Kotlin 코틀린 핵심 파악하기

핵심만 골라 배우는
안드로이드 스튜디오 3 & 프로그래밍
특별부록

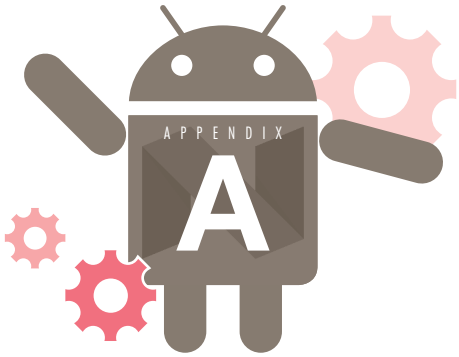
심재철 지음

Jpub 제이펍

※ 드리는 말씀

- 이 부록은 《핵심만 골라 배우는 안드로이드 스튜디오 3 & 프로그래밍》을 구매한 독자를 위해 특별 제작된 무료 배포 버전입니다.
- 이 부록은 안드로이드 정식 개발 언어로 공표된 코틀린(Kotlin)을 쉽고 빠르게 파악하는 데 도움이 되도록 원서와는 별도로 작성된 것입니다.
-
- 이 부록에 기재된 내용을 기반으로 한 운용 결과에 대해 저자, 소프트웨어 개발자 및 제공자, 제이펍 출판사는 일체의 책임을 지지 않으므로 양해 바랍니다.
- 이 부록에 기재한 회사명 및 제품명은 각 회사의 상표 및 등록명입니다.
- 이 부록에서는 ™, ©, ® 등의 기호를 생략하고 있습니다.
- 이 부록에서 사용하고 있는 실제 제품 버전은 독자의 학습 시점에 따라 책의 버전과 다를 수 있습니다.
- 《핵심만 골라 배우는 안드로이드 스튜디오 3 & 프로그래밍》은 전국의 대형 서점 및 인터넷 서점에서 구매하실 수 있습니다.
- 상기 도서의 예제 프로젝트 파일은 다음의 사이트에서 다운로드할 수 있습니다.
  - *https://github.com/Jpub/AndroidStudio3*
- 특별 부록과 관련된 문의사항은 옮긴이나 출판사로 연락주시기 바랍니다.
  - 옮긴이: jcspro@hanafos.com
  - 출판사: readers.jpub@gmail.com

차례



코틀린 핵심 파악하기: 개요와 실습 환경 구축

구글 I/O 2017에서 코틀린(Kotlin)이 안드로이드 애플리케이션의 공식 개발 언어로 발표되었다. 따라서 안드로이드 애플리케이션은 자바와 코틀린을 같이 사용하여 개발할 수 있다. 코틀린으로 작성된 소스 코드는 코틀린 컴파일러에 의해 JVM(자바 가상머신)에서 실행되는 자바 바이트 코드(클래스 파일)로 생성되므로 자바와 100% 호환된다. 즉, 이상적으로는 자바 애플리케이션이 실행 가능한 환경이라면 코틀린 애플리케이션도 실행 가능하다는 의미이다. 이와 더불어 코틀린은 새로운 언어답게 여러 가지 장점을 갖고 있으므로, 앞으로는 안드로이드 애플리케이션 개발자가 주로 사용하는 프로그래밍 언어가 될 것이다. 그리고 언어의 기본적인 문법이나 중요 개념들이 자바와 유사하므로 기존 자바 개발자는 코틀린을 쉽게 익힐 수 있다. 이번 장에서는 코틀린 언어의 개요와 코틀린을 배우기 위해 실습에 필요한 환경을 설정하는 내용을 알아본다.

A.1 코틀린 언어 개요

코틀린은 OSS(오픈 소스 소프트웨어)로 젯브레인(JetBrains)에서 개발한 프로그래밍 언어이다. 2011년부터 개발이 시작되어 2016년에 1.0 정식 버전이 발표되었으며, 현재도 계속 진화 중이다. 이미 얘기했듯이, 코틀린 애플리케이션은 JVM에서 실행되므로 다음 네 가지 형태의 애플리케이션을 만들 수 있다. 컴퓨터의 운영체제에서 독자적으로 실행되는 네이티브(Native) 애플리케이션, 웹 브라우저에서 실행되는 자바스크립트, 서버에서 실행되는 Http 서블릿, 그리고 구글 안드로이드 애플리케이션이다.

코틀린의 특징을 요약하면 다음과 같다.

- JVM에서 실행되므로 자바와 완전히 호환되며 크로스 플랫폼을 지원한다.
- 자바처럼 **정적 타입**의 언어이다. 즉, 모든 표현식의 타입을 컴파일 시점에서 알 수 있다는 의미이다. 따라서 프로그램에서 사용하는 모든 변수나 객체를 컴파일러가 미리 검증할 수 있으므로 실행 시점의 오류 발생을 막아준다. 그리고 **타입 추론**(type inference) 기능이 있어서 변수를 선언하고 사용할 때 변수의 타입을 명시적으로 지정하지 않아도 처리해준다. 예를 들어, val a = 100과 같이 변수를 선언하면 컴파일러가 a 변수를 정수 타입(코틀린에서는 Int로 나타냄)으로 처리한다. 초기화 값이 정수인 100이기 때문이다.
- 자바와 동일하게 객체지향 프로그래밍을 지원한다. 이와 더불어 기존 클래스로부터 상속받지 않고도 해당 클래스의 기능을 추가할 수 있는 확장(extension) 함수와 확장 속성을 사용할 수 있다.
- **함수형 프로그래밍**을 지원한다. 클래스와는 별도로 함수를 선언하고 사용할 수 있으며, 다양한 형태의 함수와 람다식(Lambdas expression)을 지원한다. 예를 들어, 지역 함수(함수 내부에 다른 함수를 선언하고 사용), 멤버 함수(클래스 내부에 함수를 선언하고 사용하며 자바의 메서드와 같은 개념), 제네릭 함수(함수에 제네릭 타입 사용), 다른 함수를 인자로 받아 실행시킬 수 있는 상위차(higher-order) 함수, 이름이 없는 익명 함수, 성능 향상을 위한 인라인(inline) 함수, 재귀 호출을 최적화하는 재귀(tail recursive) 함수, 기존 클래스에 기능을 추가하는 확장(extension) 함수 등이다.
- 문법과 코드가 **간결하여** 작성 부담이 훨씬 줄어든다. 예를 들어, 클래스의 게터(getter)와 세터(setter) 등이 자동 생성된다. 주석(comment)은 한 줄 주석(//로 시작)과 여러 줄 주석(/*..*/) 모두를 사용할 수 있으며, 자바와는 다르게 여러 줄 주석의 경우에 중첩이 가능하다. 또한 각 명령문 끝에 세미콜론(;)을 붙이지 않아도 된다.
- Null 값으로 인해 발생할 수 있는 NullPointerException 예외가 생기지 않도록 **언어 자체에서 배려**하고 있으므로 자바보다 안전하다. 예를 들어, null 값을 가질 수 있는 변수는 변수 타입 끝에 물음표(?)를 붙여서 간단하게 null 가능(nullable) 타입으로 지정하며(예를 들어, val s: String? = null), 이 경우 null 값인지 확인하는 코드가 없으면 컴파일 에러로 처리된다. 그리고 이런 모든 것을 간단한 코드로 처리할 수 있게 언어의 구성 요소를 지원한다.
- OSS이므로 코틀린(컴파일러, 라이브러리, 각종 도구를 포함)을 Apache 2 라이선스에 의거 무상으로 사용할 수 있으며, 코틀린 애플리케이션을 개발하는데 유용한 OSS IDE도 같이 사용

할 수 있다. 대표적인 IDE로는 안드로이드 스튜디오, IntelliJ IDEA, 이클립스(Eclipse) 등이 있으며, 코틀린 플러그인을 추가로 설치하면 코틀린 언어를 사용하여 애플리케이션을 개발할 수 있다. 특히 **안드로이드 스튜디오**의 경우는 3.0 버전부터 코틀린 플러그인을 포함하며, 코틀린으로 안드로이드 애플리케이션을 쉽게 개발할 수 있는 여러 가지 기능이 추가되어 있다.

- 또한 자바 코드를 코틀린 코드로 변환해주는 **변환기**를 방금 얘기한 세 가지 IDE에서 모두 지원하므로 기존의 자바 코드를 코틀린 코드로 쉽게 변환할 수 있으며, 코틀린 언어를 파악하는데도 도움이 된다.

참고로, 코틀린 언어의 공식 홈페이지는 *https://kotlinlang.org/*이며, 이곳에서 코틀린의 다양한 정보를 얻을 수 있다.

A.2 자바 JDK 설치하기

코틀린 애플리케이션은 JVM에서 실행되므로 개발 시에 JDK(Java Development Kit)가 설치되어 있어야 한다. 그리고 JDK 6이상부터 사용 가능하지만, 새로운 기능(예를 들어, 람다식)을 사용하려면 JDK 8 이상을 설치해야 한다. 따라서 가장 최신 버전을 설치하는 것이 좋다. JDK가 이미 설치되어있는 경우는 "A.3 코틀린 컴파일러 설치하기"로 건너뛰기 바란다.

A.2.1 윈도우에서 설치하기

다음의 오라클(Oracle) 사이트에서 최신 윈도우용 JDK를 다운로드하자.

URL *http://www.oracle.com/technetwork/java/javase/downloads/index.html*

이 웹 페이지의 맨 앞에 나오는 것이 최신 버전이다. JDK DOWNLOAD 버튼을 클릭하면 다운로드 페이지가 나올 것이다. 거기에서 중간에 있는 Accept License Agreement를 클릭한 후, Windows x86(32비트) 또는 Windows x64(64비트)용 exe 파일을 클릭하여 다운로드하면 된다. 그리고 다운로드된 exe를 실행시켜 설치하자(설치할 위치만 지정하면 되므로 간단하다).

설치가 끝나면 다음과 같은 절차로 Path와 JAVA_HOME 두 개의 환경 변수를 지정한다. 윈도우 7에서는 다음과 같이 한다.

1. 시작 메뉴의 컴퓨터에서 오른쪽 마우스 버튼을 클릭한 후 속성을 선택한다.
2. 시스템 창에서 **고급 시스템 설정** 클릭 ➡ **고급** 탭 선택 ➡ **환경 변수** 버튼을 누른다.

3. 환경 변수 대화상자에서 시스템 변수의 **새로 만들기** 버튼을 누른다. 변수 이름은 JAVA_HOME으로 입력하고(모두 대문자임에 유의), 변수 값은 JDK가 설치된 디렉터리를 입력한 후 **확인** 버튼을 누른다.

   예를 들어, JDK가 C:\Java\jdk1.8.0_131에 설치되었다면 다음을 입력한다.

   ```
   C:\Java\jdk1.8.0_131
   ```

   만일 JAVA_HOME 변수가 이미 있다면 **편집** 버튼을 누르고 변수 값만 변경한다.

4. 그다음에는 시스템 변수의 Path 변수를 찾고 **편집** 버튼을 누른다. 문자열의 맨 끝에 다음을 추가한다.

   ```
   ;%JAVA_HOME%\bin
   ```

   이때 문자열의 맨 끝에 세미콜론(;)이 없다면 이처럼 제일 앞에 세미콜론(;)을 붙여야 한다.

5. 각 대화상자에서 **확인** 버튼을 눌러 속성 창을 닫는다.

윈도우 8.1에서는 다음과 같이 환경 변수를 지정한다.

1. 시작 화면에서 화면의 오른쪽 아래 모서리로 마우스 커서를 이동한 후, 메뉴의 **검색**(Search)을 선택하고 **제어판**(Control Panel)을 입력한다. 검색 결과에서 제어판 아이콘이 나타나면 클릭한다.
2. 제어판 창이 열리면 오른쪽 위의 범주(Category)를 **큰 아이콘**(Large Icons)으로 선택한다. 그리고 아이콘 목록에서 **시스템**(System)을 선택한다.
3. 앞의 윈도우 7에서 설명한 2번부터 5번까지를 똑같이 수행한다.

윈도우 10에서는 다음과 같이 환경 변수를 지정한다.

1. 바탕 화면에서 시작 버튼을 클릭하고 메뉴의 **설정**을 선택한다.
2. 시스템 버튼을 클릭하고 왼쪽 제일 밑의 **정보**를 클릭한다.
3. 앞의 윈도우 7에서 설명한 2번부터 5번까지를 똑같이 수행한다.

A.2.2 맥 OS X에서 설치하기

맥 OS X 최신 버전에는 기본적으로 자바가 설치되어 있지 않다. 자바의 설치 유무를 확인하기 위해 터미널 창(terminal window)을 열고 다음 명령을 실행해보자.

```
java -version
```

혹시 이전에 자바를 설치한 적이 있다면 자바 버전을 보여주는 내용이 터미널 창에 나타날 것이다. 그러나 자바가 설치되지 않은 경우는 오라클 자바 웹 페이지를 보여주는 More Info 버튼이 있는 대화상자와 함께 다음 메시지가 나타난다.

```
No Java runtime present, requesting install
```

앞에서 얘기한 오라클 자바 웹 페이지에 접속하여 맥 OS X용 자바 SE 8을 다운로드하자. 그리고 다운로드된 디스크 이미지(.dmg 파일)를 열고 아이콘을 더블 클릭하여 자바 패키지를 설치한다.

Java for OS X 인스톨러 창이 나타나고 JDK 설치가 진행될 것이다. 그리고 설치가 완료되면 터미널 창에서 다음 명령을 실행하자. 앞에서처럼 자바 버전 정보가 터미널 창에 나올 것이다.

```
java -version
```

A.3 코틀린 컴파일러 설치하기

코틀린 언어를 처음 배울 때는 컴퓨터의 운영체제에서 독자적으로 실행되는 네이티브 애플리케이션을 작성하고 테스트하는 것이 좋다. 그리고 그렇게 하려면 코틀린 컴파일러를 설치해야 한다. *https://github.com/JetBrains/kotlin/releases* 페이지를 접속한 후 아래쪽의 Downloads에 있는 ZIP 파일을 다운로드 받자.

그다음에 원하는 디렉터리에 압축을 풀면 코틀린 컴파일러의 설치가 끝난다. 예를 들어, 윈도우 시스템에서 C:\Kotlin에 설치했다면 이 디렉터리 밑의 kotlinc\bin에 코틀린 컴파일러인 kotlinc가 있다. 앞에서 JDK를 설치할 때와 동일한 방법으로 시스템의 path 환경 변수에 코틀린 컴파일러의 경로를 추가하자.

그리고 명령행(윈도우의 명령 프롬프트 창, 리눅스나 OS/X에서는 터미널 창)에서 다음을 입력하여 코틀린 컴파일러가 잘 실행되는지 확인한다. 제대로 실행되면 설치된 컴파일러의 버전이 출력된다.

```
kotlinc -version
```

A.4 코틀린으로 네이티브 애플리케이션 작성과 실행하기

코틀린을 사용해서 네이티브 애플리케이션을 생성할 때는 다음 두 가지 방법으로 할 수 있다. 첫 번째는, 텍스트 편집기를 사용하여 소스 코드를 작성한 후 명령행에서 실행하는 방법이고, 두 번째는, IntelliJ IDEA나 이클립스와 같은 IDE를 사용하는 방법이다(안드로이드 스튜디오는 안드로이드 애플리케이션을 개발할 때만 사용한다). 당연히 두 번째 방법을 사용하겠지만, 일단 명령행에서 하는 방법을 알아보자.

A.4.1 명령행에서 애플리케이션 작성과 실행하기

각자 사용하는 텍스트 편집기를 사용하여 다음의 코틀린 소스 코드를 작성한 후 hello.kt라는 이름의 파일로 저장한다. (코틀린 소스 코드 파일의 확장자는 .kt이어야 한다)

```
fun main(args: Array<String>) {
    printHello(1)
}
fun printHello(msgType: Int) {
    when (msgType) {
        1, 2 -> println("안녕하세요?")
        else -> println("메시지 번호 오류")
    }
}
```

자바와 마찬가지로 코틀린에서도 네이티브 애플리케이션은 main() 함수부터 시작한다. 그리고 함수는 fun 키워드로 선언한다. 여기서는 printHello() 함수를 추가로 지정하였으며, 이 함수는 정수 타입(Int)의 인자를 하나 받는다. 그리고 그 인자의 값이 1이나 2일 때는 "안녕하세요?"를 출력하고 그 외에는 "메시지 번호 오류"를 출력한다. (코틀린의 when은 자바의 switch~case와 유사하지만 훨씬 간결하고 기능도 다양하다.)

작성이 다 되었으면 명령행에서 다음과 같이 컴파일을 한다.

```
kotlinc hello.kt -include-runtime -d hello.jar
```

여기서 -include-runtime은 코틀린의 런타임 라이브러리를 우리 애플리케이션에 포함시키라는 의미이다. 그렇게 해야만 독자적인 애플리케이션으로 실행될 수 있기 때문이다. 그러나 만일 다른 코드에서 라이브러리로 사용하기 위해 컴파일할 때는 -include-runtime을 지정하지 않으면 된다.

-d 다음에는 클래스 파일을 생성할 디렉터리 위치나 jar 파일을 지정한다. 여기서는 jar 파일로 생성하며, 이 파일에는 우리의 클래스 파일과 코틀린 런타임 라이브러리가 모두 포함된다. 코틀린 컴파일러의 옵션은 kotlinc -help를 실행하면 모두 볼 수 있다.

컴파일이 에러 없이 수행된 후에는 다음과 같이 실행할 수 있다(자바 JDK도 제대로 설치되어 있어야 한다).

```
java -jar hello.jar
```

출력 결과는 다음과 같다.

```
안녕하세요?
```

컴파일의 결과로 생성된 hello.jar 파일의 내용을 압축 프로그램(알집이나 반디집 등)으로 보면 HelloKt.class 파일이 생성되어 있음을 알 수 있다. 코틀린 컴파일러는 이처럼 소스 파일 이름의 첫 자를 대문자로 바꾸고 그 뒤에 Kt를 붙인 .class 파일을 생성한다. 다음과 같이 javap를 실행해보자.

```
javap -classpath hello.jar HelloKt
```

아래의 결과가 출력된다.

```
Compiled from “hello.kt”
public final class HelloKt {
  public static final void main(java.lang.String[]);
  public static final void printHello(int);
}
```

이 내용을 보면 알 수 있듯이, 코틀린 컴파일러는 .kt 파일을 하나의 .class 파일로 생성한다. 또한 각 함수는 기본적으로 자바의 public static final, 즉 어디서든 호출 가능하고(public), 클래스 인스턴스를 생성하지 않고 클래스 이름으로 호출할 수 있으며(static), 상속 받는 서브 클래스에서 오버라이딩 할 수 없는(final) 형태로 정의된다는 것을 알 수 있다. 클래스도 기본적으로 서브 클래스로 상속할 수 없게 되어있다(상속 가능하게 하려면 class 키워드 앞에 open 을 지정해야 한다).

코틀린 컴파일러는 쉘(shell) 형태의 대화식 모드로 실행할 수도 있다. 이때는 kotlinc 대신 kotlinc-jvm을 실행하면 >>> 프롬프트가 나타난다. 이것을 코틀린 REPL(Read-Eval-Print Loop)이라고 한다. REPL은 사용자로부터 명령어와 표현식 등을 입력 받아서 바로 실행한 후 그 결과를 출력해서 보여주는 대화식 형태의 컴퓨터 프로그래밍 환경을 말한다.

```
Welcome to Kotlin version 1.1.2-5 (JRE 1.8.0_131-b11)
Type :help for help, :quit for quit
>>>
```

>>> 다음에 println("안녕하세요?")를 입력하고 Enter 키를 누르면 곧바로 "안녕하세요?"가 출력된다. 이것은 코틀린 명령어를 바로 실행해 볼 때 편리하게 사용할 수 있다. 그리고 :quit을 입력하면 코틀린 REPL이 종료된다.

A.4.2 이클립스에서 애플리케이션 작성과 실행하기

지금까지는 명령행에서 네이티브 애플리케이션을 컴파일하고 실행하는 방법을 알아보았다. 그러나 IntelliJ IDEA나 이클립스와 같은 IDE를 사용하는 것이 여러모로 편리하다. 우선, 이클립스의 설치와 코틀린 프로젝트 생성 방법부터 알아보자. *https://www.eclipse.org/downloads/*를 접속한 후 아래쪽의 DOWNLOAD 버튼을 눌러서 이클립스를 다운로드 한다. 다운로드된 파일을 실행하면 다음 화면이 나타난다.

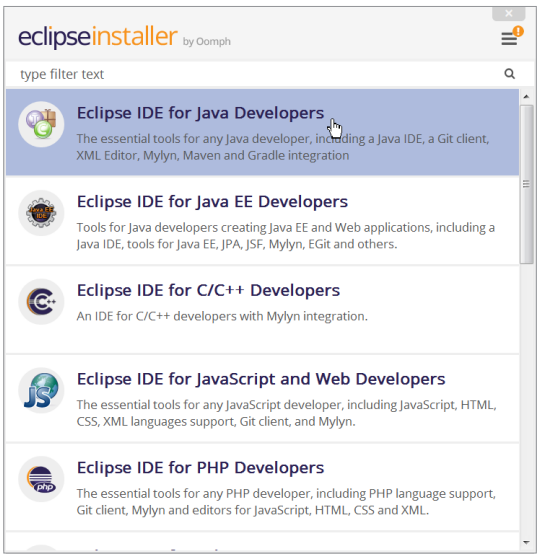


그림 A-1

이클립스는 여러 가지 용도로 사용할 수 있다. 코틀린을 사용할 때는 일단 이클립스를 설치한 후에 코틀린 플러그인으로 변경하면 되므로 어느 것을 선택해도 관계 없지만, 일단 "Eclipse IDE for java Developers"를 클릭한다. 그러면 설치할 위치를 선택하는 화면이 나타난다.

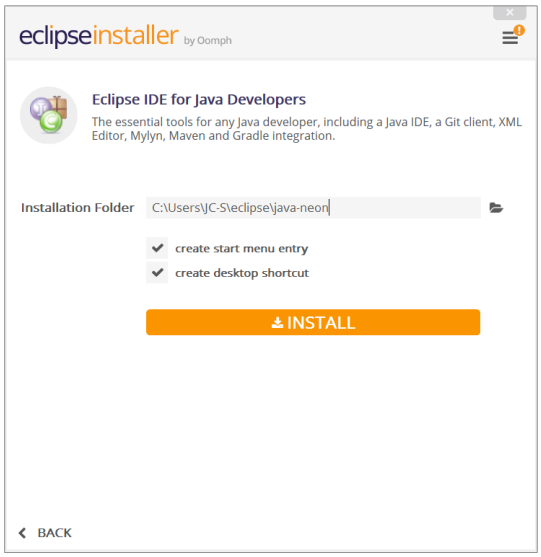


그림 A-2

기본 값을 그대로 두고 INSTALL 버튼을 클릭한 후 다음 대화상자에서 Accept Now 버튼을 클릭하면 설치가 시작된다. 그리고 설치가 끝나면 나타나는 대화상자에서 LAUNCH 버튼을 클릭하면 이클립스가 시작되면서 작업 디렉터리를 선택하는 대화상자가 나타난다.

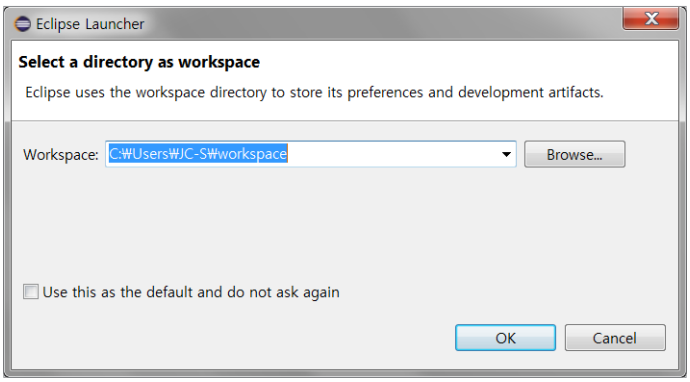


그림 A-3

각자 원하는 디렉터리를 지정한 후 OK 버튼을 클릭하면 이클립스 메인 화면이 나타난다. 이제는 코틀린 플러그인을 설치해야 한다. 그림 A-4처럼 Help ➡ Eclipse Marketplace...를 선택한다.

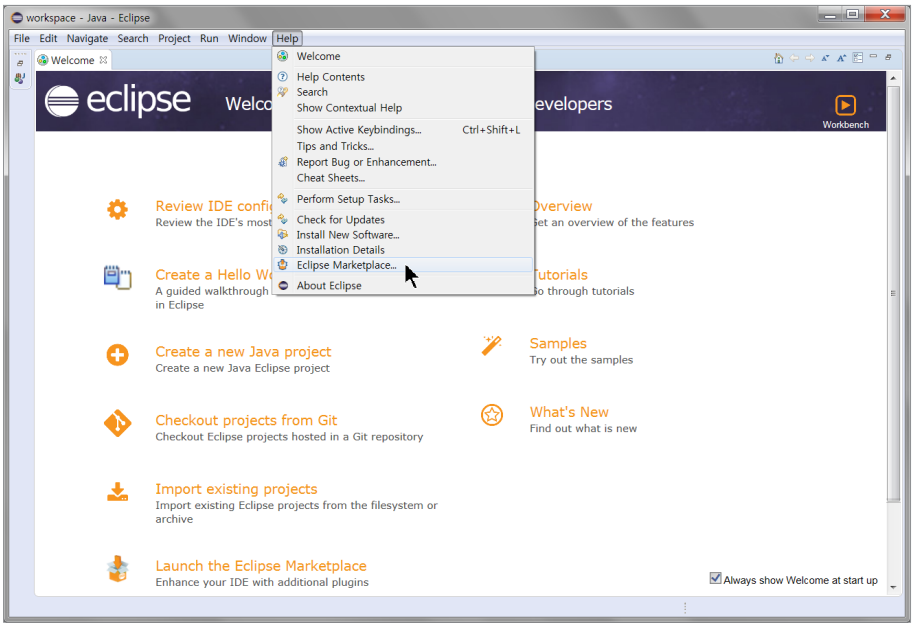


그림 A-4

그리고 그림 A-5처럼 검색 필드에 Kotlin을 입력하고 코틀린 플러그인이 나타나면 Install 버튼을 클릭한다.

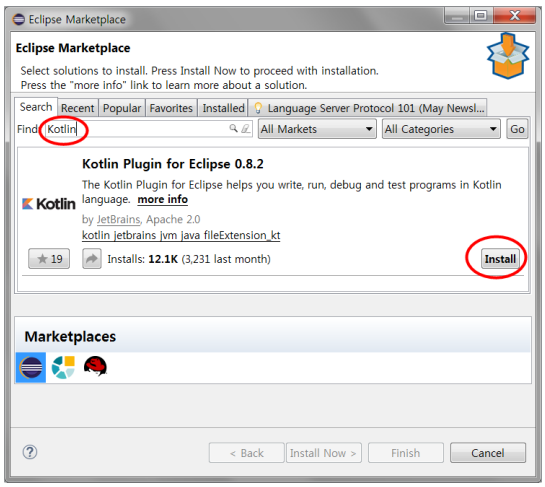


그림 A-5

그다음에 그림 A-6처럼 "I accept the terms of the license agreement"를 선택하고 Finish 버튼을 클릭하면 플러그인이 설치되고 이클립스를 다시 시작해야 한다는 대화상자가 나타난다. Yes 버튼을 눌러서 이클립스를 다시 시작시킨다.

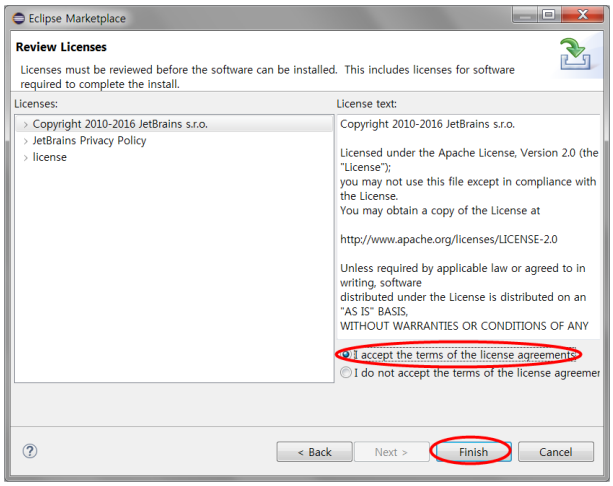


그림 A-6

이제는 코틀린 플러그인이 설치되었으므로 이클립스에서 코틀린을 사용할 수 있다. 그렇게 하려면 퍼스펙티브를 전환해야 한다. 그림 A-7처럼 Window ➡ Perspective ➡ Open Perspective ➡ Other...를 선택한다.

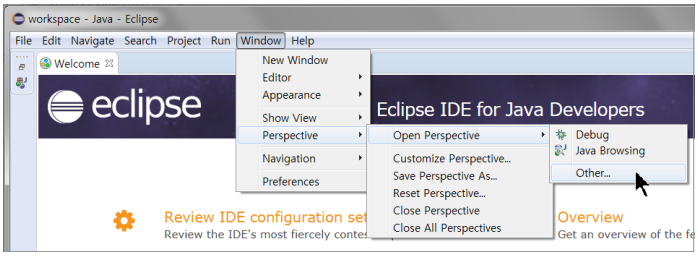


그림 A-7

그리고 퍼스펙티브 선택 대화상자에서 Kotlin을 선택한 후 OK 버튼을 클릭한다. 여기까지 되었으면 코틀린을 사용할 수 있는 환경이 준비된 것이다. 이제부터는 코틀린 프로젝트를 생성할 것이다. 그림 A-8처럼 File ➡ New ➡ Kotlin Project를 선택한다.

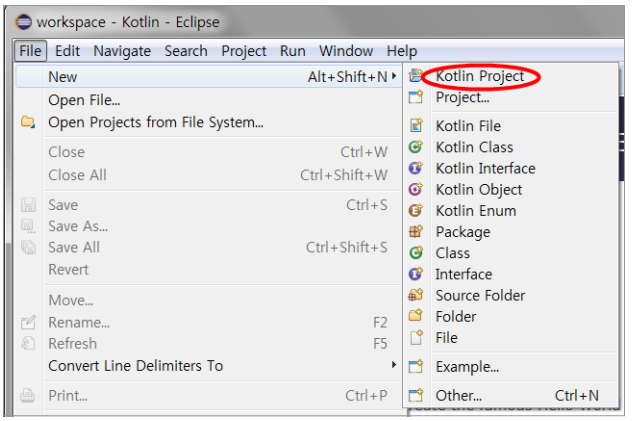


그림 A-8

그리고 그림 A-9의 프로젝트 지정 대화상자에서 각자 원하는 프로젝트 이름을 입력하고(여기서는 kotlin) Finish 버튼을 클릭한다.

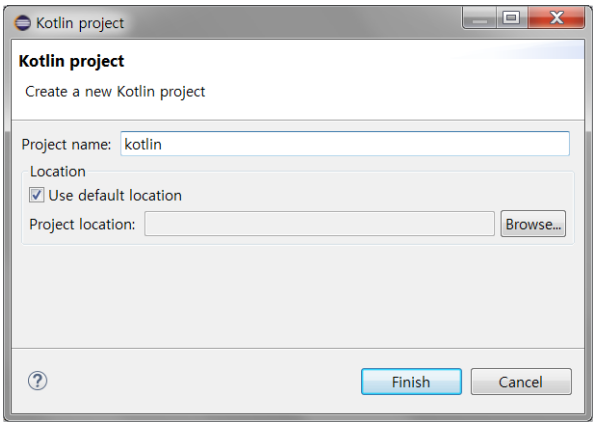


그림 A-9

이제는 코틀린 프로젝트가 생성되었으므로, 코틀린 소스 코드 파일을 생성해야 한다. 그림 A-10처럼 패키지 탐색기 창의 프로젝트 밑에 있는 src 폴더에서 오른쪽 마우스 버튼을 클릭한 후 New ➡ Kotlin File을 선택하자.

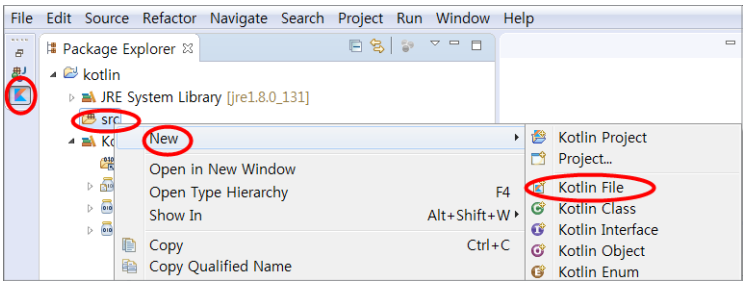


그림 A-10

그림 A-11의 대화상자가 나타나면 파일 이름에 hello를 입력하고 Finish 버튼을 클릭한다. 소스 폴더는 기본값을 그대로 둔다.

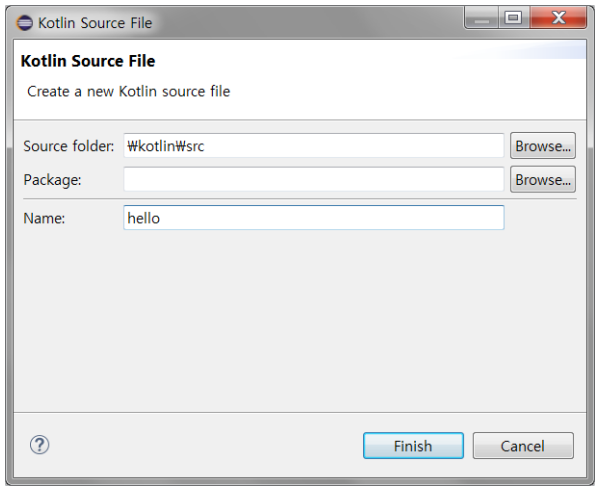


그림 A-11

그러면 hello.kt 파일이 자동으로 생성되고 편집기 창에 나타난다. 그림 A-12와 같이 편집기 창에서 hello.kt의 코드를 입력하자.

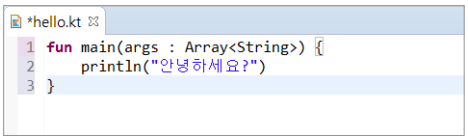


그림 A-12

코드 작성이 끝났으므로 이제는 컴파일과 실행을 하면 된다. hello.kt의 코드 화면에서 오른쪽 마우스 버튼을 누른 후 그림 A-13처럼 Run As ➡ Kotlin Application을 선택한다.

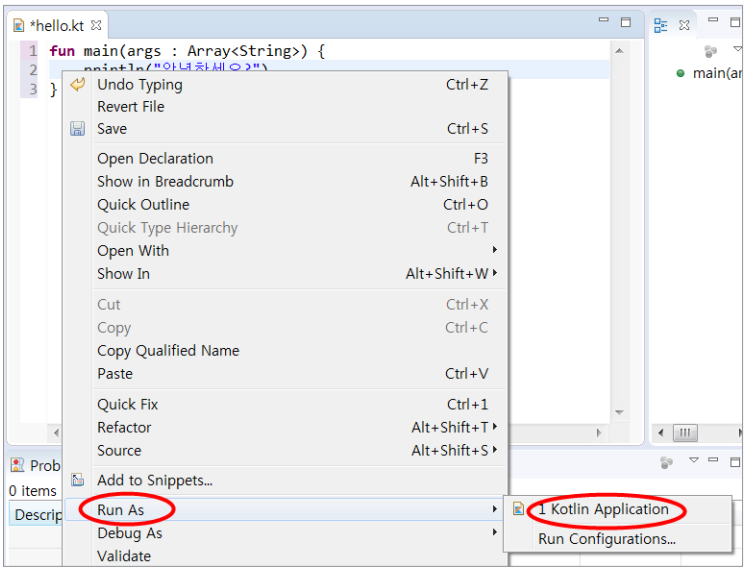


그림 A-13

그러면 컴파일과 실행이 끝나고 그림 A-14처럼 콘솔 창에 "안녕하세요?"가 실행 결과로 나타날 것이다.

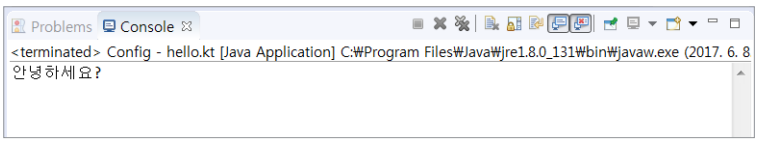


그림 A-14

이후로 새로운 프로젝트를 생성할 때는 그림 A-8의 코틀린 프로젝트 생성부터 시작하면 된다 (여기서는 이클립스 IDE 자체의 자세한 사용 방법은 설명하지 않는다).

A.4.3 IntelliJ IDEA에서 애플리케이션 작성과 실행하기

다음은 IntelliJ IDEA를 설치하고 코틀린 프로젝트를 생성하는 방법을 알아보자. 우선, *https://www.jetbrains.com/idea/download/index.html*을 접속한 후 Community 버전의 DOWNLOAD 버튼을 눌러서 다운로드 한다. 그리고 다운로드된 파일을 실행하면 다음 대화상자가 나타난다.

그림 A-15

Next 버튼을 클릭하면 그림 A-16의 설치 위치 지정 대화상자가 나타난다.

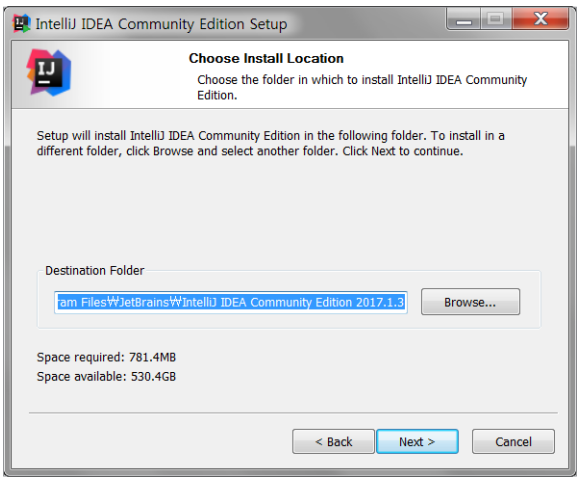


그림 A-16

각자 원하는 위치를 지정하고 Next 버튼을 클릭하면 설치 옵션을 선택할 수 있는 그림 A-17의 대화상자가 나타난다.

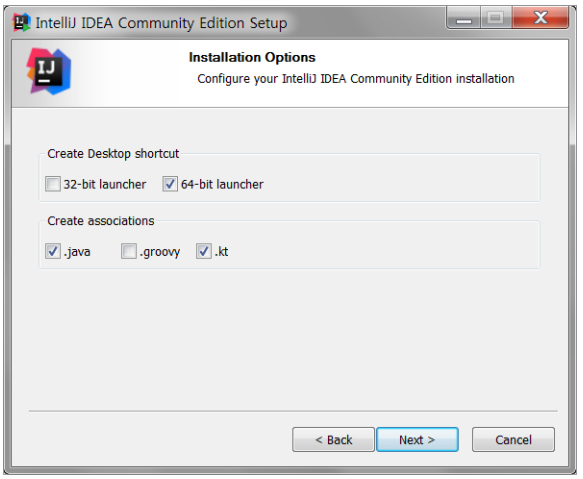


그림 A-17

실행할 버전(32비트 또는 64 비트)을 선택하고 Next 버튼을 클릭한다.

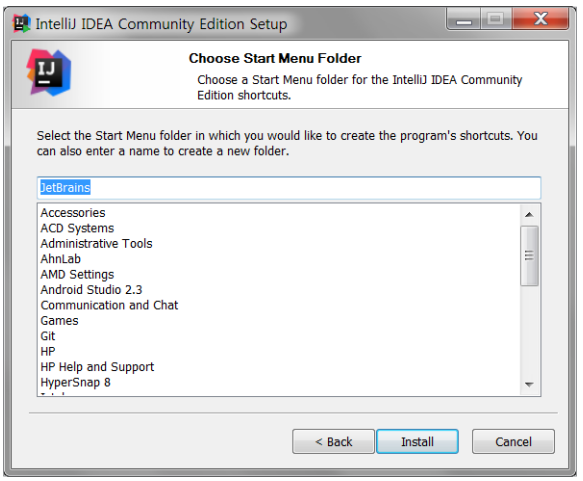


그림 A-18

끝으로 Install 버튼을 클릭하면 설치가 시작되며, 완료된 후에는 그림 A-19의 대화상자가 나타난다.

그림 A-19

"Run IntelliJ IDEA Community Edition"을 선택한 후 Finish 버튼을 누르면 IntelliJ IDEA가 최초 실행되며, 만일 이전에 IntelliJ IDEA Community Edition을 설치한 적이 있으면 그것의 설정을 가져올 수 있도록 그림 A-20의 대화상자가 나타난다.

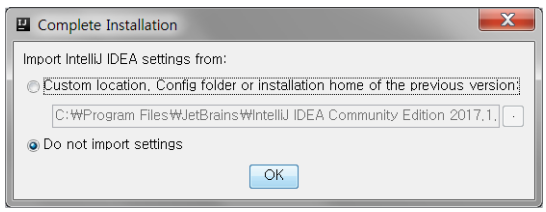


그림 A-20

처음 설치이거나 또는 이전의 설정을 가져올 필요가 없으면 "Do not import settings"를 선택하고 OK 버튼을 클릭한다. 그리고 다음 대화상자에서 Accept 버튼을 클릭하면 IntelliJ IDEA의 UI를 선택하는 그림 A-21의 대화상자가 나타난다.

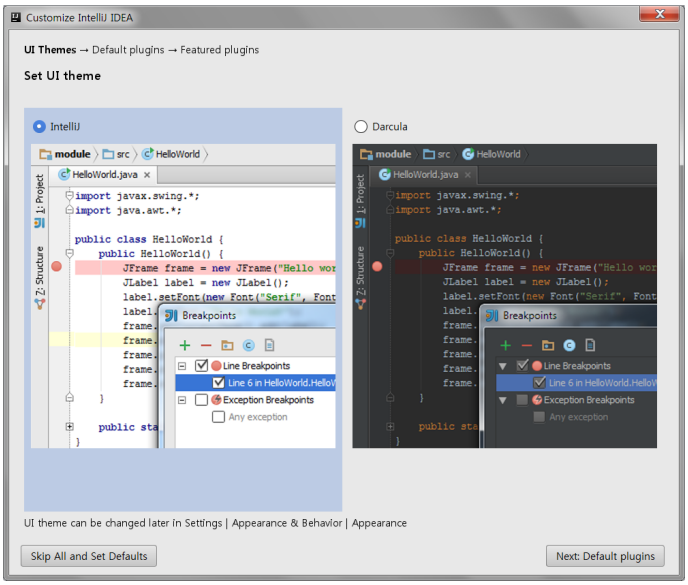


그림 A-21

각자 선호하는 형태를 선택하고 Next: Default plugins 버튼을 클릭하면 그림 A-22의 대화상자가 나타난다. 여기서는 각종 도구에 사용할 플러그인을 활성화 또는 비활성화할 수 있다.

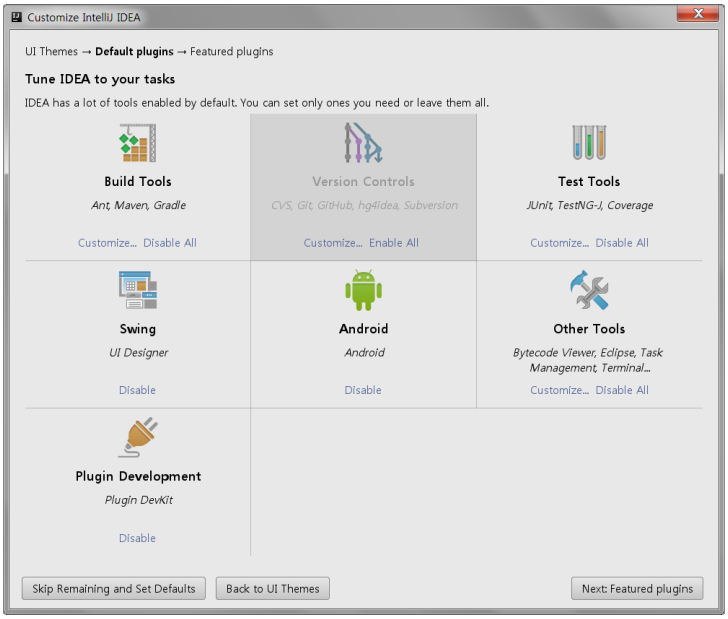


그림 A-22

기본 선택을 그대로 두고 Skip Remaining and Set Defaults 버튼을 클릭하면 모든 설정이 끝나고 그림 A-23의 IntelliJ IDEA 시작 대화상자가 나타난다.

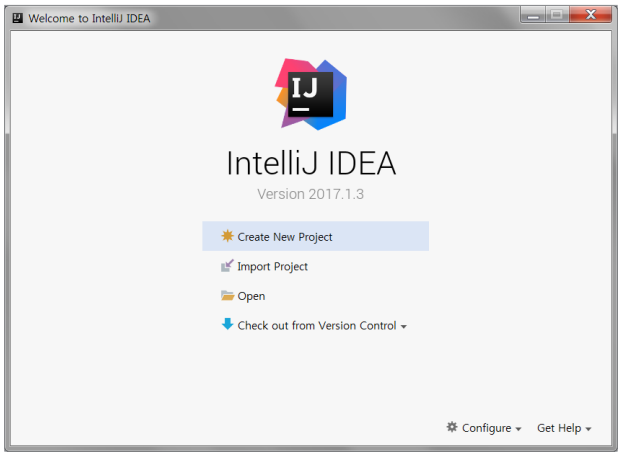


그림 A-23

여기까지 되었으면 코틀린을 사용할 수 있는 환경이 준비된 것이다. 이제부터는 코틀린 프로젝트를 생성할 것이다. 그림 A-23에서 Create New Project를 선택하면 그림 A-24의 대화상자가 나타난다.

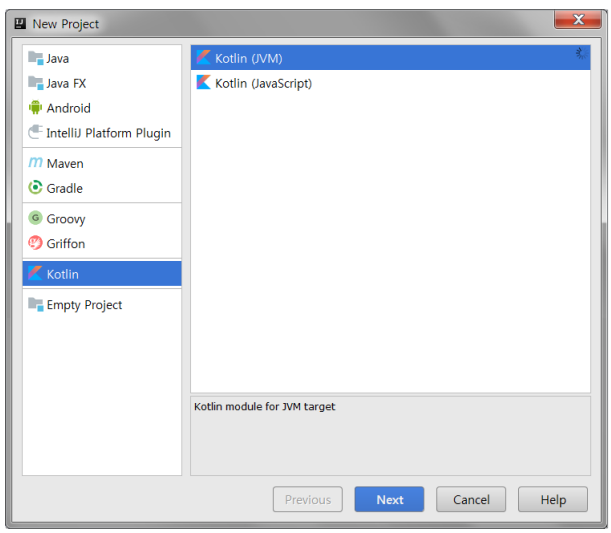


그림 A-24

여기서는 코틀린 프로젝트를 생성할 것이므로, 왼쪽 패널에서 Kotlin을 선택하고 오른쪽 패널에서는 Kotlin(JVM)을 선택한 후 Next 버튼을 클릭하면 그림 A-25의 대화상자가 나타난다.

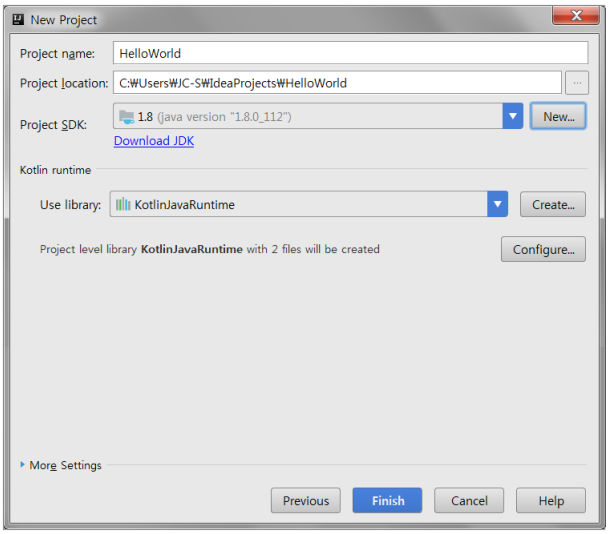


그림 A-25

프로젝트 이름에 HelloWord를 입력하고 위치는 각자 지정한 후 Finish 버튼을 클릭하면 새로운 코틀린 프로젝트가 생성되고 왼쪽의 프로젝트 도구 창에 나타난다. 코틀린 프로젝트가 생성되었으므로 이제는 코틀린 소스 코드 파일을 생성하면 된다. 그림 A-26처럼 프로젝트 밑의 src 폴더에서 오른쪽 마우스 버튼을 클릭한 후 New ➡ Kotlin File/Class를 선택한다.

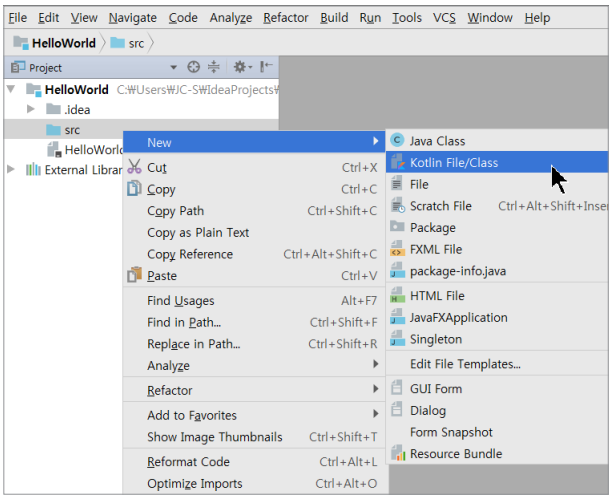


그림 A-26

그리고 그림 A-27의 대화상자가 나타나면 파일 이름에 app을 입력하고(각자 원하는 이름을 입력해도 된다) OK 버튼을 클릭한다.

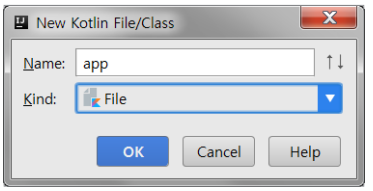


그림 A-27

그러면 app.kt 파일이 자동 생성되고 중앙의 편집기 창에 열릴 것이다. 이때 빈 화면의 아무 곳이나 클릭한 후 main을 입력하고 Tab 키를 누르면 main() 함수 코드를 자동으로 생성해준다. 그림 A-28과 같이 println("안녕하세요?")를 입력하자.

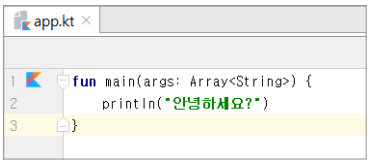


그림 A-28

코드 작성이 끝났으므로 이제는 컴파일과 실행을 하면 된다. 그림 A-29에 타원으로 표시된 코틀린 아이콘을 클릭한 후 Run 'AppKt'를 선택한다.

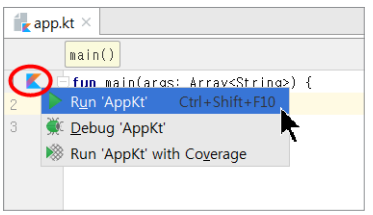


그림 A-29

그러면 그림 A-30처럼 Run 도구 창에 "안녕하세요?"가 실행 결과로 나타날 것이다.

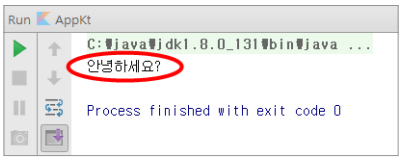


그림 A-30

이후로 새로운 프로젝트를 생성할 때는 그림 A-24의 코틀린 프로젝트 생성부터 시작하면 된다.

IntelliJ IDEA는 안드로이드 스튜디오와 사용 방법이 거의 유사하다. 안드로이드 스튜디오가 IntelliJ IDEA를 기반으로 개발되었기 때문이다. 그러나 안드로이드 애플리케이션을 개발할 때는 안드로이드 스튜디오를 사용하는 것이 좋다. 안드로이드에 특화된 여러 가지 다른 기능을 추가로 제공하기 때문이다.

코틀린 언어를 배우기 위해 사용할 IDE는 각자의 취향이나 사용 경험에 따라 결정하면 될 것이다. 이와는 별도로, 코틀린 공식 사이트에서는 온라인으로 연결된 상태에서 코틀린 언어를 실습할 수 있는 기능을 제공한다. https://kotlinlang.org/를 접속한 후 위의 메뉴에서 TRY ONLINE을 클릭하면 그림 A-31의 화면이 나타난다.

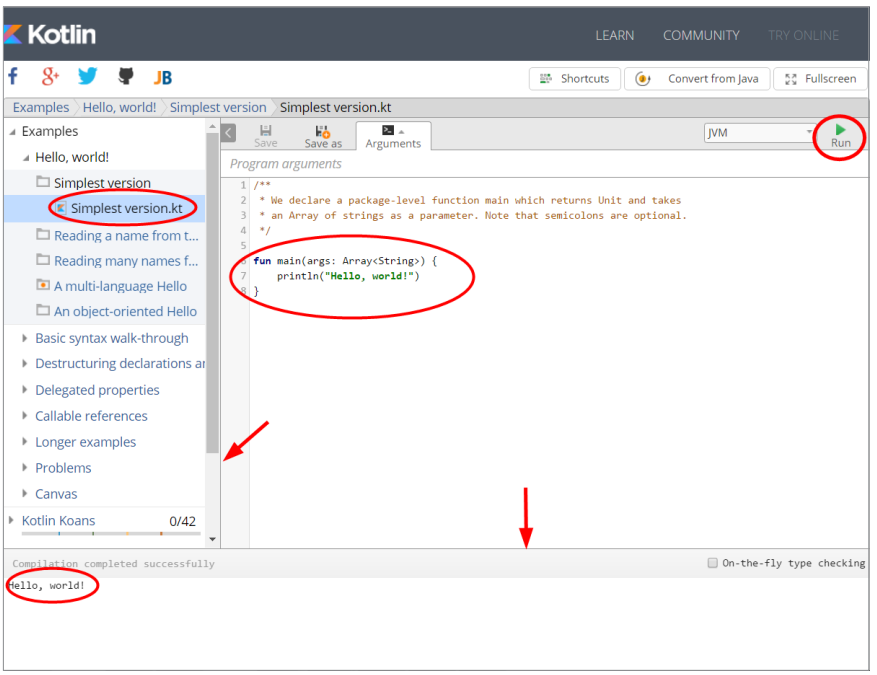
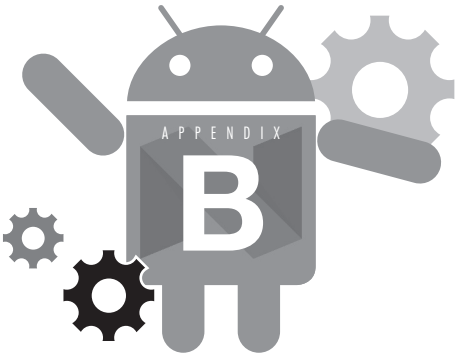


그림 A-31

왼쪽 패널에는 코틀린의 명령어와 기능들이 요약되어 있으며, 각 항목 왼쪽의 작은 삼각형을 클릭하면 상세 항목을 확장하거나 축소하여 볼 수 있다. 그리고 그 중 하나를 클릭하면 샘플 소스 코드를 오른쪽의 편집기 패널에 보여준다. 여기에 나타난 코드는 우리가 원하는 대로 수정할 수 있다. 또한 오른쪽 위의 Run 버튼을 클릭하면 곧바로 컴파일되어 실행되며, 출력된 결과는 아래쪽 패널에 나타난다. 또한 각 패널의 경계선을 마우스로 끌면 패널 크기를 조정할 수 있다.

A.5 요약

이번 장에서는 코틀린 언어의 개요와 네이티브 애플리케이션을 개발하면서 코틀린을 배우는 데 필요한 방법을 알아보았다. 다음 장에서는 예제 코드와 더불어 코틀린 언어의 구성 요소와 기본 문법을 자세히 알아볼 것이다.



코틀린 핵심 파악하기: 구성 요소와 문법

이번 장에서는 코틀린 언어를 빠른 시간 내에 익힐 수 있도록 예제 코드와 더불어 핵심 구성 요소와 문법을 알아본다. 다른 프로그래밍 언어를 사용해본 경험이 있다면 더 쉽게 파악할 수 있을 것이다. 자바를 알고 있다면 더욱 좋다. 그리고 예제 코드의 작성과 테스트는 앞 장에서 알아본 방법 중 어느 것을 사용해도 되지만, 여기서는 IntelliJ IDEA를 사용할 것이다. 이번 장의 목표는 코틀린 언어의 핵심을 빠른 시간 내에 파악할 수 있게 하는 것이다.

B.1 코틀린 프로그램 구조

코틀린 프로그램의 기본적인 구성 요소를 이해하기 쉽도록 하기 위해 우선 간단한 예제 코드 작성부터 시작해보자.

IntelliJ IDEA를 실행했을 때 나타나는 웰컴 스크린에서 Create New Project를 선택하거나 또는 IntelliJ IDEA의 메인 메뉴에서 File ➡ New ➡ Project...를 클릭하면 프로젝트 종류를 선택하는 대화상자가 나타난다(부록A의 그림 A-24참조). 그리고 왼쪽 패널에서 Kotlin을 선택하고 오른쪽 패널에서는 Kotlin(JVM)을 선택한 후 Next 버튼을 클릭하면 프로젝트 정보 입력 대화상자가 나타난다(그림 A-25참조).

프로젝트 이름에 Kotlin01을 입력하고 위치는 각자 지정한 후 Finish 버튼을 클릭하면 새로운 코틀린 프로젝트가 생성되어 왼쪽의 프로젝트 도구 창에 나타난다. 이제는 코틀린 프로젝트

가 생성되었으므로 코틀린 소스 코드 파일을 생성하면 된다. Kotlin01 프로젝트 밑의 src 폴더에서 오른쪽 마우스 버튼을 클릭한 후 New ➡ Kotlin File/Class를 선택한다. 그리고 대화상자에서 파일 이름에 Hello를 입력하고(각자 원하는 이름을 입력해도 된다) OK 버튼을 클릭한다.

그러면 Hello.kt 파일이 자동 생성되고 중앙의 편집기 창에 열릴 것이다. 이때 빈 화면의 아무 곳이나 클릭한 후 main을 입력하고 Tab 키를 누르면 main( ) 함수 코드를 자동으로 생성해준다. 아래와 같이 코드를 입력하자.

```
fun main(args: Array<String>) {
    println(makeMessage1(1))
    println(makeMessage1(2))
    println(makeMessage2(1))
    println(makeMessage2(2))

}
fun makeMessage1(msgType: Int) : String {
    return if (msgType == 1) "안녕하세요?" else "또 만났군요!!"
}
fun makeMessage2(msgType: Int) = if (msgType == 1) "날씨 좋죠?" else "참 맑군요!!"
```

코드 작성이 끝났으면 main( ) 함수 왼쪽의 코틀린 아이콘()을 클릭한 후 Run 'HelloKt'를 선택하여 코드를 컴파일하고 실행한다. 그러면 그림 B-1처럼 Run 도구 창에 "안녕하세요?", "또 만났군요!!", "날씨 좋죠?", "참 맑군요!!"가 차례대로 나타날 것이다.

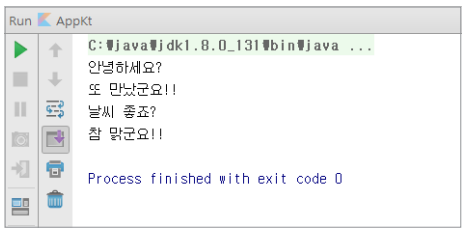


그림 B-1

(이후의 예제 코드를 실습할 때는 지금과 동일한 방법으로 새로운 코틀린 파일을 생성하고 코드를 작성하면 된다.)

앞의 코드를 자세히 살펴보자. 컴퓨터에서 독자적으로 실행되는 코틀린 네이티브 애플리케이션의 경우는 main( ) 함수부터 실행이 시작되며, 이 함수는 명령행에서 전달되는 인자를 배열로 받을 수 있다(코틀린 안드로이드 애플리케이션은 액티비티로 실행되므로 main( ) 함수가 없다). 우선,

코틀린에서는 함수를 선언할 때 fun 키워드를 함수 이름 앞에 넣는다. 그리고 함수는 소스 코드 파일의 어디든 바로 정의할 수 있다(함수의 자세한 내용은 B.7에서 설명한다). 또한 함수의 매개변수(인자)나 변수의 타입은 이름 다음에 콜론(:)을 추가한 후 지정하며, 함수의 반환값 타입은 함수 정의 문장 끝에 콜론을 추가한 후 지정한다. 여기서는 msg( ) 함수의 반환값 타입이 String이다. 함수의 반환값이 없을 때 코틀린에서는 Unit 타입으로 지정한다(자바의 void). 그러나 Unit 타입은 생략해도 되며, 이때는 함수 내부에서 return 문도 생략할 수 있다.

"B.3.5 배열"에서 자세하게 설명하겠지만, 코틀린에서는 배열이 Array 클래스로 정의되어 있다. 그리고 Array<String>처럼 배열에 저장되는 요소의 타입을 제네릭 타입으로 나타낸다(여기서는 <String>이다).

코틀린 표준 라이브러리에서는 자바 표준 라이브러리의 클래스와 메서드에 대한 래퍼(wrapper)를 제공한다. 따라서 System.out.println( ) 대신 그냥 println( )을 호출하면 된다.

또한, 자바에서는 문장의 제일 끝에 세미콜론(;)을 붙여야 하지만 코틀린에서는 그럴 필요 없다.

마지막으로, 코틀린에서는 함수의 실행 코드가 하나의 표현식으로만 구성될 때는 함수 선언 시에 반환값 타입을 생략하고 다음 코드과 같이 대입문에 정의할 수 있다.

```
fun makeMessage2(msgType: Int) = if (msgType == 1) "날씨 좋죠?" else "참 맑군요!!"
```

여기서는 if 문이 하나의 표현식으로 수행되고 그 결과는 String 타입이다. 따라서 우리가 반환값의 타입을 명시하지 않아도 코틀린 컴파일러가 추론(inference)해 준다. 그러나 함수의 실행 코드가 여러 개의 표현식으로 되어 있을 때는 다음과 같이 함수를 선언해야 한다.

```
fun makeMessage2(msgType: Int) : String {
    return if (msgType == 1) "날씨 좋죠?" else "참 맑군요!!"
}
```

또한, if 문은 다음과 같이 변수를 초기화하는 대입문에도 사용할 수 있다. 코틀린에서는 if 문을 표현식으로 간주하기 때문이다.

```
val greeting = if (msgType == 1) "날씨 좋죠?" else "참 맑군요!!"
```

(여기서 val은 불변 값의 변수를 나타내며 잠시 후에 설명할 것이다.)

이 코드에서는 msgType의 값이 1일 때 "날씨 좋죠?"가, 그렇지 않을 때는 "참 맑군요!!"가 greeting 변수에 지정된다.

"B.3.4 문자열 리터럴"에서 자세히 설명하겠지만, 코틀린에서는 변수 이름 앞에 $ 기호를 붙여서 **문자열 템플릿**으로 만든 후 문자열 접합에 사용할 수 있다(또한 자바처럼 + 연산자도 사용 가능하다). 예를 들어, 바로 앞의 greeting 변수 이름 앞에 $ 기호를 붙인 후, println("안녕하세요? $greeting")을 실행하면, greeting 변수의 문자열 값에 따라 "안녕하세요? 날씨 좋죠?" 또는 "안녕하세요? 참 맑군요!!"가 출력된다.

B.2 코틀린 변수

코틀린의 변수는 지역(local) 변수이며, 전역(global) 변수의 개념이 없다. 그리고 다음 두 가지 형태가 있다. 일단 초기화되면 값을 변경할 수 없는 불변(immutable) 변수와 얼마든지 변경 가능한(mutable) 변수이다. 불변 변수는 상수이며 val(value를 의미) 키워드로 선언한다(자바의 final과 동일). 가변 변수는 var(variable을 의미)로 선언하며 일반적인 변수이다. 모든 변수는 코드에서 실제 사용하기 전에 반드시 한 번 초기화되어야 하며, 선언된 타입은 변경할 수 없다. 코틀린에서 변수를 선언하는 예를 들면 다음과 같다.

```
val a: Int = 7
val b = 10;
val c: Int
var d: Int = 0
```

여기서 변수 a, b, c는 불변 변수이며, d는 가변 변수이다. a에는 타입과 초기값을 명시적으로 모두 지정하였으며, 정수 타입의 값인 7을 갖는다. b에는 초기값인 10만 지정하였다. 이때는 코틀린 컴파일러가 타입을 추론하여 정수 타입인 Int로 처리한다. c에는 타입만 지정하고 초기값은 주지 않았다. d에는 타입과 초기값 모두를 지정하였으며, 얼마든지 값이 변경될 수 있다.

그러나 val 변수가 객체를 참조하는 경우는 변수 자신의 참조 값은 변경할 수 없지만(달리 말해, 초기값으로 참조되는 객체 외의 다른 객체를 참조할 수 없다), 참조되는 객체가 갖는 데이터는 변경이 가능하다는 것을 알아두자.

B.3 코틀린의 타입

B.3.1 기본 타입

코틀린에는 여러 가지 종류의 타입이 있다. 우선, 모든 프로그래밍 언어에 있는 기본(primitive) 타입을 알아보자. 자바를 포함해서 대부분의 프로그래밍 언어에서는 기본 타입이 언어의 구성 요소로 되어 있으며 사전 약속된 키워드로 나타낸다. 예를 들어, 정수 타입은 int로 표기. 그러나 코틀린의 기본 타입은 자바와 달리 모두 클래스로 정의되어 있다. 예를 들어, 정수 타입은 Int이다. 그러나 우리가 기본 타입의 변수를 사용할 때 일일이 객체로 생성해서 사용하는 그런 수고는 하지 않아도 된다.

정수를 나타내는 기본 타입에는 Byte(8비트), Short(16비트), Int(32비트), Long(64비트)이 있으며, 실수를 나타내는 기본 타입에는 Float(32비트), Double(64비트)이 있다.

자바와는 다르게 코틀린에서는 숫자 타입 간의 변환을 자동으로 해주지 않으므로 주의해야 한다. 예를 들면 다음과 같다.

```
val a = 100;
val b: Long = a
val b: Long = a + 1L
```

이 경우 첫 번째 코드에서는 변수 a에 정수 타입(Int)의 값인 100을 정상적으로 지정한다. 그러나 두 번째 코드에서는 타입 불일치로 컴파일 에러가 발생한다. 숫자 타입의 변수에서 직접 값을 받을 때 자바에서는 받는 쪽의 타입 크기가 더 크면 자동으로 값을 확장하여 처리해주지만 코틀린에서는 그렇지 않다. 따라서 이때는 다음과 같이 변환 함수(자바에서는 클래스 메서드)를 호출해야 한다.

```
val a = 100;
val b: Long = a.toLong()
```

그러나 세 번째 코드에서는 컴파일 에러가 생기지 않는다. 오른쪽의 표현식인 a + 1L을 처리할 때(여기서 L은 잠시 후에 알아볼 Long 타입의 리터럴 값을 나타낸다) 컴파일러가 표현식의 결과를 Long 타입으로 추론해 주기 때문이다. 따라서 변수 b는 Long 타입의 값인 101로 초기화 된다.

코틀린에서는 숫자 타입 간의 변환을 해주는 함수를 제공하며 그 내역은 다음과 같다(toChar( ) 함수는 문자 타입으로 변환해준다).

```
toByte(): Byte, toShort(): Short, toInt(): Int, toLong(): Long
toFloat(): Float, toDouble(): Double
toChar(): Char
```

문자를 나타내는 타입에는 Char(16비트) 타입이 있으며, 부울값(true 또는 false)을 나타내는 타입에는 Boolean 타입이 있다. 코틀린에서는 Char 타입의 값을 직접 숫자로 처리할 수 없으므로 이때는 toInt( ) 함수를 사용해서 변환해야 한다(Boolean 타입은 변환 함수가 없다).

B.3.2 문자열 타입

문자열 타입은 String으로 나타낸다(내부적으로는 자바의 String 클래스를 사용). String 타입의 변수값은 변경되지 않는다. 이것은 우리가 변수값을 변경할 수 없다는 의미가 아니라 변경된 변수값을 갖는 String 객체가 새로 생성된다는 것이다. 예를 들어, 다음 코드를 보자.

```
var s: String = "변경 전의 값"
println(s)
s = "변경 후의 값"
println(s)
```

여기서는 문자열 변수로 s를 선언하고 "변경 전의 값"으로 초기화하였다. 그리고 "변경 후의 값"으로 변경하였다. 우리가 보는 관점에서는 s 변수의 값이 변경되었다. 그러나 내부적으로는 변경된 값을 갖는 새로운 String 객체가 메모리의 다른 곳에 생성된 것이다. 이것은 자바와 동일하며, 메모리 낭비나 시스템의 부담이 생길 수 있으므로 주의해야 한다.

B.3.3 기본 타입의 리터럴

기본 타입의 리터럴은 다음과 같이 나타낸다.

- **Long 타입:** L이며, 대문자만 가능하다. 예를 들면 100L.
- **Float 타입:** F 또는 f이며, 예를 들면 123.7F, .234f, 2e3f.
- **Double 타입:** 실수는 기본적으로 Double 타입으로 간주되므로 별도 표기가 없으며, 예를 들면 0.71, 7.0, 1.3e10, 1.3e-10.
- **16진수:** 0x 또는 0X를 앞에 붙이며, 예를 들면 0XABFE, 0xcdfe.

문자 리터럴의 경우는 '1', 'A' 등과 같이 홀따옴표(')를 사용하며, 이스케이프 시퀀스를 나타낼 때는 역슬래시(\)를 사용한다. 예를 들어, 탭 문자의 경우는 '\t' 또는 '\u009'(유니 코드)로 나타낼 수 있다.

숫자 리터럴(Long, Float, Double) 값이 표현식에 포함되어 있을 때는 코틀린 컴파일러가 자동으로 타입을 변환해주므로 변환 함수를 사용하지 않아도 된다. 예를 들면 다음과 같다.

```
val x: Byte = 2
val y = x + 1L
```

두 번째 코드에서는 Byte 타입의 값과 Long 타입의 값을 더하여 y에 넣지만 컴파일 에러가 발생하지 않는다. 코틀린 컴파일러가 타입을 자동 변환하여 처리해주기 때문이다.

B.3.4 문자열 리터럴

문자열 리터럴은 두 가지 방법으로 나타낼 수 있다. 우리가 주로 사용하는 첫 번째 방법의 예는 다음과 같다.

```
val s = "삶이 그대를 속일지라도\n슬퍼하거나 노하지 말라\n"
```

이처럼 겹따옴표 사이에 원하는 문자열 값을 주면 된다. 그리고 줄바꿈을 나타내는 \n과 같은 이스케이프 시퀀스 문자를 포함시킬 수 있다. 두 번째는 문자열 값을 소스 코드에 작성한 그대로 지정하는 방법이며, 예를 들면 다음과 같다.

```
val s = """
    삶이 그대를 속일지라도
    슬퍼하거나 노하지 말라
    슬픈 날엔 참고 견디라
    즐거운 날이 오고야 말리니
    """
println(s)
```

이처럼 겹따옴표 세 개를 사용하면 왼쪽 여백과 줄바꿈이 모두 반영되어 우리가 입력한 그대로 출력된다. 그리고 왼쪽 여백 없이 출력하고 싶을 때는 다음과 같이 trimMargin( ) 함수를 사용할 수 있으며, 여백 제거를 나타내는 문자를 이 함수의 인자로 전달하면 된다. 인자를 전달하지 않으면 "|" 문자가 기본으로 사용된다.

```
val s = """
    >마음은 미래를 바라느니
    >현재는 한없이 우울한 것
```

```
    >모든 것 하염없이 사라지나
    >지나가 버린 것 그리움 되리니 - (푸쉬킨)
""".trimMargin(">")
```

또한 문자열 리터럴에 변환 함수를 직접 호출하여 합당한 타입으로 변환할 수 있다. 예를 들면 다음과 같다.

```
val c = "77".toInt()
val d = "123.5".toDouble()
println("$c, $d")
```

첫 번째 코드에서는 문자열 타입의 정수를 Int 타입으로 변환하며, 두 번째 코드에서는 실수를 Double 타입으로 변환한다. 따라서 이 코드를 실행하면 77, 123.5가 출력된다.

그리고 코틀린에서는 변수 이름 앞에 $ 기호를 붙여서 **문자열 템플릿**을 만들어 문자열 접합에 사용할 수 있다. 단, 배열과 표현식의 경우는 ${args[0]} 또는 ${x+y}와 같이 중괄호({})를 앞뒤로 붙인다. 또한 $ 기호 앞에 역슬래시(\)를 붙이면 템플릿으로 인식하지 않고 일반 문자로 처리한다. 예를 들어, println("\$greeting")의 경우는 greeting을 변수로 인식하지 않으므로 $greeting이 출력된다. 템플릿의 사용 예는 다음과 같다.

```
val count = 77
val s1 = "Count = $count"                  // s1은 "Count = 77"이 된다
val s2 = "$s1의 길이는 ${s1.length}"        // s2는 "Count == 77의 길이는 10"이 된다.
println(s1)
println(s2)
```

끝으로, 코틀린 언어 1.1 버전부터는 리터럴 값을 알아보기 쉽게 값의 중간에 밑줄(_)을 사용할 수 있다. 예를 들면 다음과 같다.

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
println("$oneMillion, $creditCardNumber, $socialSecurityNumber, $hexBytes, $bytes")
```

소스 코드에는 이런 형태로 작성했지만, 코드가 실행될 때는 중간의 밑줄들은 모두 제외된 상태로 값이 지정되며, 다음과 같이 출력된다.

```
1000000, 1234567890123456, 999999999, 4293713502, 3530134674
```

B.3.5 배열

대부분의 프로그래밍 언어에서는 언어 구성 요소로 배열을 갖고 있다. 그러나 이번 장 앞에서 얘기했듯이, 코틀린에는 배열이 Array 클래스로 정의되어 있다. 그리고 Array<String>처럼 배열에 저장되는 요소의 타입을 제네릭 타입으로 나타낸다(여기서는 <String>이다). 따라서 인덱스 연산자(B.4.10 절 참조)인 대괄호([])는 배열을 선언하고 생성할 때 사용하지 않으며, 배열의 각 요소를 읽거나 쓸 때만 사용한다.

코틀린에서는 여러 가지 방법으로 배열을 생성할 수 있다. 첫 번째로, arrayOf( ) 함수를 사용하면 쉽다. 예를 들면 다음과 같다.

```
val item = arrayOf("사과", "바나나", "키위")
```

이 경우 item[0]의 값은 "사과", item[1]의 값은 "바나나", item[2]의 값은 "키위"가 된다. 그리고 item 배열의 타입은 지정하지 않아도 된다. 요소 타입이 String이라는 것을 컴파일러가 추론할 수 있기 때문이다.

item 배열의 모든 요소 값을 출력할 때는 다음과 같이 for 루프를 사용하면 된다(for 루프는 B.6.2에서 설명한다).

```
for (fruit in item) {
    println(fruit)
}
```

두 번째는, Array 클래스의 생성자를 사용하는 방법이며, 동일한 코드를 다음 두 가지 형태로 작성할 수 있다.

```
val num = Array<String>(5, { i -> (i * i).toString() })
```

또는

```
val num = Array<String>(5) { i -> (i * i).toString() }
```

생성자에는 요소의 개수, 그리고 중괄호({}) 안에 **람다식**(lambda)을 지정할 수 있다(람다식은 B.9에서 설명한다). 또한, 배열의 요소 타입을 나타내는 제네릭 타입(여기서는 <String>)은 생략해도 된다. 요소 타입이 String이라는 것을 컴파일러가 추론할 수 있기 때문이다.

여기서 num[0]의 값은 "0", num[1]의 값은 "1", num[2]의 값은 "4", num[3]의 값은 "9", num[4]의 값은 "16"이 된다. i 값이 0부터 1씩 증가하면서 곱해진 후 num 배열의 해당 인덱스(i) 요소 값으로 초기화되기 때문이다.

이렇게 생성된 num 배열의 모든 요소는 다음과 같이 for 루프로 출력할 수 있다.

```
for (item in num) {
    println(item)
}
```

또한 코틀린에서는 기본 타입의 요소를 저장하는 배열을 별도의 클래스로 갖고 있다. 예를 들어, Int 타입의 값을 저장하는 배열은 IntArray이다. 이외에도 ByteArray, ShortArray, LongArray, FloatArray, DoubleArray, CharArray, BooleanArray가 있다. 그리고 이 배열 클래스들은 코틀린 컴파일러가 컴파일할 때 자바의 기본 타입 배열로 변환한다(예를 들어, IntArray는 int[]로 변환).

기본 타입 배열의 선언과 생성도 앞에서 설명한 것과 유사하게 두 가지 방법으로 할 수 있다. 단, 클래스 이름과 함수 이름만 다르다. 예를 들면 다음과 같다.

```
val intItem: IntArray = intArrayOf(1, 2, 3, 4, 5)
```

```
val intNum = IntArray(5, { i -> (i * i) })
```

또는

```
val intNum = IntArray(5) { i -> (i * i) }
```

배열의 각 요소 값은 인덱스를 사용해서 읽거나 쓸 수 있다. 예를 들면 다음과 같다.

```
intNum[0] = intNum[1] + intNum[2]
```

B.4 연산자와 연산자 오버로딩

코틀린의 연산자(operator)는 다른 언어와 색다르다. 연산자를 표기하는 형태(예를 들어, 덧셈은 +)와 연산자 우선순위(하나의 표현식에 여러 개의 연산자가 있을 때 어떤 연산자가 먼저 처리되는지를 결정)는 다른 언어와 유사하지만, 코틀린에서는 내부적으로 연산자를 오버로딩(overloading)한 함수를 사용해서 연산을 처리한다. 예를 들어, Int 타입의 변수 a와 b의 값을 더한다고 할 때 우리는 a+b로 코드를 작성하지만, 내부적으로는 덧셈(+) 연산자를 오버로딩한 a.plus(b)를 수행하도록(a의 plus( ) 함수를 실행) 컴파일러가 코드를 변환해준다. 따라서 피연산자인 a와 b가 기본 타입이나 객체 중 어떤 것일지라도 우리는 덧셈을 +로만 나타내면 되므로 굉장히 편리하다(코틀린의 기본 타입은 클래스로 정의되어 있기 때문이다). 또한, 우리가 정의한 클래스의 객체 간에 덧셈 연산이 필요할 때는 우리 클래스에서 plus( ) 함수를 오버로딩하면 된다(이때 함수 이름 앞에 operator 키워드를 붙인다). (자바에서는 기본 타입의 덧셈 연산을 +로 나타내지만, 객체 간의 덧셈 연산에서는 +를 사용할 수 없고 add( )와 같은 메서드를 별도로 호출해야 하므로 불편하다.)

B.4.1 산술 연산자

산술(arithmetic) 연산자의 표기와 변환되어 실행되는 코드(오버로딩된 함수) 및 의미는 다음과 같다.

표기	변환 코드	의미
a + b	a.plus(b)	a와 b의 값을 더한다
a – b	a.minus(b)	a의 값에서 b의 값을 뺀다
a * b	a.times(b)	a의 값과 b의 값을 곱한다
a / b	a.div(b)	a의 값을 b의 값으로 나눈다.
a % b	a.rem(b), a.mod(b)	a의 값을 b의 값으로 나눈 후 나머지를 구한다

*, /, % 연산자의 우선순위는 동일하지만 +, - 보다는 높으므로 같은 표현식에 있을 때 먼저 실행된다.

기본 타입의 변수를 연산할 때는 우리가 굳이 연산자를 오버로딩하지 않아도 된다. 이미 오버로딩되어 있기 때문이다. 그러나 우리가 정의한 클래스의 객체 간에 연산을 편리하게 하려면 연산자를 오버로딩하는 방법을 알아 둘 필요가 있으므로, 이번 장 앞에서 작성한 Kotlin01 프로젝트에 코틀린 파일을 추가하여 실습해보자(클래스에 관해서는 B.8에서 설명한다).

Kotlin01 프로젝트 밑의 src 폴더에서 오른쪽 마우스 버튼을 클릭한 후 New ➡ Kotlin File/Class를 선택한다. 그리고 대화상자에서 파일 이름에 Hello2를 입력하고(각자 원하는 이름을 입력해도 된다) OK 버튼을 클릭한다. 편집기 창에 열린 Hello2.kt 파일의 아무 곳이나 클릭한 후 main을 입력하고 Tab 키를 눌러서 main( ) 함수 코드를 생성한다. 그다음에 아래와 같이 코드를 입력하자.

```
fun main(args: Array<String>) {
    val m1 = Score(100, 200)
    val m2 = Score(300, 400)
    println(m1 + m2)
}

data class Score(val a: Int, val b: Int) {
    operator fun plus(other: Score): Score {
        return Score(a + other.a, b + other.b)
    }
}
```

코드 작성이 끝났으면 main( ) 함수 왼쪽의 코틀린 아이콘()을 클릭한 후 Run 'Hello2Kt'를 선택하여 코드를 컴파일하고 실행한다. 그러면 그림 B-2처럼 Run 도구 창에 실행 결과가 출력될 것이다.

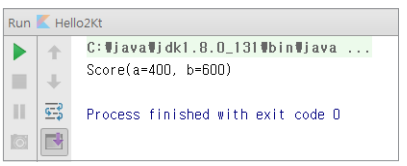


그림 B-2

앞의 코드를 자세히 살펴보자. 여기서는 우리의 클래스인 Score를 정의하였다. 이 클래스는 두 게임의 점수를 a와 b 속성으로 갖고 있다. 클래스는 class 키워드로 정의한다. 그리고 이처럼 데이터를 갖는 클래스를 정의할 때 코틀린에서는 데이터 클래스로 정의하면 편리하다(class 키워드 앞에 data를 지정하며, 자세한 내용은 B.8.8에서 설명한다). 데이터 클래스에 공통적으로 필요한

equals( ), hashCode( ), toString( ), copy( )와 같은 함수를 코틀린 컴파일러가 자동으로 생성해 주기 때문이다.

데이터 클래스의 속성은 클래스 이름 다음에 괄호 속에 정의하면 된다(괄호는 이 클래스의 기본 생성자를 나타내며, 자세한 내용은 B.8.1에서 설명한다). 그리고 Score 클래스에서는 덧셈(+) 멤버 함수로 연산자를 오버로딩하고 있다. 연산자를 오버로딩하는 함수는 맨 앞에 operator 키워드를 붙여야 한다. 그리고 앞의 표에 있듯이, 덧셈의 오버로딩 함수는 plus( )이므로 함수 이름을 반드시 plus로 지정해야 한다. 여기서는 plus( ) 함수의 인자로 다른 Score 객체를 받으므로 other 키워드를 사용하였다. 또한 새로운 Score 객체를 반환하므로 반환 타입은 Score이다.

main( ) 함수에서는 두 개의 Score 객체를 생성한 후 + 연산자를 사용해서 두 객체의 덧셈을 수행하고 결과를 출력한다. 이때 Score 클래스에 오버로딩된 plus( ) 함수가 실행되어 두 객체의 a와 b 점수를 더한 값을 갖는 새로운 Score 객체가 생성되고 반환된다.

연산자를 오버로딩할 때 이처럼 클래스의 멤버 함수로 해도 되지만, 클래스 외부에서 확장(extension) 함수로 할 수도 있다. 확장 함수를 사용하면 클래스를 변경하지 않고 새로운 기능(함수)을 추가할 수 있다. 이때는 Hello2.kt 파일에 오버로딩 함수를 별도로 정의하면 된다. 단, 확장할 클래스를 알려주어야 하므로, 함수 이름 앞에 클래스 참조를 추가해야 한다(확장 함수는 B.7.3에서 설명한다). 다음과 같이 진한 글씨의 코드를 추가하여 곱셈 연산자도 오버로딩해보자.

```
fun main(args: Array<String>) {
    val m1 = Score(100, 200)
    val m2 = Score(300, 400)
    println(m1 + m2)
    println(m1 * m2)

}

data class Score(val a: Int, val b: Int) {
    operator fun plus(other: Score): Score {
        return Score(a + other.a, b + other.b)
    }
}

operator fun Score.times(other: Score): Score {
    return Score(a * other.a, b * other.b)
}
```

여기서는 곱셈 연산자로 실행되는 times( ) 함수를 확장 함수를 사용해서 오버로딩하였다.

이 코드를 컴파일하고 실행하면 그림 B-3처럼 Run 도구 창에 실행 결과가 출력될 것이다.

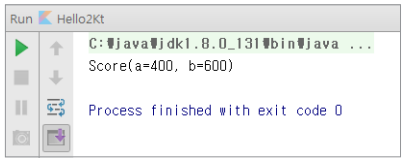


그림 B-3

B.4.2 단항 연산자

단항(unary) 연산자의 표기와 변환되어 실행되는 코드(오버로딩된 함수) 및 의미는 다음과 같다.

표기	변환 코드	의미
+a	a.unaryPlus( )	a의 값을 양수로 변환
-a	a.unaryMinus( )	a의 값을 음수로 변환
!a	a.not( )	Boolean 타입의 부정(true는 false로, false는 true로)
++a, a++	inc	a의 값에 1을 더함
--a, a--	dec	a의 값에서 1을 뺌

다음과 같이 확장 함수를 추가한 후 main( ) 메서드에서 println(-m1)을 실행하면 결과를 확인할 수 있다.

```
operator fun Score.unaryMinus(): Score {
    return Score(-a, -b)
}
```

여기서는 Score 객체를 인자로 받지 않는다. 단항 연산자는 현재의 Score 객체로 연산하기 때문이다. 그리고 연산 후에는 새로운 Score 객체가 생성되어 반환된다.

B.4.3 복합 대입 연산자

복합 대입 연산자는 a += b와 같이 사용하며, 이것은 a = a + b와 동일하다. 단, 이때 a는 값의 변경이 가능한 변수이어야 하므로 기본 타입 변수는 var로 선언해야 한다. 또한 객체의 경우는 참조 변수인 a와 b의 값이 변경될 필요는 없으므로 var이나 val 모두 가능하다.

표기	변환 코드	의미
a += b	a.plusAssign(b)	a의 값에 b의 값을 더한 후 a에 넣음
a -= b	a.minusAssign(b)	a의 값에서 b의 값을 뺀 후 a에 넣음
a *= b	a.timesAssign(b)	a의 값에 b의 값을 곱한 후 a에 넣음
a /= b	a.divAssign(b)	a의 값을 b의 값으로 나눈 후 a에 넣음
a %= b	a.modAssign(b)	a의 값을 b의 값으로 나눈 후 나머지를 a에 넣음

B.4.4 비트 연산자

비트(bitwise) 연산자는 피연산자의 각 비트 값에 대해 연산을 수행한다. 코틀린에서는 비트 연산자를 오버로딩하고 있지 않으므로, 자바의 <<, >>, >>>처럼 이 연산자를 나타내는 표기 방법이 없다. 대신에 피연산자 사이에 사전 정의된 함수를 넣어서 비트 연산을 수행한다. 이런 함수를 중위(infix) 함수라고 하며, 자세한 내용은 B.7.3에서 설명한다. 비트 연산 함수는 다음과 같다. (Byte, Short, Int, Long, Float, Double과 같이 부호가 있는 숫자 타입의 부호 비트는 제일 왼쪽에 있다.)

함수명	의미
shl	부호 비트는 그대로 두고 왼쪽으로 비트 이동(Signed shift left)
shr	부호 비트는 그대로 두고 오른쪽으로 비트 이동(Signed shift right)
ushr	부호 비트를 포함해서 오른쪽으로 비트 이동(Unsigned shift right)
and	대응되는 각 비트에 대해 논리 and 연산 수행(Bitwise and)
or	대응되는 각 비트에 대해 논리 or 연산 수행(Bitwise or)
xor	대응되는 각 비트에 대해 논리 xor(exclusive or) 연산 수행(Bitwise xor)
inv	0또는 1의 비트 값을 반대로 바꿈(Bitwise inversion)

(참고로, shl은 자바의 <<, shr은 >>, ushr은 >>>, and는 &, or는 |, xor는 ^와 동일하다.)

비트 연산자의 사용 예는 다음과 같다.

```
println(8 shr 2)
println(8 shl 4)
println(0xC0 and 0x0C)
println(0xC0 or 0x0C)
println(0xC0 xor 0xC0)
```

첫 번째 결과는 2($8/2^2$), 두 번째는 128($8*2^4$), 세 번째는 0, 네 번째는 204, 마지막은 0이 된다.

오른쪽으로 비트 값을 이동하는 것은 2^m만큼 나누는 것이고, 왼쪽으로 이동은 2^m만큼 곱하는 것이기 때문이다. 또한 비트 and는 대응 비트가 모두 1일 때만 1이며, or는 둘 중 하나가 1이면 1, xor는 대응 비트가 같으면 0이 된다.

여기서 왼쪽 피연산자는 정수 타입의 값을 갖는 변수나 표현식 어느 것도 가능하며, 예를 들면 다음과 같다.

```
val a = 8
println(a+8 shr 2)
```

여기서는 4가 결과로 출력된다. 그리고 a+8은 (a+8)로 괄호를 씌우지 않아도 된다. shr( ) 함수 호출보다 + 연산자의 우선순위가 높기 때문이다.

B.4.5 논리 연산자

논리(logical) 연산자는 피연산자(변수나 표현식)에 대해 논리 연산(true 또는 false)을 수행한다. 비트 연산의 경우처럼 논리 연산도 피연산자 사이에 사전 정의된 함수를 넣어서 수행한다(이것도 비트 연산자처럼 중위 함수로 정의되어 있다). 그리고 당연한 얘기지만, 논리 연산의 각 피연산자는 true 또는 false 값이어야 하며, 변수나 표현식 모두 될 수 있다. 논리 연산 함수는 다음과 같다.

함수명	의미
and	좌우의 피 연산자에 대해 논리 and 연산 수행(Logical or)
or	좌우의 피 연산자에 대해 논리 or 연산 수행(Logical or)
not	좌우의 피 연산자에 대해 논리 not 연산 수행(Logical or)

(참고로, and는 자바의 &&, or는 ||, not은 !와 동일하며, 코틀린 1.1 버전에서는 아직까지 함수 대신 &&, ||, !를 사용할 수 있다.)

논리 연산자의 사용 예는 다음과 같다.

```
val b1: Boolean = true
val b2: Boolean = false
val b3: Boolean = false
if ((b1 || b2) && !b3)
  println("True!!")
if ((b1 or b2) and b3.not())
  println("True!!")
```

이 코드의 첫 번째 if와 두 번째 if는 동일한 의미이며, 둘 다 "True!!"를 출력한다.

B.4.6 동등 비교 연산자

코틀린의 동등 비교 연산자는 다음과 같다.

표기	변환 코드
a == b	a?.equals(b) ?: (b == null)
a === b	오버로딩 안 됨
a != b	!(a?.equals(b) ?: (b == null)
a !== b	오버로딩 안 됨
not	좌우의 피 연산자에 대해 논리 not 연산 수행

코틀린에서는 기본 타입과 객체 모두 == 연산자가 equals( ) 함수 호출로 변환된다는 점이 자바와 다르다. 자바의 경우 기본 타입을 == 연산자로 비교하면 변수의 값이 같은지 비교한다. 그러나 객체의 경우는 우리 클래스에서 equals( ) 메서드를 오버라이딩 해야만 객체의 값을 비교할 수 있으며, 그렇지 않으면 변수의 참조 값을 비교하게 되므로 혼란을 초래할 수 있다. 그러나 코틀린에서는 == 연산자를 사용할 때 항상 해당 클래스에 오버라이딩된 euqals( ) 함수를 호출하도록 되어 있다. 또한 비교하는 변수가 null 값인지 확인하므로 NullPointerException 예외가 발생하지 않도록 처리해준다(==은 같은지 비교하고 !=은 다른지 비교한다). 이와는 달리, 변수의 참조 값이 같은지(똑같은 객체를 참조하는지) 비교할 때는 === 연산자를 사용하고 반대로 다른지 비교할 때는 !== 연산자를 사용한다.

앞의 표에서 a == b의 변환 코드가 어떤 의미인지 알아보자.

```
a?.equals(b) ?: (b == null)
```

여기서는 우선 변수 a의 값이 null이 아닌지 확인하고(?. 연산자의 기능이며, B.5.2에서 설명한다) 아니면 a 객체의 equals(b) 함수를 호출한다. 그리고 equals(b) 함수를 실행한 결과 값이 null이 아니면 그 값을 반환하며, null이면 (b == null)의 결과 값을 반환한다(이런 기능을 해 주는 연산자가 "?:"이며, 이것을 엘비스 연산자라고 한다. 이 내용은 B.5.3에서 설명한다). 궁극적으로 ==과 != 연산자는 null 값이 가능한 변수의 동등 비교를 할 때도 안전하게 사용할 수 있다.

또한 데이터 클래스(자세한 내용은 B.8.8에서 설명한다)로 정의하면 equals( ) 함수를 자동으로 오버라이딩해 주므로 편리하다. 과연 그런지 알아볼 겸, 앞의 B.4.1에서 정의한 Score 클래스 객체를 생성하여 비교하는 다음 코드를 m2 변수 다음에 추가해보자.

```
val m2 = Score(300, 400)
val m3 = Score(300, 400)
println(m2 == m3)
println(m2 === m3)
println(m2.equals(m3))
```

실행 결과는 true, false, true가 된다. Score는 데이터 클래스이므로 코틀린 컴파일러가 equals( ) 함수를 자동으로 오버라이딩하여 두 객체의 모든 속성 값을 정상적으로 비교했기 때문이다. 단, 비교되는 속성들은 해당 클래스의 기본 생성자(B.8.1 참조)에 정의되어야 한다. 그리고 === 연산자의 결과는 false이다. 두 개의 Score 객체(인스턴스)가 다른 곳에 존재하기 때문이다.

B.4.7 그 밖의 비교 연산자

동등 비교 연산자 외의 다른 비교 연산자는 다음과 같다.

표기	변환 코드
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

이 연산자들은 모두 compareTo( ) 함수 호출로 변환되며, 이것은 자바와 마찬가지로 Comparable 인터페이스에 정의된 함수를 구현한 것이다. 위의 표에서 보듯이, a가 b보다 큰지를 비교하는 경우에는 a.compareTo(b) > 0가 실행된다. 여기서 compareTo( ) 함수의 반환값은 Int 타입이며, a가 b보다 작으면 음수 값이, 크면 양수 값이 반환된다.

기본 타입의 경우는 이미 compareTo( ) 함수가 정의되어 있지만, 우리가 정의한 클래스의 객체를 비교하려면 Comparable 인터페이스와 그것에 정의된 compareTo( ) 함수를 구현해야 한다. 예를 들어, 이름과 전화번호를 갖는 고객(Customer) 클래스를 정의하고, 각 고객 객체의 이름과 전화번호를 비교한다면 다음과 같이 할 수 있다(클래스와 인터페이스는 B.8에서 설명한다). Kotlin01 프로젝트의 src 폴더 밑에 새로운 코틀린 파일을 생성하고 다음 코드를 작성해보자.

```
fun main(args: Array<String>) {
    val p1 = Customer("홍길동", "010-1234-5678")
    val p2 = Customer("김선달", "010-5678-1234")
    println(p1 < p2)
    println(p1 > p2)
}

class Customer(val name: String, val phone: String) : Comparable<Customer> {
    override fun compareTo(other: Customer): Int {
        return compareValuesBy(this, other, Customer::name, Customer::phone)
    }
}
```

클래스에서 인터페이스를 구현할 때는 클래스 선언문 제일 끝에 콜론(:)을 붙이고 구현할 인터페이스 이름을 지정한다(자바에서는 : 대신에 implements 키워드를 사용).

이 코드를 실행하면 false와 true가 차례대로 나올 것이다. 이름의 문자열 비교에서 p1 고객의 이름이 p2보다 작기 때문이다. 여기서 사용한 compareValuesBy( ) 함수는 compareTo( ) 함수를 쉽게 구현할 수 있도록 코틀린 라이브러리에서 제공한다. 그러나 compareTo( ) 함수는 우리 나름대로 구현해도 된다. 단, 비교 연산자 왼쪽의 피연산자가 오른쪽 피연산자보다 작을 때는 음수를, 클 때는 양수를, 같을 때는 0을 반환해야 한다.

B.4.8 In 연산자

List나 Map과 같이 객체를 저장하는 데이터 구조를 컬렉션(collection)이라고 한다. 코틀린은 자바와 동일한 컬렉션 클래스들을 사용한다. 그리고 확장된 API를 추가로 제공한다. In 연산자는 특정 객체가 컬렉션에 저장되어 있는지 확인하는 연산자이다. 그러나 이런 연산자는 특정 함수를 호출할 때 알기 쉽게 사용하도록 하는 것이 목적이므로 컨벤션(convention)이라고도 한다. In 연산자는 다음과 같다.

표기	변환 코드
a in b	b.contains(a)
a !in b	!b.contains(a)

a in b에서는 컬렉션 b에 a가 있으면 true가 반환되며, a !in b에서는 없을 때 true가 반환된다.

B.4.9 범위(..) 연산자

범위 값을 가지며 a..b로 표기하면 컴파일러가 a.rangeTo(b)로 코드를 생성한다. 이 경우 a부터 b까지의 범위 값이 생성된다. main( ) 함수 끝에 다음 코드를 추가하고 실행해보자.

```
val start = LocalDate.now()
val end = start..start.plusDays(15)
println(start.plusWeeks(1) in end)
println(end)
```

첫 번째 println( )에서는 true가 출력되고, 두 번째 println( )에서는 2017-06-20..2017-07-05와 같이 범위 값이 출력될 것이다.

B.4.10 인덱스 연산자

배열은 인덱스를 사용해서 저장된 요소를 읽거나 쓸 수 있다. 또한 우리가 정의하는 컬렉션 클래스를 포함해서 코틀린의 컬렉션 중에도 그렇게 할 수 있는 것이 있다. 이때는 get( )과 set( )을 operator 함수로 정의하여 인덱스 연산자인 대괄호([])를 오버로딩하면 된다.

표기	변환 코드
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

이 경우 인덱스 연산자인 대괄호([])가 get( ) 또는 set( ) 함수 호출 코드로 변환된다.

B.4.11 Invoke 연산자

함수를 호출할 때는 이름에 괄호를 붙이는 표기 방법을 사용한다. 어찌 보면 이상하게 생각될 수도 있지만 코틀린에서는 클래스의 인스턴스도 이런 방법으로 호출할 수 있다. 즉, operator 키워드를 지정하여 연산자를 오버로딩한 invoke( ) 함수를 클래스에 정의하면 된다(클래스는 B.8에서 설명한다). 그리고 이 클래스의 인스턴스를 생성한 후 참조 변수에 괄호를 붙여서 호출한다. 다음 표에서 a는 이런 객체의 참조 변수이며, 우리가 a( )로 코드를 작성하면 컴파일러가

a.invoke( ) 함수 호출로 변환해 준다. 물론 이때 인자도 얼마든지 전달할 수 있다.

표기	변환 코드
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

앞에서 작성한 소스 코드 파일에 다음의 클래스 정의 코드를 추가한다.

```
class InvokeOperator(val makeMessage1: String) {
    operator fun invoke(makeMessage2: String) {
        println("$makeMessage1 $makeMessage2!")
    }
}
```

그리고 main( ) 함수에 다음 코드를 추가하자.

```
val instance1 = InvokeOperator("코틀린을")
instance1("배우자")
```

이 코드를 실행하면 "코틀린을 배우자!"가 출력된다. 이와 같이 하면 InvokeOperator 인스턴스를 함수처럼 호출할 수 있다. 내부적으로는 instance1("배우자")가 instance1.invoke("배우자")로 변환되어 실행되기 때문이다.

이런 형태로 코드를 작성하면 이해하기 어려울 수 있다. 따라서 일반적인 코드보다는 DSL(Domain Specific Language)에서 유용하게 사용된다. 예를 들어, 프로젝트 자동화 관리 도구인 그래들(gradle)에서 모듈 의존 관계(dependencies)를 정의할 때 사용하면 편리하다(안드로이드 스튜디오에서도 그래들을 사용한다).

B.4.12 타입 확인 연산자: is, !is

코틀린에서는 객체가 해당 클래스 타입인지 확인할 때 is 연산자를 사용한다(자바의 instanceof). !is는 해당 타입이 아닌지를 검사한다. 다음 예를 보자.

```
val b: String = "코틀린을 배우자"
if (b is String) {
    println("String 타입임")
} else {
    println("String 타입이 아님")
}
```

여기서 b는 String 객체이므로 String 타입이다. 따라서 "String 타입임"이 출력된다. 또 다른 예를 보자.

```
open class A {}
class B : A() {}

val x = B()
if (x is A) {
    println("A 타입임")
} else {
    println("A 타입이 아님")
}
if (x is B) {
    println("B 타입임")
} else {
    println("B 타입이 아님")
}
```

여기서 클래스 B는 A 클래스로부터 상속받는 서브 클래스이다. 따라서 B 클래스의 인스턴스(객체)는 자신의 클래스인 B의 타입이면서 동시에 A 타입도 된다. 그러므로 "A 타입임"과 "B 타입임"이 출력된다.

클래스에 관해서는 B.8에서 자세히 설명하겠지만, 참고로 코틀린에서는 class 키워드로 클래스를 정의하며, open 키워드는 해당 클래스가 상속 가능하도록 할 때 지정해야 한다. 그리고 상속을 나타낼 때는 콜론(:)을 사용한다(자바에서는 extends).

B.5 Null 타입 처리 메커니즘

자바 언어에서 가장 문제가 될 수 있는 것이 런타임 시에(애플리케이션을 실행할 때) 흔히 발생할 수 있는 NullPointerException 예외(에러)이다. 이것은 객체의 메서드를 호출할 때 해당 객체의 참조 변수값이 null이라서 호출할 수 없으므로 발생한다. 그러므로 코틀린에서는 언어 자체에 새로운 기능을 추가하여 런타임이 아닌 컴파일 시점에 NullPointerException 예외 발생을 미리 방지할 수 있게 하였다. 그러나 이로 인해 개발자가 더 많은 코드를 작성하게 하는 부담을

준다면 불편할 것이다. 따라서 null 참조와 관련된 타입 처리 기능, 즉 타입과 연산자를 추가하여 개발자의 부담도 덜고 에러도 미연에 방지할 수 있게 배려하고 있다.

B.5.1 Null 가능 타입

코틀린에서는 기본적으로 모든 타입에 null 값을 허용하지 않는다(코틀린에서는 기본 타입도 클래스로 정의되어 있으므로 사실상 모든 타입이 객체 참조로 사용된다고 생각하면 된다). 따라서 null 값을 허용하려면 타입 이름 끝에 물음표(?)를 붙여야 한다. 예를 들어, 다음 코드를 보자.

```
val s1: String? = null
val s2: String = null
val s3: String = s1
```

첫 번째 코드는 정상적이다. null 값이 될 수 있다고 코틀린 컴파일러에게 알려주었기 때문이다. 그러나 두 번째 코드에서는 다음의 컴파일 에러가 발생한다.

```
Error: Null cannot be a value of a non-null type String
```

null 값을 허용하지 않는 타입에 null을 지정했기 때문이다.

세 번째 코드 역시 다음의 컴파일 에러가 발생한다.

```
Error: Type mismatch: inferred type is String? but String was expected
```

메시지는 다르지만 의미는 마찬가지다. 즉, null 값을 받으려면 String 대신 String?으로 지정해야 한다는 것이다.

String 객체는 문자열의 길이를 반환해주는 length 속성이 있다. 따라서 다음 코드와 같이 문자열의 길이(문자의 개수)를 출력할 수 있으며, 출력 결과는 8이다(문자는 유니 코드를 지원한다). 그리고 여기서는 a의 값을 변경할 것이므로 val이 아닌 var 변수로 선언하였다.

```
var a: String = "코틀린을 배우자"
println(a.length)
```

그러나 다음과 같이 하면 컴파일 에러가 발생한다.

```
a = null
```

null 값을 가질 수 없는 변수에 null을 넣으려고 했기 때문이다. 객체 참조 변수가 null 값을 가짐으로 해서 발생할 수 있는 NullPointerException 예외를 이처럼 컴파일 시점에 미리 막아준다.

그렇다면 a를 null 값이 가능한 변수로 지정하면 어떻게 될까?

```
var a: String? = "코틀린을 배우자"
println(a.length)
```

이때는 코틀린 컴파일러가 a.length를 컴파일 에러로 처리한다. a 변수는 null 값을 가질 수 있으므로 length 속성을 참조할 때 안전하지 않기 때문이다. 따라서 다음과 같이 null 값이 아닐 때만 a.length를 참조하는 코드를 if 문에 넣어 주어야 한다.

```
if (a != null) println(a.length)
```

그러나 null 참조 값을 방지하기 위해 이런 식으로 우리가 매번 코드를 작성해 주어야 한다면 불편할 것이다. 따라서 코틀린에서는 이런 일을 하는 연산자를 별도로 제공한다.

B.5.2 Null 처리 연산자 #1: "?."

"?." 연산자는 객체의 속성이나 함수(자바의 메서드)를 안전하게 참조/호출할 수 있게 해준다. 즉, 객체 참조가 null이 아닐 때만 참조하게 한다. 바로 앞의 다음 코드는,

```
if (a != null) println(a.length)
```

아래와 같이 "?." 연산자를 사용하여 간단하게 작성할 수 있다.

```
println(a?.length)
```

여기서 "?."은 a 참조의 값이 null 이 아닐 때만 String 객체인 a의 length 속성을 참조하라는 의미이다. 따라서 a의 값이 null일 경우에도 NullPointerException은 발생하지 않으며 a?.length는 null 값을 반환한다. 따라서 이 연산자를 사용하면 항상 객체의 속성이나 함수를 안전하게 참조/호출할 수 있으며, 우리가 if 문과 같은 코드를 추가로 작성하지 않아도 되므로 편리하다.

코틀린에서도 자바처럼 함수의 반환값이 객체 참조일 때는 점(.) 연산자를 사용하여 연쇄(chain) 호출이 가능하다. 그렇다면 이때는 "?." 연산자를 어떻게 사용할 수 있을까? 다음 예를 보자. 여기서는 문자열에서 부분 문자열을 추출한 후 그것의 길이를 출력한다.

```
var a: String? = "코틀린을 배우자"
println(a?.substring(5)?.length)          // 출력 결과는 3이다
```

여기서 a?.substring(5)의 결과는 "배우자"를 값으로 갖는 String 객체이다. 따라서 a?.substring(5)?.length로 연쇄 호출하여 그것의 길이를 알 수 있다. 이처럼 연쇄 호출 시에는 계속해서 "?." 연산자를 사용하면 된다.

즉, "?." 연산자는 참조 변수의 null 여부 확인 및 참조되는 객체의 속성 참조와 함수 호출을 겸하는 역할을 한다.

B.5.3 Null 처리 연산자 #2: "?:"

"?:" 연산자는 "로큰롤의 왕"으로 유명했던 엘비스 프레슬리(Elvis Preslev)의 이모티콘과 유사하다고 해서 엘비스 연산자라고 한다. 이 연산자의 왼쪽 피연산자 값이 null이 아니면 그 피연산자의 결과 값을 반환하고, null이면 오른쪽 피연산자의 결과 값을 반환한다. 여기서 피연산자는 변수나 함수 호출 모두 가능하며, 피연산자가 함수 호출일 때는 해당 함수가 실행되어 그 결과 값이 반환된다. 이 연산자도 참조 변수의 null 여부 확인 및 참조되는 객체의 속성 참조와 함수 호출을 겸하는 역할을 한다. 다음 예를 보자.

```
var a: String? = "코틀린을 배우자"
val b = a?.length ?: 0      // val b: Int = if (a != null) a.length else 0
println(b)
```

이 코드의 두 번째 줄에서는 a의 문자열 값이 있으면 그 길이를 변수 b에 넣으며, 없으면(null이면) 0을 넣는다. 만일 엘비스 연산자가 없다면, 이와 동일한 기능을 수행하는 주석 표시된 코드를 우리가 작성해 주어야 한다.

엘비스 연산자의 오른쪽 피연산자는 함수 호출이나 함수의 return문 또는 예외를 발생시키는(던지는) throw 문이 될 수도 있다.

B.5.4 Null 처리 연산자 #3: "!!"

"!!" 연산자는 참조 변수의 값이 null이 아니면 정상적으로 코드를 수행하고 null이면 런타임 시에 NullPointerException 예외를 발생시킨다. 그렇다면 NullPointerException을 컴파일 시점에서 미리 방지해 준다는 코틀린의 방침과 어긋나므로 왜 이런 연산자가 필요한지 이상하다고 생각할 수 있을 것이다. 이것은 우리가 작성하는 클래스와 함수에서는 사용할 일이 거의 없으며, 시스템 라이브러리의 변수를 참조하거나 함수들을 호출할 때 언제 발생할지 모르는 NullPointerException을 우리가 명시적으로 파악하고자 할 때 필요하다. 다음 예를 보자.

```
var a: String? = "코틀린을 배우자"
val b = a!!.length
println(b)
```

이 코드는 앞에 나왔던 코드와 유사하다. 그러나 첫 번째 줄의 코드를 var a: String? = null로 변경하여 a가 null이 될 때는 런타임 시에 NullPointerException을 발생시킨다는 것만 다르다. 그리고 애플리케이션의 실행이 중단된다.

B.5.5 Null 처리 연산자 #4: as, as?

객체의 타입을 변환(casting)할 때는 as 연산자를 사용한다(단, 기본 타입의 경우는 as 연산자를 사용할 수 없고 변환 함수를 사용해야 한다). 예를 들면 다음과 같다.

```
val s1: String? = s2 as String?
```

여기서는 s2가 String 타입에 적합한 것일 때 s2의 값을 String 타입으로 변환하여 s1에 넣는다. 그러나 만일 s2의 값이 String 타입에 부적합하다면 ClassCastException 예외가 발생되고 실행이 중단된다. 따라서 예외가 발생되지 않도록 안전하게 변환하려면 다음과 같이 as?를 사용한다.

```
val s1: String? = s2 as? String?
```

이렇게 하면 s2의 값이 String 타입에 부적합할 때 예외가 생기지 않고 null 값이 반환된다.

B.6 코드 실행 제어

모든 프로그래밍 언어와 마찬가지로 코틀린에도 코드의 실행을 제어하는 언어 구성 요소들이 있다. 조건문과 반복문 등이다. 코틀린에서는 조건문으로 if와 when을 사용하며, 반복문으로 for와 while 및 do~while 루프를 사용한다. 이 내용은 알기 쉬우므로 예제 코드를 중심으로 빨리 파악해보자.

B.6.1 if

우선, if는 "B.1 코틀린 프로그램 구조"에서 설명한 대로 여러 형태로 사용할 수 있으니 참조하자. 또 다른 형태는 다음과 같다.

```
val result = if (param == 1) {
    "one"
} else if (param == 2) {
    "two"
} else {
    "three"
}
```

이 코드에서 알 수 있듯이, 다른 언어와 마찬가지로 if – else if – else를 사용할 수 있다. 그러나 중요한 차이점이 있다. 코틀린에서는 if 문을 하나의 표현식으로 간주하므로 이 코드처럼 변수의 대입문에 if 문을 사용할 수 있다. 이 경우 if 문의 실행 결과 값이 왼쪽의 result 변수에 지정된다.

B.6.2 when

when은 자바의 switch~case와 유사하지만 더 간결하고 기능도 뛰어나다. 여러 가지 형태의 사용 예를 보면 다음과 같다. 코틀린 파일을 새로 생성하고 작성한 후 실행해보자.

```
fun main(args: Array<String>) {
    whenUsage(2, 50, "서울시")
}

fun whenUsage(inputType: Int, score: Int, city: String) {
    //---------- #1
    when (inputType) {
        1 -> println("타입-1")
        2, 3 -> println("타입-2,3")
        else -> {
            println("미확인")
```

```
        }
    }

    //---------- #2
    when (inputType) {
        checkInputType(inputType) -> {
            println("타입 정상")
        }
        else -> print("타입 비정상")
    }

    //---------- #3
    val start = 0
    val end = 100
    when (score) {
        in start..(end / 4) -> println("우수함")
        50 -> println("평균임")
        in start..end -> println("범위에 있음")
        else -> print("범위를 벗어남")
    }

    //---------- #4
    val isSeoul = when (city) {
        is String -> city.startsWith("서울")
        else -> false
    }
    println(isSeoul)

    //---------- #5
    when {
        city.length == 0 -> println("도시명을 입력하세요")
        city.substring(0, 2).equals("서울") -> println("서울이군요")
        else -> println(city)
    }
}

fun checkInputType(inputType: Int): Int {
    if (inputType >= 1 && inputType < 3) {
        return inputType;
    }
    return -1
}
```

이 코드에서는 when 문의 다양한 사용 예를 보여준다. 그리고 모든 when 문을 whenUsage( ) 함수에 넣어서 실행하기 편하게 하였다. 따라서 main( ) 함수에서 whenUsage( )를 호출하면 모든 코드를 테스트할 수 있다. 그리고 이때 우리가 필요한 인자의 값을 전달하면 된다.

//로 표시된 주석 #1에 있는 when에서는 변수의 값을 검사한 후 일치하는 값일 때 우리가 원하는 코드를 실행시키는 예를 보여준다. 이때 "값 -> 실행 코드 블록"의 형태로 작성한다. 또한 쉼표(,)를 사용하여 여러 개의 값을 한꺼번에 검사할 수도 있다. 그리고 첫 번째 값이 일치하지 않으면 차례대로 다음 값을 확인하며 모두 일치하지 않으면 else ->의 코드를 실행한다. 단, 일치하는 값이 있으면 그것에 지정된 코드 블록을 실행한 후 when 문을 벗어난다(자바의 switch-case 문에서는 break;가 있어야 함)는 것을 알아 두자. inputType의 값이 2이므로 여기서는 "타입-2,3"이 출력된다.

주석 #2의 when에서는 함수를 실행한 결과를 확인할 수도 있음을 보여준다. 여기서는 제일 끝의 checkInputType( ) 함수를 실행한 후 반환된 값을 검사하여 처리한다. inputType의 값이 2이므로 checkInputType( ) 함수에서도 2가 반환되어 "타입 정상"이 출력된다.

주석 #3의 when에서는 in 연산자와 범위 연산자(..)를 같이 사용하여 원하는 범위의 값에 있는지 검사한 후 처리한다. 여기서는 score 값이 50이므로 "평균임"이 출력된다.

주석 #4의 when에서는 is 연산자를 사용하여 city 변수의 타입이 String인지 확인한다. 그리고 String이면 문자열의 맨 앞이 "서울"로 시작하는지 확인한 후 그 결과를 isSeoul 변수에 넣는다. 이처럼 변수의 대입문에도 when을 사용할 수 있다. 여기서는 "서울"이 맞으므로 true 값이 isSeoul 변수에 지정되고 결과로 출력된다. 이때 isSeoul 변수의 타입은 코틀린 컴파일러가 Boolean 타입으로 추론해준다.

마지막으로 주석 #5의 when에서는 String 클래스의 함수들을 사용하여 city 변수의 문자열 값을 검사하는 예를 보여준다. city.length는 문자열의 길이를 반환하며, city.substring(0, 2).equals("서울")에서는 문자열의 맨 앞부터 두 글자를 추출한 후 "서울"과 같은 지 비교한다. 여기서는 "서울이군요"가 출력된다.

이 코드를 보면 알 수 있듯이, 코틀린의 when은 자바의 switch-case보다 간결하고 기능도 다양하다.

코틀린에서는 when도 if처럼 표현식으로 간주된다. 따라서 if로 했듯이, 대입문의 오른쪽에 사용될 수 있다.

B.6.3 for 루프

for 루프에서는 in 연산자를 사용하여 반복 처리한다. 사용 예를 보면 다음과 같다. 앞의 main( ) 함수 끝에 다음 코드를 추가하고 실행해보자.

```
//---------- #1
val item1 = listOf("사과", "바나나", "키위")
for (item in item1) {
    println(item)
}

//---------- #2
val item2 = listOf("사과", "바나나", "키위")
for (index in item2.indices) {
    println("item at $index is ${item2[index]}")
}

//---------- #3
val item3 = arrayOf("사과", "바나나", "키위")
for (index in item3.indices) {
    println("item at $index is ${item3[index]}")
}
```

주석 #1의 for에서는 세 개의 String 값을 저장한 List를 새로 생성한다. 이때 listOf( ) 함수를 사용하면 이처럼 쉽게 List 객체를 생성할 수 있다. 단, 이렇게 생성된 List는 읽기만 가능하고 저장된 값을 변경할 수 없다. 그다음에 for를 사용하여 List의 각 요소 값을 하나씩 출력한다.

주석 #2의 for에서도 listOf( ) 함수를 사용해서 List를 생성한 후 for를 사용하여 List의 각 요소 값을 하나씩 출력한다. 그러나 다른 점이 있다. List에 저장된 요소의 인덱스 값을 참조한다는 것이다. 이렇게 하려면 이 코드의 **item2.indices**처럼 데이터 값을 저장한 List(또는 다른 컬렉션이나 배열)의 indices 변수를 참조해야 한다(이 변수는 인덱스의 유효한 범위 값을 갖는다). 그다음에 for 루프 내부에서는 각 요소의 인덱스를 참조할 수 있다.

주석 #3의 for는 주석 #2의 for와 동일하게 실행된다. 단, List가 아닌 배열에 데이터가 저장된다는 것만 다르다. 코틀린에서 배열을 생성하는 방법은 여러 가지가 있지만, arrayOf( ) 함수를 사용하면 쉽게 배열을 생성할 수 있다.

이 코드를 컴파일하고 실행하면 그림 B-4처럼 Run 도구 창에 실행 결과가 출력될 것이다.

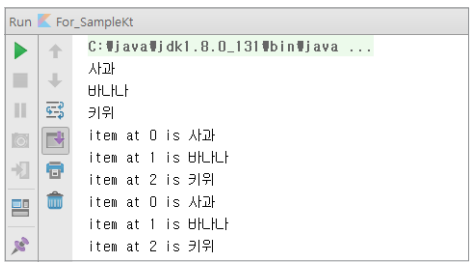


그림 B-4

for 루프에서 우리가 카운터(루프 반복 횟수)를 지정하고 사용할 때는 다음과 같은 방법으로 하면 된다.

```
for (i in 1..100) { ... }                // 1부터 100까지 반복, i 값은 1씩 증가
for (i in 1 until 100) { ... }           // 1부터 100까지 반복하되 100은 제외, i 값은 1씩 증가
for (i in 2..10 step 2) { ... }          // 2부터 10까지 반복하되, i 값은 2씩 증가
for (i in 10 downTo 1) { ... }           // 10부터 시작하여 i 값을 1씩 감소하면서 반복
```

B.6.4 while과 do-while 루프

while과 do-while 루프는 다른 프로그래밍 언어와 동일하다. 사용 예를 보면 다음과 같다. 앞의 main( ) 함수 끝에 다음 코드를 추가하고 실행해보자.

```
    //---------- while
    val items = listOf("사과", "바나나", "키위")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }

    //---------- do - while
    index = 0
    do {
        println("item at $index is ${items[index]}")
        index++
    } while (index < items.size)
```

이 코드에서도 앞의 for와 동일하게 listOf( ) 함수를 사용해서 List를 생성한다. 그리고 while 루프와 do-while 루프에서 List의 모든 요소를 출력한다. 여기서 for 루프와 다른 점은 List의 인덱스를 참조하는 변수를 우리가 생성하고 그 값을 1씩 증가 시키면서(따라서 index 변수를 val이 아닌 var로 선언해야 한다) 요소를 가져온다는 것이다. 이때 요소의 개수를 갖고 있는 items.size 변수를 참조하여 루프를 끝낸다.

while은 루프를 시작하기 전에 조건을 검사하며, do-while은 일단 한 번 실행한 후 조건을 검사한다는 것이 다르다.

B.7 함수

B.7.1 함수 선언과 호출

이번 장 앞에서 이미 알아보았지만, 코틀린에서는 함수를 선언할 때 fun 키워드를 함수 이름 앞에 지정한다. 그리고 함수는 소스 코드 파일의 어디든 바로 정의할 수 있다(또는 클래스 내부의 멤버로 정의할 수도 있다). 또한 함수의 매개변수는 **변수이름:타입**의 형태로 지정한다. 함수에서 반환하는 값의 타입은 함수 정의 문장 끝에 콜론을 추가한 후 지정한다. 함수 선언의 예를 들면 다음과 같다.

```
fun min(valueLeft: Int, valueRight: Int): Int {
    return if(valueLeft < valueRight) valueLeft else valueRight
}
```

여기서 min은 함수 이름이며, 괄호 안의 (valueLeft: Int, valueRight: Int)는 매개변수이다. 또한 세미콜론(:) 다음에 지정한 Int는 이 함수의 반환 타입이다. 만일 이 함수에서 값을 반환하지 않는다면 Int 대신 Unit 타입으로 지정한다(자바의 void). 그러나 Unit 타입은 생략해도 되며, 이 때는 함수 내부에서 return 문도 생략할 수 있다.

그리고 중괄호({}) 안에 있는 코드는 함수의 몸체이며 함수가 호출되면 실행된다.

return if(valueLeft < valueRight) valueLeft else valueRight는 다음 코드와 동일하다.

```
if(valueLeft < valueRight) {
    rerurn valueLeft
} else {
    return valueRight
}
```

코틀린에서는 if를 명령문보다는 표현식으로 간주하므로, return 문이나 대입문 등에 바로 추가할 수 있다(표현식은 값을 산출하여 반환하는 것이기 때문이다).

이와 더불어 앞의 함수 선언을 다음과 같이 더 축약된 형태로 할 수도 있다.

```
fun min(valueLeft: Int, valueRight: Int) =
                                if(valueLeft < valueRight) valueLeft else valueRight
```

여기서는 min( ) 함수의 반환 타입이 생략되어 있다. 그러나 컴파일 에러가 생기지 않는다. 왜냐하면, 오른쪽의 if 표현식 결과가 Int 타입이므로, 함수의 반환 타입이 Int라는 것을 코틀린 컴파일러가 추론해 주기 때문이다.

함수의 매개변수는 **변수이름:타입**의 형태로 지정하며, 두개 이상일 때는 쉼표(,)로 구분한다. 그리고 코틀린에서는 **지명 인자**(named argument)도 지원한다. 즉, 함수를 호출할 때 각 매개변수의 값만 전달하는 것이 아니고 매개변수 이름과 값을 같이 지정할 수 있다는 의미이다. 따라서 매개변수의 의미를 알기 쉽게 해줄 수 있다.

또한 매개변수의 기본값을 정의할 수도 있으며, 기본값에는 표현식(수식이나 다른 함수 호출 등)을 지정할 수 있다. 예를 들면 다음과 같다.

```
fun min(valueLeft: Int = 0, valueRight: Int = 0) =
                                if(valueLeft < valueRight) valueLeft else valueRight
```

이 경우 min( ) 함수를 호출할 때 매개변수의 모두 또는 일부를 생략하면 생략된 매개변수의 값이 0으로 지정되어 처리된다.

따라서 min( ) 함수는 다음 여섯 가지 형태로 호출할 수 있다.

```
min(100, 50)
min(100)                                    // valueRight 매개변수는 기본값인 0이 됨
min()                                       // 두 매개변수 모두 기본값인 0이 됨
min(valueLeft=50, valueRight=100)           // 지명 인자를 사용해서 값을 전달
min(valueRight=100, valueLeft=50)           // 지명 인자를 사용하면 정의된 순서와 무관하게 값을 전달
min(valueLeft=50)                           // 지명 인자와 기본값을 모두 사용하여 호출됨
```

코틀린에서는 함수의 매개변수로 다른 함수를 전달하여 실행시킬 수 있다. 이 내용은 "B.9.4 상위차 함수"에서 설명한다.

B.7.2 가변 인자

함수를 호출할 때 인자의 개수를 가변적으로 전달할 수 있다. 이때는 vararg 키워드를 사용한다(자바에서는 타입명…). 그리고 가변 인자들은 배열로 전달된다. 예를 들어, 다음 코드를 보자.

```
fun <T> newList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts)   // ts는 배열이다
        result.add(t)
    return result
}
```

우리가 정의한 newList( ) 함수는 가변 인자를 배열로 받은 후 컬렉션(여기서는 List의 일종인 ArrayList)을 생성하고 각 인자를 저장한 후 반환한다. 단, 반환되는 ArrayList는 val로 지정되어 있으므로 변경할 수 없다. 이 함수는 다음과 같이 호출하여 간편하게 컬렉션을 만들어 사용할 때 편리하다.

```
val list = newList(1, 2, 3)
```

앞의 코드에서 <T>는 제네릭 타입을 나타내며, newList( ) 함수를 호출할 때 인자에 지정된 타입으로 대체되며, ArrayList에 저장되는 요소의 타입을 나타낸다. 따라서 어떤 타입의 객체를 인자로 지정하더라도 이 함수를 사용할 수 있다.

또한 newList( ) 함수를 호출할 때 각 요소와 함께 배열도 같이 전달할 수 있다. 이때는 배열명 앞에 * 연산자만 붙여주면 된다. 예를 들면 다음과 같다.

```
val e = arrayOf(7, 8, 9)                    // Int 타입의 요소 세 개를 저장한 배열을 생성한다
val list = newList(1, 2, 3, *e, 5)
println(list)
```

두 번째 줄의 코드에서는 1, 2, 3, 7, 8, 9, 5의 요소를 저장한 ArrayList를 생성한다. 여기서 *e는 배열의 요소를 하나씩 가져와서 인자로 전달하라는 의미이며, *를 확산(spread) 연산자라고 한다.

앞의 newList( ) 함수 코드를 코틀린 파일(.kt)의 main( ) 함수 밖에 추가하고 바로 위의 코드 세 줄을 main( ) 함수 안에 추가한 후 실행해 보면 [1, 2, 3, 7, 8, 9, 5]가 출력될 것이다.

여기서 정의한 newList( ) 함수와 동일한 기능을 수행하는 listOf( ) 함수가 코틀린 표준 라이브러리에도 있다. 따라서 우리 코드에서 배열을 간단하게 생성할 때는 arrayOf( ) 함수를 사용하고, List 컬렉션을 생성할 때는 listOf( ) 함수를 사용하면 편리하다.

B.7.3 함수의 종류

함수는 어떤 범위(scope)로 사용하는가에 따라 몇 가지 종류로 구분할 수 있다. 첫 번째로, 코틀린에서는 함수를 클래스 외부에 정의할 수 있다. 즉, 소스 코드 파일(.kt)에 바로 정의하여 사용한다는 의미이다. 이것을 **최상위 수준**(top level) 함수라고 한다. 이런 함수는 애플리케이션에서 공통으로 사용하는 기능을 처리하는데 필요하다. 반면에 자바의 경우는 모든 것이 클래스에 정의되어야 하므로 그렇게 할 수 없다. 따라서 특정 클래스에 static 메서드로 모아 두고 사용해야 한다. 코틀린에 최상위 수준 함수가 있는 이유가 그 때문이다. 이번 장의 맨 앞에서 작성했던 makeMessage1( )과 makeMessage2( ) 함수가 최상위 수준 함수의 예이다.

부록A에서 알아보았듯이, 코틀린에서는 .kt 파일을 컴파일하여 JVM의 .class 파일로 생성한다. 그리고 .kt 파일에 선언한 최상위 수준 함수는 기본적으로 자바의 public static final 메서드가 된다. 따라서 최상위 수준 함수를 다른 .kt 파일 코드에서 사용할 때는 import 문으로 해당 함수의 위치(패키지이며 B.11에서 알아본다)를 지정한 후 호출하면 된다.

최상위 수준 함수와 더불어, 최상위 수준의 속성(property)도 사용할 수 있으며, 이것 역시 .kt 파일의 클래스 외부에 선언한다(다른 언어의 전역 변수와 유사함). 이 속성은 애플리케이션 데이터를 따로 유지하므로 사용하는 것이 바람직하지 않지만, 애플리케이션 전체에서 사용하는 상수를 정의할 때는 유용하다. 그리고 이때는 const 키워드를 지정한다. 예를 들면 다음과 같다.

```
const val ERROR_INVALID_CUSTOMER 1
```

이 경우 코틀린 컴파일러가 다음과 같이 자바 코드로 생성한다.

```
public static final Int ERROR_INVALID_CUSTOMER = 1;
```

두 번째로, 함수는 클래스 내부에 정의하여 사용할 수 있다. 이것을 멤버(member) 함수라고 한다(자바의 인스턴스 메서드). 예를 들면 다음과 같다.

```
class Customer() {
    fun checkID() { .. }
}
Customer().checkID()
```

여기서 checkID( )는 멤버 함수이며, Customer 클래스의 인스턴스를 생성(Customer( ))하여 호출할 수 있다(클래스는 B.8에서 알아본다).

세 번째로, 지역(local) 함수를 선언하여 사용할 수 있다. 이것은 다른 함수 내부에 포함된 함수이다. 사용 예를 보면 다음과 같다. (IntelliJ/IDEA에서 src 폴더 밑에 새로운 코틀린 파일을 생성한 후 아래 코드를 입력하고 실행해보자.)

```
fun main(args: Array<String>) {
    println(calcCombination(45, 6))       // 로또 복권의 모든 조합 가능한 번호 개수를 출력
}

fun calcCombination(whole: Int, selected: Int): Double {
    if ((selected > whole) || (selected <= 0) || (whole <= 0)) {
        return -1.0
    } else if (selected == whole) {
        return 1.0
    }

    fun calcFactorial(num: Int): Double {
        var total: Double = 1.0
        for (i in num downTo 1) {
            total *= i
        }
        return total
    }

    return calcFactorial(whole) /
            (calcFactorial(whole - selected) * calcFactorial(selected))
}
```

이 코드에서는 조합의 값을 구하는 calcCombination( ) 함수를 정의하고 사용하며, 이 함수 내부에서는 지역 함수로 calcFactorial( )을 정의하고 사용한다. 조합의 값을 구하려면 계승 값을 세번 계산해야 한다. 따라서 여기서는 지역 함수를 정의하여 계승 값을 계산하였다. 그리고 main( ) 함수에서는 ${}_{45}C_6$(45개에서 6개를 조합하는 경우의 수)의 결과를 출력한다. 이것은 로또 복권의 모든 조합 가능한 번호 개수이며 8,145,060이다. 따라서 1등에 당첨될 확률은 1/8,145,060이다.

지역 함수에서는 자신을 포함하는 외부 함수의 인자나 변수를 그냥 사용할 수 있다.

네 번째로, 제네릭(generic) 함수를 선언하고 사용할 수 있다. 예를 들어, B.7.2에서 작성했던 다음 함수를 보자.

```
fun <T> newList(vararg ts: T): List<T> {
```

여기서 <T>가 제네릭 타입이다. 즉, List에 저장할 요소의 타입을 미리 정하지 않고 함수 호출 코드에서 타입을 지정할 수 있다는 의미이다. 따라서 이 함수는 다양한 타입의 인자를 전달하여 호출할 수 있으며, 함수가 실행된 후에는 지정된 타입의 객체들을 요소로 저장한 List가 생성되어 반환된다.

다섯 번째로, **확장**(extension) 함수를 선언하고 사용할 수 있다. 이것은 특정 클래스로부터 상속받지 않고 해당 클래스의 기능을 확장할 때 사용한다. 예를 들어, 코틀린의 컬렉션 중에는 MutableList가 있으며, 이것은 저장된 요소를 변경할 수 있는 List를 나타낸다. 그리고 mutableListOf( ) 함수를 사용하면 MutableList 객체를 쉽게 생성할 수 있다. 이때 저장된 요소를 바꿔치기 하는 기능의 swap( ) 함수를 추가할 필요가 있다면, MutableList로부터 상속받는 서브 클래스를 정의하고 멤버 함수를 추가하면 된다. 그러나 확장 함수를 사용하면 그렇게 하지 않아도 쉽게 기능을 추가할 수 있다. 예를 들면 다음과 같다.

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]                  // 여기서 'this'는 현재의 MutableList객체를 의미한다
    this[index1] = this[index2]
    this[index2] = tmp
}
```

이 코드에서 보듯이, 확장 함수를 정의할 때는 함수 이름 앞에 대상이 되는 클래스를 지정해야 한다. 그리고 여기서는 어떤 타입의 요소든 MutableList에 저장할 수 있도록 제네릭 타입을 <T>로 지정하였다. 또한, 함수 내부에서 사용하는 this 키워드는 현재 사용 중인 MutableList 객체를 참조한다.

확장 함수가 정의되었으므로, 이제는 MutableList를 생성하고 다음과 같이 확장 함수인 swap( )을 호출할 수 있다.

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2)
```

여기서는 세 개의 Int 타입 요소를 저장한 MutableList를 생성한 후 첫 번째 요소와 세 번째 요소를 바꿔치기 한다(첫 번째 요소의 위치는 0부터 시작한다).

여섯 번째로, 중위(infix) 함수가 있다. 이 함수는 이항 연산자처럼 두 개의 피연산자 사이에 함수 호출을 넣는다. 이것은 새로운 부류의 함수라기 보다는 사용을 편리하게 하기 위해 코틀린에 추가된 기능이다. 대표적인 사용 예가 B.4.4에서 알아보았던 비트 연산자이다. 비트 연산자의 한 가지 사용 예를 다시 보면 다음과 같다.

```
println(8 shr 2)
```

여기서는 정수값인 8의 각 비트를 오른쪽으로 2비트 이동시켜 출력하므로 결과는 2(8/2^2)이다. 이처럼 중위 함수를 사용하면 마치 연산자처럼 사용할 수 있으므로 편리하다. 중위 함수를 정의할 때는 fun 키워드 앞에 infix 키워드를 추가한다. 실제로 코틀린의 Int 클래스에는 다음과 같이 shl( )이 중위 함수로 정의되어 있다.

```
infix fun shl(bitCount: Int): Int {
    ...
}
```

따라서 정수 타입의 값에 대해 이 함수를 다음 두 가지 방법으로 호출하여 사용할 수 있다.

```
8 shl 2
```

또는

```
8.shl(2)
```

첫 번째 코드는 중위 함수의 호출 형태이고, 두 번째 코드는 일반적인 함수 호출 형태이다. 여기서 8은 정수 타입의 값을 갖는 변수나 표현식 중 어느 것으로도 대체 가능하며 예를 들면 다음과 같다.

```
val a = 8
println(a+8 shr 2)
```

여기서는 4가 결과로 출력된다. 그리고 a+8은 (a+8)로 괄호를 씌우지 않아도 된다. shr( ) 함수 호출보다 + 연산자의 우선순위가 높기 때문이다.

중위 함수를 정의하려면 세 가지 조건을 충족해야 한다. 첫 번째로, 클래스의 멤버 함수이거나 확장 함수이어야 한다. 두 번째는, 매개변수가 한 개여야 한다. 세 번째는, infix 키워드로 함수가 정의되어야 한다.

일곱 번째로, 재귀(tail recursive) 함수가 있다. 일련의 코드를 반복해서 실행해야 하는 알고리즘의 경우는 루프를 사용하거나 재귀 함수를 사용할 수 있다(여기서 tail은 꼬리나 끝부분을 말하며, 재귀 함수의 몸체 코드가 실행되고 return하기 전에 다시 반복 수행한다고 해서 붙여진 것이다). 단, 재귀 함수를 사용할 때는 스택 오버플로의 위험이 따르고 시스템의 부담도 커진다. 따라서 코틀린에서는 이런 단점을 줄이고 실행 속도도 빠르며 효율적인 루프를 사용하는 재귀 함수를 제공한다. 이때는 다음 예와 같이 fun 키워드 앞에 tailrec 키워드를 붙인다.

```
tailrec fun calcFactorial1(num: Int): Double {
    if (num == 1) {
        return 1.0
    }
    else {
        return (num * calcFactorial1(num - 1)).toDouble()
    }
}
```

이 재귀 함수에서는 계승 값을 구해준다. 예를 들어, println(calcFactorial1(5))를 실행하면 120이 출력된다(5*4*3*2*1). 여기서 tailrec 키워드를 제거해도 마찬가지로 재귀 함수로 사용할 수 있지만 비효율적이므로 그렇게 하지 않는 것이 좋다.

반복 루프를 사용해서 이와 동일한 기능을 수행하는 함수를 작성하면 다음과 같다.

```
fun calcFactorial2(num: Int): Double {
    var total: Double = 1.0
    for (i in num downTo 1) {
        total *= i
    }
    return total
}
```

println(calcFactorial2(5))를 실행하면 재귀 함수와 동일하게 120.0이 출력된다.

마지막으로, 상위차(higher-order) 함수를 선언하고 사용할 수 있다. 이것은 다른 함수나 람다식(lambda)을 인자로 받아 실행시키거나 반환할 수 있는 함수이다. 또한 이름이 없는 익명

(anonymous) 함수를 사용할 수 있으며, 성능을 개선하기 위해 인라인(inline) 함수를 사용할 수 있다. 이 함수들은 "B.9 람다식"에서 설명한다.

B.8 클래스와 인터페이스

여기서는 객체지향의 기본 개념은 설명하지 않는다. 따라서 여러분이 자바나 C++와 같은 객체지향 프로그래밍 언어를 사용해 본 경험이 있다고 간주하고 코틀린에서는 어떤 식으로 객체지향 개념을 구현하는지 알아볼 것이다. 코틀린은 JVM에서 실행되는 코드(바이트 코드 또는 클래스 파일이라고 함)를 생성하므로 자바와 유사한 면도 있지만 키워드나 방식이 다른 점도 많고 새로 추가된 좋은 기능들도 있으므로 차이점을 알아 둘 필요가 있다.

B.8.1 클래스 선언과 생성자

클래스는 class 키워드를 사용하여 선언하며, 자신의 멤버 속성과 멤버 함수를 가질 수 있다. 예를 들어, 친구(Friend)를 나타내는 클래스를 다음과 같이 선언할 수 있다(자바처럼 클래스 이름의 첫 자는 대문자로 한다).

```
class Friend { }
```

또는

```
class Friend
```

이것은 가장 간단한 클래스 선언이다. Friend 클래스는 자신의 멤버 속성이나 함수를 갖고 있지 않으므로 이때는 클래스 몸체를 나타내는 중괄호({})도 생략할 수 있다. 그러나 이런 식의 클래스를 선언할 일은 없을 것이다. Friend 클래스에 이름, 전화번호, 친구 유형의 세 가지 속성이 필요하다고 할 때 이 클래스를 정의하는 다양한 방법을 알아보자. src 폴더에 새로운 코틀린 파일을 생성한 후 다음 코드를 작성한다.

```
fun main(args: Array<String>) {
    val f1 = Friend1()
    f1.name = "박문수"
    f1.tel = "010-123-4567"
    f1.type = 5
    println(f1.name+","+f1.tel+","+f1.type)
```

```
    val f2 = Friend2()
    f2.name = "홍길동"
    f2.tel = "010-456-1234"
    f2.type = 5                  // 이 속성에 지정한 setter인 set() 함수가 호출되어 실행된다.
    println(f2.name+","+f2.tel+","+f2.type)

    val f3 = Friend3(name = "김선달", tel = "010-345-6789", type = 5)
    println(f3.name+","+f3.tel+","+f3.type)

    val f4 = Friend4("전신주", "010-333-5555", 3)
    println(f4.name+","+f4.tel+","+f4.type)
}

class Friend1 {
    var name: String = ""
    var tel: String = ""
    var type: Int = 4            // 1: "학교", 2: "회사" 3: "SNS" 4: "기타"
}

class Friend2 {
    var name: String = ""
    var tel: String = ""
    var type: Int = 4            // 1: "학교", 2: "회사" 3: "SNS" 4: "기타"
        set(value: Int) {
            if(value < 4) field = value   // 1: "학교", 2: "회사" 3: "SNS" 4: "기타"
            else  field = 4
        }
}

class Friend3 (var name: String, var tel: String, var type: Int)
{
    init {
        type = if(type < 4) type else 4   // 1: "학교", 2: "회사" 3: "SNS" 4: "기타"
    }
}

class Friend4 {
    var name: String
    var tel: String
    var type: Int

    constructor(name: String, tel: String, type: Int) {
        this.name = name
        this.tel = tel
        this.type = if(type < 4) type else 4  // 1: "학교", 2: "회사" 3: "SNS" 4: "기타"
    }
}
```

이 코드의 Friend1부터 Friend4까지의 네 개 클래스는 친구를 나타내는 동일한 클래스이다. 단, 정의하는 방법과 생성자(constructor)가 다를 뿐이다.

우선, Friend1 클래스에서는 자바처럼 속성(property)을 정의한다. 그러나 생성자를 정의하지 않았으므로 속성을 직접 초기화해 주어야 한다. 그리고 이처럼 아무 생성자도 정의하지 않으면 매개변수도 없고 아무 일도 하지 않는 기본 생성자가 자동으로 생성된다(이 점은 자바와 같다). 또한 속성도 변수처럼 val(값이 변경 안됨)이나 var(값을 변경 가능)로 지정할 수 있다.

여기서 사전에 알아 둘 것이 있다. 클래스에 지정된 멤버 변수는 값을 갖는 필드(field)이다. 그리고 필드의 값을 읽는 것을 getter라고 하며 변경하는 것을 setter라고 한다. 자바에서는 필드와 getter/setter를 별도로 처리하지만(우리가 지정하고 호출한다), 코틀린의 속성은 내부적으로 필드와 (자동 생성된) getter/setter가 연계되어 동작하도록 언어 자체에 배려되어 있다. 즉, 속성의 값을 읽거나 변경할 때 자동으로 getter/setter가 호출된다는 의미이다. 따라서 우리가 getter/setter를 별도로 정의하지 않아도 되지만, 필요하다면 커스텀 getter/setter를 정의할 수 있다.

예를 들어, 여기서는 type 속성이 친구의 유형을 나타내며, 1부터 4까지의 값만 지정되어야 한다. 그러나 Friend1 클래스의 인스턴스를 생성하면 어떤 값이든 지정 가능하다. 따라서 type 속성의 값이 1부터 4까지만 변경 가능하도록 하려면 이 속성의 커스텀 setter를 지정해야 한다.

따라서 Friend2 클래스에서는 type 속성의 커스텀 setter를 정의하였다. 이때 코틀린에서는 자바와 다르게 해당 속성에 set( ) 함수로 지정한다. 그리고 set( ) 함수 내부에서 해당 속성을 액세스하려면 **field** 키워드를 사용해야 한다. type 속성의 set( ) 함수는 f2.type = 5처럼 속성을 액세스할 때 자동 호출된다.

field 키워드에 대해 조금 더 알아보자. 우리는 Friend2 클래스에 필드처럼 type이라는 이름의 속성을 정의하고 사용하면 된다. 그러나 코틀린에서는 내부적으로 속성 이름 앞에 밑줄(_)을 추가한 _type이라는 private 필드를 생성하고 값을 보존한다. 이것을 **후원 필드**(backing field)라고 하며, 멤버 변수로 사용된다. 그러나 _type을 우리가 코드에서 직접 액세스할 수는 없으므로 대신에 field 키워드를 사용하는 것이다. 이렇게 하는 이유는 코틀린 언어 자체에서 클래스의 캡슐화를 지원하기 위해서다.

또한 코틀린에서는 클래스의 인스턴스를 생성할 때 마치 함수를 호출하듯이 코드를 작성한다. 예를 들면, 앞의 main( ) 함수에서 val f1 = Friend1( )이나 val f3 = Friend3(name = "김선달", tel = "010-345-6789", type = 5)와 같이 한다(자바의 new 키워드를 사용하지 않는다).

Friend3 클래스에서는 자바와 코틀린의 또 다른 점을 보여준다. 자바에서는 기본 생성자와 오버로딩된 생성자 모두 클래스 몸체({} 내부)에 정의한다. 그러나 코틀린에서는 클래스 헤더(class

키워드부터 몸체의 시작을 나타내는 { 전까지) 안에 기본 생성자를 정의한다. 그리고 만일 생략하면 매개변수나 실행 코드가 없는 기본 생성자를 코틀린 컴파일러가 자동으로 생성해준다. 또한 앞의 다음 코드는,

```
class Friend3 (var name: String, var tel: String, var type: Int)
```

아래의 코드와 동일하다.

```
class Friend3 constructor(var name: String, var tel: String, var type: Int)
```

코틀린에서는 constructor 키워드로 생성자를 나타내기 때문이다. 그러나 기본 생성자에서는 편의상 이 키워드를 생략할 수 있다. 단, private과 같은 접근 제한자를 constructor 앞에 지정할 때는 생략할 수 없다(private을 지정하면 생성자로 클래스 인스턴스를 생성할 수 없으므로, 인스턴스 생성을 우리가 제어할 수 있다).

Friend3 클래스에서는 이 클래스의 모든 속성을 기본 생성자에 정의하고 있다. 이 경우 매개변수 이름과 같은 이름의 속성이 정의된다. 기본 생성자에 정의된 속성의 값을 초기화할 때 별도의 처리가 필요한 경우에는 init 키워드로 정의한 블록에서 하면 된다. 이것을 **초기화 블록**이라 한다. 그리고 이렇게 하면 main( ) 함수의 다음 코드에서 Friend3 인스턴스를 생성할 때 기본 생성자의 init 코드 블록이 실행되어 생성자 인자로 전달된 type 속성의 값을 검사하게 된다.

```
val f3 = Friend3(name = "김선달", tel = "010-345-6789", type = 5)
```

또한 여기서는 지명 인자(named argument)를 사용해서 생성자의 인자를 전달한다(B.7.1 참조). 그러나 다음과 같이 인자의 이름 없이 호출해도 된다.

```
val f3 = Friend3("김선달", "010-345-6789", 5)
```

코틀린 클래스에서도 자바처럼 기본 생성자 외에 추가로 보조 생성자를 가질 수 있다. 이때는 클래스 몸체 내부에 constructor 키워드로 정의해야 한다.

Friend4 클래스에서는 기본 생성자는 정의하지 않고 보조 생성자를 정의하였다. 따라서 Friend4 클래스의 인스턴스를 생성할 때는 앞의 main( ) 함수에 있는 다음 코드와 같이 보조

생성자를 사용해야 한다. 보조 생성자가 있을 때는 기본 생성자가 자동으로 생성되지 않기 때문이다.

```
val f4 = Friend4("전신주", "010-333-5555", 3)
```

그리고 보조 생성자에서는 this 키워드를 사용해서 현재 생성되는 객체의 속성을 참조할 수 있다.

끝으로, 기본 생성자나 보조 생성자 중 어디에서 속성을 정의하더라도 val 또는 var 모두 사용할 수 있다. 물론 val로 지정하면 한번 초기화한 후 값을 변경할 수 없다.

코틀린에서 클래스의 선언과 생성자 및 인스턴스를 생성하는 방법을 이제는 알았을 것이다. 또한 속성과 getter/setter에 대해서도 이해했을 것이다. 자바와는 다른 점이 많으니 잘 파악해 두자.

B.8.2 멤버 함수

클래스 내부의 멤버로는 앞에서 알아 본 속성과 생성자 및 초기화 블록을 가질 수 있다. 그러나 이외에도 멤버 함수와 중첩 또는 내부 클래스 및 다른 객체를 멤버로 포함할 수 있다. 우선, 멤버 함수에 대해 간략하게 알아보자.

클래스 내부에 정의된 함수는 멤버 함수로 간주된다. 따라서 해당 클래스의 인스턴스를 생성하여 호출할 수 있다. 정의하는 방법은 기본적으로 일반 함수와 동일하지만, 클래스 상속과 관련된 접근 제한자를 추가로 지정할 수 있다(이 내용은 B.8.3에서 설명한다). 예를 들어, 앞의 코드에서 Friend1 클래스에 친구의 이름을 출력하는 멤버 함수를 정의한다면 다음과 같이 할 수 있다.

```
class Friend3 {
..
    fun printName():Unit = println(this.name)
}
```

여기서 println( ) 함수는 반환값이 없으므로 :Unit를 지정했지만 생략해도 된다. 그리고 main( ) 함수에서는 다음과 같이 호출할 수 있다.

```
val f3 = Friend3("김선달", "010-345-6789", 5)
f3.printName()
```

이처럼 멤버 함수는 객체 참조에 점(.)을 붙여 호출한다.

코틀린 클래스의 멤버 속성과 멤버 함수는 기본적으로 public 가시성을 가지므로 어디서든 사용할 수 있다. 그러나 접근 제한자를 지정하면 사용 가능한 범위를 제한할 수 있다.

B.8.3 접근 제한자

코틀린에서는 클래스, 클래스 멤버, 최상위 수준 변수와 함수에 다음의 접근 제한자를 사용할 수 있다. 우선, 사용 가능한 범위를 제한하는 접근 제한자는 다음과 같다.

접근 제한자	클래스 멤버일 때	최상위 수준으로 선언되었을 때
public(기본값)	어디서든 사용 가능	어디서든 사용 가능
internal	같은 모듈에서만 사용 가능	같은 모듈에서만 사용 가능
protected	서브 클래스에서만 사용 가능	해당 없음
private	클래스 내부에서만 사용 가능	코틀린 파일 내부에서만 사용 가능

우리가 접근 제한자를 지정하지 않으면 코틀린에서는 기본적으로 public으로 간주되므로 어디서든 사용 가능하게 된다. 그리고 자바의 경우는 .java 파일에 클래스만 지정할 수 있지만, 코틀린에서는 .kt 파일에 클래스는 물론 다른 변수나 함수도 정의할 수 있다. 이것을 최상위 수준 변수/함수라고 한다.

internal은 코틀린 특유의 접근 제한자이며, 같은 모듈(module) 내에서만 사용 가능함을 의미한다. 여기서 모듈은 두 개 이상의 코틀린 파일들이 같이 컴파일되어 생성된 것을 말한다. 예를 들어, 여러 개의 코틀린 파일들이 있는 IntelliJ IDEA 프로젝트나 이클립스 프로젝트를 컴파일하여 빌드하는 경우다.

코틀린에서도 자바처럼 package 키워드로 패키지를 생성할 수 있지만(B.11 참조), 패키지 수준의 접근 제한자는 없다. 같은 패키지에 있는 클래스와 함수끼리는 기본적으로 액세스가 가능하기 때문이다.

protected는 멤버가 정의된 클래스와 이 클래스의 서브 클래스에서만 사용 가능하며, 최상위 수준에는 적용되지 않는다. (같은 패키지의 다른 클래스와 서브 클래스 모두에서 사용 가능한 자바와는 다르니 주의하자.)

private은 해당 변수나 함수가 정의된 클래스나 코틀린 파일 내부에서만 사용 가능하다.

이와 더불어 클래스와 클래스 멤버에만 사용 가능한 접근 제한자 및 그 의미는 다음과 같다.

접근 제한자	클래스	클래스 멤버
final(기본값)	서브 클래스를 만들 수 없음	슈퍼 클래스의 멤버를 오버라이딩할 수 없음
open	서브 클래스를 만들 수 있음	슈퍼 클래스의 멤버를 오버라이딩할 수 있음
abstract	추상 클래스를 의미함	함수에만 해당되며 몸체(실행 코드)가 없음
override	해당 없음	슈퍼 클래스의 멤버를 오버라이딩함

이러한 접근 제한자를 지정하지 않으면 클래스나 클래스 멤버는 기본적으로 final이다. 따라서 클래스는 서브 클래스를 만들 수 없으며, 클래스 멤버는 상속 받은 슈퍼 클래스의 멤버를 오버라이딩할 수 없다(자바와는 정반대이다).

open은 final과 정반대이다. open을 지정한 클래스는 서브 클래스를 만들 수 있고, 클래스 멤버에 open을 지정하면 서브 클래스에서 오버라이딩할 수 있다(코틀린에서는 멤버 함수는 물론 속성도 오버라이딩할 수 있다.)

abstract를 지정하면 추상 클래스가 되며, 자바처럼 인스턴스를 생성할 수 없다. 대개는 추상 클래스의 멤버(속성이 아닌 함수만 해당됨)가 구현 코드가 없는 추상 함수지만, 구현 코드가 있는 멤버 함수도 같이 가질 수 있다. 또한 abstract 멤버 함수는 기본적으로 open이다. 따라서 추가로 open을 지정하지 않아도 된다. 추상 클래스와 추상 멤버 함수는 서브 클래스에서 상속받아 오버라이딩하여 구현하도록 하는 것이 목적이기 때문이다

서브 클래스에서 슈퍼 클래스의 멤버 속성이나 멤버 함수를 오버라이딩할 때는 반드시 override를 지정해야 한다. 그리고 이때 속성의 경우는 슈퍼 클래스의 속성과 타입이 같아야 하며, 함수의 경우는 슈퍼 클래스 함수의 시그니처(인자 개수와 타입 및 반환 타입)가 같아야 한다. 또한 오버라이딩된 서브 클래스의 함수는 public이 된다. 따라서 이 클래스에서 또 다시 상속받는 서브 클래스에서 오버라이딩하지 못하게 하려면 final과 override를 같이 지정하면 된다.

B.8.3 클래스 상속과 멤버 오버라이딩

클래스는 상속(inheritance)을 통해서 서브 클래스를 만들 수 있다. 이때 상속해 주는 바로 상위 클래스를 슈퍼 클래스라고 한다. 상속을 다르게 말하면, 서브 클래스가 슈퍼 클래스의 코드를 재사용하면서 슈퍼 클래스의 기능을 확장할 수 있는 것이다. 상속은 클래스 간의 관계를 나타내며, 여러 레벨에 걸쳐 이루어질 수 있다. 코틀린이나 자바의 표준 라이브러리에 있는

클래스들도 상속을 사용하여 모델링되어 있다. 코틀린에서는 다중 상속(multiple inheritance)을 지원하지 않으므로, 서브 클래스는 하나의 슈퍼 클래스로부터만 상속받을 수 있다.

코틀린의 모든 클래스는 기본적으로 Any라는 이름의 슈퍼 클래스로부터 상속 받게 되어있다(자바의 Object 클래스와 유사하다). Any 클래스는 코틀린의 모든 클래스에서 공통으로 필요한 세 개의 함수(equals( ), hashCode( ), toString( ))와 확장된 속성 및 함수를 추가로 갖고 있다.

클래스 간의 상속 관계를 코드로 나타낼 때는 간단하게 콜론(:)을 사용한다(자바는 extends). 예를 들면 다음과 같다.

```
open class Father(nameF: String, ageF: Int)
class Child(nameC: String, ageC: Int) : Father(nameC, ageC)
```

여기서 Child는 Father의 서브 클래스이다. Father 클래스는 이름(nameF)과 나이(ageF) 속성을 가지며 기본 생성자에서 초기화된다. Child 클래스는 Father로부터 상속받은 이름(nameC)과 나이(ageC) 속성을 가지며 마찬가지로 기본 생성자에서 초기화된다. 서브 클래스의 생성자에서는 슈퍼 클래스의 속성도 초기화해주어야 한다. 그리고 이 코드처럼 기본 생성자가 있을 경우에는 서브 클래스의 기본 생성자 인자를 슈퍼 클래스의 기본 생성자 인자로 전달한다. 따라서 콜론(:) 다음에 Father 클래스 이름만 지정하면 안되고 Father(nameC, ageC)로 해야 한다.

서브 클래스에 기본 생성자가 정의되지 않고 보조 생성자가 정의된 경우에는 서브 클래스의 보조 생성자에서 콜론(:)과 super 키워드를 사용하여 슈퍼 클래스의 속성을 초기화한다. 예를 들면 다음과 같다.

```
open class Father(nameF: String, ageF: Int)
class Child : Father {
    constructor(nameC: String, ageC: Int) : super(nameC, ageC)
}
```

서브 클래스에서는 슈퍼 클래스로부터 상속받은 속성과 함수 모두를 오버라이딩할 수 있다. 단, 이때는 슈퍼 클래스의 해당 속성과 함수를 open 접근 제한자로 지정해야 한다. 예를 들면 다음과 같다.

```
open class Father(nameF: String, ageF: Int) {
    open fun sleep() {}
    fun looseWeight() {}
}

class Child : Father {
    constructor(nameC: String, ageC: Int) : super(nameC, ageC)

    override fun sleep() {}
}
```

여기서는 Child 클래스에서 Father 클래스의 sleep( ) 함수를 오버라이딩한다. 이때 Farther 클래스에는 open을, Child 클래스에는 override를 지정해야 한다. 그러나 Father 클래스의 looseWeight( ) 함수는 open이 지정되지 않았으므로 Child에서 오버라이딩할 수 없다.

코틀린에서는 속성의 오버라이딩도 가능하며, 이때는 멤버 함수와 마찬가지로 슈퍼 클래스의 속성에는 open을, 서브 클래스의 속성에는 override를 지정해야 한다. 그리고 이런 속성은 기본 생성자에 정의하지 않고 클래스 몸체에 정의해야 한다.

슈퍼 클래스에 val로 지정된 속성은 서브 클래스에서 var로 변경하여 오버라이딩할 수 있다. 그러나 그 반대는 안 된다. 왜냐하면, 슈퍼 클래스의 속성을 val로 지정하면 getter가 자동 생성되므로 서브 클래스에서 var로 변경하여 오버라이딩하더라도 setter만 추가로 생성하면 되기 때문이다. 그러나 그 반대일 때는 슈퍼 클래스에 생성된 setter가 무의미해지기 때문이다.

B.8.4 인터페이스 구현과 오버라이딩

코틀린에서는 자바처럼 interface 키워드로 인터페이스를 정의한다. 인터페이스는 주로 추상 함수를 갖지만 구현 코드가 있는 함수도 가질 수 있다. 그리고 해당 인터페이스를 구현하는 추상이 아닌 일반 클래스에서는 반드시 그것의 추상 함수를 구현해야 한다. 또한 인터페이스에는 속성이 아닌 필드(getter/setter가 없고 값만 가짐)를 가질 수 있다.

인터페이스에는 final, open, abstract, override 키워드 모두 사용하지 않는다. 인터페이스와 그것에 정의된 추상 함수는 클래스에서 구현하기 위해 정의된 것이기 때문이다. 다음 예에서는 음악을 연주하는데 필요한 악기 속성과 연주 함수를 정의하는 PlayMusic 인터페이스가 있다. 그리고 전문가를 나타내는 Professional 클래스와 아마추어를 나타내는 Amateur 클래스 모두에서 이 인터페이스를 구현한다.

```
interface PlayMusic {
    val musicalInstrument: String
    fun play()
}

class Professional(override val musicalInstrument: String) : PlayMusic {
    override fun play() {}
}

class Amateur : PlayMusic {
    override var musicalInstrument: String = "Piano"
    override fun play() {}
}
```

인터페이스에는 추상 및 일반 함수와 필드를 정의할 수 있다. 인터페이스의 필드는 var과 val 모두 지정 가능하지만 클래스 속성과 달리 getter/setter가 생성되지 않는다. 또한 인터페이스에 val로 지정된 필드도 구현 클래스에서 var로 변경하여 오버라이딩할 수 있다.

인터페이스의 구현을 나타낼 때는 클래스처럼 콜론(:) 다음에 구현할 인터페이스를 지정한다. 그리고 구현 클래스에서는 구현할 속성이나 함수에 override 키워드를 지정해야 한다.

클래스에서 클래스 상속과 인터페이스 구현을 같이 할 때는 클래스 헤더의 콜론(:) 다음에 쉼표(,)를 사용해서 지정하면 된다. 예를 들면 다음과 같다.

```
open class MusicType {
    open fun sing() {}
}

interface PlayMusic {
    val musicalInstrument: String
    fun play()
}

class Professional(override val musicalInstrument: String) : MusicType(), PlayMusic {
    override fun play() {}
    override fun sing() {}
}
```

여기서는 Professional 클래스에서 MusicType 클래스로부터 상속받고 PlayMusic 인터페이스의 구현도 같이 한다. 그리고 만일 슈퍼 클래스에서 상속받는 함수와 인터페이스의 구현 함수 이름이 같을 때는 super<타입> 키워드를 사용해서 참조하면 된다. 예를 들어, play( ) 함수가

PlayMusic 인터페이스와 MusicType 클래스 모두에 정의되어 있다면, super<MusicType>.play( ) 또는 super<PlayMusic>.play( )로 해당 슈퍼 타입의 play( ) 함수를 호출할 수 있다.

B.8.5 추상 클래스와 오버라이딩

추상 클래스는 인터페이스처럼 추상 함수를 가지므로 이 클래스의 서브 클래스에서는 추상 함수를 반드시 오버라이딩해야 한다. 추상 클래스는 또한 일반 함수와 속성도 가질 수 있다. 추상 클래스는 주로 클래스 상속 계층에서 상위의 슈퍼 클래스를 정의할 때 사용한다.

앞의 접근 제한자에서 설명했듯이, 추상 클래스는 abstract 키워드로 지정하며 자바처럼 인스턴스를 생성할 수 없다. 바로 앞 코드의 PlayMusic 인터페이스를 추상 클래스로 변경한 코드는 다음과 같다.

```
abstract class PlayMusic {
    val musicalInstrument: String = "피아노"
    abstract fun play()
    fun sing() {}
}

class Professional() : PlayMusic() {
    override fun play() {}
}
```

추상 클래스의 속성은 기본적으로 final이므로, 서브 클래스에서 오버라이딩할 수 없다. 반면에 abstract로 지정된 추상 함수는 반드시 서브 클래스에서 오버라이딩해야 한다.

B.8.6 "object" 키워드

객체지향 프로그래밍에서는 현실 세계나 소프트웨어의 객체를 클래스로 추상화한 후(모델링) 이 클래스로부터 인스턴스(객체)를 생성하여 사용한다. 이렇게 하면 틀을 사용해서 붕어빵을 구워 내듯이 같은 타입의 객체를 원하는 대로 생성할 수 있다. 또한 서로 다른 부류의 클래스 간의 관계를 상속 개념으로 구현하여 코드를 재사용하고 기능을 확장할 수 있다.

그러나 때로는 클래스를 정의하여 인스턴스를 생성하지 않고 특정 객체 자체를 생성하여 사용하면 편리한 경우가 있다. 단, 이때는 인스턴스를 만드는 틀의 역할을 하는 클래스가 없기 때문에 그것과 같은 타입의 객체를 여러 개 생성할 수 없다. 그리고 이런 객체는 코드에 정의된 곳에 생성되어 동작하며, 다른 클래스 내부에 이런 객체를 포함하면 그 클래스의 인스턴스를 생성하지 않고 그 객체를 바로 사용할 수 있다.

코틀린에서는 "object" 키워드를 사용해서 지금까지 얘기한 객체를 쉽게 구현하고 사용할 수 있다. 이때 다음의 세 가지 방법이 있다. **객체 선언**(object declaration), **동반 객체**(companion object), **객체 표현식**(object expression)이다.

첫 번째로 객체 선언을 알아보자. 이것은 말 그대로 코드의 필요한 곳에서 클래스 대신 객체를 선언하고 생성하여 사용하는 방법이다. 따라서 하나의 인스턴스만 생성하고 공유하여 사용하는 **싱글톤**(singleton)을 구현할 때 좋은 방법이다. 만일 이런 객체를 클래스로 정의하고 매번 인스턴스를 생성하여 사용하게 한다면 메모리 낭비가 심하게 되고 시스템의 부담도 커지게 된다. 예를 들어, 애플리케이션이 실행되는 동안에 수시로 변하는 상태 정보를 메시지 번호와 내용으로 기록할 필요가 있다고 해보자. 실제 데이터는 데이터베이스 등에 저장되겠지만, 이런 일을 처리하는 객체가 필요하다. 이때 object 키워드 다음에 이 객체의 이름을 지정하고, 클래스처럼 중괄호({}) 안에 속성과 함수를 선언하면 된다. Kotlin01 프로젝트의 src 밑에 새로운 코틀린 파일을 생성하고 다음 코드를 작성해보자.

```
fun main(args: Array<String>) {
    for(i in 1..10) {
        StateManager.msgNumber += 1
        StateManager.msgContent = StateManager.msgNumber.toString() + "번째 메시지"
        StateManager.storeMessage()
    }
}

object StateManager {
    var msgNumber: Int = 0
    var msgContent: String = ""

    fun storeMessage() {
        // 데이터를 저장하는 코드
        println("메시지 번호 = " + msgNumber + ", 내용 = " + msgContent)
    }
}
```

여기서 StateManager는 싱글톤 객체이며, 메시지 번호와 내용을 속성으로 갖는다. 그리고 데이터를 저장하는 멤버 함수를 갖고 있다(여기서는 출력만 한다).

클래스처럼 객체 선언에서도 속성과 함수 및 초기화 블록 등을 포함할 수 있다. 단, 생성자는 가질 수 없다. 최초로 액세스되는 시점에 하나의 객체가 생성되기 때문이다(이것을 늦 초기화(lazy initialization)라고 한다).

객체 선언으로 생성되는 객체의 속성이나 함수는 객체 이름을 사용해서 액세스할 수 있다. main( ) 함수에서는 for 루프를 10회 반복 실행하면서 StateManager 객체의 메시지 번호와 내용을 변경하고 출력한다.

객체 선언에서는 클래스로부터 상속 받거나 인터페이스를 구현할 수도 있다. 따라서 기존 클래스의 싱글톤 객체를 생성하여 사용하거나, 또는 시스템 프레임워크의 인터페이스를 구현해야 할 필요가 있을 때 유용하다. 이때는 다음과 같이 콜론(:) 다음에 슈퍼 클래스나 인터페이스를 지정하고 함수를 오버라이딩하면 된다. 인터페이스를 구현하는 예는 다음과 같다.

```
interface CaptureMessage {
    fun capture()
}

object StateManager : CaptureMessage {
    ..
    override fun capture() {}
}
```

여기서는 StateManager 객체에서 CaptureMessage 인터페이스의 capture( ) 함수를 구현한다.

싱글톤은 시스템의 자원 사용과 부담을 줄이고 같은 객체를 공유할 수 있다는 장점이 있다. 그러나 애플리케이션이 다중 스레드로 실행될 때 반드시 하나의 객체만 생성되도록 동기화 처리를 해 주어야 하므로 주의해야 한다.

object 객체 선언은 클래스 내부에서도 할 수 있다. 이 경우 포함하는 클래스의 인스턴스가 여러 개 생성되더라도 여전히 object 객체는 하나만 생성된다. 예를 들면 다음과 같다.

```
class Outer () {
    object InnerObject {
        var count: Int = 0
        fun printCount() {
            println(count)
        }
    }
}
```

여기서는 Outer 클래스 내부에 object 객체를 선언하였다. 이 객체는 다음과 같이 포함하는 클래스 이름으로 참조할 수 있다.

```
Outer.InnerObject.printCount()
// Outer.printCount()는 에러가 발생한다
```

object 객체의 멤버 속성과 함수는 포함클래스명.object객체명으로 참조해야 한다.

코틀린 컴파일러에서는 object 객체를 컴파일하여 INSTANCE라는 이름의 static 필드를 갖는 클래스로 생성한다. 그리고 이 필드에는 싱글톤 객체의 참조가 저장된다. 따라서 자바 코드에서 코틀린의 object 객체를 사용할 때는 INSTANCE 필드 이름을 추가해야 한다. 예를 들어, 위의 Outer.InnerObject.printCount()를 자바 코드에서는 Outer.InnerObject.**INSTANCE**.printCount()로 사용해야 한다.

object 키워드를 사용하는 두 번째 방법으로 동반 객체를 알아보자.

바로 앞에서 얘기했듯이, 클래스 내부에 object 객체 선언을 사용하면 자바의 static 메서드나 static 필드처럼 object 객체의 함수와 속성을 사용할 수 있다. object 객체를 포함하는 클래스의 인스턴스가 생성되지 않아도 되기 때문이다. 코틀린에서는 최상위 함수(코틀린 파일의 클래스 외부에 따로 선언된 함수)를 사용해도 동일한 효과를 볼 수 있다. 그러나 최상위 함수에서는 클래스 내부의 private 멤버는 액세스할 수 없다. 클래스 내부에서 object 객체 선언을 사용할 때도 마찬가지다. 예를 들어, 클래스의 생성자를 private으로 지정하면 생성자를 사용한 보통의 방법으로는 인스턴스를 생성하지 못한다. 따라서 해당 클래스의 팩토리(Factory) 메서드를 호출하여 인스턴스 생성을 제어할 수 있다. 또한 바로 앞의 Outer.InnerObject.printCount()처럼 object 객체의 멤버를 액세스할 때는 이 객체를 포함하는 클래스와 object 객체를 모두 지정해야 하므로 사용이 불편하다(자바에서는 클래스 이름만 지정하여 static 멤버를 액세스할 수 있다).

코틀린에는 자바의 static과 같은 키워드가 없다. 따라서 companion 키워드를 추가하여 object 객체 선언과 같이 사용하도록 하였다. 이것을 동반 객체라고 하며, 이 객체를 포함하는 클래스에서는 자신의 멤버인 것처럼 인식한다. 따라서 object 객체를 더 편리하게 사용할 수 있다. 동반 객체의 예를 보면 다음과 같다.

```
fun main(args: Array<String>) {
    OuterClass.printMsg()
}

class OuterClass {
    companion object ComObj {              // 이름없이 companion object { 로 해도 된다
        fun printMsg() {
```

```
            println("동반 객체의 함수가 호출됨")
        }
    }
}
```

여기서는 OuterClass에서 동반 객체인 ComObj를 포함한다. 그리고 main( ) 함수에서 동반 객체의 printMsg( ) 함수를 호출한다. 이때 동반 객체의 이름을 지정하지 않고 포함하는 클래스의 멤버인 것처럼 호출할 수 있다. 따라서 동반 객체의 이름인 ComObj를 생략해도 된다.

앞에서 얘기한 팩토리 메서드를 동반 객체로 구현하는 예를 보면 다음과 같다.

```
fun main(args: Array<String>) {
    val m = MyClass.create()    // val m = MyClass()는 에러임
    m.printMyClass()
}

class MyClass {
    private constructor()
    companion object {
        fun create(): MyClass = MyClass()
    }
    fun printMyClass() {
        println("팩토리 객체의 함수가 호출됨")
    }
}
```

여기서는 MyClass에서 이름이 없는 동반 객체를 포함하고 있으며, 생성자를 private으로 지정하였다. 그리고 팩토리 메서드 역할을 수행하는 동반 객체의 create( ) 함수에서 MyClass의 인스턴스를 생성한다. 이처럼 동반 객체에서는 자신을 포함하는 클래스의 private 멤버도 액세스할 수 있다.

main( ) 함수에서는 MyClass.create( )를 호출하여 MyClass 인스턴스를 생성한다. 이때 MyClass( )로는 인스턴스를 생성할 수 없다. MyClass 생성자가 private이기 때문이다.

자바 코드에서 이름이 없는 코틀린 동반 객체를 사용할 때는 Companion을 추가해야 한다. 예를 들어, 위의 MyClass.create( )를 자바 코드에서는 MyClass.Companion.create( )로 해야 한다. 그러나 이름이 있을 때는 Companion 대신 동반 객체 이름을 사용하면 된다.

동반 객체에서도 클래스로부터 상속 받거나 인터페이스를 구현할 수 있으며, 그 방법은 앞의 객체 선언과 동일하다. 또한 동반 객체는 자신을 포함하는 클래스가 메모리에 로드될 때 비로소 초기화된다.

object 키워드를 사용하는 마지막 방법으로 객체 표현식을 알아보자. 이것은 익명(anonymous) 객체(자바의 익명 내부 클래스)를 생성하는 방법이다. 다음 예를 보자.

```
fun countClicks(window: Window) {
    var count = 0

    window.addMouseListener(
        object : MouseAdapter() {
            override fun mouseClicked(e: MouseEvent) {
                count++
            }
            override fun mouseEntered(e: MouseEvent) {
                ..
            }
        }
    )
    ..
}
```

여기서는 마우스 이벤트 리스너를 객체 표현식으로 구현하고 있다. 얼핏 보기에는 객체 선언과 같아 보인다. 그러나 이름이 없으며, MouseAdapter 클래스의 서브 클래스 인스턴스로 object 표현식 객체를 생성한다. 그리고 슈퍼 클래스인 MouseAdapter의 두 함수를 오버라이딩한다. 이 object 표현식 객체(참조)는 이벤트 리스너를 등록하는 addMouseListener( ) 함수의 인자로 전달된다. 따라서 이름을 지정하지 않는다. 그리고 Window 객체에서 마우스 클릭이 발생하면 object 표현식 객체의 mouseClicked( ) 함수가 실행되어 count 변수의 값이 하나 증가한다.

object 표현식 객체에서는 자신이 위치한 함수의 변수를 사용하고 값을 변경할 수도 있다. 또한 하나 이상의 인터페이스를 구현할 수도 있다. 그리고 중요한 것이 있다. object 표현식 객체는 싱글톤이 아니다. 매번 참조될 때마다 즉시로 새로운 인스턴스가 생성되어 사용된다는 것에 유의하자.

B.8.7 중첩 클래스와 내부 클래스

중첩(nested) 클래스와 내부(inner) 클래스는 모두 다른 클래스 내부에 선언된 클래스를 말한다. 그리고 중첩이나 내부 클래스의 멤버는 이들을 포함하는 외부 클래스에서 액세스할 수 있다.

그러나 한 가지 중요한 차이점이 있다. 중첩 클래스에서는 외부 클래스의 멤버를 참조할 수 없지만 내부 클래스에서는 참조 가능하다(중첩 클래스는 자바의 static 중첩 클래스와 동일하다).

코틀린에서 중첩 클래스는 별도의 키워드를 사용하지 않고 정의하며, 내부 클래스는 inner 키워드를 지정한다. 다음 예를 보자.

```
fun main(args: Array<String>) {
    val result1 = ClassOuter.Nested().funcNested()
    println(result1)                        // 2가 출력됨

    val result2 = ClassOuter().Inner().funcInner() // == 1
    println(result2)                        // 1이 출력됨
}

class ClassOuter {
    private val bar: Int = 1
    class Nested {
        fun funcNested() = 2                // fun funcNested() = bar는 에러임
    }
    inner class Inner {
        fun funcInner() = bar
    }
}
```

여기서는 ClassOuter에 중첩 클래스인 Nested와 내부 클래스인 Inner가 모두 포함되어 있다. 그리고 Inner에서는 ClassOuter의 멤버 속성인 bar를 사용할 수 있지만 Nested는 사용할 수 없다는 것을 알 수 있다.

또한 외부 클래스에서 내부 클래스의 멤버를 사용할 때는 외부 클래스 인스턴스가 생성되어야 결과를 받을 수 있으므로 ClassOuter().Inner().funcInner()와 같이 해야 한다. ClassOuter의 생성자 호출 및 인스턴스를 생성하는 괄호를 빼면 안 된다.

B.8.8 데이터 클래스

우리가 정의하는 클래스 중에는 주로 데이터를 갖는 클래스들이 많이 있으며, 그 중에는 업무에 관련된 것도 있고 프로그램에서 필요해서 생성한 것들도 있다. 이런 데이터 클래스들은 비즈니스 로직을 처리하는 함수보다는 주로 데이터 값을 저장하는 속성을 갖는다. 따라서 클래스 인스턴스끼리 각 속성의 값을 비교하거나(equals() 함수), 인스턴스를 컬렉션(예를 들어, HashMap)에 저장할 때 사용할 키 값(해시 코드)을 생성하거나(hashCode() 함수), 속성의 값을 쉽게 출력하는 기능(toString() 함수)이 공통적으로 필요하다.

따라서 코틀린에서는 데이터 클래스라는 개념을 추가하였다. 즉, 우리가 클래스를 정의할 때 data 키워드를 지정하면 데이터 클래스로 간주하며, 방금 전에 설명한 기능들을 처리해주는 함수들을 코틀린 컴파일러가 자동으로 생성해주기 때문이다. 또한 각 속성의 값을 쉽게 변수로 추출해주는 componentN( ) (여기서 N은 클래스에 속성이 선언된 위치로 1부터 시작됨) 함수들도 생성해 주며, 기존 인스턴스의 속성 값을 쉽게 변경하여 사용할 수 있게 해주는 copy( ) 함수도 생성해준다.

그러나 주의할 것이 있다. 자동 생성되는 함수들이 데이터 클래스의 모든 속성을 처리할 수 있게 하려면 반드시 기본 생성자에 속성들을 지정해야 한다. 그래야만 코틀린 컴파일러가 알 수 있기 때문이다. 기본 생성자에 지정되지 않은 속성들은 자동 생성되는 함수들을 우리가 오버라이딩해서 처리해야 한다.

예를 들어, 친구(Friend) 클래스를 생각해보자.

```
data class Friend (val name: String, val age: Int, val tel: String)
```

여기서 Friend 클래스는 데이터 클래스로 정의되었으며, 기본 생성자에 이름(name)과 나이(age) 및 전화번호(tel) 속성을 지정하였다. 따라서 코틀린 컴파일러는 이 클래스 인스턴스들의 속성들을 대상으로 처리하는 equals( ), hashCode( ), toString( ), componentN( ), copy( ) 함수를 자동 생성해준다. 단, Friend 클래스에 우리가 정의하거나 또는 슈퍼 클래스로부터 상속받는 함수의 경우는 자동 생성되지 않는다.

데이터 클래스의 속성들은 val과 var 모두 가능하지만 여기처럼 val 변수로 지정하여 값을 변경하지 못하게 하는 것이 좋다. 왜냐하면, HashMap과 같은 컬렉션에 데이터 클래스 인스턴스들을 저장해서 사용할 때 이미 저장된 인스턴스의 속성 값을 변경하면(var일 때) hashCode( ) 결과값이 달라져서 문제가 생길 수 있기 때문이다. 또한 다중 스레드(multi-thread)로 실행될 때 동기화에 따른 문제도 생길 수 있다. 따라서 데이터 클래스에는 copy( ) 함수가 있어서 기존 인스턴스의 속성 값을 변경하여 새로운 인스턴스로 생성할 수 있게 해준다. 다음 코드를 보자.

```
fun main(args: Array<String>) {
    val f1 = Friend("김선달", 30, "010-123-4567")
    val f2 = Friend("홍길동", 25, "010-456-7891")

    println(f1)
    println(f2)
```

```
    println(f1==f2)
    println(f1.hashCode() == f2.hashCode())

    val f3 = f2.copy(tel="010-234-5678")
    println(f3)
    println(f3==f2)
    println(f3===f2)

}

data class Friend (val name: String, val age: Int, val tel: String)
```

여기서는 데이터 클래스로 정의된 Friend의 인스턴스로 f1과 f2 두 개를 생성한다. 그리고 두 인스턴스를 println( ) 함수로 출력한다. 이때 자동 생성된 toString( ) 함수가 호출되어 두 인스턴스의 모든 속성 값이 다음과 같이 출력된다.

```
Friend(name=김선달, age=30, tel=010-123-4567)
Friend(name=홍길동, age=25, tel=010-456-7891)
```

그리고 println(f1==f2)의 f1==f2를 실행할 때 자동 생성된 equals( ) 함수가 호출되어 두 인스턴스의 모든 속성 값을 비교한다. 여기서는 속성 값이 다르므로 결과로 false가 출력된다.

println(f1.hashCode( ) == f2.hashCode( ))에서는 각 인스턴스의 해시 코드 값을 비교하며, 결과는 false가 출력된다. hashCode( ) 역시 자동 생성된 함수이다.

그리고 val f3 = f2.copy(tel="010-234-5678")에서는 f2 인스턴스의 tel 속성 값만 변경한 새로운 인스턴스가 생성된다. 자동 생성된 copy( ) 함수는 이처럼 우리가 변경을 원하는 속성만 값을 바꾸고 나머지 속성은 원래의 값 그대로 갖는 인스턴스를 생성해준다.

따라서 println(f3)의 결과는 Friend(name=홍길동, age=25, tel=010-234-5678)이 출력된다. 그리고 f3==f2의 결과는 false이다. name과 age 속성 값은 같지만 tel 속성 값은 다르기 때문이다. f3===f2의 결과 역시 false이다. f2와 f3 변수는 서로 다른 객체의 참조를 갖고 있기 때문이다.

데이터 클래스 인스턴스의 속성 값은 쉽게 변수로 추출할 수 있다. 데이터 클래스 속성의 구조를 기준으로 각 속성 값을 변수로 추출해주는 componentN( ) (예를 들어, 첫 번째 속성은 component1( ), 두 번째 속성은 component2( )) 함수를 자동 생성하고 호출해주기 때문이다.

예를 들어, 앞 코드의 f1 인스턴스가 갖는 모든 속성 값을 각각 다른 변수에 추출하고자 할 때는 다음과 같이 하면 된다.

```
val (fname, fage, ftel) = f1
println(fname + ", " + fage + ", " + ftel)
```

이처럼 괄호 안에 하나 이상의 변수명을 지정하면 오른쪽에 지정된 인스턴스의 속성 값을 차례대로 찾아서 해당 변수에 넣어준다. 따라서 여기서는 다음 결과가 출력된다.

```
김선달, 30, 010-123-4567
```

값이 추출되는 속성의 순서는 데이터 클래스의 기본 생성자에 지정된 순서를 따른다.

이와 같이 데이터 클래스의 구조를 인지하여 변수로 추출해주는 기능을 **해체 선언**(destructuring declaration)이라고 한다. HashMap과 같은 컬렉션에 데이터 클래스의 인스턴스를 저장하면 이 기능을 사용하여 인스턴스의 값을 쉽게 변수로 추출할 수 있다. 또한 함수에서 반환값으로 데이터 클래스 인스턴스를 사용하면 여러 개의 값을 반환하는 효과를 낼 수 있다.

B.8.9 클래스 위임

코드를 재사용하는 대표적인 방법이 **클래스 상속**(inheritance)이다. 즉, 클래스 간에 개념적인 관계가 있을 때 계층 구조를 만들고 상위의 슈퍼 클래스가 갖는 속성과 함수를 하위의 서브 클래스에서 상속받아 재사용할 수 있도록 해준다. 또한 상속 받은 속성이나 함수를 서브 클래스에서 변경 가능하며(오버라이딩), 자신만의 속성이나 함수를 추가하여 기능을 확장할 수도 있다. 그러나 이런 장점과 더불어 단점도 있다. 슈퍼 클래스가 변경되면 서브 클래스에 영향을 줄 수 있고, 슈퍼 클래스의 구현 부분이 서브 클래스에 노출된다는 것이다.

따라서 상속의 단점을 보완하면서 코드를 재사용하는 또 다른 방법이 필요한 경우가 있다. 그것이 **위임**(delegation)이다. 위임은 말 그대로 해당 일을 처리할 수 있는 객체의 코드를 재사용하는 것이다. 그리고 이때 상속처럼 클래스 간의 계층 관계를 구성하지 않아도 되며, 위임을 받아 처리할 객체를 포함하면 된다. 따라서 해당 객체가 포함된 클래스 내부의 구현 부분이 공개되지 않으며 변경도 용이하다.

위임은 우리가 직접 해당 코드를 작성하여 구현할 수 있다. 그러나 코틀린 언어에 포함된 "by"

키워드를 사용하면 더 적은 양의 코드로 쉽게 구현할 수 있다. 그리고 "by" 키워드는 클래스 헤더에 정의하므로 코틀린에서는 **클래스 위임**이라고 한다.

우선, 클래스 상속을 이용해서 코드를 재사용하는 예를 생각해보자. 사각형을 나타내는 Rectangle 클래스와 원을 나타내는 Circle 클래스가 있다. 그리고 이 클래스들은 모두 도형을 나타내므로 도형에 공통적으로 필요한 속성과 함수를 가질 수 있다. 따라서 도형의 대표 클래스인 Figure 클래스를 만들고 이 클래스에 그런 속성과 함수를 정의한 후 Rectangle과 Circle 서브 클래스에서 상속받아 오버라이딩하도록 클래스 상속 구조를 만들 수 있다. 사각형과 원은 그리거나 채우는 방법이 서로 다르기 때문이다.

그다음에 이런 도형들을 창(window)에 그려주는 Window 클래스에서는 도형을 그리거나 채울 때 각 도형의 인스턴스에게 위임하여 처리하도록 하면 된다. 현재 열려 있는 코틀린 프로젝트의 src 폴더에 새로운 코틀린 파일을 생성하고 다음 코드를 작성해보자.

```
// 클래스 상속과 다형성(polymorphism)을 이용한 위임 구현
fun main(args: Array<String>) {
    val r = Rectangle()
    val c = Circle()
    Window(r).drawFigure()
    Window(r).fillFigure()
    Window(c).drawFigure()
    Window(c).fillFigure()
}

open class Figure() {
    open fun draw() {}
    open fun fill() {}
}
class Rectangle() : Figure() {
    override fun draw() {
        println("Draw rectangle")
    }
    override fun fill() {
        println("Fill rectangle")
    }
}

class Circle() : Figure() {
    override fun draw() {
        println("Draw circle")
    }
    override fun fill() {
        println("Fill Circle")
    }
}
```

```
class Window(val figure: Figure) {
    fun drawFigure() {
        figure.draw()
    }
    fun fillFigure() {
        figure.fill()
    }
}
```

여기서는 Figure 슈퍼 클래스에 도형을 그리는 draw( ) 함수와 도형을 채우는 fill( ) 함수가 정의되어 있으며, Rectangle과 Circle 서브 클래스에서는 두 함수를 상속받아 오버라이딩한다(여기서는 함수가 실행된다는 메시지만 출력하였다).

그리고 Window 클래스에서는 도형을 그려서 보여준다. 이때 Window 인스턴스를 생성할 때 기본 생성자의 인자로 전달된 도형 객체 참조를 사용한다. 단, 인자의 타입은 슈퍼 클래스인 Figure로 지정하였다. 이렇게 하면 Rectangle과 Circle 중 어떤 서브 클래스의 인스턴스가 전달되더라도 모두 받을 수 있기 때문이다(서브 클래스의 인스턴스는 자신의 클래스는 물론 슈퍼 클래스의 타입도 된다).

main( ) 함수에서는 각 도형의 인스턴스를 하나씩 생성한 후 Window 클래스 인스턴스를 생성할 때 생성자 인자로 전달한다. 그리고 Window 클래스의 함수를 호출한다. 그러면 전달된 도형 인스턴스의 타입에 따라 그것의 draw( )와 fill( ) 함수가 호출된다. 이것은 컴파일 시점이 아닌 런타임 시에 객체지향의 **다형성** 개념을 구현하는 것이다. 또한 도형을 그리거나 채우는 일을 할 때 각 도형 인스턴스의 코드가 재사용된다.

이처럼 클래스 상속을 사용해도 되지만, 다음과 같이 코틀린의 클래스 위임을 사용하면 편리하다.

```
// 클래스 위임을 이용한 위임 구현
fun main(args: Array<String>) {
    val r = Rectangle()
    val c = Circle()
    Window(r).draw()
    Window(r).fill()
    Window(c).draw()
    Window(c).fill()
}

interface Figure {
```

```
    fun draw()
    fun fill()
}
class Rectangle() : Figure {
    override fun draw() {
        println("Draw rectangle")
    }
    override fun fill() {
        println("Fill rectangle")
    }
}

class Circle() : Figure {
    override fun draw() {
        println("Draw circle")
    }
    override fun fill() {
        println("Fill Circle")
    }
}

class Window(val figure: Figure) : Figure by figure {}
```

더 앞의 클래스 상속을 이용한 코드 재사용 예와 다른 점을 비교해 보자. 여기서는 도형에 공통적으로 필요한 함수를 클래스가 아닌 Figure 인터페이스로 정의하였다. 위임을 받을 클래스에서는 클래스 상속 구조 없이 이 인터페이스만 구현하면 되기 때문이다.

그리고 코드 재사용을 위해 위임을 처리하는 Window 클래스(이것을 대표자(delegate)라고 한다)에서는 위임을 받아 실행되는 인스턴스의 함수를 호출하는 코드를 갖고 있지 않아도 된다. "by" 키워드를 지정하였기 때문이다. 이 경우 생성자 인자로 전달되는 figure(Figure 타입의 인스턴스)를 Window 객체의 내부에 저장하고, 그 인스턴스를 호출하는 Figure의 함수들을 자동 생성해 주기 때문이다. 이처럼 코틀린의 언어 요소로 제공되는 "by" 키워드를 사용하면 작성할 코드를 줄일 수 있다.

이와 더불어 위임의 대표자 역할을 하는 Window 클래스에 Figure 인터페이스 함수들을 오버라이딩할 수도 있다. 이때는 전달된 인스턴스의 함수 대신 그 함수가 호출된다. 그러나 이럴 필요는 거의 없을 것이다.

이런 형태로 위임을 구현하여 코드를 재사용하면 위임을 받아 실행되는 객체들의 클래스를 상속 구조로 만들 필요 없이 해당 인터페이스만 구현하면 된다. 또한 모델과 클래스의 변경도 용이하다.

B.8.9 enum 클래스

일련의 상수 값을 갖고 있는 열거형(enumeration)을 코틀린에서는 enum 클래스로 정의한다. 예를 들어, 친구의 유형을 enum 클래스로 정의하고 사용하는 코드는 다음과 같다.

```
fun main(args: Array<String>) {
    println(FriendType.SCHOOL)
    println(FriendType.SCHOOL.ordinal)
    println(FriendType.COMPANY.ordinal)
    println(FriendType.COMPANY.name)
    println(FriendType.valueOf("COMPANY"))

    val friends = FriendType.values()
    for (item in friends) {
        println(item)
    }

    println(getFriendTypeName(FriendType.SCHOOL))
}

enum class FriendType {
    SCHOOL, COMPANY, SNS, OTHERS
}

fun getFriendTypeName(friend: FriendType) =
    when (friend) {
        FriendType.SCHOOL -> "학교 친구"
        FriendType.COMPANY -> "회사 친구"
        FriendType.SNS -> "SNS 친구"
        FriendType.OTHERS -> "기타 친구"
    }
```

이 코드의 FriendType은 enum 클래스이며, 가장 간단하게 사용한 예이다. 여기서는 SCHOOL(학교 친구), COMPANY(회사 친구), SNS(SNS 친구), OTHERS(기타)의 네 가지 항목을 갖고 있다.

enum 클래스의 항목은 FriendType.SCHOOL과 같이 참조한다. 또한 코틀린에서는 enum 클래스의 각 항목에 대해 내부적으로 name(상수 이름이며 String 타입)과 ordinal(상수 위치이며 Int 타입) 두 개의 속성을 갖는다. 따라서 FriendType.SCHOOL.ordinal은 0을, FriendType.COMPANY.ordinal은 1을 반환한다(항목의 위치 값은 0부터 시작하며 1씩 증가한다). FriendType.COMPANY.name과 FriendType.COMPANY 및 FriendType.valueOf("COMPANY")는 모두 "COMPANY"를 반환한다. 또한 FriendType.values( )는 FriendType의 모든 항목 이름을 배열로 생성해준다.

또한 enum 클래스와 when을 같이 사용하여 우리가 알기 쉬운 문자열로 enum 항목을 나타

낼 수 있다. 여기서는 getFriendTypeName( ) 함수에서 enum 항목을 인자로 받아 친구 유형을 알기 쉬운 문자열로 반환한다. 이때 코틀린 특유의 함수 선언 방식을 사용하였다. 즉, 대입문의 왼쪽에 함수 선언문이 있고 오른쪽에는 함수의 실행 코드가 있다. 이번 장의 앞에서 설명하였듯이, if나 when은 단일 표현식으로 간주되므로 가능하며, when의 실행 결과가 함수의 반환값으로 처리된다. 만일 이렇게 하지 않고 기존 방식대로 중괄호({})를 사용해서 함수 몸체 안에 when을 넣으면 함수 선언에 반환 타입을 추가하고 when의 각 항목에도 return 문을 추가해야 한다.

코틀린에서는 enum 클래스 항목의 속성을 우리가 정의할 수도 있다. 각 항목이 단순한 상수가 아니라 객체이기 때문이다. 예를 들어, RGB 색상은 빨간색과 초록색 및 파란색의 조합으로 나타낼 수 있다. 따라서 RGB 색상은 세 개의 정수 속성을 가질 수 있다.

```
enum class RGBColor(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 rgbValue() = (r * 256 + g) * 256 + b
}
```

또한 이 코드처럼 enum 클래스에는 멤버 함수도 정의할 수 있다. 여기서는 enum 클래스에 정의된 RGB 색상의 색상 값을 반환하는 rgb( ) 함수를 멤버 함수로 갖는다. 따라서 다음과 같이 호출하면,

```
println(RGBColor.RED.rgbValue())
```

결과로 16711680이 출력된다.

이처럼 enum 클래스에 멤버 함수를 정의할 때는 다음과 같이 마지막 항목의 제일 끝에 세미콜론(;)을 붙여야 한다는 것에 유의하자.

```
INDIGO(75, 0, 130), VIOLET(238, 130, 238);
```

또한 이렇게 우리가 속성을 정의해도 모든 enum 클래스에 기본적으로 생성되는 속성인 name과 ordinal은 그대로 유지된다.

B.8.10 sealed 클래스

sealed 클래스는 자신의 서브 클래스 종류를 제한하기 위해 사용된다. 어찌 보면 이것은 바로 앞에서 설명한 enum 클래스와 유사하다. enum 클래스에 포함된 각 항목도 클래스이기 때문이다. 그러나 차이점이 있다. enum 클래스의 각 항목은 하나의 인스턴스만 가능하다. 그러나 sealed 클래스에 속하는 서브 클래스들은 일반 클래스이므로 각각 여러 개의 인스턴스를 가질 수 있다. sealed 클래스는 두 가지 형태로 사용할 수 있다. 첫 번째는, sealed 클래스의 모든 서브 클래스들을 sealed 클래스 내부에 중첩된 클래스로 사용하는 방법이다. 이 방법을 사용해서 앞의 FriendType을 enum이 아닌 sealed 클래스로 정의하면 다음과 같다.

```
sealed class FriendType {
    class School(val name: String, val schoolType: Int) : FriendType()
    class Company: FriendType()
    class Sns : FriendType()
    class Others: FriendType()
}
```

이와 같이 sealed 클래스를 정의할 때는 open 키워드를 지정하지 않아도 된다. 여기서 School 서브 클래스의 인스턴스를 생성하고 사용하는 예는 다음과 같다.

```
val s1 = FriendType.School("교동", 1)
println(s1.name)
println(s1.schoolType)
```

sealed 클래스를 정의하는 두 번째 방법은, sealed 클래스의 모든 서브 클래스들을 독립적인 클래스로 정의하는 것이다. 단 이때는 모든 서브 클래스들이 sealed 클래스와 같은 코틀린 파일 안에 있어야 한다. 이 방법을 사용해서 앞의 예를 구현한 코드는 다음과 같다.

```
sealed class FriendType
class School(val name: String, val schoolType: Int) : FriendType()
class Company: FriendType()
class Sns : FriendType()
class Others: FriendType()
```

이때는 School 서브 클래스의 인스턴스를 다음과 같이 생성한다. 이것은 일반 클래스의 인스턴스를 생성하는 방법과 같다.

```
val s1 = School("교동", 1)
```

B.9 람다식

람다식(Lambda expression)은 원래 문제 해결 과정을 수학적인 함수를 사용하여 나타낸 수식이지만, 프로그래밍 언어에서는 함수를 나타내는 표현식을 말한다. 람다식은 자바 8에 새로 추가되었으며 코틀린에서도 더욱 다양한 기능으로 지원한다. 본래 함수는 이름, 매개변수, 반환 타입, 몸체(실행 코드)로 구성된다. 그러나 람다식을 사용하면 함수를 간결하게 표현할 수 있으며 다음과 같은 일을 할 수 있다.

- 함수를 따로 정의하지 않고 간결하게 나타낼 수 있으며, 이름을 지정하지 않아도 된다.
- 람다식은 값으로 처리되므로 변수에 저장하고 실행시킬 수 있다.
- 람다식은 다른 함수의 인자로 전달되어 실행될 수 있다.

우선, 람다식을 어떻게 작성하고 사용할 수 있는지 알아보자.

B.9.1 람다식 작성 방법

현재 열려 있는 IntelliJ/IDEA프로젝트의 src 폴더 밑에 새로운 코틀린 파일을 생성하고 다음 코드를 작성해보자.

```
fun main(args: Array<String>) {
    println("합계는 ${sum1(10, 20)} 입니다")          // 기존 형태로 정의된 덧셈 함수 실행
    println("합계는 ${sum2(10, 20)} 입니다")          // 코틀린 특유의 형태로 정의된 덧셈 함수 실행

    val sum3: (Int, Int) -> Int = {x, y -> x + y} // 람다식으로 작성된 덧셈 함수  #1
    println("합계는 ${sum3(10, 20)} 입니다")

    val sum4 = {a: Int, b: Int -> a + b}          // 람다식으로 작성된 덧셈 함수 #2
    println("합계는 ${sum4(10, 20)} 입니다")
}

fun sum1(a:Int, b: Int): Int {                    // 기존 형태의 덧셈 함수
    return a + b
}

fun sum2(a:Int, b: Int): Int = a + b              // 코틀린 특유의 대입문 형태로 정의된 덧셈 함수
```

여기서는 두 개의 값을 더하는 함수를 기존 형태와 람다식으로 작성한 예를 보여준다. 네 가지의 출력 결과는 동일하다.

첫 번째로, sum1( )은 기존 방법으로 선언한 함수이다. 함수 이름은 sum1이며, 두 개의 Int 타입 매개변수를 받아서 Int 타입의 값을 반환한다. 함수 몸체의 실행 코드는 중괄호({}) 안에 정의되어 있다.

두 번째로, sum2( )는 코틀린 특유의 방법으로 선언한 함수이다. 함수 이름은 sum2이며, 두 개의 Int 타입 매개변수를 받아서 Int 타입의 값을 반환한다. 함수 몸체의 실행 코드는 표현식이므로 대입 연산자의 오른쪽에 지정할 수 있다. 또한 다음과 같이 반환 타입을 생략해도 된다. 코틀린 컴파일러가 타입을 추론해주기 때문이다(여기서는 Int).

```
fun sum2(a:Int, b: Int) = a + b
```

세 번째로, sum3는 람다식을 저장한 변수이며, 그 값은 단순한 데이터가 아니라 sum1( )이나 sum2( ) 함수와 동일한 코드이다. 이런 형태로 람다식을 표현할 때는 괄호 안에 매개변수와 타입을 정의한다. 그리고 -> 다음에 람다식의 반환 타입을 지정하고 그다음에 중괄호({}) 안에 매개변수 이름과 실행 코드를 정의한다.

네 번째로, sum4도 람다식을 저장한 변수이며, sum3과 마찬가지로 그 값은 단순한 데이터가 아니라 함수 코드이다. 이런 형태가 코틀린에서 주로 사용하는 람다식 표현 방법이다. 이때는 중괄호({}) 안에 매개변수 이름과 타입을 지정하고 -> 다음에 실행 코드를 정의한다. 이 경우에도 반환 타입이 생략되었으므로, 코틀린 컴파일러가 타입을 추론해준다(여기서는 Int).

sum3과 sum4는 함수 참조 변수이며, sum3의 **(Int, Int) -> Int** 및 sum4의 **a: Int, b: Int ->**를 **함수 타입**이라고 한다. 이 내용은 "B.9.4 상위차 함수"에서 설명한다.

네 가지 방법을 비교해보면 차이점을 금방 알 수 있을 것이다. 그리고 sum3과 sum4에 저장된 람다식은 다른 함수의 인자로 전달되어 실행될 수 있다.

B.9.2 람다식 활용

람다식은 함수로 정의된 것은 아니지만 표현식으로 전달되고 실행될 수 있기 때문에 이름이 없는 익명 함수로 많이 사용된다(실제로 코틀린에서는 람다식이 익명 클래스로 컴파일된다). 예를 들어, 다음에 설명하는 이벤트나 컬렉션 요소에 대한 처리를 할 때이다.

버튼을 클릭했을 때 실행되는 이벤트 처리 코드를 익명 클래스로 구현한 자바 코드는 다음과 같은 형태를 갖는다.

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
        /* 버튼을 클릭했을 때 실행될 코드 */
    }
});
```

여기서 진한 글씨가 button 객체의 클릭 이벤트를 처리하는 익명 클래스 인스턴스이며, OnClickListener( ) 인터페이스의 onClick( ) 메서드를 구현한다. 그리고 이 인스턴스가 새로 생성되어 setOnClickListener( ) 메서드의 인자로 전달된다.

이것을 코틀린의 람다식으로 작성하면 다음과 같이 매우 간단하다.

```
button.setOnClickListener { view -> /* 버튼을 클릭했을 때 실행될 코드 */ }
```

여기서는 setOnClickListener( ) 함수의 인자로 중괄호({}) 안의 람다식이 전달되는 것이다. 단, 이때는 람다식에서 구현하는 인터페이스에 정의된 추상 함수가 하나만 있어야 한다.

코틀린에서는 컬렉션에서 람다식을 다양하게 사용할 수도 있다. 다음 예를 보자.

```
val list = listOf(1, 2, 3, 4, 5)
println(list.filter { it % 2 == 0 })
```

이 코드에서는 다섯 개의 Int 타입 요소 값을 갖는 List를 생성한 후 List의 filter( ) 함수를 호출한다. 이 함수는 조건식을 하나의 매개변수로 받아서 해당 조건이 true(여기서는 짝수)인 모든 요소를 List로 반환한다. 이 코드에서는 filter( ) 함수의 매개변수로 람다식을 전달한다. 이처럼 하나의 매개변수만 갖는 람다식의 경우에는 기본으로 생성되는 it라는 이름으로 매개변수를 대체할 수 있다.

filter( ) 함수는 조건에 맞는 요소들만 새로운 List로 생성한다. 이와는 달리 map( ) 함수를 사용하면 기존 요소의 값을 변경하여 새로운 List로 만들 수 있다. 예를 들면 다음과 같다.

```
val list = listOf(1, 2, 3, 4, 5)
println(list.map {it + 10})
```

여기서는 map( ) 함수의 인자로 람다식을 전달하여 각 요소의 현재 값에 10을 더한 값을 갖는 새로운 List를 생성하고 출력한다.

더 앞에서 예를 들었던 Friend 클래스를 List에 저장하고 람다식을 사용하는 예를 보자. 이 클래스의 정의는 다음과 같다.

```
data class Friend (val name: String, val age: Int, val tel: String)
```

그리고 두 명의 친구 데이터를 저장하는 List는 다음과 같이 생성할 수 있다.

```
val fr = listOf(Friend("김선달", 30, "010-123-4567"), Friend("홍길동", 25, "010-456-7891"))
```

그다음에 이 List에서 30세 이상인 친구의 이름만 출력한다면 다음과 같이 할 수 있다.

```
println(fr.filter {it.age >= 30}.map(Friend::name))
```

출력된 결과는 다음과 같다.

```
[김선달]
```

여기서 fr.filter {it.age >= 30}은 30세 이상인 친구만 찾아서 저장한 새로운 List를 생성한다. 그리고 점(.)으로 연결된 map(Friend::name))에서는 그 List 요소에서 이름만 읽어서 다시 새로운 List로 생성한다. 이처럼 함수의 연쇄 호출도 가능하다. 또한 List와 같은 컬렉션에 저장된 객체의 멤버(속성이나 함수)는 **클래스이름::멤버이름**의 형태로 두 개의 콜론을 사용해서 참조할 수 있다.

B.9.3 익명 함수

실행 가능한 코드 블록을 다른 함수의 인자로 전달하는 또 다른 방법으로 **익명**(anonymous) 함수가 있다. 익명 함수는 일반 함수와 유사하지만 함수 이름과 매개변수 타입을 갖지 않는다. 앞에서 Friend 인스턴스가 저장된 List로부터 30세 이상인 친구만 찾아서 새로운 List를 생성하고 출력한 코드는 다음과 같으며, 이때는 람다식을 사용하였다.

```
println(fr.filter {it.age >= 30})
```

익명 함수를 사용하면 이 코드를 다음과 같이 작성할 수 있다. 기능은 동일하다.

```
println(fr.filter(fun (friend) = friend.age >= 30))  // 익명 함수 형태 #1
```

또한 다음 코드도 동일하지만 더 길고 장황한 형태이다. 여기서는 알기 쉽도록 여러 줄로 분리하였지만 한 줄로 붙여도 된다.

```
println(
    fr.filter(
        fun (friend): Boolean {  // 익명 함수 형태 #2
            return friend.age >= 30
        }
    )
)
```

람다식도 그렇지만 익명 함수도 이처럼 다른 함수의 인자로 전달되어 실행될 수 있다.

여기서는 filter( ) 함수에 익명 함수의 코드를 전달한다. 첫 번째 익명 함수에서는 코틀린 특유의 대입문 형태로 함수를 선언하였다. 이때는 함수의 반환 타입을 지정하지 않는다. 코틀린 컴파일러가 반환 타입을 추론해주기 때문이다. 반면에 두 번째 익명 함수는 일반 함수와 동일한 형식으로 정의되었다. 그러나 첫 번째 것에 비해 코드가 길고 작성하기 번거로울 것이다. 이 코드에서 보듯이, 익명 함수의 매개변수는 람다식과 달리 괄호 속에 지정해야 한다. 또한 익명 함수에서 return 문이 실행되면 익명 함수 자체의 실행이 끝나고 복귀되지만, 람다식의 경우는 자신을 포함하는 함수의 실행이 끝나고 복귀된다는 차이가 있다.

람다식과 익명 함수 모두 사용 가능 범위의 외부 변수(이것을 클로저(closure)라 한다)를 액세스할 수 있다. 예를 들면 다음과 같다.

```
val value: Int = 100
val list = listOf(1, 2, 3, 4, 5)
println(list.map {it + value})                              // 람다식을 인자로 전달
println(list.map(fun (it): Int {return it + value}))        // 익명 함수를 인자로 전달
```

여기서 두 개의 println( ) 출력 결과는 동일하다. 하나는 람다식으로 map( ) 함수의 인자로 전달한 것이고, 다른 하나는 익명 함수로 전달한다.

또한 value 변수는 람다식과 익명 함수의 외부에 선언된 것을 사용한다.

B.9.4 상위차 함수

상위차(higher-order) 함수는 다른 함수나 람다식을 인자로 받거나 반환할 수 있는 함수를 말한다. 예를 들어, 앞에서 알아본 컬렉션의 filter( )나 map( ) 함수는 람다식이나 익명 함수를 인자로 받아서 실행시킨다. 상위차 함수에서는 인자로 받는 함수의 타입을 매개변수로 지정해야 한다. 우선, B.9.1의 람다식 코드를 다시 보자.

```
val sum3: (Int, Int) -> Int = {x, y -> x + y}
```

이 코드에서는 두 개의 Int 값을 인자로 받아 더한 값을 반환하여 sum3 변수에 저장한다. 이 변수는 함수 참조를 갖는다. 여기서 (Int, Int) -> Int를 함수 타입이라고 하며, 인자로 전달되는 함수의 매개변수와 타입 및 반환 타입을 나타낸다. 만일 매개변수가 없는 경우는 ( ) -> Int와 같이 나타내며, 아무 것도 반환하지 않는 경우는 ( ) -> Unit처럼 반드시 Unit를 지정해야 한다 (일반 함수에서는 Unit를 생략할 수 있다).

이처럼 매개변수로 람다식이나 함수를 받아 실행시키는 상위차 함수를 정의할 때는 함수 타입을 매개변수로 지정해야 한다. 예를 들어, 두 개의 정수로 어떤 연산이든 처리 가능한 상위차 함수는 다음과 같다. 새로운 코틀린 파일을 생성하고 다음 코드를 작성해보자.

```
fun main(args: Array<String>) {
    val v1 = calc(2, 7, {a, b -> a * b})
    val v2 = calc(3, 100, {a, b -> a + b})
    val v3 = calc(50, 200, {a, b -> a - b})

    println("The result is $v1")
    println("The result is $v2")
    println("The result is $v3")
}

fun calc( value1: Int, value2: Int, execCode: (codeParam1: Int,
                                    codeParam2: Int) -> Int): Int {
    return execCode(value1, value2)
}
```

여기서는 상위차 함수인 calc( )를 정의하였다. 이 함수에서는 연산 처리할 두 개의 정수와 연산을 실행하는 함수를 인자로 받는다. 앞에서 설명한 대로 함수를 인자로 받을 때는 이 코드처럼 함수 타입을 선언해주어야 한다.

```
execCode: (codeValue1: Int, codeValue2: Int) -> Int
```

여기서 execCode는 우리가 임의로 지정하며, 매개변수 이름이면서 동시에 함수 이름의 역할도 한다. codeParam1과 codeParam2는 상위차 함수(calc( ))의 인자로 전달된 함수의 매개변수를 나타내며, -> Int는 이 함수의 반환 타입을 나타낸다.

상위차 함수의 인자로 전달된 함수를 실행시킬 때는 일반 함수를 호출할 때처럼 이 함수의 이름과 괄호를 사용하며, 괄호 안에는 전달할 인자를 지정한다. 여기서는 다음과 같이 calc( ) 함수의 인자로 받은 두 개의 정수를 전달하였다.

```
execCode(value1, value2)
```

상위차 함수인 calc( )는 다음과 같이 호출해야 한다.

```
calc(2, 7, {a, b -> a * b} )
```

첫 번째와 두 번째는 연산 처리할 정수이며, 세 번째는 연산이 수행될 람다식이다. 이 코드에서는 두 정수의 곱셈 연산을 수행한다. 그러나 a * b를 다른 수식이나 표현식으로 변경하면 그것이 실행된다.

우리가 많이 사용할 일은 없겠지만, 상위차 함수에 전달되는 람다식과 함수는 객체로 생성되어 참조되고 사용되므로 시스템의 부담이 생길 수 있다는 것을 고려하자.

B.9.5 인라인 함수

상위차 함수를 사용할 때는 런타임 시에 단점이 생길 수 있다. 각 함수가 객체로 동작하므로 메모리 할당과 사용 및 호출에 따른 시스템의 부담이 따른다. 또한 함수 외부의 변수를 사용 가능하므로 변수 액세스에 따른 부담도 추가된다. 람다식의 경우에도 코틀린 컴파일러가 익명 클래스로 컴파일한다. 따라서 람다식이 사용될 때마다 새로운 객체가 생성되므로, 함수 코드를 직접 실행하는 것보다 비효율적이고 런타임 시의 부담도 커진다. 이럴 때 인라인(inline)

함수를 사용하면 그런 단점을 해소할 수 있다. 인라인 함수로 지정할 때는 inline 키워드만 추가하면 된다. 앞에 나왔던 상위차 함수인 calc( )를 인라인 함수로 지정하면 다음과 같다.

```
inline fun calc( value1: Int, value2: Int, execCode:
                                    (codeParam1: Int, codeParam2: Int) -> Int): Int {
    return execCode(value1, value2)
}
```

인라인 함수를 호출하는 모든 코드는 코틀린 컴파일러에 의해 해당 함수의 바이트 코드로 복사 및 대체된다(인라인 처리된다). 따라서 컴파일된 코드의 크기가 증가한다. 그러나 런타임 시의 성능은 향상된다. 특히 반복 실행되는 루프에 람다식을 포함하는 함수는 인라인 함수로 지정하는 것이 좋다.

람다식이 인라인 함수의 인자로 전달될 때는 해당 함수와 람다식 모두 인라인 처리된다. 그러나 흔하지는 않지만, 인자로 전달되는 람다식이 다른 곳(객체)에 저장되는 경우에 해당 람다식은 인라인 처리될 수 없다(이때는 코틀린 컴파일러가 알려준다). 예를 들어, 인라인 함수에서 생성하는 어떤 객체의 생성자 인자로 해당 람다식이 전달되어 그 객체의 속성에 저장된 후 사용되는 경우다. 이때는 런타임 시에 객체가 생성되므로, 코틀린 컴파일러가 해당 람다식의 바이트 코드를 어디에 대체할지 알 수 없기 때문이다.

인라인 함수의 인자로 전달되는 람다식을 인라인 처리하고 싶지 않거나, 또는 인라인 처리가 불가능한 경우는 다음과 같이 noinline 키워드를 지정하면 된다.

```
inline fun makeMoney(lambda1: () -> Int, noinline lambda2: (Int) -> Unit) {
    // 함수의 실행 코드
}
```

여기서 makeMoney( ) 인라인 함수는 두 개의 람다식을 매개변수로 받는다. lambda1은 매개변수가 없으며 Int 타입의 값을 반환한다. 또한 lambda2는 Int 타입의 매개변수 하나를 받으며 아무 것도 반환하지 않는다. 이 경우 makeMoney( ) 함수의 실행 코드와 lambda1의 코드는 인라인 처리되지만, lambda2는 인라인 처리되지 않는다.

인라인 처리되는 람다식은 해당 인라인 함수 내부에서만 호출될 수 있으며, 인라인 인자로만 전달될 수 있다. 그러나 noinline이 지정된 람다식은 일반 람다식과 동일하게 사용될 수 있다(예를 들어, 변수에 저장하거나 함수의 인자로 전달).

B.10 예외 처리

코틀린 프로그램에서 발생하는 모든 에러는 클래스로 정의된 예외(exception)로 처리되며, 예외 발생과 처리 메커니즘이 언어 자체에 배려되어 있다(자바와 거의 동일하며, 실제로 대부분의 자바 예외 클래스를 사용한다). 코틀린의 모든 에러 및 예외 클래스는 Throwable 클래스의 후손이다. 이 클래스는 발생된 예외 객체(Throwable의 서브 클래스 인스턴스)와 메시지(String) 및 스택 기록(java.lang.StackTraceElement 객체를 요소로 저장한 배열)을 속성으로 갖고 있다.

B.10.1 try~catch~finally

자바와 마찬가지로, 코틀린에서도 에러나 예외가 생기면 다음과 같이 throw 문을 사용해서 해당 예외 객체를 던지면(발생시키면) 된다.

```
throw MyException("예외 발생!")
```

여기서는 MyException이라는 예외 클래스의 인스턴스를 생성하여 예외를 발생시키며, 이때 이 인스턴스는 "예외 발생!"이라는 메시지를 갖는다.

발생된 예외는 다음과 같이 try~catch~finally 블록으로 잡아내어 처리할 수 있다.

```
try {
    // 실행 중에 예외가 생길 수 있는 코드들을 여기에 넣는다
}
catch (e: SomeException) {
    // 해당 예외(여기서는 SomeException)가 발생했을 때 처리하는 코드
}
..
finally {
    // finally 블록은 생략 가능하다
}
```

try 블록에는 실행 중에 예외가 생길 수 있는 코드들을 넣는다. catch 블록은 하나 이상을 지정할 수 있으며, 발생된 예외의 타입과 일치하는 catch 블록의 코드가 실행된다. finally 블록의 코드는 예외 발생과는 무관하게 항상 실행된다. 즉, try 블록의 코드를 에러 없이 모두 실행한 후에도 finally 블록의 코드가 실행되며, 예외가 발생하여 특정 catch 블록의 코드를 실행한 후에도 finally 블록의 코드는 실행된다. 따라서 파일을 닫는 등의 마무리 작업을 처리하는 코드를 finally 블록에 두면 된다.

위의 catch (e: SomeException)에서 e는 발생된 예외 객체 참조이며, 이름은 우리가 임의로 지정하면 된다. 그리고 콜론(:) 다음에는 확인하고자 하는 예외 객체의 타입을 지정한다.

B.10.2 Checked와 Unchecked 예외

자바에서는 메서드에서 throw 문으로 예외를 발생시킬 수 있는 경우 해당 메서드 헤더에 throws 키워드로 명시해야 한다. 그리고 그런 메서드를 사용하는 코드에서는 반드시 그런 예외를 처리하는 try~catch 블록이 있어야 한다. 만일 없으면 컴파일러가 에러를 발생시킨다. 이와 같은 예외를 checked 예외라고 한다. 쉽게 말해, 프로그래머가 반드시 예외를 확인하도록 만든 메커니즘이다. 이와는 달리, 상황에 따라 발생할 수는 있지만 checked가 아닌 것은 unchecked 예외라고 한다. 이런 예외들은 프로그래머가 알아서 처리해 줄 예외들이다.

그러나 자바와는 달리 코틀린에는 checked와 unchecked 예외라는 개념이 없다. 모든 예외는 필요할 때 프로그래머가 알아서 처리한다. 그 이유는 다음과 같다.

자바의 라이브러리 메서드에서는 주로 IOException과 같은 것이 checked 예외로 지정되어 있으며, 이 예외는 파일을 저장할 때 하드웨어나 기타 시스템 에러 등의 이유로 발생할 수 있다. 그러므로 프로그래머가 IOException예외 발생을 확인한다고 하더라도 특별히 조치할 것이 없다. 따라서 이런 예외를 발생시킬 수 있는 라이브러리 메서드들을 사용할 때마다 try~catch로 그런 예외를 확인하는 코드를 작성해야 한다는 것은 비효율적이므로 코틀린에서는 checked 예외를 없앤 것이다.

B.10.3 try~catch는 표현식이다

코틀린에서는 if나 when처럼 try가 표현식으로 간주되므로 대입문에 바로 나올 수 있다. 예를 들면 다음과 같다.

```
val err: Int? = try { parseInt(value) }
                catch (e: NumberFormatException) { null }
```

여기서는 문자열 형태의 숫자를 정수(Int)로 변환하는 parseInt( ) 함수를 호출한다. 그러나 인자로 전달되는 value의 값에 숫자가 아닌 것이 있을 때는 이 함수에서 NumberFormatException 예외를 발생시킬 수 있기 때문에 try 블록에 넣은 것이다. 그리고 만일 NumberFormatException 예외가 발생되면, catch 블록의 코드가 실행되므로 err 변수에는 null 값이 대입된다. 그러나 에러 없이 실행되면 Int 타입으로 변환된 정수 값이 대입된다.

또한 try~catch와 더불어 throw도 표현식으로 간주된다. 따라서 다음과 같이 엘비스 연산자에 사용될 수 있다.

```
val s = fr.name ?: throw IllegalArgumentException("이름을 입력하세요")
```

여기서는 fr.name의 값이 null이 아니면 그 값을 s에 넣고 null이면 IllegalArgumentException 예외를 발생시킨다. 이때 throw 표현식의 타입은 Nothing이라는 키워드로 나타내는 특별한 타입이며, 이것은 값이 없고 코드 위치를 나타내는데 사용된다. Nothing은 함수의 반환 타입을 지정하는데도 사용할 수 있다.

함수를 사용해서 앞의 코드와 동일한 기능을 수행하는 코드를 작성하면 다음과 같다.

```
val err = fr.name ?: fail("이름을 입력하세요")

fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

B.11 package와 import

코틀린에서는 하나의 소스 코드(.kt) 파일에 하나 이상의 클래스, 함수, 속성을 포함시킬 수 있으며, 이것들을 최상위 수준의 클래스, 함수, 속성이라고 한다.

또한 자바와 마찬가지로 코틀린에서도 패키지(package) 단위로 클래스, 함수, 속성들을 모아 두고 사용할 수 있다. 이때는 소스 코드 파일의 맨 앞에 package 키워드를 지정하면 된다. 그리고 클래스, 함수, 속성들이 다른 소스 코드 파일에 있더라도 패키지가 같으면 바로 사용할 수 있다. 예를 들면 다음과 같다.

```
package mypkg.classes
```

이렇게 지정하면 같은 소스 코드 파일에 있는 클래스, 함수, 속성들은 모두 같은 패키지에 속하게 된다(여기서는 mypkg.classes).

여기서 mypkg는 상위 패키지이고 classes 패키지는 mypkg의 하위 패키지이다. 자바에서는 패키지 구조가 컴퓨터 파일 시스템의 디렉터리 구조와 일치한다. 따라서 mypkg 디렉터리 밑에

classes 서브 디렉터리가 생긴다. 그러나 코틀린에서는 하나의 .kt 파일에 여러 개의 클래스나 함수를 둘 수 있고 .kt 파일 이름을 우리가 원하는 대로 지정할 수 있다. 따라서 패키지 구조와 디렉터리 구조가 일치하지 않아도 된다. 그러나 코틀린 코드와 자바 코드를 같이 사용할 수 있으므로, 가급적이면 자바처럼 패키지 구조와 디렉터리 구조를 일치시켜 사용하는 것이 좋다.

다른 패키지에 있는 클래스, 함수, 속성들을 사용하려면 import 문을 사용해서 그것들의 위치를 알려주어야 한다. 우리 코드를 컴파일할 때 코틀린 컴파일러가 위치를 알아야 찾을 수 있기 때문이다. 단, import 문은 package 문 다음에 지정한다. 그리고 특정 패키지의 모든 클래스, 함수, 속성을 나타낼 때는 *를 사용하면 편리하다. 예를 들어, 앞의 mypkg.classes 패키지에 있는 모든 클래스, 함수, 속성을 나타낼 때는 다음과 같이 한다.

```
import mypkg.classes.*
```

만일 mypkg.classes 패키지의 특정 클래스나 함수를 나타낼 때는 다음과 같이 패키지 다음에 지정하면 된다(여기서 MyClass는 클래스이고, MyFunction은 함수이다).

```
import mypkg.classes.MyClass
import mypkg.classes.MyFunction
```

코틀린 파일(.kt)에는 클래스는 물론이고 클래스 외부에도 함수나 속성을 정의할 수 있다(자바에서는 이렇게 할 수 없다). 앞에서 얘기했듯이, 이런 함수나 속성을 최상위 수준 함수나 속성이라고 하며 같은 코틀린 파일에 있는 다른 클래스나 함수에서 바로 사용할 수 있다. 따라서 import 문에는 클래스는 물론 함수도 바로 지정할 수 있다.

코틀린에서도 클래스, 클래스 멤버, 최상위 수준 변수와 함수에 접근 제한자를 사용할 수 있다(B.8.3 참조). 그러나 코틀린에는 패키지 수준의 접근 제한자는 없다. 같은 패키지에 있는 클래스, 함수끼리는 기본적으로 액세스가 가능하기 때문이다.