第K章 プログラミング言語 Kotlin

Kotlin はジェットブレインズ (JetBrains) 社の Andrey Breslav, Dmitry Jemerov らにより開発されたマルチパラダイムのプログラミング言語である。登場したのは 2011 年である。 ( ) 上で動作し、Java で書かれたプログラムと相互呼出しが可能である。また、JavaScript にコンパイルすることも可能になっている。 など現代的な特徴を取り入れ、"より良い" Java を目指して設計されている。 Android のアプリケーションの開発言語として Google の推奨言語にもなっている。

ここでは Java と異なる部分を中心に Kotlin を説明する。

K.1 Kotlin の実行

新しいプログラミング言語を学習するときの慣習により、最初に、画面に"Hello World!"と表示するだけのプログラムを作成する。次のようなプログラムを好みのエディターで作成する。

ファイル名 Hello.kt

```
1 fun main() {
2     println("hello, world!")
3 }
```

これを というコマンドラインツールを用いてコンパイルする。

```
> kotlinc Hello.kt -include-runtime -d Hello.jar
```

コンパイルが成功すれば、同じディレクトリーに Hello.jar というファイルができている。これは Hello.kt に対する class ファイルと、必要な実行時ライブラリーをひとまとめにしたものである。 実行するときは java コマンドの -jar オプションを用いる。

```
> java -jar Hello.jar
```

また kotlinc コマンドを引数なしで実行すると、 （対話的な処理系）が起動する。これは Java の jshell コマンドに対応する。対話的な処理系では、プロンプト（通常、>>> ）のあとに式を入力すれば、その値を出力する。

```
Type :help for help, :quit for quit
>>> 1 + 2
res0: kotlin.Int = 3
```

Kotlin の統合開発環境 (IDE) としては Eclipse などもあるが、 JetBrains 社が開発している IntelliJ IDEA が（当然ながら）有力である。 また、Java 用のビルドツールとして知られている Maven や Gradle なども Kotlin に対応している。

K.2 定義と宣言

パッケージとインポート

パッケージ (package) やインポート (import) の書き方は Java とほとんど同じである。ただし、最後のセミコロンは省略できる。また、パッケージとソースを配置するディレクトリーを＿＿＿＿＿＿＿＿＿＿。

関数定義

Kotlin ではクラスの外で（トップレベルで）関数を定義することができる。逆に言うと、Java のスタティック (static) メソッドやスタティックフィールドのような概念はない。（ただし companion object というものが使われることがあるが、ここでは説明を割愛する。）

ファイル名 Twice.kt

```kotlin
fun twice(x: Int): Int {
    return x + x
}

fun thrice(x: Int) = x + x + x

fun sayHello(): Unit {
    println("Hello!")
}

fun sayGoodbye() {
    println("Goodbye!")
}

fun main() {
    println(twice(2))
    println(thrice(2))
    sayHello()
    sayGoodbye()
}
```

関数 twice はもっとも基本的な関数定義のカタチを例示している。関数定義は fun というキーワードから開始する。そのあとに関数名が続き、丸括弧の間「(」～「)」に仮引数をコンマで区切って宣言する。仮引数の宣言は、変数: 型 というカタチをしている。さらに、閉じ丸括弧の後にコロン「＿」に続けて戻り値の型を書く。

ブレースの間「{」～「}」に文の並びを書く。なお、文末のセミコロン「;」は省略できる場合が多い。

関数 thrice で例示されているように、すぐに return する関数は「＿」のあとにすぐに式を書くことも可能である。処理系が戻り値の型を推論できる場合は、戻り値の型の宣言を省略できる。

その下の sayHello 関数のように戻り値を持たない場合は、戻り値の型は Unit とする。また、sayGoodbye 関数で示されているように、戻り値型が Unit 場合は宣言を省略することが可能である。

Kotlin のプログラムは＿＿＿という名前の関数から実行される。この main 関数は、この例のように無引数であるか、または文字列 (String) の配列を引数に取る。

print 関数と println 関数

関数 print は、引数を出力する。関数 println は引数を出力してから改行を出力する。

変数宣言

変数もクラスの外に定義することができる。代入で値を変更しない変数は ___ というキーワードで宣言する。（Java 言語で修飾子 final がついている変数に相当する。）一方、代入で値を変更する変数は ___ というキーワードで宣言する。型を宣言する場合は、変数名のあとにコロン「:」に続けて型を書くが、初期値から型を推論できる場合はこの部分は省略できる。

ファイル名 VarTest.kt

```
1 val x: Int = 1
2 val y = 2
3
4 var v = 9
5
6 fun main() {
7     v -= 1
8     println("x =$x, y = $y, v = $v")
9 }
```

文字列リテラルと補間 (interpolation)

文字列リテラルは二重引用符「"」～「"」で囲む。文字列の中の「$変数名」や「${式}」はその変数や式の値に置き換えられる。

K.3 制御構造

条件判断

条件判断の if や if ～ else の書き方は C 言語や Java と大差はないが、Kotlin の場合は if は"式"であり、値を持つことができる。

また when 式という、C や Java の switch 文に相当する構文もあるが、ここでは説明を割愛する。

繰り返し

ループの for 文は in というキーワードとともに、次のカタチで用いる。

```
for（変数 in 式）文またはブロック
```

この in の次の式は配列などの"集まり"である。変数を集まりの各要素に割り当ててループ本体を繰り返す。

さらに for 文の in の次の式には range と呼ばれる「 ___ 」を使った式を書くこともできる。for（i in 1..10）{ … } なら 1 から 10 までの 10 回の繰り返しになる。

また、while 文や do ～ while 文も、C 言語や Java と書き方は同じなので、説明は省く。（ただし、do ～ while 文の最後のセミコロン「;」は必要ない。）

例外処理

例外処理の try ～ catch も Java と書き方は同じだが、Kotlin の場合 try ～ catch は文ではなく“式”である。

K.4 演算子と組み込み型

基本型

Kotlin の数値型は Int, Long, Float, Double のように、＿＿＿＿＿＿＿＿＿＿という点を除き Java と同じ名前を持っている。文字型は Char、真偽値型は Boolean である。Java であるようなプリミティブ型 (int, char, ...) とラッパー型 (Integer, Character, ...) の使い分けは必要ない。

配列・リスト

Kotlin の配列は総称型で、型パラメーターを持っている。Java と同様、型パラメーターはブラケット「＿」～「＿」のあいだに書く。例えば、文字列 (String) を要素とする配列の型は Array<String> と表される。プログラムのエントリーポイントである main 関数が、コマンドライン引数を表す文字列の配列を引数として受け取るときは、その宣言は「fun main(args: ＿＿＿＿＿＿＿＿) { … }」となる。

配列を作成するには可変個引数の関数＿＿＿＿＿を使う。

```
1 val ss: Array<String> = arrayOf("Hayashi", "Takamatsu", "Kagawa", '
```

Kotlin の基本的なコレクション (collection) 型としては配列 Array<T> のほかにリスト List<T>、セット Set<T>、マップ Map<K, V> などがある。Array はアクセスの効率は良いが、要素数を一度決めたら変更することができないのに対し、List はアクセスの効率は犠牲になるが、要素数を容易に変更することができる。リストを作成するには可変個引数の関数＿＿＿＿＿を用いる。

```
1 val fruits: List<String> = listOf("apple", "banana", "orange", "pea
```

ナル (null) 許容型

Kotlin と Java の大きな違いの一つは、通常の Kotlin の型はナル (null) を許容しないというところである。例えば、String 型の変数に null を代入することは型エラーになる。

```
1 var s: String = null  // コンパイル時にエラー
```

型のあとに「＿」をつけると null を許容する型になる。

```
1 var t: String? = null  // これは OK
```

このように null を許容する型の式に対してドット「.」演算子でプロパティーやメソッドをアクセスすることは許されていない。

```
1 println(t.length)  // コンパイル時にエラー
```

そこで「?.」演算子を用いると null を許容する型の式のプロパティーやメソッドにアクセスできて、レシーバーが null のときは結果も null になる。また「?:」演算子は左オペランドが null でないときはその値を、null のときは、右オペランドの値を返す。

```
println(t?.length)      // これは OK --- t が null なら null
println(t?.length ?: 0) // t が null なら  0
```

コメント

Kotlin のコメントは Java と同じで、「//」から「\n」までと「/*」から行末まで、である。

タプル

Pair, Triple はそれぞれ 2 つ組、3 つ組を表すデータ型である。

ファイル名 Quadratic.kt

```
import kotlin.math.*

fun quadratic(a: Double, b: Double, c: Double): Pair<Double, Double
    val d = b * b - 4 * a * c
    val sq = sqrt(d)
    return Pair((-b + sq) / (2 * a), (-b - sq) / (2 * a))
}

```

関数 quadratic は二次方程式 $ax^2 + bx + c = 0$ の 2 解 $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ を Pair として返している。

Pair や Triple からは destructuring declaration （分割宣言(?)）という記法で、構成要素を取り出すことができる。キーワードの val や var のあとに、丸括弧で囲った変数のコンマ区切りの並びを書く。

ファイル名 Quadratic.kt

```
fun main() {
    val a = 1.0; val b = -1.0; val c = -1.0
    val (x1, x2) = quadratic(a, b, c)
    println("方程式の解は、$x1、$x2 です。")
}
```

この例では、Pair の第 1 成分が x1 に、Pair の第 2 成分が x2 に取り出される。

Kotlin の標準ライブラリーには 4 つ組以上のタプルは用意されていないが、データクラス (data class) という宣言で、このような分割宣言を利用できるデータ型を定義することができる。データクラスの説明はここでは割愛する。

K.5 クラスとオブジェクト

クラス定義

クラスはキーワード class を使って定義する。Kotlin のクラスはデフォルトでは final である。つまりサブクラスを定義することができない。サブクラスを定義する場合は class の前に open というキーワードをつけなければいけない。

ファイル名 Point.kt

```
open class Point(var x: Double, var y: Double) {
    open fun show() {
        print("($x, $y)")
    }
    fun move(dx: Double, dy: Double) {
        x += dx; y += dy
    }
    fun moveAndShow(dx: Double, dy: Double) {
        move(dx, dy)
        show()
    }
}

```

キーワード class のあとにクラス名が続く。そのあとに丸括弧「(」～「)」の中に囲んで、コンストラクターの仮引数の宣言を続ける。このコンストラクターはプライマリー（つまり第一の）コンストラクターと呼ばれる。 引数の型や数が異なる、第二、第三のコンストラクターを定義することも可能だが、ここではその説明を割愛する。

プライマリーコンストラクターの引数で、そのままプロパティー（property — Kotlin のインスタンス変数）の初期値となるものは、再代入可能なプロパティーの場合は、キーワード var を、再代入不可能なプロパティーには、キーワード val を変数宣言の前につける。こうしておけば、改めてクラス定義の本体にプロパティーの宣言を書く必要はない。

ブレース「{」～「}」の間がクラス定義の本体で、メソッドや、プライマリーコンストラクターで定義していないプロパティーの宣言などを書く。

Kotlin のメソッドはデフォルトではオーバーライド (override) 不可である。サブクラスでオーバーライドできるようにするには、 fun の前に　　というキーワードをつけなければならない。また、サブクラスでオーバーライドするときは、 fun の前に　　というキーワードをつける。

ファイル名 Point.kt

```
class ColorPoint(x: Double, y: Double, var color: String): Point(x,
    override fun show() {
        print("<font color='$color'>($x, $y)</font>")
    }
}

class DeepPoint(x: Double, y: Double, var depth: Int): Point (x, y)
    override fun show() {
        if (depth <= 0) {
            print("$x, $y")
        } else {
            print("${"(".repeat(depth)}$x, $y${")".repeat(depth)}")
        }
    }
}
```

サブクラスを定義するときは、プライマリーコンストラクターの後にコロン「　」に続けてスーパークラスのコンストラクターの式を書く。 ColorPoint と DeepPoint のコンストラクターでは、 x と y は定義されるクラスのプロパティーではない （スーパークラスの Point で既に定義されている）ので var や val がついていないことに注意する。

ファイル名 PointTest.kt

```
fun main() {
    val points = arrayOf(Point(1.1, 2.5),
                         ColorPoint(2.1, 3.9, "red"),
                         DeepPoint(1.9, 0.25, 5))
    for (p in points) {
        p.moveAndShow(1.5, -0.5)
        println()
    }
}
```

関数 main を上のように定義するとき、このプログラムは

```
(2.6, 2.0)
```

と出力する。コンストラクターを呼び出すときに new というキーワードはつけないことにも注意する。

Kotlin のクラスのプロパティーには読み・書きにそれぞれゲッター (getter)・セッター (setter) というメソッドを関連づけることができる。その文法の詳細の説明は割愛する。

インタフェース

インタフェースを定義するときは interface というキーワードを用いる。そのあとにインタフェース名を続け、ブレース「{」～「}」の間に、プロパティーやメソッドの宣言を書く。

ファイル名 Movable.kt

```
interface Movable {
    fun tick()
    fun getLoc(): Point
}

```

あるクラスがインタフェースを実装 (implements) するときは、スーパークラスと同じようにコロン「:」のあとにインタフェース名を書く。複数のインタフェースを実装するときは、インタフェース名をコンマ「,」で区切って並べて書く。

オブジェクト

匿名クラスのオブジェクトを生成するときは、object というキーワードを用いる。それに続くブレース「{」～「}」の間に、プロパティーやメソッドの定義を書く。スーパークラスやインタフェースを指定するときは、object というキーワードに続けてコロン「:」のあとにスーパークラスのコンストラクターの呼び出し式、またはインタフェース名（の並び）をコンマ区切りで書く。

ファイル名 Movable.kt

```
fun main() {
    val m = object: Movable {
        var x = 0.0
        var y = 0.0
        override fun tick() {
            x += 24; y += 15
            if (x > 100) x -= 100
            if (y > 100) y -= 100
```

```
            if (x < 0)   x += 100
            if (y < 0)   y += 100
        }
        override fun getLoc(): Point = Point(x, y)
    }
    for (i in 1..10) {
        m.tick()
        m.getLoc().show()
    }
}
```

インタフェースのメソッドを実装するときは override キーワードが必要となる。

拡張関数

Kotlin では既存のクラスにメソッドを追加するかのように見せかけることができる。これを＿＿＿＿(＿＿＿＿＿＿＿＿)と呼ぶ。

拡張関数は fun クラス名.メソッド名(…) で定義を開始する。関数内部でレシーバー（ドット「.」の左側の引数）のメソッドやプロパティーは、その名前だけでアクセスすることができる。また、レシーバーそのものはキーワード＿＿で参照することができる。

ファイル名 ExtensionTest.kt

```
import kotlin.math.*

fun Point.rotate(theta: Double) {
    val x0 = x
    val y0 = y
    x = x0 * cos(theta) - y0 * sin(theta)
    y = x0 * sin(theta) + y0 * cos(theta)
}

```

次の例では上で定義した拡張関数の rotate を本来のメソッド（show や move など）と同じ文法で呼び出していることがわかる。

ファイル名 ExtensionTest.kt

```
fun main() {
    valjav p = Point(1.0, 0.0)
    p.rotate(PI / 6)
    p.show()
}
```

ただし、拡張関数はクラスを拡張したように"見せかけている"だけなので、本当にクラスを拡張しているのではない。拡張関数は動的束縛ではなく、静的に（コンパイル時に）推論される型から選択される。例えば次の例では、main で、変数 p の型が Point と宣言されているので、p.bar() の呼び出しには Point.bar() のほうが選択されて、

ファイル名 ExtensionTest2.kt

```
fun Point.bar() {
    print("Point class: ")
    show()
    println()
}

fun ColorPoint.bar() {
    print("ColorPoint class:")
```

```
    show()
    println()
}

fun main() {
    val p: Point = ColorPoint(1.0, 0.0, "black")
    p.bar()
}
```

```
```

と出力する。

K.6 関数リテラルと高階関数

関数リテラル

Kotlin の関数リテラルは、キーワード fun を使う書き方（　　）と、ブレース「{」～「}」・矢印「->」を使う書き方（　　）がある。

- 匿名関数:「fun (仮引数宣言の並び): 型 { 文の並び } 」または「fun (仮引数宣言の並び): 型 = 式」
- ラムダ式:「{ 仮引数宣言の並び -> 文の並び }」

ここで仮引数宣言の並びはコンマ「,」区切りである。また、関数の型は「(型, 型, …, 型) -> 型」のように書く。矢印の左辺の丸括弧の中が引数の型、右辺が戻り値の型である。

高階関数

高階関数 (higher-order function) は関数を引数としたり、戻り値としたりする関数である。Kotlin の標準ライブラリーには、いくつかの便利な高階関数が用意されている。例えば　　メソッドはリストや配列などのコレクションの要素に引数の関数を一斉に適用し、その戻り値のコレクションを返す。また　　メソッドはコレクションの要素のなかで、引数の関数の値を真にするような要素だけのコレクションを返す。

ファイル名 LambdaTest.kt

```
fun main() {
    val xs: List<Int> = listOf(2, 3, 5, 8)
    println(xs.map { x -> x * x })
    println(xs.filter { x -> x % 2 != 0 })
}
```

Kotlin では、関数・メソッドの最後の引数がラムダ式の場合、　　という約束事がある。特に、ラムダ式が唯一の引数であるときは丸括弧自体を省略して良い。

結局、上のプログラムは次のように出力する

```
```

シーケンス

シーケンス (Sequence) は配列やリストと同様にコレクションだが、遅延評価される（つまり、必要になるまで評価されない）という点が異なる。このため無限個の要素を持つシーケンスを定義することもできる。シーケンスを生成する関数としては generateSequence, sequence などがある。前者は初期値 a と"漸化式" f を受け取り、無限列 $a, f(a), f(f(a)), f(f(f(a))), \ldots$ を返す。後者は無引数のラムダ式を受け取り、ラムダ式の中の文を実行して　　関数に渡される値を順に生成する。いったん yield 関数が呼び出されると、ラムダ式の中の文の実行は中断されるが、シーケンスの次の要素が必要になったときに、yield の呼び出しの　　、やはり、次に yield に渡される値を返す。

もちろん無限列はそのまま出力しようとすると、プログラムが止まらなくなってしまう。次のプログラムでは take(n) メソッドで先頭の *n* 個の要素からなる有限列を取り出して、 toList() メソッドでリストに変換して出力している。

ファイル名 SequenceTest.kt

```
fun main() {
    val nums = generateSequence(1) { x -> x + 3 }
    println(nums.take(10).toList())
    val fibs = sequence {
        var a = 1; var b = 1
        while (true) {
            yield(a)
            val c = a
            a = b; b += c
        }
    }
    println(fibs.take(10).toList())
}
```

このプログラムは次のように出力する。

```

```

次のプログラムは、シーケンスによるエラトステネスの篩のアルゴリズムの実装である。ここで elementAt(n) は *n* 番目の要素を取り出すメソッド、 drop(n) は先頭の *n* 個の要素を除いた残りを返すメソッドである。

ファイル名 Primes.kt

```
fun main() {
    val primes = sequence {
        var nums = generateSequence(2) { x -> x + 1 }
        while (true) {
            val a = nums.elementAt(0)
            yield(a)
            nums = nums.drop(1).filter { b -> b % a != 0 }
        }
    }
    println(primes.take(20).toList())
}
```

このプログラムは次のように出力する

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 6
```