닌 컴파일 시간에 알 수 있는 걸 정적타입이라고 합니다. 이 때문에, 코틀린에서는 타입 추론(type inference)이 가능합니다. 간략한 코드는 파이선(python)이나 스위프트(swift)처럼 최근 등장하는 언어들의 일반적 경향입니다. 자바로 작성한 안드로이드 앱을 코틀린으로 바꾸면 코드 길이가 최소 30% 이상 줄어듭니다. 다중 패러다임 언어란 객체지향언어 및 함수형 언어의 특징을 동시에 갖고 있다는 뜻입니다.

이 장에서는 코틀린의 기본 사용법을 간단히 소개합니다. 좀 더 자세한 내용은 안드로이드 앱 개발에서 필요할 때 추가로 소개합니다. 예를 들면, 배열은 7장 항목 선택을 다를 때 상세히 다룹니다.

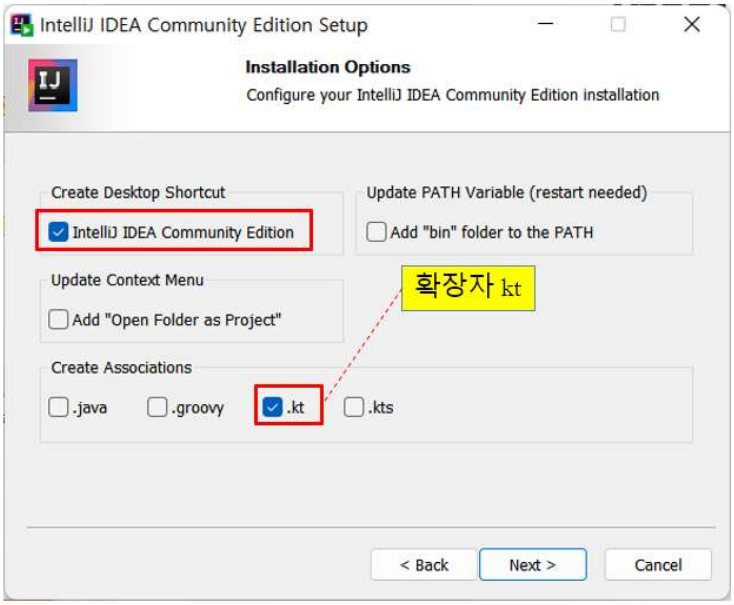
코틀린을 배우는 목적은 안드로이드 앱 개발을 위해서입니다. 아직 코틀린은 자바처럼 범용 언어가 아니며 사용 범위가 안드로이드 앱 개발로 제한적입니다. 따라서 안드로이드 앱 개발을 위해 코틀린을 배울 때 2가지에 초점을 맞추기를 바랍니다. 첫째, 안드로이드 앱은 객체지향 프로그램입니다. 둘째, 안드로이드 앱 코드를 간략화할 때 핵심은 람다 식입니다.

3.1 코틀린 설치 및 실행

코틀린을 실행하기 위해서는 젯 브레인(Jet Brains)에서 만든 IntelliJ IDEA[1]를 내려받아 설치해야 합니다. 웹 페이지(https://www.jetbrains.com/idea/) 화면 중앙에 놓인 Download를 클릭하면 아래에 보인 웹페이지로 이동합니다. 컴퓨터 운영체제를 선택[2]하고, Community Edition(idealC-2022.1.4.exe, 592Mb)을 내려받습니다[3].



idealC-2022.1.4.exe 파일을 이중 클릭해 설치를 진행합니다. 기본 설치 위치는 c:\Program Files\JetBrains 폴더입니다. 바탕화면에 단축 아이콘(desktop shortcut)을 생성하는 옵션과 IDEA와 연결할 파일 확장자(associations)를 kt로 하는 옵션을 선택하고 설치를 진행합니다.



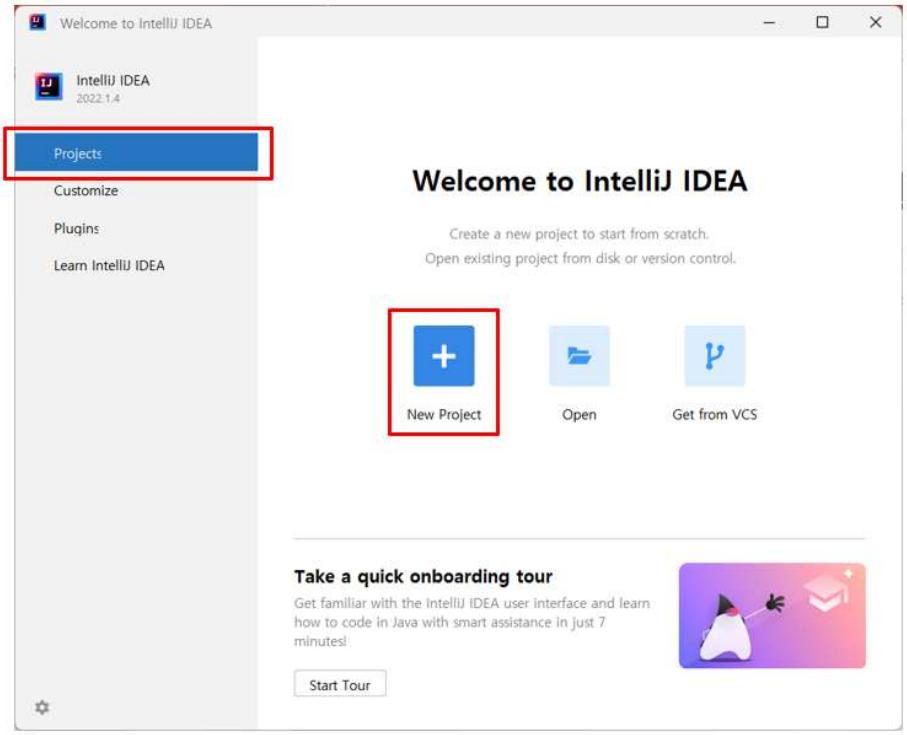
IDEA를 실행하면 저작권에 대한 동의를 요청하는 창이 나타납니다. 저작권 동의는 필수입니다. 이어서 데이터 공유(data sharing)에 동참할 것인지 묻는 창이 나타납니다. 참여하고 싶지 않으면 "Don't send"를, 참여하려면 "Send Anonymous Statistics"를 선택합니다.

---

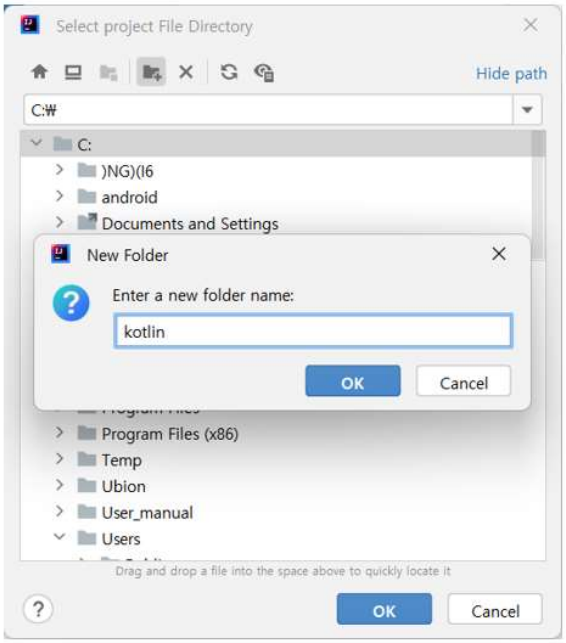
1) 간단히 IDEA로 부르겠습니다.
2) 여기서는 Windows입니다. 따로 선택하지 않아도 자동으로 인식합니다.
3) 현재(2022년 7월) 버전은 2022.1.4입니다.

이제 초기 화면을 볼 수 있습니다. 기본 컬러 테마는 Darcula입니다. 왼쪽 **Customize** 탭을 눌러 컬러 테마를 "IntelliJ Light"로 변경하면 아래 화면으로 바뀝니다.

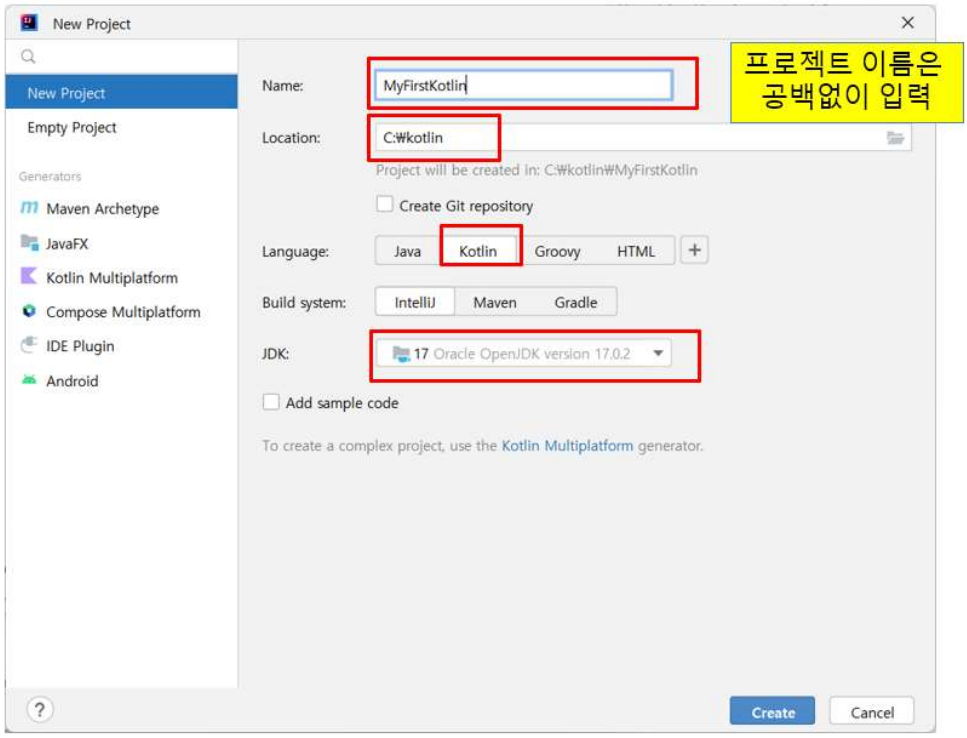
젯 브레인(Jet Brain)에서 안드로이드 스튜디오를 만들었기 때문에 IDEA도 화면 구성이나 사용 방법이 스튜디오와 매우 유사합니다. **Projects** 탭에서 New Project를 선택합니다.



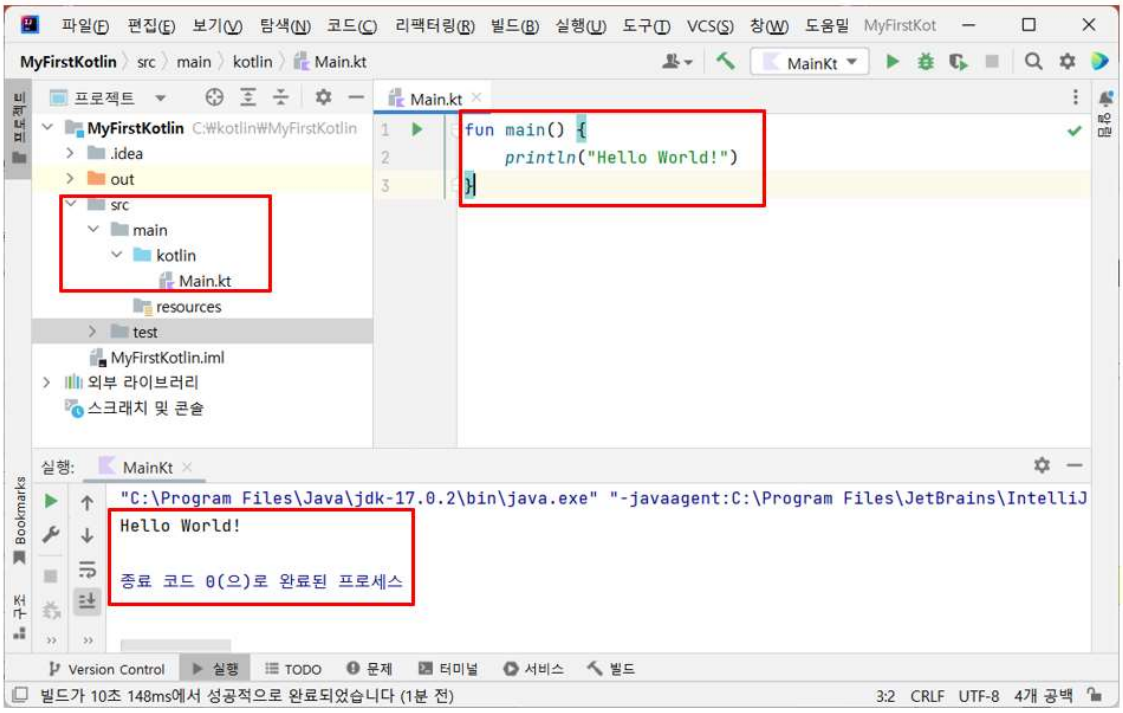
New Projet 창에서 프로젝트를 저장할 폴더를 지정해야 합니다. Location 오른쪽의 폴더 아이콘을 클릭하여 c:\ 드라이브에 새 폴더 **kotlin**을 만듭니다.



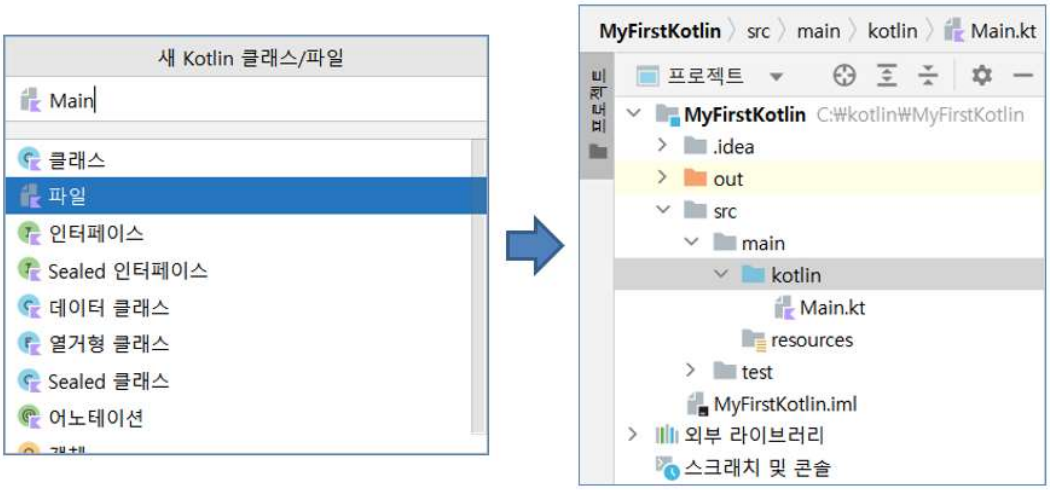
프로젝트 이름을 **MyFirstKotlin**으로 입력합니다. 프로젝트 이름은 반드시 공백없이 입력해야 합니다. IDEA는 JVM에서 코틀린을 실행하므로 JDK가 설치되어 있어야 합니다. 안드로이드 스튜디오를 설치할 때 이미 JDK를 설치했기 때문에, IDEA가 JDK가 설치된 경로를 알고 있습니다. Language는 'Kotlin'을 선택합니다. **Create** 버튼을 누르면 새 프로젝트가 생성됩니다.



전체화면은 프로젝트 창, 편집 창, 콘솔(console) 창 3개로 이루어져 있습니다. 화면 아래에 놓인 콘솔 창은 프로그램을 실행해야 활성화됩니다.



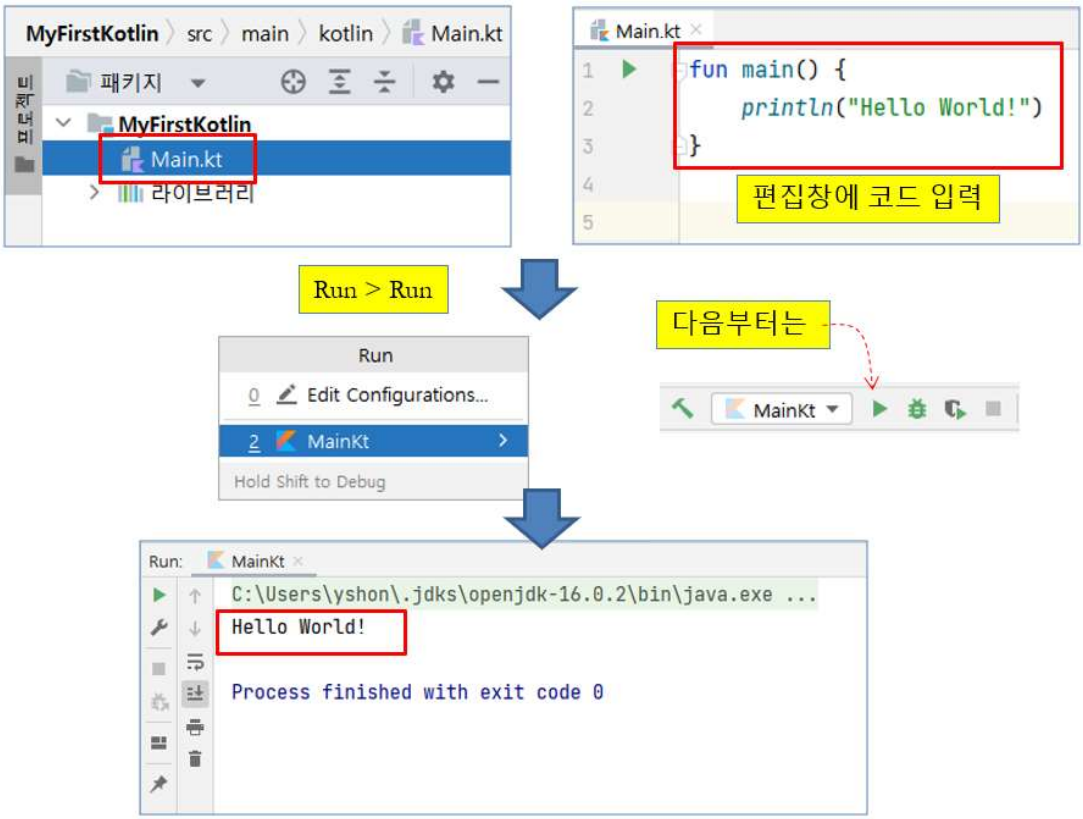
프로젝트 탐색 창에서 **프로젝트** 모드로 설정하고, **MyFirstKotlin > src > main > kotlin** 폴더로 이동합니다. kotlin 폴더를 클릭하고 오른쪽 버튼을 눌러 **새로 만들기 > Kotlin 클래스/파일**을 선택합니다. 타입은 "파일"을 선택하고, 이름은 **Main**을 입력합니다. kotlin 폴더에 Main.kt 파일이 생성되며, 파일 확장자는 kt입니다.



Main,kt 탭을 클릭하고, 편집 창에 아래 코드를 입력합니다.

```
fun main( ) {
    println("Hello World!")
}
```

실행을 위해 **실행(Run) > 실행(Run)**에서 Main.kt를 선택합니다. 다음 실행부터는 단축 아이콘을 클릭하면 됩니다.



3.2 코틀린 개요

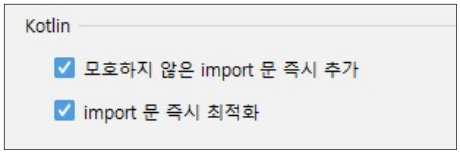
코틀린이 어떤 언어인지 살펴보겠습니다. 코틀린은 함수형 언어와 객체지향 언어 특징을 모두 갖는 다중 패러다임 언어입니다. 3.8절부터 객체지향언어를 다루기 때문에, 여기서는 주로 함수형 언어 관점에서 설명합니다.

■ 코틀린 프로그램 구조 : 함수

함수형 프로그래밍에서는 반드시 main() 함수를 포함해야 합니다. main() 함수는 프로그램 실행을 위한 진입점(entry point)[4] 역할을 합니다. 지금처럼 함수만 있는 경우는 파일 이름을 자유롭게 지정할 수 있습니다.

```
fun main( ) {
    println("Hello World!")
}
```

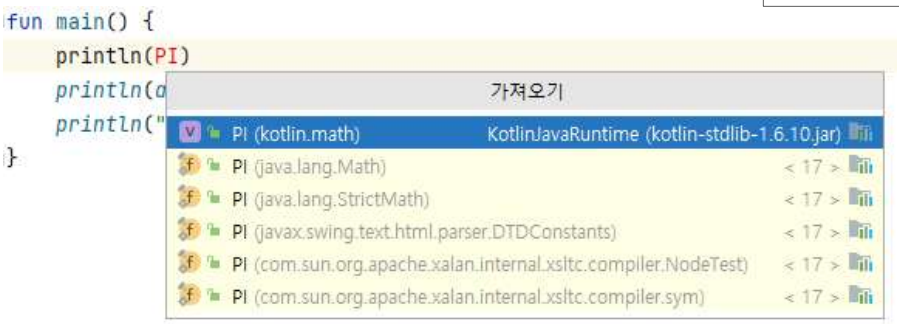
외부에서 정의한 함수를 사용할 때는 import 문이 필요합니다. import 문은 간단한 설정을 통해 자동 입력할 수 있습니다. **설정 > 에디터 > 일반 > 자동 가져오기**에서 아래 옵션을 모두 설정합니다. 안드로이드 스튜디오에서 설정하는 방식과 같습니다.



아래 예제는 파이(PI) 상수와 abs() 함수를 사용합니다. abs() 함수는 절댓값을 반환합니다. PI를 제공하는 클래스가 여러 개이기 때문에 한 개를 선택해야 합니다. kotlin.math에서 정의한 PI 상수를 선택합니다. abs() 함수 역시 kotlin.math에서 정의한 abs() 함수를 선택합니다. 2개의 import 문이 추가됩니다.

실행 결과는 오른쪽과 같습니다.

```
3.141592653589793
12.6
Hello World!
```

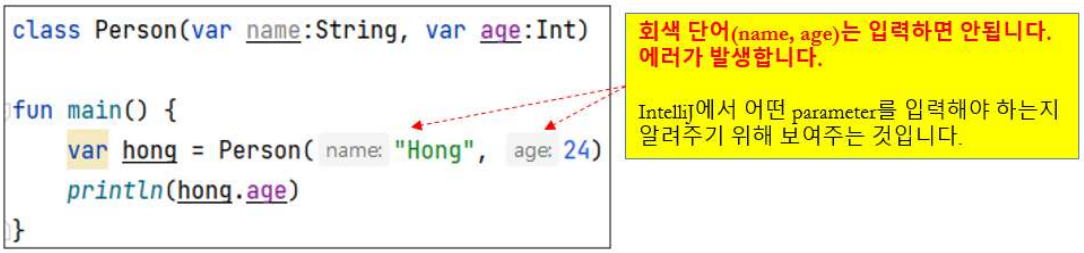



```
import kotlin.math.PI
import kotlin.math.abs

fun main( ) {
    println(PI)
    println(abs(-12.6))
    println("Hello World!")
}
```

4) CPU가 프로그램 실행을 위해 메모리에 저장된 코드 중 가장 먼저 참조해야 하는 코드를 말합니다.

■ 코틀린 프로그램 구조 : 함수 + 클래스

하나의 프로그램에 클래스와 함수가 함께 있는 구조입니다. 자바처럼 파일 이름을 클래스 이름으로 지정하지 않아도 됩니다. 다중 패러다임 언어는 최근 언어의 일반적 추세입니다. 기계학습이나 데이터 분석에 사용하는 파이썬도 다중 패러다임 언어입니다.



■ 코틀린 함수 특징

함수 sum()은 Int 타입 값을 반환하며, 함수 print_sum()은 반환 값이 없습니다.

```
fun main() {
    println(sum(23, 47))
    print_sum(12, 23)
}

fun sum(a:Int, b:Int): Int {
    return a + b
}

fun print_sum(a:Int, b:Int): Unit {
    println("sum of $a and $b is ${a+b}")
}
```

코틀린에서는 타입을 유추하는 기능이 있어 반환 값의 타입을 생략할 수 있습니다. 또, return 문 없이 반환 값을 직접 함수로 전달하는 수식으로 간략화할 수 있습니다. 함수를 변수처럼 취급하는 셈입니다.

```
fun sum(a:Int, b:Int) = a + b
```

함수 print_sum()은 반환 값이 없어 Unit라고 표기했지만, 생략할 수 있습니다.

```
fun print_sum(a:Int, b:Int) {
    println("sum of $a and $b is ${a+b}")
}
```

반환 값이 없는 함수 print_sum()도 블록 문이 아닌 수식으로 나타낼 수 있습니다.

```
fun print_sum(a:Int, b:Int) = println( "sum of $a and $b is ${a+b}" )
```

■ **변수 수식어: val과 var**

변수를 선언할 때 변수 이름 앞에 수식어 **val** 또는 **var**를 붙여야 합니다. val은 value의 약자로, val 변수는 불변 변수(immutable variable) 또는 읽기 전용 변수(read-only variable)입니다. val 변수는 값을 바꿀 수 없습니다. 자바에서 final로 선언한 것과 같습니다. val 변수는 선언할 때 초기화해야 합니다. val 변수를 초깃값 없이 선언만 하면, 이후 한 번만 초깃값을 할당할 수 있습니다. 초깃값을 할당한 이후에는 val 변수의 값은 변경할 수 없습니다.

아래 예에서, //(double slash)는 주석(comment)이며, 주석 내용은 // 이후부터 그 줄의 끝까지입니다. 주석은 실행과 무관하며, 코드를 설명하기 위해 사용합니다.

```
fun main() {
    val a:Int = 1 // 값 할당.
    val b = 2 // Int 타입 추론.
    val c:Int // 초깃값을 할당하지 않았을 때는 반드시 type을 선언해야 함.
    c = 3 // 초깃값 할당.
}
```

반대로 var은 variable의 약자로, var 변수는 가변 변수(mutable variables) 또는 쓸 수 있는 변수(writeable variable)입니다. var 변수는 언제든 값을 바꿀 수 있습니다.

```
fun main() {
    var x = 5      // Int 타입 추론
    x += 1         // 값 변경
}
```

3.3 코틀린 기초 : 기본 타입

코틀린 기본 타입은 숫자 타입, Char, Boolean, String이 있습니다. 이 밖에도 최상위 기본 클래스이면서 어떤 타입과도 호환 가능한 **Any**와 숫자를 대표하는 타입 **Number**가 있습니다. 타입 변환은 1:1로 타입 변환 함수를 적용하는 방식과 여러 개 타입 중 적절한 타입 하나를 추론하는 smart cast[5]가 있습니다. 또 널(null) 객체를 참조하는 예외(exception) 상황을 방지하기 위해, 변수 타입에 safe call(안전 호출)을 추가할 수 있습니다.

■ 기본 타입: 숫자

코틀린 타입의 첫 글자는 대문자입니다. 숫자 타입은 Double(64), Float(32), Long(64), Int(32), Short(16), Byte(8)가 있습니다. 괄호 안의 숫자는 비트(bit) 크기입니다. 부호 없는 숫자 타입은 정수에만 사용하며, 접두사 **U**(Unsigned)를 붙입니다. **U**Long, **U**Int, **U**Short, **U**Byte 입니다.

상수(literal constant)는 따로 타입을 지정할 필요가 없습니다. 실수(소수점이 있는 숫자)는 Double 타입으로 인식합니다. 123.5, 123.5e10은 모두 Double 타입입니다. Float 타입으로 만들려면, 123.5**F**, 123.5**f**와 같이, 숫자 뒤에 f 또는 F를 붙여야 합니다.

정수 상수는 Int 타입으로 인식합니다. Long 타입으로 만들려면 123**L**과 같이 대문자 L[6]을 붙입니다. 16진수(Hexadecimals) 상수는 **0x**0F처럼 접두사 0x[7]를 붙입니다. 이진수 상수는 **0b**00001011처럼 접두사 0b[8]를 붙입니다.

아래 예는 각 숫자 타입이 나타낼 수 있는 최솟값, 최댓값 및 비트 크기를 출력합니다.

```
fun main() {
    println("Byte : " + Byte.MIN_VALUE + ", " + Byte.MAX_VALUE + ", "
            + Byte.SIZE_BITS)
    println("Short : " + Short.MIN_VALUE + ", " + Short.MAX_VALUE + ", "
            + Short.SIZE_BITS)
    println("Int : " + Int.MIN_VALUE + ", " + Int.MAX_VALUE + ", "
            + Int.SIZE_BITS)
    println("Long : " + Long.MIN_VALUE + ", " + Long.MAX_VALUE + ", "
            + Long.SIZE_BITS)
    println("Float : " + Float.MIN_VALUE + ", " + Float.MAX_VALUE + ", "
            + Float.SIZE_BITS)
    println("Double : " + Double.MIN_VALUE + ", " + Double.MAX_VALUE + ", "
            + Double.SIZE_BITS)
}
```

5) type cast. cast는 (타입을) 변환한다는 뜻으로 사용합니다.
6) 소문자 'l'을 붙이면 에러입니다. 소문자 'l'은 숫자 '1'과 혼동할 수 있기 때문입니다.
7) 0x 또는 0X를 사용합니다. 'x'는 16진수(hexadecimal)의 'x'입니다.
8) 0b 또는 0B를 사용합니다. 'b'는 이진수(binary)의 'b'입니다.

실행 결과입니다. 비트 크기 대신 바이트 크기를 출력하려면 **타입.SIZE_BYTES**로 바꾸면 됩니다.

```
Byte : -128, 127, 8
Short : -32768, 32767, 16
Int : -2147483648, 2147483647, 32
Long : -9223372036854775808, 9223372036854775807, 64
Float : 1.4E-45, 3.4028235E38, 32
Double : 4.9E-324, 1.7976931348623157E308, 64
```

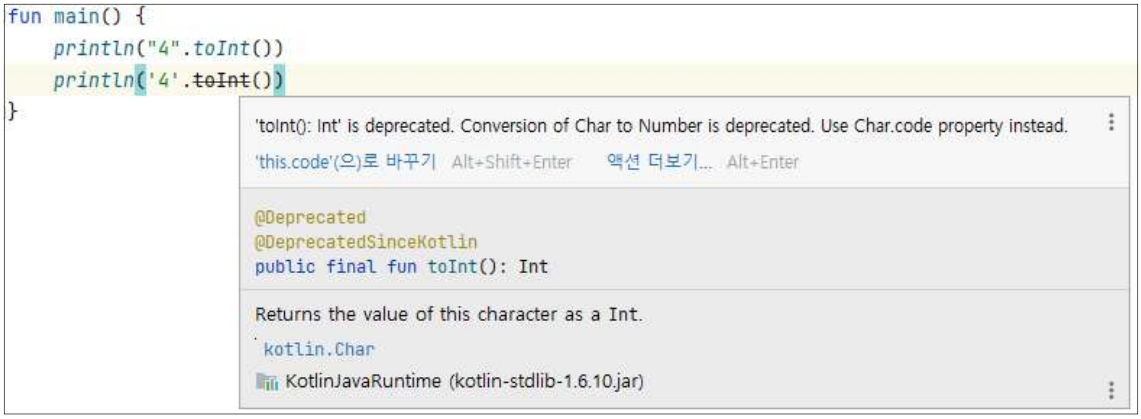
타입 변환을 하려면 타입 변환 함수를 사용합니다. 타입 변환 함수는 **to타입()** 입니다. to**Byte()**, to**Short( )**, to**Long( )**, to**Float( )**, to**Double( )**, to**Char( )**, **toString()** 처럼 to 다음에 변환할 타입을 지정합니다.

코틀린에서 모든 타입은 클래스(class)입니다. 아래 예에서 변수 f는 Float 타입 객체입니다. **객체.타입변환함수()** (**f.toInt( )**)를 사용해 Float 타입 객체를 Int 타입으로 변환할 수 있습니다. 실행 결과 정수 변수 i는 3을 출력합니다.

```
fun main() {
    var f: Float = 3.9f
    var i: Int = f.toInt()
    println()
    println("f = $f, i = $i")
}
```

타입 변환 함수에서 toInt()는 주의해서(?) 사용해야 합니다. 아래처럼 문자열(String) 상수 "4"를 toInt()로 변환하면 숫자 4로 바뀌지만, 문자(Char) 상수 '4'를 toInt()로 변환하면 ASCII 코드 52가 출력됩니다.

이때, toInt()는 deprecated(비추천)되었다고 표시됩니다. 비추천한다는 것은 함수 사용에 문제가 발생해 앞으로 지원하지 않으니 사용에 유의하라는 뜻입니다. deprecated된 함수는 실행할 수는 있지만, 엉뚱한 결과를 얻을 수 있기 때문입니다.



따라서, toInt()와 기능은 같지만 이름 및 사용 방법이 바뀐 대안을 찾아야 합니다. toInt()의 대안이 **digitToInt**(radix : Int)입니다. 이 함수의 파라미터는 진수(radix)입니다. 위 코드는 다음처럼 수정하면 됩니다. ASCII 코드 52에 해당하는 문자는 숫자 4입니다.

```
fun main() {
    var str = String.format("%c, %c", Char(52), Char(53))
    println(str)                          // 4, 5.
    println('4'.digitToInt(10))           // 4. 변환할 정수의 진수는 10진수.
    println('4'.digitToInt( ))            // 진수 10은 생략 가능.
}
```

다음 예를 통해 코틀린 숫자 타입 변환을 정리하겠습니다. 변수 f, d는 숫자 상수를 사용해 타입을 추론할 수 있어 타입을 생략할 수 있습니다. 변수 i는 타입을 지정하지 않으면 Int 타입으로 인식하기 때문에 Short 타입 선언이 필요합니다. 변수 fd와 i2f는 타입 변환 함수를 적용하였습니다. 함수 typeCheck()를 사용해 5개 변수의 타입을 확인할 수 있습니다.

typeCheck() 함수의 파라미터 타입은 **Any**입니다. Any는 임의의 타입이며, 어떤 타입과도 호환할 수 있습니다. **when** 문은 자바의 switch-case 문과 같으며, 3.4절에서 자세히 설명합니다.

```
fun main() {
    var f = 3.14f
    var d = 3.14
    var i:Short = 3     // Short 타입을 지정하지 않으면 Int 타입으로 인식
    var fd = f.toDouble()
    var i2f = i.toFloat()

    typeCheck(f)
    typeCheck(d)
    typeCheck(i)
    typeCheck(fd)
    typeCheck(i2f)
}

fun typeCheck(v:Any) {     // Any는 임의의 타입: 어떤 타입과도 호환이 가능.
    when (v) {             // 자바의 switch-case 구문
        is Short -> println("the type of $v is Short.")  // is는 자바의 instanceOf와 같음.
        is Int -> println("the type of $v is Int.")
        is Float -> println("the type of $v is Float.")
        is Double -> println("the type of $v is Double.")
    }
}
```

.

실행 결과입니다.

```
the type of 3.14 is Float.
the type of 3.14 is Double.
the type of 3 is Short.
the type of 3.140000104904175 is Double.
the type of 3.0 is Float.
```

■ 기본 타입: Char

Char 타입 변수는 홑 따옴표(‘)로 묶은 문자만 할당할 수 있습니다. ASCII 코드에 해당하는 숫자를 직접 할당하면 에러가 발생합니다. 65는 문자 ’A’의 ASCII 코드입니다.

```
val c:Char = 'A'
val c2:Char = 65
```

ASCII 코드를 toChar()를 사용하여 Char 타입으로 변환하면 됩니다. 유니코드(unicode)는 ‘\u’로 나타냅니다. 한글 음절 ‘한’의 유니코드는 '\ud55c'입니다. 실행 결과 “A, B”와 ‘한’을 출력합니다.

```
fun main() {
    val code:Int = 65                // 문자 'A'의 ASCII 코드
    val han_code:Char = '\ud55c'  // 한글 음절 '한'의 unicode

    println(code.toChar() + ", " + (code+1).toChar())
    println(han_code)
}
```

아래 예는 숫자(48부터 57)에 속하는 ASCII 코드만 문자로 변환하고, 이 범위 밖의 ASCII 코드는 ‘-’로 출력합니다. 실행 결과 “0123456789---”를 출력합니다.

```
fun main() {
    for (i in 48..60)
        print("${decimalValue(i)}")
    println()
}

fun decimalValue(i:Int) =
    if (i in 48..57) i.toChar() else '-'
```

■ 기본 타입: Boolean

Boolean 타입 변수의 값은 true(참) 또는 false(거짓)입니다. Boolean 타입 변수는 논리 연산식에서 주로 사용합니다. 논리 연산식에 사용하는 논리 연산은 &&(논리 AND), ||(논리 OR), !(논리 NOT)입니다. 특히 ‘!’은 변수의 Boolean 상태(또는 스위치 상태)를 바꿀 때 사용합니다. true(스위치 ON) → false(스위치 OFF) → true(스위치 ON) → ... 와 같이 상태 순환을 반복하는 것을 상태 토글(toggle)이라고 부릅니다.

```
fun main() {
    var foo:Boolean = true
    val bar = false

    println( foo && bar )    /* false */
    println( foo || bar )     /* true */
    println( !foo )           /* false */

    foo = !foo               // 상태 toggle.
    println( foo )           // false
}
```

■ 기본 타입: String

String 타입 변수는 겹따옴표(“)로 묶은 문자열만 할당할 수 있습니다. 문자열은 문자 배열로써, 특정 위치의 문자를 인덱스(index)를 사용하여 참조할 수 있습니다. 문자열의 인덱스 범위는 0부터 (문자열 길이-1)까지입니다. 인덱스를 사용하여 문자열 원소를 참조할 때 **배열객체.**get(index) 또는 **배열객체**[index]를 사용합니다.

```
fun main() {
    var foo:String = "My First Kotlin"

    val size = foo.length
    for (i in 0 until size)      /* until은 size-1 까지임 */
        print(foo[i])
    println()
    println("first char = ${foo[0]}, last char = ${foo[size-1]}")

    var ch1:Char = foo.get(3)
    var ch2:Char = foo[9]
    println("length=$size, ch1 = $ch1 and ch2 = $ch2")
}
```

실행 결과입니다.

```
My First Kotlin
first char = M, last char = n
length=15, ch1 = F and ch2 = K
```

코틀린은 힙(Heap) 메모리 중 String pool에 문자열을 저장하고 관리합니다. String 타입 객체는 불변(immutable) 객체, 즉 읽기 전용(read-only) 객체입니다. var로 선언했다고 해서 불변 객체가 가변 객체로 바뀌는 것이 아닙니다, String 타입 객체의 문자 원소가 바뀌면, 바뀐 문자를 포함하는 새로운 String 타입 객체가 생성됩니다. var로 선언한 foo 객체는 이렇게 생성된 새 객체를 참조할 뿐입니다. 이전에 foo가 참조하던 문자열 객체는 메모리를 그대로 차지하고 있습니다. 코틀린 String은 **5.4절**에서 좀 더 자세히 설명합니다. foo와 foo2는 같은 문자열을 참조합니다.

```
fun main() {
    var foo:String = "My First Kotlin"
    foo = foo.substring(0, 9) + "python"
    println(foo)

    var foo2 = foo.replace("Kotlin", "python")
    println(foo2)
    println(foo == foo2)
}
```

실행 결과입니다.

```
My First python
My First python
true
```

■ 규격화된 String 객체 출력: String.format()

String.format()은 규격화된 문자열(formatted string)을 출력합니다. 단, 데이터 타입에 맞게 출력 서식을 지정해야 합니다. String 타입 변수의 출력 서식은 %s, Int 타입은 %d, Float 타입은 %f입니다. Float 타입은 **%.nf**처럼 소수점 이하 **n**번째 자리까지 출력을 제한할 수 있습니다. Int 타입 변수의 출력 서식을 **%nd**로 지정하면, 출력 자릿수를 **n**으로 지정하고 오른쪽 맞춤을 지정한 것입니다. **%5d**는 xx234로 출력합니다. 여기서 x는 공백입니다. **%-5d**와 같이 '-'를 붙이면 왼쪽 맞춤을 지정한 것이며, 234xx와 같이 출력합니다.

```
import kotlin.math.PI

fun main() {
    val pi:Float = PI.toFloat()
    val digit = 234
    val str = "Hello"
    println("PI = %.5f, %5d, %10s".format(pi, digit, str))

    val length = 3000
    val lengthStr:String = String.format("Length: %d meters", length)
    println(lengthStr)
}
```

실행 결과입니다.

```
PI = 3.14159,   234,      Hello
Length: 3000 meters
```

■ 문자열 템플릿(String templates)

문자열 템플릿(template)은 앞에서 정의한 변수를 문자열 안에서 참조할 수 있습니다. 변수 이름 앞에 반드시 $를 붙여야 합니다. 문자열 템플릿에 수식을 포함할 때는 중괄호로 묶어야만 합니다. "$s.length"는 중괄호로 묶지 않았기 때문에 $s만 문자열 템플릿이고, ".length"는 출력 문자열입니다. 반면 "${s.length}"는 중괄호로 묶은 수식이기 때문에, String 타입 객체 s의 길이를 출력합니다.

```
fun main() {
    var a = 1
    val s1 = "a is $a"

    a = 2
    val s2 = "${s1.replace("is", "was")}, but now is $a"
    println(s2)

    val s = "abc"
    println("$s.length is ${s.length}")
}
```

실행 결과입니다.

```
a was 1, but now is 2
abc.length is 3
```

문자열 템플릿에 특수 기호를 포함하려면 **${'특수기호'}**를 사용하거나 escape char(backslash 기호)를 사용해야 합니다. 특수 기호인 '$'를 출력하기 위해 **${'$'}**로 복잡하게 표현하는 대신 **\$**로 간단히 표현하는 게 좋습니다.

```
fun main() {
    val expr:String = "My First Kotlin"
    val amount = 10

    val lengthStr = "text length: ${expr.length}"
    val priceStr = "price: USD ${'$'}$amount"
    val priceStr2 = "price: USD \$$amount"   // 첫 번째 $는 dollar($) 기호
                                             // 두 번째 $는 문자열 템플릿 기호
    println(lengthStr)
    println(priceStr)
    println(priceStr2)
}
```

실행 결과입니다.

```
text length: 15
price: USD $10
price: USD $10
```

■ safe call(안전 호출)

프로그램 실행 중에 널(null) 변수(객체)를 참조하면, 예외(=에러)가 발생하면서 실행이 중단됩니다. 즉, NullPointerException(NPE)이 발생합니다. 이를 방지하기 위해 코틀린에서는 변수가 null 값을 갖는지를 프로그래머가 직접 지정하도록 하였습니다. 기본은 null 값을 갖지 않는 것입니다. null 값을 갖지 않는 변수에 null을 할당하면 에러가 발생합니다. 변수가 null 값을 갖는 것을 허용하려면, 변수를 선언할 때 타입 뒤에 물음표(?) 기호를 붙여야 합니다.

```
var str: String
str = null        // 에러 발생
```

```
var str:String?
str = null
```

null 값을 갖는 변수를 사용할 때 신경을 써야 할 부분이 꽤 있습니다. 예제를 통해서 확인해 보겠습니다.

함수에서 safe call로 정의한 파라미터를 전달받을 때도 "**String?**"처럼 safe call 타입으로 선언해야만 합니다. nullCheck() 함수는 변수 str이 null인지를 조사합니다.

```
fun main( ) {
    var str:String? = null
    nullCheck(str)
}

fun nullCheck(s:String?) {
    if (s == null)
        println("\"$s\" is null")
    else
        println("\"$s\" is NOT null")
}
```

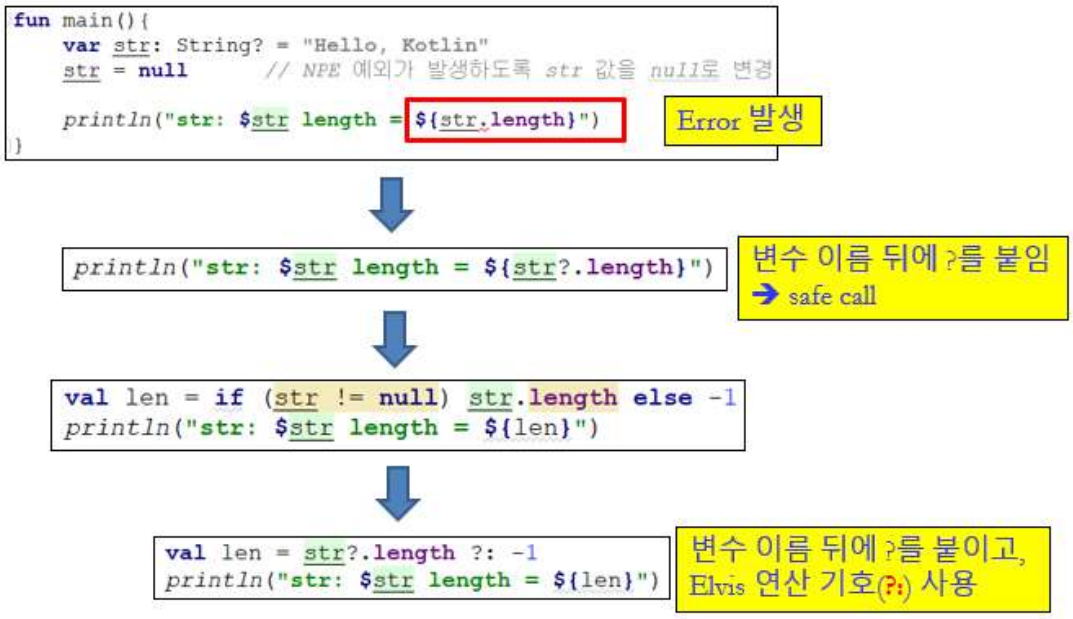
null 값과 빈 문자열("")은 같지 않습니다. 빈 문자열("")로 초기화한 str2와 실행 결과를 비교해 보기 바랍니다. safe call로 정의한 객체의 속성이나 메소드를 호출할 때도 "s?.isEmpty( )"와 같이 객체 이름 뒤에 '?'를 붙여서 사용해야 합니다.

```
fun main( ) {
    var str:String? = null
    nullCheck(str)
    emptyCheck(str)

    var str2:String? = ""
    nullCheck(str2)
    emptyCheck(str2)
}

fun nullCheck(s:String?) { ... }

fun emptyCheck(s:String?) {
    if (s?.isEmpty() == true)
        println("\"$s\" is empty")
    else
        println("\"$s\" is NOT empty")
}
```

실행 결과입니다.

```
"null" is null
"null" is NOT empty
"" is null
"" is empty
```

safe call 표현은 Elvis 연산 기호(?:)와 함께 사용하면 코드를 압축해서 표현할 수 있습니다. Elvis 연산 기호의 기능은 if-else 문과 같습니다. Elvis 연산 기호 앞에 놓인 수식은 조건이 참(true)일 때 값이며, 연산 기호 뒤에 놓인 값은 조건이 거짓(false)일 때 값입니다. 변수 len은 str 객체가 null이면 -1, str 객체가 null이 아니면 str?.length 값이 할당됩니다.



■ smart cast(자동 형 변환)

smart cast는 코틀린 컴파일러가 변수의 타입을 유추하는 것을 말합니다. 아래 예에서 Number는 숫자 타입을 대표하는 타입입니다. 즉, Number 타입은 어느 숫자 타입과도 호환됩니다. Any 타입은 최상위 기본 클래스로써 어떤 타입과도 호환 가능합니다.

변수 num에 "8L"을 할당하면, 변수 num의 타입은 Long 타입으로 바뀝니다. 마찬가지로 변수 str은 Any 타입으로 선언했지만, 문자열 "Hello, Kotlin"을 할당하면서 구체적 타입인 String 타입으로 바뀝니다. 이러한 예가 smart cast(자동 형 변환)입니다.

함수 typeCheck()의 파라미터 타입은 Any입니다. typeCheck(num)를 호출하면, 변수 num 타입이 Long 타입임을 확인할 수 있습니다. typeCheck(8)을 호출하면, 정수 8은 Int 타입으로 인식합니다. 이 예도 마찬가지로 smart cast 예입니다.

```
fun main() {
    var num:Number = 8L
    val str:Any

    typeCheck(num)  // Long 타입으로 인식
    typeCheck(8)      // Int 타입으로 인식
    var ld = num as Long
    typeCheck(ld)     // Long 타입으로 인식

    str = "Hello, Kotlin" // String 타입으로 인식
    if (str is String)
        println("\"$str\" is ${str.javaClass}")
}

fun typeCheck(x: Any) {
    if (x is Int)
        println("$x is ${x.javaClass}")
    else if (x !is Int)
        println("$x is NOT Int. The type is ${x.javaClass}")
}
```

실행 결과입니다.

```
8 is NOT Int. The type is class java.lang.Long
8 is class java.lang.Integer
8 is NOT Int. The type is class java.lang.Long
"Hello, Kotlin" is class java.lang.String
```

위 예에서 "**num as Long**"은 "**대표_타입 as 구체적_타입**"의 의미입니다. **대표_타입**은 Number나 Any이며, as를 사용해 **구체적_타입**으로 변환합니다.

```
var ld = num as Long
```

3.4 코틀린 기초 : 조건문과 반복문

조건문은 if 문과 when 문이 있으며, 반복문은 for 문과 while 문이 있습니다. 이미 다른 언어를 사용하면서 조건문이나 반복문은 잘 알고 있을 것입니다. 예제 중심으로 코틀린의 조건문과 반복문 사용법을 알아보겠습니다.

■ 조건문: if 문

2개 숫자 중 큰 값을 출력하는 프로그램은 대개 아래처럼 작성합니다.

```
fun main() {
    val a = 12
    val b = 7

    var greater = maxValue(a, b)
    println("The greater value is $greater")
}

fun maxValue(x:Int, y:Int):Int {
    if ( x > y ) {
        println("$x is chosen")
        return x
    } else {
        println("$y is chosen")
        return y
    }
}
```

그러나 코틀린에서는 함수의 블록 및 return 문을 없애고, 할당문을 사용합니다. if 블록이나 else 블록에서 마지막 변수가 함수의 반환 값입니다.

```
fun maxValue(x:Int, y:Int) =
    if ( x > y ) {
        println("$x is chosen")
        x
    } else {
        println("$y is chosen")
        y
    }
```

또는 변수에 직접 수식을 대입할 수 있습니다.

```
var greater =
    if ( a > b ) {
        println("$a is chosen")
        a
    } else {
        println("$b is chosen")
        b
    }
```

이번에는 if-else if 문을 사용해 보겠습니다. "readLine()!!"은 콘솔(console)에서 값을 읽어오는 문장입니다. "!!"은 "null 값은 절대 안 됨!" (non-null assertion)이란 표시입니다.

```
val score = readLine( )!!.toFloat()
```

readLine()!!을 if-else 문으로 풀어 쓰면 아래와 같습니다.

```
val score = if (readLine( ) != null)
                readLine()?.toFloat()
            else
                throw NullPointerException("Expression 'readLine()' must not be null")
```

조건식에서 값의 범위를 지정할 때 "**in (하한)..(상한)**"을 사용하면, 범위에 (하한)과 (상한)이 모두 포함됩니다. 즉 논리 AND 연산 기호를 사용한 아래 조건식은

```
if (score >= 80.0 && score <= 89.9)
```

"in (하한)..(상한)"을 사용하면, 아래처럼 간단히 나타낼 수 있습니다

```
if (score in 80.0..89.9)
```

아래 예제는 성적(score)을 입력받아 등급(grade)을 출력하는 프로그램입니다. if-else if 문은 when 문으로 바꿀 수 있습니다.

```
fun main() {
    print("Enter the score : ")
    val score = readLine()!!.toFloat()
    var grade = 'F'

    if (score >= 90)
        grade = 'A'
    else if (score in 80.0..89.9)  // else if (score >= 80.0 && score <= 89.9)
        grade = 'B'
    else if (score in 70.0..79.9)
        grade = 'C'

    println("Your grade is $grade")
}
```

■ **조건문: when 문**

자바의 switch-case 문이 코틀린에서는 when 문으로 바뀌었습니다. switch-case 문은 각 case의 실행문 마지막에는 반드시 break 문이 필요합니다. 실수로 break 문을 빠트리면, 다음 case 문을 실행하는 오류가 발생하곤 했습니다. 코틀린에서는 따로 break 문이 필요 없으며, 한 가지 조건에 해당하는 문장을 실행하면 when 문을 빠져나옵니다.

when 문의 기본 구조는 "**조건 -> 실행문**"입니다. 정해진 조건에 포함하지 않는 경우는 뭉뚱그려 키워드 **else**를 사용합니다. 실제 모든 조건을 일일이 다 열거할 수 없으므로, else 조건 사용은 거의 필수입니다.

```
fun main() {
    checkValue(1)
    checkValue(3)
    checkValue(5)
}

fun checkValue(x:Int) {
    when (x) {
        1 -> println("x is 1")
        2 -> println("x is 2")
        3, 4 -> println("x is 3 or 4")
        else -> {
            println("x is greater than or equal to 5.")
        }
    }
}
```

실행 결과입니다.

```
x is 1
x is 3 or 4
x is greater than or equal to 5.
```

앞에서 if-else로 구현했던 성적 등급 프로그램을 when 문으로 다시 구현했습니다.

```
fun main() {
    print("Enter the score : ")
    val score = readLine()!!.toFloat()
    var grade:Char

    when (score) {
        in 90.0..100.0 -> grade = 'A'
        in 80.0..89.9 -> grade = 'B'
        in 70.0..79.9 -> grade = 'C'
        else -> grade = 'F'
    }
    println("score: $score, grade: $grade")
}
```

when 문의 다른 사용 예를 보겠습니다. enum class는 나열(enumeration) 클래스로써 문자열을 나열한 순서대로 정수를 할당합니다. 아래 Color 클래스에서는 RED는 0, ORANGE는 1, YELLOW는 2로 차례대로 정수를 할당합니다. 실제 코드에서는 숫자 대신 문자열을 사용하므로, 코드를 읽기가 쉬워지겠죠.

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

enum class에서 정의한 문자열을 참조할 때는 **(클래스_이름).문자열**을 사용합니다. enum class는 객체를 생성하지 않고 클래스 이름을 사용합니다. Color 클래스의 ORANGE를 참조한다면, Color.ORANGE를 사용하면 됩니다. getMnemonics()와 getWarmth() 함수는 모두 할당문 형태로 구현했습니다.

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

fun main() {
    println(getMnemonics(Color.BLUE))
    println(getWarmth(Color.ORANGE))
}

fun getMnemonics(color: Color): String =
    when (color) {
        Color.RED -> "Richard"      // Mnemoic 단어 뜻은 단어를 기억하기 쉽게
        Color.ORANGE -> "OKay"      // 도와주는 역할입니다.
        Color.YELLOW -> "Young"     //  물리 공식 Q=CV를 '콱 씨버' 로 기억하는 것처럼
        Color.GREEN -> "Give"       // ORANGE를 O.K.로 기억하는 것이죠.
        Color.BLUE -> "Buy"         // 일종의 연상 기억법입니다.
        Color.INDIGO -> "Inside"    // 컴퓨터 용어로 연관 메모리인 셈입니다.
        Color.VIOLET -> "Vain"
    }

fun getWarmth(color: Color):String =
     when (color) {
        Color.RED, Color.ORANGE, Color.YELLOW -> "Warm"
        Color.GREEN -> "Neutral"
        Color.BLUE, Color.INDIGO, Color.VIOLET -> "Cold"
     }
```

■ 반복문: for 문과 while 문

369 게임 해본 적 있나요? 1, 2 다음에 3은 숫자를 말하지 않고 손뼉을 치는 게임이죠. 369 게임과 비슷한 게임이 fizz buzz 게임입니다. 3의 배수이면 'Fizz'를, 5의 배수이면 'Buzz'를, 15의 배수이면 'FizzBuzz'를 말하는 게임입니다. 369 게임보다는 난도가 높은 편입니다. fizz buzz 게임은 루프 문 실습할 때 자주 사용하는 예제입니다.

```
fun main() {
    for (i in 1..100) {
        print(fizzBuzz(i))
        if (i % 10 == 0) println()
    }
    for (i in 100 downTo 1 step 2) {
        print(fizzBuzz(i))
        if (i % 10 == 0) println()
    }
}

fun fizzBuzz(i:Int):String =
    when {
        i % 15 == 0 -> "FizzBuzz"
        i % 3 == 0 -> "Fizz"
        i % 5 == 0 -> "Buzz"
        else -> "$i"
    }
```

실행 결과 중 일부입니다.

```
12Fizz4BuzzFizz78FizzBuzz
11Fizz1314FizzBuzz1617Fizz19Buzz
Fizz2223FizzBuzz26Fizz2829FizzBuzz
3132Fizz34BuzzFizz3738FizzBuzz
41Fizz4344FizzBuzz4647Fizz49Buzz
Fizz5253FizzBuzz56Fizz5859FizzBuzz
```

for 문은 while 문과 do-while 문으로도 구현 가능합니다.

```
fun main() {
    for (i in 1..100) {
        print(fizzBuzz(i))
        if (i % 10 == 0) println()
    }
    for (i in 100 downTo 1 step 2) {
        print(fizzBuzz(i))
        if (i % 10 == 0) println()
    }
}

fun fizzBuzz(i:Int):String {
    return when {
        i % 15 == 0 -> "FizzBuzz"
        i % 3 == 0 -> "Fizz"
        i % 5 == 0 -> "Buzz"
        else -> "$i"
    }
}
```

while 루프

```
var j = 1
while (j <= 100) {
    print(fizzBuzz(j))
    if (j % 10 == 0) println()
    j++
}
```

do-while 루프

```
var j = 1
do {
    print(fizzBuzz(j))
    if (j % 10 == 0) println()
    j++
} while (j<=100)
```

3.5 코틀린 기초 : 배열

배열은 7장에서 자세히 다루기 때문에, 여기서는 간단히 살펴보겠습니다. 안드로이드 앱에서는 주로 1차원 배열을 다룹니다.

arrayOf() 생성자를 사용하여 1차원 배열 객체를 생성합니다. arrayOf()의 파라미터 개수는 제한이 없으며, 파라미터의 타입에도 제한이 없습니다.

```
val numbers:Array<Int> = arrayOf(1, 2, 3)
val mixedArray:Array<Any> = arrayOf(7, "Kotlin", false)

println(numbers.contentToString(
println(mixedArray.contentToStri
```

```
public inline fun <reified @PureReifiable T> arrayOf(
    vararg elements: T
): Array<T>
```

한 가지 타입의 값으로 초기화하여 1차원 배열 객체를 생성하거나, 여러 종류 타입의 값들을 뒤섞은 1차원 배열 객체를 만들 수도 있습니다.

```
fun main() {
    val numbers:Array<Int> = arrayOf(1, 2, 3)
    val mixedArray:Array<Any> = arrayOf(7, "Kotlin", false)

    println(numbers.contentToString())
    println(mixedArray.contentToString())
}
```

배열 원소의 타입을 한 가지로 제한할 수 있습니다. arrayOf<**타입**>()을 사용하거나 **char**ArrayOf, **boolean**ArrayOf, **long**ArrayOf, **short**ArrayOf, **byte**ArrayOf, … 처럼 사용할 수도 있습니다. 1차원 배열의 원소는 인덱스(index)를 사용하여 참조할 수 있습니다. 배열의 인덱스 범위는 0부터 (**배열객체.size**−1)까지입니다. 인덱스를 사용하여 배열의 원소를 참조할 때 **배열객체.**get(index) 또는 **배열객체**[index]를 사용합니다. 배열의 첫 번째 원소와 마지막 원소를 참조하려면, **배열객체.**first()와 **배열객체.**last()를 각각 사용하면 됩니다.

```
fun main() {
    val intOnlyArray = arrayOf<Int>(1, 2, 3)
    val intOnlyArray2 = intArrayOf(4, 5, 6, 7)

    val i = intOnlyArray[0]
    val i2 = intOnlyArray2.get(2)
    println("i=$i, i2=$i2")
    println("${intOnlyArray.size}, ${intOnlyArray2.size}")
    println("${intOnlyArray.first()}, ${intOnlyArray2.last()}")
}
```

실행 결과입니다.

```
i=1, i2=6
3, 4
1, 7
```

배열 원소의 값을 변경할 때는 할당문이나 set()을 사용하지만, 할당문 사용을 추천합니다. 아래 코드는 배열 intOnlyArray2의 4번째 원소의 값을 9로 변경하기 위해 할당문과 set()을 사용한 예입니다.

```
intOnlyArray2[3] = 9
intOnlyArray2.set(3, 9)
```

arrayOf<Int>()로 생성한 배열 객체와 intArrayOf() 생성한 배열 객체의 타입은 전혀 다릅니다. 따라서 함수로 파라미터를 전달할 때 파라미터의 타입도 다릅니다.

```
fun main() {
    val intOnlyArray:Array<Int> = arrayOf<Int>(1, 2, 3)
    val intOnlyArray2:IntArray = intArrayOf(4, 5, 6, 7)

    intOnlyArray[0] = 0
    intOnlyArray2[3] = 9
    showElement(intOnlyArray)
    showElement2(intOnlyArray2)
}

fun showElement(arr:Array<Int>) {
    for (i in arr)
        print("$i\t")
    println()
}

fun showElement2(arr:IntArray) {
    for (i in arr)
        print("$i\t")
    println()
}
```

실행 결과입니다.

```
0   2   3
4   5   6   9
```

다양한 타입의 파라미터와 호환 하도록 만들려면 T-파라미터를 정의하면 됩니다. 아래 예제는 배열 객체의 타입과 무관하게 배열 원소를 출력하는 통합 함수를 정의하였습니다.

```
fun main() {
    val words: Array<String> =
        arrayOf("python", "kotlin", "swift")
    val intOnlyArray = arrayOf(4, 5, 6, 7)

    unified<String>(words)
    unified<Int>(intOnlyArray)
}

fun <T> unified(arr:Array<T>) {
    for (i in arr)
        println(i)
}
```

■ 2차원 배열 객체

2차원 배열 객체는 1차원 배열 객체를 원소로 포함하면 됩니다. 원소를 출력할 때는 바깥쪽 for 루프는 행을, 안쪽 for 루프는 열을 지정합니다. 직접 2차원 배열의 원소를 지정할 때는 2개의 인덱스가 필요하며, 첫 번째 인덱스는 행을, 두 번째 인덱스는 열을 가리킵니다.

```
fun main() {
    val array1 = arrayOf(1, 2, 3)
    val array2 = arrayOf(4, 5, 6)
    val array3 = arrayOf(7, 8, 9)

    val arr2d = arrayOf(array1, array2, array3)

    for (e1 in arr2d) {
        for (e2 in e1)
            print("$e2 ")
        println()
    }

    println(arr2d[0][1]) // 1행 2열
    println(arr2d[1][1]) // 2행 2열
    println(arr2d[2][1]) // 3행 2열
}
```

실행 결과입니다.

```
1 2 3
4 5 6
7 8 9
2
5
8
```