Kotlin 実習マニュアル

第零版

大黒学

Kotlin 実習マニュアル・第零版
著者——大黒学

2018 年　9 月　4 日（火）　第零版発行

目次

第 1 章　Kotlin の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、Python、Haskell、Swift、……というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「Kotlin 実習マニュアル」）は、Kotlin というプログラミング言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

1.2　言語処理系

1.2.1　ソフトウェアとハードウェア

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ソフトウェア」(software) と「ハードウェア」(hardware) と呼ばれます。ソフトウェアというのはプログラムなどのデータのことで、ハードウェアというのは物理的な装置のことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類ごとに決まっているひとつのものだけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムは、「ソースコード」(source code) と呼ばれます。それに対して、何らかのハードウェアが理解することができる言語で書かれたプログラムは、「バイナリーコード」(binary code) と呼ばれます。

1.2.2 コンパイラとインタプリタ

ソースコードをコンピュータのハードウェアに理解させるためには、そのためのプログラムが必要になります。そのような、ソースコードをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります）。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、ソースコードを機械語に翻訳してバイナリーコードを作るプログラムのことで、インタプリタというのは、ソースコードがあらわしている動作をコンピュータに実行させるプログラムのことです。

ソースコードを機械語に翻訳することを、プログラムを「コンパイルする」(compile) と言います。

1.2.3 JVM

物理的には存在しないコンピュータのハードウェアを仮想的に作り出すプログラムは、「仮想マシン」(virtual machine) と呼ばれます。

仮想マシンの実例のひとつとして、「Java 仮想マシン」(Java Virtual Machine, JVM) と呼ばれるものがあります。これは、Windows、macOS、Linux など、さまざまな OS の上で動作する仮想マシンです。この仮想マシンのバイナリーコードは、まったく同じものがそれらの OS の上で動作します。

Java 仮想マシンの機械語へ翻訳するコンパイラを持つプログラミング言語は、「JVM 言語」(JVM language) と呼ばれます。JVM 言語には、Java、Scala、Clojure、Groovy などがあります。Kotlin も、JVM 言語のひとつです。

1.2.4 Kotlin のコンパイラ

Kotlin の公式サイト (`https://kotlinlang.org/`) には、Kotlin のコンパイラが置かれているサイトへのリンクや、Kotlin についてのさまざまな文書があります。

Kotlin のコンパイラは、Kotlin で書かれたプログラムを JVM の機械語に翻訳することができます。

1.2.5 プログラムの入力

プログラムをファイルに保存したり、すでにファイルに保存されているプログラムを修正したりしたいときは、「テキストエディター」(text editor) と呼ばれるソフトを使います（テキストエディターは、単に「エディター」(editor) と呼ばれることもあります）。

それでは、何らかのテキストエディターを使って、Kotlin のプログラムを入力して、それをファイルに保存してみましょう。ちなみに、Kotlin のプログラムを保存するファイルに付ける拡張子は、`.kt` です。

次のプログラムを入力して、`hello.kt` という名前のファイルに保存してください。

```
fun main(args: Array<String>) {
    println("こんにちは、世界。")
}
```

このプログラムは、「こんにちは、世界。」という文字列を出力して、さらに改行を出力する、という動作をします。

1.2.6 Kotlin のプログラムのコンパイル

Kotlin で書かれたプログラムは、

`kotlinc` パス名

というコマンドによって、JVM の機械語にコンパイルすることができます。この中の「パス名」のところには、プログラムが保存されているファイルのパス名を書きます。

それでは、先ほど入力したプログラムを JVM の機械語にコンパイルしてみましょう。コマンドを入力するためのアプリ（Linux や macOS ならばターミナル、Windows ならば PowerShell）を起動して、プログラムのファイルがあるディレクトリ（フォルダ）をカレントディレクトリにして、

```
kotlinc hello.kt
```

というコマンドを入力してみてください。そうすると、コンパイラによってプログラムがコンパイルされます。

Kotlin のコンパイラが出力する JVM のバイナリーコードは、 `.class` という拡張子を持つ、「Java クラスファイル」(Java class file) と呼ばれるファイルに保存されます（Java クラスファイルは、単に「クラスファイル」(class file) と呼ばれることもあります)。

`hello.kt` という名前のファイルに保存されている Kotlin のプログラムをコンパイラにコンパイルさせると、コンパイラは、JVM のバイナリーコードを、

```
HelloKt.class
```

という名前のファイルに保存します。このように、Kotlin のプログラムをコンパイルすることによってできた JVM のバイナリーコードが保存されるファイルの名前は、ソースコードのファイル名から `.kt` という拡張子を取り除いて、名前の先頭の文字を大文字にして、名前の末尾に `Kt` を追加して、 `.class` という拡張子を追加したものになります。

1.2.7 JVM のバイナリーコードの実行

Kotlin で書かれたプログラムをコンパイルすることによってできた JVM のバイナリーコードは、

```
kotlin クラス名
```

というコマンドによって実行することができます。この中の「クラス名」というところには、Java クラスファイルの名前から `.class` という拡張子を取り除いた部分を書きます。

それでは、先ほどコンパイルしてできたプログラムを実行してみましょう。

先ほどのコンパイルでできた JVM のバイナリーコードは、

```
HelloKt.class
```

という名前のファイルに保存されていますので、

```
kotlin HelloKt
```

というコマンドを入力すれば、そのバイナリーコードを実行することができます。試してみましょう。そうすると、

```
こんにちは、世界。
```

という文字列が出力されるはずです。

1.2.8 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、翻訳または実行しようとしたプログラムにエラーが含まれていた場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、エラーを含んでいる次の Kotlin のプログラムをファイルに保存してください。

プログラムの例 `error.kt`

```
fun main(args: Array<String>) {
    brintln("こんにちは、世界。")
}
```

先ほどのプログラムとの相違点は、 `println` という正しい名前が、 `brintln` という間違った名前に変わっているというところです。このプログラムを Kotlin のコンパイラにコンパイルさせると、コンパイラは、次のようなエラーメッセージを出力します。

```
error.kt:2:5: error: unresolved reference: brintln
    brintln("こんにちは、世界。")
    ^
```

このエラーメッセージは、 `brintln` というのが何の名前なのか分からない、ということを述べています。

1.3 REPL

1.3.1 REPL の基礎

言語処理系は、「REPL」と呼ばれるものと、そうでないものとに分類することができます。REPL というのは、read-eval-print loop の略称で、次の三つの動作を延々と繰り返す、という動作をする処理系のことです。

(1) プログラムまたはその断片を読み込む (read)。
(2) 読み込んだプログラムまたはその断片を実行する (eval)。
(3) 結果を出力する (print)。

ですから、REPL を使うことによって、プログラムまたはその断片をキーボードから直接入力して、それがどのように動作するかということを即座に確かめる、ということができます。

1.3.2 REPL の起動

Kotlin のコンパイラも、REPL として動作させることができます。
Kotlin のコンパイラは、コマンドライン引数を何も書かずに、

```
kotlinc
```

というコマンドをシェルに入力することによって起動した場合、REPL として動作します。
それでは、実際に、Kotlin のコンパイラを REPL として動作するように起動してみてください。そうすると、

```
>>>
```

というプロンプトが表示されるはずです。

1.3.3 REPL のコマンド

Kotlin の REPL には、さまざまなものを入力することができます。「コマンド」(command) と呼ばれるものもそのひとつです。Kotlin の REPL のコマンドは、すべて、コロン (:) で始まります。
たとえば、`:help` というコマンドを Kotlin の REPL に入力すると、コマンドの使い方が表示されます。
それでは、ヘルプを出力させてみましょう。`:help` というコマンドを入力して、そののち改行を入力してみてください。

```
>>> :help
Available commands:
:help                    show this help
（中略）
>>>
```

1.3.4 REPL の終了

Kotlin の REPL は、`:quit` というコマンドを入力することによって終了させることができます。
それでは、実際に REPL を終了させてみてください。REPL が終了すると、Linux や macOS の場合はターミナルのプロンプトが、Windows の場合はコマンドプロンプトのプロンプトが表示されます。

1.3.5 式の入力

Kotlin のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。式については、第 2 章で詳しく説明することになりますが、とりあえずここでは、数学で使われる式のように、何らかの計算を書きあらわしたものだと考えてください。
Kotlin の REPL には、コマンドだけではなくて、式を入力することもできます。Kotlin の REPL に式を入力すると、Kotlin の REPL は、それがあらわしている計算を実行して、その結果を出力します。
それでは、式を Kotlin の REPL に入力してみましょう。たとえば、`5 + 3` という式を入力して、そののち改行を入力してみてください。そうすると、次のように、入力した式があらわしている計算が実行されて、`8` という結果が出力されます。

```
>>> 5 + 3
8
```

1.3.6 ロード

Kotlin の REPL は、ファイルに保存されているソースコードをコンパイルして、その結果としてできたバイナリーコードを実行する、ということもできます。REPL がソースコードをコンパイルして、バイナリーコードを実行することを、プログラムを「ロードする」(load) と言います。

それでは、Kotlin のプログラムをファイルに保存して、そのプログラムを REPL にロードさせてみましょう。

まず、次のプログラムを入力して、ファイルに保存してください。

プログラムの例 `world.kt`

```
println("こんにちは、世界。")
```

このプログラムは、「こんにちは、世界。」という文字列を出力して、さらに改行を出力する、という動作をします。

ファイルに保存されているプログラムを REPL にロードさせたいときは、

`:load` パス名

という形のコマンドを REPL に入力します。この中の「パス名」のところには、ロードさせたいプログラムが格納されているファイルのパス名を書きます。ですから、先ほど入力したプログラムは、

```
:load world.kt
```

というコマンドを REPL に入力することによって、REPL にロードさせることができます。

```
>>> :load world.kt
こんにちは、世界。
```

1.4 文

1.4.1 プログラムの構造

プログラムという文書は、構文的な部品から構成されます。そして、プログラムを構文的に構成している部品には、いくつかの階層があります。もっとも下にある階層は文字で、もっとも上にある階層はプログラム全体です。

Kotlin というプログラミング言語の場合、プログラムを構成している部品の中間的な階層として、「文」(statement) と呼ばれるものや、「式」(expression) と呼ばれるものがあります。

第 1.3 節で、

```
println("こんにちは、世界。")
```

というプログラムを紹介しましたが、このプログラムは、その全体が 1 個の式になっています。また、この式の中にある、

```
"こんにちは、世界。"
```

という部分も、1 個の式です。このように、式という部品は、その中にさらに式が含まれていることもあります。つまり、式という部品は入れ子にすることができる、ということです。式だけではなくて、文も、入れ子にすることができます。

ちなみに、文字列を二重引用符 (") で囲んだものは、囲まれている文字列のデータを生成する、ということをあらわしている式です。このような、特定のデータを生成する式については、第 2.2 節で詳しく説明します。

Kotlin では、すべての式は、そのままの形のものを文としても扱うことができます。たとえば、

```
println("こんにちは、世界。")
```

という式も、この形のままで、文として扱うことができます。

文として扱われる式は、「式文」(expression statement) と呼ばれます。

1.4.2　文の列

Kotlin のプログラムの中には、文を、いくつでも並べて書くことができます。プログラムの中に文がいくつか並んでいる場合、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

文を並べることによって文の列を作る場合、それぞれの文は、改行またはミコロン (;) で区切る必要があります。改行を使って文を区切ると、文は縦に並ぶことになります。文を縦ではなく横に並べたいときは、

文; 文; 文; ··· 文

というように、それらの文をセミコロンで区切ります。

それでは、実際に、2 個以上の文から構成される文の列を書いて、それがどのように実行されるかということを試してみましょう。

プログラムの例　sequence.kt

```
println("菜の花や")
println("月は東に")
println("日は西に")
```

実行例

```
>>> :load sequence.kt
菜の花や
月は東に
日は西に
```

1.5　空白と改行と注釈

1.5.1　空白と改行

空白という文字（スペースキーを押したときに入力される文字）と改行という文字（エンターキーを押したときに入力される文字）は、Kotlin のプログラムの意味に影響を与えません。たとえば、

```
println("こんにちは、世界。")
```

という式は、

```
println　（　"こんにちは、世界。"　）
```

と書いたとしても同じ意味になりますし、

```
println(
"こんにちは、世界。"
)
```

と書いたとしても同じ意味になります。

ただし、二重引用符 (") で囲まれている文字列に空白を挿入した場合は、その空白を含んだ文字列のデータが生成されます。たとえば、

```
"こ　ん　に　ち　は　、　世　界　。"
```

という式は、

こ　ん　に　ち　は　、　世　界　。

という文字列のデータを生成します。

名前の途中には、空白も改行も挿入することができません。ですから、println という名前を、

```
p r i n t l n
```

と書くことはできません。

1.5.2　REPL における改行

Kotlin の REPL には、式を入力することも文を入力することもできます。

Kotlin の REPL は、改行が入力されたとき、入力された式または文が、そこまでで完成しているかどうかを判断して、すでに完成していると判断した場合は、それを実行します。それに対して、まだ完成していないと判断した場合は、 ... というプロンプトを出力することによって、続きを入力することを人間に要求します。たとえば、5 + 3という式を入力する場合、5 +まで入力したのちに改行を入力したとすると、REPL は、式がまだ完成していないと判断して、 ... というプロンプトを出力します。

```
>>> 5 +
... 3
8
```

1.5.3 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。

注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。

Kotlin には、「この部分は注釈である」ということをコンパイラに認識してもらう方法が、二つあります。

そのうちのひとつは、スラッシュスラッシュ(//) を使う方法です。プログラムの中に // を書くと、その直後から最初の改行までが注釈だと認識されます。たとえば、Kotlin のコンパイラは、

```
// 私は注釈です。
```

という記述を、注釈だと認識します。

それでは、REPL を使って、スラッシュスラッシュから改行までの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。まず、次の式を REPL に入力してみてください。

```
5 + 3 + 7
```

そうすると、REPL は、この式があらわしている計算を実行して、その結果として得られた 15 という整数を出力します。

それでは、次の式を REPL に入力してみてください。

```
5 + 3 // + 7
```

この場合、入力した式の中にある // + 7 という部分は、スラッシュスラッシュと改行のあいだに書かれていますので、Kotlin の REPL はその部分を注釈だと認識します。ですので、REPL は、計算の結果を 8 と出力します。

プログラムの一部分を注釈として認識してもらう方法の二つ目は、スラッシュアスタリスク(/*) とアスタリスクスラッシュ(*/) でそれを囲むという方法です。たとえば、Kotlin のコンパイラは、

```
/* 私は注釈です。 */
```

という記述を、注釈だと認識します。

それでは、REPL を使って、スラッシュアスタリスクからアスタリスクスラッシュまでの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。次の式を REPL に入力してみてください。

```
5 + /* 3 + */ 7
```

この場合、入力した式の中にある /* 3 + */ という部分は、スラッシュアスタリスクとアスタリスクスラッシュのあいだに書かれていますので、Kotlin の REPL はその部分を注釈だと認識します。ですので、REPL は、計算の結果を 12 と出力します。

改行を含んでいる注釈、つまり 2 行以上の注釈も、その全体を /* と */ で囲むことによって、注釈だと認識してもらうことができます。たとえば、Kotlin のコンパイラは、

```
/* 私は、改行を
含んでいる注釈です。 */
```

という記述を、注釈だと認識します。

1.5.4 コメントアウトとアンコメント

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「アンコメントする」(uncomment) と言います。

第2章 式

2.1 式の基礎

2.1.1 式と評価と値

Kotlin のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。

式というのは、コンピュータによる何らかの動作をあらわしています。ですから、コンピュータは、式があらわしている動作を実行することができます。ただし、式の場合は、「実行する」(execute) とは言わずに、「評価する」(evaluate) と言うのが普通です。

式を評価すると、その結果として一つのデータが得られます。式を評価することによって得られるデータは、その式の「値」(value) と呼ばれます。

「評価する」という言葉は、「値を求める」というニュアンスを含んでいます。式を実行することを、「実行する」と言わずに「評価する」と言うのは、「式の実行というのは、単なる動作の実行ではなくて、値を求めるという志向性を持った動作の実行である」という意識が強く働いているからです。

2.1.2 式の構造

式というのは、プログラムというものを組み立てるための部品のようなものだと考えることができます。

多くの場合、式という部品は、より単純な式を組み合わせることによって作られています。たとえば、`5 + 3`というのはひとつの式ですが、この式は、より単純な式を組み合わせることによって作られています。

`5 + 3`という式の中にある`5`という部分は、5 という整数を求めるという動作をあらわしている式です。したがって、REPL に対して`5`と入力すると、その式が評価されて、その値が出力されます。

```
>>> 5
5
```

同じように、`3`という部分も、3 という整数を求めるという動作をあらわしている式です。つまり、`5 + 3`という式は、`5`という式と`3`という式を、`+`というものをあいだにはさんで結びつけることによってできているわけです。

どんな式でも、それ自体を、さらに複雑な式の部品にすることができます。たとえば、`5 + 3`という式を部品にして、`5 + 3 - 7`という式を作ることができます。式というのは、組み合わせることによっていくらでも複雑なものを作ることができるという、そんな性質を持っている部品なのです。

2.2 リテラル

2.2.1 リテラルの基礎

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、`481`のような、何個かの数字を並べることによってできる列は、リテラルの一種です。

リテラルを評価すると、それによって生成されたデータが、その値として得られます。たとえば、`481`というリテラルを評価すると、それによって生成された 481 という整数のデータが、その値として得られます。

```
>>> 481
```

```
481
```

2.2.2 整数リテラル

整数のデータを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。
整数リテラルは、次の 3 種類に分類することができます。

- 10 進数リテラル (decimal integer literal)
- 16 進数リテラル (hexadecimal integer literal)
- 2 進数リテラル (binary integer literal)

これらの整数リテラルの相違点は、それらの名前が示しているとおり、整数を表現するための基数です。つまり、それぞれの整数リテラルは、10、16、2 のそれぞれを基数として整数を表現します。

10 進数リテラルは、`481` とか `3007` というような、数字だけから構成される列です。たとえば、`481` という 10 進数リテラルを評価すると、481 というプラスの整数が値として得られます。

16 進数リテラルと 2 進数リテラルは、基数を示す接頭辞を先頭に書くことによって作られます。基数を示す接頭辞は、16 進数は `0x` で、2 進数は `0b` です。たとえば、`0xff` と `0b11111111` は、どちらも、255 という整数を生成します。

```
>>> 0xff
255
>>> 0b11111111
255
```

2.2.3 浮動小数点数リテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。

浮動小数点数のデータを生成するリテラルは、「浮動小数点数リテラル」(floating point literal) と呼ばれます。

`0.003`、`41.56`、`723.0` というような、ドット (`.`) という文字の左右に数字の列を書いたものは、浮動小数点数のデータを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

```
>>> 3.14
3.14
```

浮動小数点数を生成するリテラルとしては、

a `e` b

という形のものを書くことも可能です（ `e` は大文字でもかまいません)。a のところには、ドットを 1 個だけ含むか、またはまったく含まない数字の列を書くことができて、b のところには、数字の列または左側にマイナスのある数字の列を書くことができます。この形のリテラルを評価すると、

$a \times 10^b$

という浮動小数点数が生成されます。

リテラル	意味
`3e8`	3×10^8
`6.022e23`	6.022×10^{23}
`6.626e-34`	6.626×10^{-34}

```
>>> 3e8
3.0E8
```

2.2.4 マイナスの数値を生成する式

マイナス (`-`) という文字を書いて、その右側に整数または浮動小数点数のリテラルを書くと、その全体は、マイナスの数値を生成する式になります。たとえば、`-56` という式はマイナスの 56

という整数を生成して、-0xff はマイナスの 255 という整数を生成して、-8.317 はマイナスの 8.317 という浮動小数点数を生成します。

```
>>> -56
-56
```

マイナスの数値を生成するこのような式の先頭に書かれるマイナスという文字は、リテラルの一部分ではなくて、第 2.3 節で説明することになる「演算子」(operator) と呼ばれるものです。

2.2.5 文字リテラル

文字のデータを生成するリテラルは、「文字リテラル」(character literal) と呼ばれます。

文字リテラルは、一重引用符 (') で文字を囲むことによって作られます。たとえば、

```
'A'
```

という記述は、文字リテラルです。

文字リテラルは、一重引用符で囲まれた中にある文字のデータを生成します。たとえば、

```
'A'
```

という文字リテラルを評価すると、A という文字のデータが生成されて、その文字のデータが値として得られます。

```
>>> 'A'
A
```

2.2.6 文字列リテラル

文字列のデータを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、二重引用符 (") で文字列を囲むことによって作られます。たとえば、

```
"namako"
```

という記述は、文字列リテラルです。

文字列リテラルは、二重引用符で囲まれた中にある文字列のデータを生成します。たとえば、

```
"namako"
```

という文字列リテラルを評価すると、namako という文字列のデータが生成されて、その文字列のデータが値として得られます。

```
>>> "namako"
namako
```

0 個の文字列から構成される文字列は、「空文字列」(empty string) と呼ばれます。空文字列は、2 個の連続した二重引用符を書くことによって生成することができます。

```
>>> ""

>>>
```

2.2.7 エスケープシーケンス

文字の中には、たとえば改ページや水平タブのように、そのままでは文字リテラルや文字列リテラルの中に書くことができない特殊なものもあります。

そのような特殊な文字のデータを生成する文字リテラルや、そのような文字を含む文字列のデータを生成する文字列リテラルを書きたいときには、「エスケープシーケンス」(escape sequence) と呼ばれる文字列が使われます。エスケープシーケンスは、必ず、バックスラッシュ(\) という文字で始まります[1]。

エスケープシーケンスには、次のようなものがあります。

\n	改行	\t	水平タブ
\r	キャリッジリターン	\b	バックスペース
\'	一重引用符	\"	二重引用符
\\	バックスラッシュ	\$	ドルマーク
\u*nnnn*	4 桁の 16 進数 *nnnn* を文字コードとする Unicode 文字		

[1] バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

エスケープシーケンスを一重引用符で囲んだものは、そのエスケープシーケンスが意味している文字のデータを生成する文字リテラルになります。

```
>>> '\''
'
>>> '\\'
\
>>> '\u004d'
M
```

エスケープシーケンスを含む文字列を二重引用符で囲むことによってできた文字列リテラルは、そのエスケープシーケンスが意味している文字を含む文字列のデータを生成します。

```
>>> "namako\numiushi\nkurage"
namako
umiushi
kurage
```

2.2.8 未加工文字列リテラル

文字列のデータを生成するリテラルとしては、文字列リテラルのほかに、「未加工文字列リテラル」(raw string literal) と呼ばれるものもあります。

未加工文字列リテラルは、二重引用符を 3 個連続して並べたもの（ `"""` ）で文字列を囲むことによって作られます。たとえば、 `"""namako"""` は、未加工文字列リテラルです。

未加工文字列リテラルも、文字列リテラルと同様に、3 連続の二重引用符で囲まれた中にある文字列を生成します。たとえば、 `"""namako"""` という未加工文字列リテラルは、 `namako` という文字列を生成します。

文字列リテラルと未加工文字列リテラルとの相違点のひとつは、バックスラッシュが特別扱いされるかどうかという点です。文字列リテラルの中に含まれているバックスラッシュは、エスケープシーケンスの 1 文字目として解釈されます。それに対して、未加工文字列リテラルの場合、その中に含まれているバックスラッシュは、特別扱いされることなく、無条件にそれ自身として解釈されます。ですから、未加工文字列リテラルは、Windows のパス名のような、バックスラッシュを何個も含む文字列を生成したいときに便利です。

```
>>> """c:\prog\tex\bin"""
c:\prog\tex\bin
```

文字列リテラルと未加工文字列リテラルとの第 2 の相違点は、改行という文字を含む文字列を生成する方法です。文字列リテラルの場合、改行を含む文字列を生成したいときは、エスケープシーケンスの `\n` を使います。それに対して、未加工文字列リテラルの場合は、改行をそのまま書きます。

```
>>> """namako
... umiushi
... kurage"""
namako
umiushi
kurage
```

2.2.9 文字列テンプレート

文字列リテラルと未加工文字列リテラルは、「式を評価して、その値を文字列の一部分にする」という機能を持っています。文字列リテラルと未加工文字列リテラルが持っているこの機能は、「文字列テンプレート」(string template) と呼ばれるものを書くことによって利用することができます。

文字列テンプレートは、

```
${式}
```

と書きます。つまり、ドルマーク (`$`) を書いて、その直後に、中括弧 (`{}`) で式を囲んだものを書くわけです。

文字列リテラルまたは未加工文字列リテラルの中に文字列テンプレートを書いておくと、そのリテラルが評価されたときに、文字列テンプレートの中の式も評価されて、その式の値を文字列に変換したものが文字列の一部分になります。

```
>>> "namako${0xff}umiushi"
namako255umiushi
>>> """namako${0xff}umiushi"""
namako255umiushi
```

2.3　演算子

2.3.1　演算子の基礎

Kotlin では、頻繁に必要となる単純な動作は、「演算子」(operator) と呼ばれるものを書くことによって記述することができます。演算子があらわしている動作は、「演算」(operation) と呼ばれます。

演算子を使いたいときは、演算子と式とを組み合わせた式を書きます。演算子は、それと式とを組み合わせた式の構造によって、次の 2 種類のどちらかに分類されます。

- 二項演算子 (binary operator)
- 単項演算子 (unary operator)

二項演算子があらわしている動作は「二項演算」(binary operation) と呼ばれ、単項演算子があらわしている動作は「単項演算」(unary operation) と呼ばれます。

2.3.2　二項演算子

「二項演算子」(binary operator) と呼ばれるグループに所属している演算子は、

式	二項演算子	式

という構造の式を作ります。

二項演算子を含む式を評価すると、原則的には、まず演算子の左右の式が評価されて、それらの式の値に対して、二項演算子があらわしている動作が実行されて、その結果が式全体の値になります。

2.3.3　算術演算子

数値に対する計算をあらわしている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。そして、算術演算子があらわしている動作は、「算術演算」(arithmetic operation) と呼ばれます。

二項演算子でかつ算術演算子であるような演算子としては、次のようなものがあります。

a `+` b　　a と b とを足し算（加算）する。
a `-` b　　a から b を引き算（減算）する。
a `*` b　　a と b とを掛け算（乗算）する。
a `/` b　　a を b で割り算（除算）した商を求める。
a `%` b　　a を b で割り算（除算）したあまりを求める。

2.3.4　整数に対する算術演算

算術演算を整数に対して実行すると、その結果も整数になります。

```
>>> 30 + 7
37
>>> 30 - 7
23
>>> 30 * 7
210
>>> 30 % 7
2
```

割り算（除算）の商を求める `/` という算術演算子は、割られる数（被除数）と割る数（除数）の両方が整数の場合は、整数の範囲だけで割り算をします。そして、その場合は、他の算術演算子と同じように、その結果は整数になります。

```
>>> 30 / 7
4
```

2.3.5 浮動小数点数に対する算術演算

算術演算を浮動小数点数に対して実行すると、その結果も浮動小数点数になります。

```
>>> 5.3 + 2.7
8.0
>>> 3e2 * 2e1
6000.0
```

算術演算の対象となる二つの数値の一方が浮動小数点数で他方が整数の場合も、結果は浮動小数点数です。

```
>>> 11.0 + 4
15.0
>>> 11 + 4.0
15.0
```

割り算（除算）の商を求める / という算術演算子は、割られる数（被除数）と割る数（除数）のどちらかが浮動小数点数の場合、整数の範囲で割り切れなければ、小数点より下の桁も求めます。

```
>>> 11 / 4
2
>>> 11.0 / 4
2.75
>>> 11 / 4.0
2.75
```

2.3.6 文字列の連結

+ という演算子は、数値の加算だけではなくて、文字列の連結という意味も持っています。

+ の左右に書かれた式の値の両方が文字列だった場合、 + は、数値の加算という動作ではなくて、文字列の連結という動作をします。

```
>>> "kitsune" + "udon"
kitsuneudon
```

+ の左右に書かれた式の値の一方が文字列で、他方が数値だった場合は、数値を文字列に変換したものと文字列とが連結されます。

```
>>> "uso" + 800
uso800
```

+ の左右に書かれた式の値の一方が文字列で、他方が文字だった場合は、文字列と文字とが連結されます。

```
>>> 'a' + "theism"
atheism
```

2.3.7 優先順位

ひとつの式の中に 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
2 + 3 * 4
```

という式は、

```
[2 + 3] * 4
```

という構造なのでしょうか。それとも、

```
2 + [3 * 4]
```

という構造なのでしょうか。

この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。

優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。

* と / と % は、 + と - よりも高い優先順位を持っています。ですから、

```
2 + 3 * 4
```

という式は、

```
2 + [3 * 4]
```

という構造だと解釈されます。

```
>>> 2 + 3 * 4
14
```

2.3.8　結合規則

ひとつの式の中に同じ優先順位を持っている 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
10 - 5 + 2
```

という式は、

```
[10 - 5] + 2
```

という構造なのでしょうか。それとも、

```
10 - [5 + 2]
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

+、-、*、/、%の結合規則は、左結合です。したがって、

```
10 - 5 + 2
```

という式は、

```
[10 - 5] + 2
```

という構造だと解釈されます。

```
>>> 10 - 5 + 2
7
```

2.3.9　丸括弧

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、+ と * とでは、* のほうが優先順位が高くなっていますので、

```
2 + 3 * 4
```

と書いたのでは、期待した結果は得られません。

演算子の優先順位や結合規則に縛られずに、自分が望んだとおりに式を解釈してほしい場合は、ひとまとまりの式だと解釈してほしい部分を、丸括弧 (()) で囲みます。そうすると、丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとまとまりの式だと解釈されます。

```
>>> (2 + 3) * 4
20
>>> 10 - (5 + 2)
3
```

2.3.10　単項演算子

「単項演算子」(unary operator) と呼ばれるグループに所属している演算子は、

```
[単項演算子] [式]
```

という構造の式か、または、

式 単項演算子

という構造の式を作ります。式の前に置かれる単項演算子は「前置単項演算子」(prefix unary operator) と呼ばれ、式の後ろに置かれる単項演算子は「後置単項演算子」(postfix unary operator) と呼ばれます。

Kotlin では、すべての単項演算子は、どの二項演算子よりも高い優先順位を持っています。

2.3.11 符号の反転

- という前置単項演算子は、数値の符号（プラスかマイナスか）を反転させる算術演算子です。

```
>>> -(3 + 5)
-8
>>> -(3 - 5)
2
```

2.4 関数呼び出し

2.4.1 関数

Kotlin では、何らかの動作を意味しているデータのことを「関数」(function) と呼びます。

コンピュータは、関数というデータがあらわしている動作を実行することができます。

関数があらわしている動作をコンピュータに実行させることを、関数を「呼び出す」(call) と言います。

関数というのはあくまでデータであって、それがあらわしている動作を実行するのはあくまでコンピュータです。しかし、プログラムを書く人間としては、「関数自体が、自分があらわしている動作を実行する」というイメージで考えるほうが思考が単純になります。ですから、このチュートリアルでも、これからは、関数自体が動作をするというイメージで説明をしていきたいと思います。

2.4.2 ライブラリー

言語処理系の中に取り込んで使うことのできる、言語処理系の外部にある機能は、「ライブラリー」(library) と呼ばれます。そして、言語処理系とともに配布されるライブラリーは、「標準ライブラリー」(standard library) と呼ばれます。

Kotlin のコンパイラも、標準ライブラリーとともに配布されています。

2.4.3 ユーザー定義関数と標準ライブラリー関数

関数を生成して、その関数に名前を与えることを、関数を「宣言する」(declare) と言います。

関数に与えられた名前は、「関数名」(function name) と呼ばれます。

Kotlin では、「関数宣言」(function declaration) と呼ばれる記述をプログラムの中に書くことによって、関数を自由に宣言することができます（関数宣言の書き方については、第 3 章で説明することにしたいと思います）。プログラムの中に関数宣言を書くことによって宣言された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

Kotlin の標準ライブラリーには、さまざまな関数も含まれています。標準ライブラリーに含まれている関数は、「標準ライブラリー関数」(standard library function) と呼ばれます。標準ライブラリー関数は、プログラムの中に関数宣言を書かなくても利用することができます。

2.4.4 引数と戻り値

関数は、何らかのデータを受け取って動作します。関数が受け取るデータは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます）。

関数は、自分の動作が終了したのちに、自分を呼び出した者にデータを返すことができます。関数が返すデータは、「戻り値」(return value) と呼ばれます。

関数 f を動作させて、引数としてデータ d をそれに渡すことを、「f を d に適用する」(apply f to d) と言うこともあります。

2.4.5　関数呼び出しの書き方

関数を呼び出したいときは、関数を呼び出すという動作をあらわす式を書きます。そのような式は、「関数呼び出し」(function invocation) と呼ばれます。

関数呼び出しは、

関数名(式$_1$, 式$_2$, ･･･)

と書きます。「関数名」のところには呼び出したい関数の名前を書いて、式$_1$, 式$_2$, ･･･のところには関数に渡す引数を求める式を書きます。

たとえば、hoge という名前の関数があって、この関数は引数として 3 個の整数を受け取るとしましょう。このとき、

```
hoge(78, 36, 24)
```

という関数呼び出しを書くことによって、hoge という関数を呼び出して、78、36、24 という整数を引数としてそれに渡すことができます。

2.4.6　関数呼び出しの評価

関数呼び出しは式ですから、評価することができます。関数呼び出しを評価すると、関数を呼び出してその関数に引数を渡すという、その関数呼び出しがあらわしている動作が実行されます。そして、呼び出された関数が返した戻り値が、関数呼び出しの値になります。たとえば、

```
moge(84)
```

という関数呼び出しを評価すると、moge という関数が呼び出されて、引数として 84 が渡されるわけですが、この関数呼び出しの値は、そのときに moge が返した戻り値です。

2.4.7　大きいほうの数値を求める標準ライブラリー関数

すべての標準ライブラリー関数には名前が与えられています。ですから、関数呼び出しの中に、標準ライブラリー関数の名前と、その関数に渡す引数を求める式を書くことによって、その関数を呼び出して、それに引数を渡すことができます。

Kotlin の標準ライブラリーには、kotlin.math.max という関数が含まれています。この関数は、引数として 2 個の数値を受け取って、それらのうちの大きいほうを戻り値として返します。

それでは、kotlin.math.max を呼び出す関数呼び出しを REPL に入力することによって、2 個の数値のうちの大きいほうを求めてみましょう。

```
>>> kotlin.math.max(5, 8)
8
```

2.4.8　引数と改行を出力する標準ライブラリー関数

第 1.3.6 項で、

```
println("こんにちは、世界。")
```

というプログラムを紹介しましたが、このプログラムは、その全体が、println という名前の関数を呼び出す関数呼び出しになっています。この関数呼び出しは、println を呼び出して、

```
"こんにちは、世界。"
```

という文字列を引数としてそれに渡します。

println は、引数として受け取ったデータを出力して、そののち 1 個の改行を出力する、という動作をする標準ライブラリー関数です。引数を何も渡さなかった場合は、改行だけを出力します。

```
>>> println()

>>>
```

println は、どんなデータを引数として受け取った場合も、戻り値として、「kotlin.Unit」という名前を持つデータを返します。ですから、println を呼び出す関数呼び出しの値は、常に kotlin.Unit です。

Kotlin の REPL は、入力された関数呼び出しの値が Kotlin.Unit だった場合、その値を出力しません。ですから、 println を呼び出す関数呼び出しを REPL に入力した場合、その値は出力されません。

println の戻り値は、それを println に出力させることによって確かめることができます。

```
>>> println(println("namako"))
namako
kotlin.Unit
```

2.4.9 引数を出力する標準ライブラリー関数

println と同じように、 print という標準ライブラリー関数も、受け取った引数を出力します。ただし、 println とは違って、そののちに改行を出力するということはしません。

```
>>> print("kitsune"); println("udon")
kitsuneudon
```

2.4.10 パッケージ

Kotlin のライブラリーは、「パッケージ」(package) と呼ばれる箱に格納されています。

kotlin.Unit という名前に含まれている kotlin という部分は、Kotlin の標準ライブラリーが格納されているパッケージの名前です。

パッケージは、その中にさまざまなものを入れることができる箱です。原則として、パッケージの中にあるものを指定するためには、そのパッケージの名前とその中にあるものの名前をドット (.) で区切って並べた、

パッケージ名 . 名前

という形の名前を書く必要があります。このような名前は、「完全修飾名」(fully qualified name) と呼ばれます。

kotlin.Unit という名前も、完全修飾名です。この名前は、 kotlin という名前のパッケージの中にある Unit という名前のものを指定しています。

パッケージの中には、パッケージを入れることもできます。たとえば、 kotlin というパッケージは、その中にさまざまなパッケージを持っています。 math というパッケージもそのひとつです。

kotlin.math.max という完全修飾名は、 kotlin というパッケージの中にある math というパッケージの中にある max という関数を指定しています。

2.4.11 import 宣言

プログラムの中に「 import 宣言」(import declaration) と呼ばれるものを書いておくと、パッケージの中にあるものを指定するときに、パッケージ名を省略することができるようになります。

import 宣言は、

import 完全修飾名

と書きます。たとえば、

```
import kotlin.math.max
```

という import 宣言を書くことによって、 kotlin.math.max は、単に max と書くことによって指定することができるようになります。

```
>>> import kotlin.math.max
>>> max(63, 27)
63
```

import 宣言を書くことによって、パッケージの中にあるものを指定するときにパッケージ名を省略することができるようにすることを、パッケージの中にある何々を「インポートする」(import) と言います。

println や print という標準ライブラリー関数は、 kotlin.io というパッケージの中に含まれているのですが、これらの標準ライブラリー関数は、デフォルトでインポートされていますので、 import 宣言を書かなくてもパッケージ名を省略することができます。同じように、 kotlin.Unit も、デフォルトでインポートされています。

import 宣言を書くことによって、特定のパッケージの中にあるすべてのものについて、そのパッケージ名を省略した名前によってそれらを指定することができるようにする、ということも可能です。それをしたいときは、完全修飾名の末尾の名前を書く場所に、名前ではなくアスタリスク (*) を書きます。たとえば、

```
import kotlin.math.*
```

という import 宣言を書くことによって、kotlin.math というパッケージの中にあるすべてのものについて、パッケージ名を省略した名前によってそれらを指定することができるようになります。

import 宣言を書くことによって、パッケージの中にあるすべてのものについて、パッケージ名を省略した名前によってそれらを指定することができるようにすることを、パッケージを「インポートする」(import) と言います。

2.5 プロパティーとメソッド

2.5.1 オブジェクト

第 2.1.1 項で、「式を評価すると、その結果として一つのデータが得られます」と書きましたが、厳密に言うと、これは正しくありません。実は、Kotlin では、式を評価することによって得られるものは、単なるデータではなくて、「オブジェクト」(object) と呼ばれるものなのです。

オブジェクトというのは、箱のようなものだと考えることができます。

オブジェクトという箱の中には、2 種類のものを入れることができます。ひとつの種類はオブジェクトで、もうひとつの種類は「メソッド」(method) と呼ばれるものです。メソッドについては、第 2.5.4 項で説明します。

2.5.2 プロパティー

オブジェクトという箱は、その中にオブジェクトを持つことができます。ただし、オブジェクトに対して、オブジェクトを自由に出し入れすることができる、というわけではありません。

オブジェクトに対してオブジェクトの出し入れをするためには、「プロパティー」(property) と呼ばれるものを使う必要があります。プロパティーというのは、オブジェクトという箱の中と外とのあいだで、オブジェクトの出し入れをする窓口のことだと考えることができます。

プロパティーに与えられた名前は、「プロパティー名」(property name) と呼ばれます。

プロパティーという窓口が扱っているオブジェクトは、

式 . プロパティー名

という形の式を評価することによって求めることができます。この中の「式」のところには、求めたいオブジェクトを扱っているプロパティーを持っているオブジェクトを求める式を書きます。そして、「プロパティー名」のところには、求めたいオブジェクトを扱っているプロパティーの名前を書きます。

2.5.3 文字列の長さ

文字列を構成している文字の個数は、その文字列の「長さ」と呼ばれます。

文字列のオブジェクトは、length という名前のプロパティーを持っています。このプロパティーは、文字列の長さのオブジェクトを扱っています。

実行例

```
>>> "umiushi".length
7
```

2.5.4 メソッド

「メソッド」(method) というのは、オブジェクトの中にある関数のことです。

メソッドに与えられた名前は、「メソッド名」(method name) と呼ばれます。

メソッドの仕事というのは、基本的には、自分が所属しているオブジェクトの中にあるオブジェクトの操作です。ひとつのオブジェクトはいくつかのメソッドを持っていて、それぞれのメソッドは、自分に固有の仕事を実行します。ですから、オブジェクトというのは、自分の中にあるオブジェクトを操作するためのさまざまな機能を持っている箱のことだと考えることができます。

オブジェクトにはさまざまな種類があります。たとえば整数のオブジェクトや文字列のオブジェクトなどです。オブジェクトがどんなメソッドを持っているかというのは、そのオブジェクトの種類ごとに決まっています。たとえば、文字列のオブジェクトは、その文字列から一部分だけを取り出すメソッドや、その文字列を整数へ変換するメソッドや、その文字列を浮動小数点数へ変換するメソッドなどを持っています。

2.5.5 メソッド呼び出し

メソッドを呼び出すためには、そのための式を評価する必要があります。メソッドを呼び出す式は、「メソッド呼び出し」(method invocation) と呼ばれます。

メソッド呼び出しは、

$\boxed{式_1}$. $\boxed{メソッド名}$ ($\boxed{式_2}$, $\boxed{式_3}$, ・・・)

という形の式です。この中の「$\boxed{式_1}$」のところには、呼び出したいメソッドを持っているオブジェクトを求める式を書きます。「メソッド名」のところには、呼び出したいメソッドの名前を書きます。そして、$\boxed{式_2}$, $\boxed{式_3}$, ・・・のところには、メソッドに渡す引数を求める式を書きます。

2.5.6 部分文字列を取り出すメソッド

文字列の一部分となっている文字列は、「部分文字列」(substring) と呼ばれます。

文字列のオブジェクトは、`substring` という名前のメソッドを持っています。これは、文字列から部分文字列を取り出して、その部分文字列を戻り値として返すメソッドです。

`substring` は、

s.substring(i, j)

というメソッド呼び出しを書くことによって呼び出すことができます。この式の中の s というところには、文字列のオブジェクトが値として得られる式を書きます。そして、i と j のところには整数を求める式を書きます。そうすると、i 番目の文字から始まって $(j-1)$ 番目の文字で終わる部分文字列が s から取り出されて、その部分文字列が戻り値として返ってきます（文字の番号は、先頭の文字を 0 番目と数えます)。たとえば、

```
"isoginchaku".substring(3, 9)
```

というメソッド呼び出しを書くことによって、`isoginchaku` という文字列の 3 番目の文字から始まって 8 番目の文字で終わる部分文字列を求めることができます。

それでは、インタラクティブシェルを使って、文字列のオブジェクトが持っている `substring` というメソッドを呼び出してみてください。

実行例

```
>>> "isoginchaku".substring(3, 9)
gincha
```

2.5.7 余分な改行と空白を削除するメソッド

文字列のオブジェクトは、`trimMargin` という名前のメソッドを持っています。これは、文字列から余分な改行と空白を削除することによってできた文字列を戻り値として返すメソッドです。このメソッドが削除するのは、文字列の先頭にある改行、それぞれの行の先頭から特定の文字までの空白の列、そして、末尾にある改行と空白の列です。

引数を何も渡さないで `trimMargin` を呼び出した場合、削除する空白の列の末尾を指定する文字は、縦棒 (|) です。

```
>>> """
...     |namako
...     |umiushi
...     |kurage
...     """.trimMargin()
namako
umiushi
kurage
```

引数として文字列を渡して `trimMargin` を呼び出した場合、その文字列が、削除する空白の列の末尾を指定することになります。

```
>>> """
...     //namako
...     //umiushi
...     //kurage
...     """.trimMargin("//")
namako
umiushi
kurage
```

2 行以上の文字列を未加工文字列リテラルで生成する場合、trimMargin を使うことによって、改行や空白でそれを読みやすく整形することができます。

2.5.8 文字列を整数へ変換するメソッド

文字列のオブジェクトは、toInt という名前のメソッドを持っています。これは、文字列を整数に変換して、その整数を戻り値として返すメソッドです。

toInt は、

s.toInt()

というメソッド呼び出しを書くことによって呼び出すことができます。この式の中の s というところには、整数をあらわしている文字列のオブジェクトが値として得られる式を書きます。

それでは、インタラクティブシェルを使って、文字列のオブジェクトが持っている toInt というメソッドを呼び出してみてください。

実行例

```
>>> "700".toInt() + "60".toInt()
760
```

文字列のオブジェクトが持っている toInt は、

s.toInt(r)

というメソッド呼び出しを書くことによって呼び出すこともできます。この式の中の s というところには、文字列のオブジェクトが値として得られる式を書きます。そして、r のところには、位取り記数法の基数を求める式を書きます（基数として使うことができるのは、2 から 36 までの文字列です）。そうすると、toInt は、s を r 進数とみなして、それを整数に変換した結果を戻り値として返します。たとえば、

```
"11111111".toInt(2)
```

というメソッド呼び出しを書くことによって、11111111 という文字列を 2 進数とみなして、それを整数に変換した結果を求めることができます。

それでは、インタラクティブシェルを使って、さまざまな基数の位取り記数法で表記した文字列を整数に変換してみてください。

実行例

```
>>> "11111111".toInt(2)
255
>>> "377".toInt(8)
255
>>> "ff".toInt(16)
255
>>> "7v".toInt(32)
255
```

ちなみに、すでに確かめたように、文字列のオブジェクトが持っている toInt を、引数を渡さないで呼び出した場合は、文字列を 10 進数とみなして、それを整数に変換した結果が返ってきます。

2.5.9 文字列を浮動小数点数へ変換するメソッド

文字列のオブジェクトは、toDouble という名前のメソッドを持っています。これは、文字列を浮動小数点数に変換して、その浮動小数点数を戻り値として返すメソッドです。

toDouble は、

s.toDouble()

というメソッド呼び出しを書くことによって呼び出すことができます。この式の中の s というところには、数値をあらわしている文字列のオブジェクトが値として得られる式を書きます。

それでは、インタラクティブシェルを使って、文字列のオブジェクトが持っている toDouble というメソッドを呼び出してみてください。

実行例

```
>>> "0.7".toDouble() + "0.06".toDouble()
0.76
```

2.5.10 数値を文字列へ変換するメソッド

整数または浮動小数点数のオブジェクトは、toString という名前のメソッドを持っています。これは、整数または浮動小数点数を文字列に変換して、その文字列を戻り値として返すメソッドです。

toString は、

x.toString()

というメソッド呼び出しを書くことによって呼び出すことができます。この式の中の x というところには、整数または浮動小数点数のオブジェクトが値として得られる式を書きます。

それでは、インタラクティブシェルを使って、整数と浮動小数点数のオブジェクトが持っている toString というメソッドを呼び出してみてください。

実行例

```
>>> 700.toString() + 60.toString()
70060
>>> 0.7.toString() + 0.06.toString()
0.70.06
```

整数のオブジェクトが持っている toString は、

n.toString(r)

というメソッド呼び出しを書くことによって呼び出すこともできます。この式の中の n というところには、整数のオブジェクトが値として得られる式を書きます。そして、r のところには、位取り記数法の基数を求める式を書きます（基数として使うことができるのは、2 から 36 までの整数です)。そうすると、n を r 進法で表記した文字列が戻り値として返ってきます。たとえば、

255.toString(2)

というメソッド呼び出しを書くことによって、255 という整数を 2 進法で表記した文字列を求めることができます。

それでは、インタラクティブシェルを使って、整数をさまざまな基数の位取り記数法で表記した文字列を求めてみてください。

実行例

```
>>> 255.toString(2)
11111111
>>> 255.toString(8)
377
>>> 255.toString(16)
ff
>>> 255.toString(32)
7v
```

ちなみに、すでに確かめたように、整数のオブジェクトが持っている toString を、引数を渡さないで呼び出した場合は、整数を 10 進法で表記した文字列が返ってきます。

2.5.11 文字コードを文字へ変換するメソッド

整数のオブジェクトは、toChar という名前のメソッドを持っています。これは、整数を、それを文字コードとする文字に変換して、その文字を戻り値として返すメソッドです。

実行例

```
>>> 97.toChar()
a
```

2.5.12　文字を文字コードへ変換するメソッド

文字のオブジェクトは、toInt という名前のメソッドを持っています。これは、文字を、それの文字コードの整数に変換して、その整数を戻り値として返すメソッドです。

実行例

```
>>> 'a'.toInt()
97
```

第 3 章　識別子

3.1　識別子の基礎

3.1.1　識別子とは何か

Kotlin のプログラムでは、しばしば、オブジェクトに名前を与えておいて、その名前によってオブジェクトを識別する、ということをします。オブジェクトに名前として与えることのできるものは、「識別子」(identifier) と呼ばれます。

識別子を名前としてオブジェクトに与えることを、識別子にオブジェクトを「代入する」(assign) と言います。この操作を、識別子をオブジェクトに「束縛する」(bind) と言うこともあります。

3.1.2　識別子の作り方

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、英字、数字、アンダースコア (_) です。
- 識別子の先頭の文字として、数字を使うことはできません。
- 予約語 (reserved word) と同じものは識別子としては使えません。「予約語」(reserved word) というのは、用途があらかじめ予約されている単語のことで、次のようなものがあります。

```
as        else    !in         package   try
as?       false   interface   return    typealias
break     for     is          super     val
class     fun     !is         this      var
continue  if      null        throw     when
do        in      object      true      while
```

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  a8  namako  back_to_the_future
```

英字の大文字と小文字は区別されますので、たとえば、a と A は、それぞれを異なるオブジェクトに束縛することができます。

最後の例のように、アンダースコアは、複数の単語から構成される識別子を作るときに、空白の代わりとして使うことができます。

識別子として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`　使うことのできない文字を含んでいる。

`8a`　先頭の文字が数字。

`val`　同じ予約語が存在する。

3.2　変数

3.2.1　変数の基礎

データを入れることのできる箱は、「変数」(variable) と呼ばれます。

変数は、識別子によって識別されます。変数に与えられた識別子は、「変数名」(variable name) と呼ばれます。

変数を作ることを、変数を「宣言する」(declare) と言います。

データを変数に入れることを、データを変数に「代入する」(assign) と言います。

変数を宣言すると同時にそれにデータを代入することを、変数を「初期化する」(initialize) と言います。そして、そのときに変数に代入されるデータを、その変数の「初期値」(initial value) と呼びます。

3.2.2 参照

Kotlin でも、変数を扱うことができます。ただし、Kotlin の場合、変数に入れることができるデータは、「参照」(reference) と呼ばれるものだけです。参照というのは、オブジェクトを指し示すデータのことです。オブジェクトは、それを指し示す参照を変数に入れておくことによって、それを保持しておくことができます。

Kotlin では、オブジェクトを指し示す参照を変数に入れることを、「オブジェクトを変数に代入する」と言います。つまり、オブジェクトを変数に代入したとしても、実際に変数に入るのはそのオブジェクトを指し示す参照であって、オブジェクトそのものではないということです。

オブジェクトを指し示す参照が変数に入っているとき、その変数はそのオブジェクトを「参照している」(refer) と言います。

3.2.3 変更可能な変数と変更不可能な変数

Kotlin の変数には、変更可能な (mutable) ものと変更不可能な (immutable) ものという 2 種類のものがあります。

変更可能な変数は、オブジェクトをそこに何度でも代入することができます。それに対して、変更不可能な変数は、宣言のときに初期化することはできますが、そののちに別のオブジェクトを代入することはできません。

すでに宣言されている変更可能な変数に対して別のオブジェクトを代入する方法については、第 3.4 節で説明します。

3.2.4 変数宣言

変数は、「変数宣言」(variable declaration) と呼ばれる文を書くことによって宣言することができます。

変更不可能な変数を宣言して初期化する変数宣言は、基本的には、

`val` 識別子 `=` 式

と書きます（`val` は value に由来する略語）。それに対して、変更可能な変数を宣言して初期化する変数宣言は、基本的には、

`var` 識別子 `=` 式

と書きます（`var` は variable に由来する略語）。どちらの場合も、「識別子」のところには変数名として変数に与えたい識別子を書いて、「式」のところには初期値として変数に代入したいオブジェクトを求める式を書きます。

変数宣言を実行すると、変数が宣言されて、それが初期化されます。

3.2.5 変数名の値

変数名は、式として評価することができます。変数名を評価すると、その変数に代入されているオブジェクトが、その値として得られます。

```
>>> val namako = 48
>>> namako
48
>>> val umiushi = "I am a seahare."
>>> umiushi
I am a seahare.
```

3.3 型

3.3.1 型の基礎

プログラミングにおいては、同じ性質を持つデータの集合のことを「型」(type) と呼びます。

Kotlin においては、オブジェクトというものはかならず、何らかの型に所属しているものとして扱われます。

オブジェクト O が型 T に所属しているとき、「O は T を持つ」という言い方をすることもあります。

型は、「型名」(type name) と呼ばれる名前によって識別されます。単純な型は、「クラス名」(class name) と呼ばれる 1 個の識別子によって識別されますが、何個かのクラス名から構成される型名を持つ型もあります。

3.3.2 基本型

Kotlin のプログラムが扱うオブジェクトの型には、さまざまなものがあります。それらのうちでもっとも基本的な一群の型は、「基本型」(basic type) と呼ばれます。

基本型を持つオブジェクトとしては、数値、文字、文字列などがあります。

数値、文字、文字列のオブジェクトが持っている型は、次のようなクラス名によって識別されます。

`Int`	整数の型のひとつ。通常の整数リテラルの値は、この型を持つ。
`Double`	浮動小数点数の型のひとつ。通常の浮動小数点数リテラルの値は、この型を持つ。
`Char`	文字の型。
`String`	文字列の型。

3.3.3 変数の型

Kotlin においては、オブジェクトだけではなくて、変数も型を持っています。変数に代入することができるオブジェクトは、その変数と同じ型を持つものだけです。

変数の型は、それが宣言されたときに、それに代入される初期値の型によって決定されます。たとえば、

```
val namako = 48
```

という変数宣言によって変数を宣言した場合、初期値の 48 が `Int` という型を持つ整数ですので、`namako` という変数も `Int` という型を持つことになります。

3.3.4 型を明示した変数の宣言

変数の型は、初期値の型によって自動的に決定されるわけですが、変数宣言の中に、変数が持つべき型を明示することも可能です。

型を明示して、変更不可能な変数を宣言する変数宣言は、

```
val 識別子 : 型名 = 式
```

と書きます。同じように、型を明示して、変更可能な変数を宣言する変数宣言は、

```
var 識別子 : 型名 = 式
```

と書きます。

```
>>> val namako: Int = 48
>>> namako
48
>>> val umiushi: String = "I am a seahare."
>>> umiushi
I am a seahare.
```

明示した型と初期値の型とが一致しない場合は、エラーになります。

```
>>> val hitode: Int = "I am a starfish."
error: type mismatch: inferred type is String but Int was expected
val hitode: Int = "I am a starfish."
                  ^
```

3.4 代入演算子

3.4.1 代入演算子の基礎

第 3.2.3 項で説明したように、変数には変更可能なものと変更不可能なものがあって、前者はオブジェクトをそこに何度でも代入することができますが、後者にできる代入は初期化だけです。

変更可能な変数を宣言したのちに、そこにオブジェクトを代入したいときは、「代入演算子」(assingment operator) と呼ばれる演算子を使います。代入演算子は、代入という動作をする演算に与えられた名前です。

代入演算子は、そのすべてが二項演算子で、それら以外のどの演算子よりも低い優先順位を持っています。

代入演算子は、式ではなくて文を作る演算子です。代入演算子によって作られる文は、「代入文」(assignment statement) と呼ばれます。

3.4.2 左辺値と右辺値

Kotlin では、式を評価すると、その結果としてオブジェクトが得られます。式を評価した結果として得られたオブジェクトは、その式の「値」(value) と呼ばれます。

式を評価することによって得られるものは、必ずしもオブジェクトだけとは限りません。場合によっては、オブジェクトだけではなくて、オブジェクトを指し示す参照を入れることのできる箱が得られることもあります。

式を評価することによって得られる箱は、その式の「左辺値」(left value) と呼ばれます。それに対して、式を評価することによって得られるオブジェクトは、その式の「右辺値」(right value) と呼ばれます。「値」(value) という言葉は、左辺値と右辺値の総称です。

値として左辺値を持つ式は、必ず右辺値も持ちます。左辺値を持つ式の右辺値は、得られた左辺値の中に格納されているオブジェクトです。

変数名は、左辺値が得られる式の一例です。変数名を評価すると、その名前を持っている変数が左辺値として得られます。そして、その変数の内容が右辺値として得られます。

3.4.3 単純代入演算子

= という演算子は、「単純代入演算子」(simple assignment operator) と呼ばれます。これは、代入演算子のうちで、もっとも単純な動作をするものです。

= を使って代入を実行する代入文は、

$\boxed{式_1}$ = $\boxed{式_2}$

と書きます。この形の代入文を実行すると、$式_1$ を評価することによって得られた左辺値に対して、$式_2$ を評価することによって得られたオブジェクトが代入されます。

それでは、単純代入演算子を使って変数にオブジェクトを代入して、そののち変数の内容を調べてみましょう。

```
>>> var a = 76
>>> a
76
>>> a = 38
>>> a
38
```

代入しようとしたオブジェクトの型と変数の型とが一致しない場合は、エラーになります。

```
>>> var a = 83
>>> a
83
>>> a = "I am a string."
error: type mismatch: inferred type is String but Int was expected
a = "I am a string."
    ^
```

3.4.4 拡張代入演算子

+=、 -=、 *=、 /=、 %= という 5 個の演算子は、「拡張代入演算子」(augmented assignment operator) と呼ばれます。

拡張代入演算子を使って代入を実行する代入文は、= を使った代入文と同じように、

式$_1$ 拡張代入演算子 式$_2$

と書きます。この形の代入文を評価すると、式$_1$ を評価することによって得られた右辺値と、式$_2$ を評価することによって得られた右辺値に対して演算が実行されて、その結果が、式$_1$ を評価することによって得られた左辺値に代入されます。

ところで、「演算が実行されて」と書きましたが、ここで実行される「演算」というのは、いったい何なのでしょうか。

拡張代入演算子は、すべて、イコール (=) の左側に何らかの二項演算子を書いた形になっています。左辺値の内容と式の値に対して実行される演算は、イコールの左側に書かれた二項演算子があらわしている演算です。つまり、☆が二項演算子だとすると、

a ☆= b

という代入文は、

a = a ☆ b

という代入文と同じ意味になるということです（ただし、前者は a が 1 回しか評価されないのに対して、後者は a が 2 回評価される、という相違があります)。たとえば、

```
a += 8
```

という代入文は、

```
a = a + 8
```

という代入文と同じ意味になります。

それでは、拡張代入演算子を使って、変数の内容を変化させてみましょう。

```
>>> var b = 7
>>> b
7
>>> b += 8
>>> b
15
```

3.5　インクリメントとデクリメント

3.5.1　インクリメントとデクリメントの基礎

代入演算子を使うことによって、オブジェクトを指し示す参照を入れることのできる箱の内容を変化させることができるわけですが、箱の内容を変化させる演算子は、代入演算子だけではありません。「インクリメント演算子」(increment operator) と呼ばれる ++ という演算子と、「デクリメント演算子」(decrement operator) と呼ばれる -- という演算子も、箱の内容を変化させます。

数値のオブジェクトが代入されている箱に対して、その数値よりも 1 だけ大きい数値のオブジェクトを代入することを、箱を「インクリメントする」(increment) と言います。そして、数値のオブジェクトが代入されている箱に対して、その数値よりも 1 だけ小さい数値のオブジェクトを代入することを、箱を「デクリメントする」(decrement) と言います。

インクリメント演算子は、箱をインクリメントする演算子で、デクリメント演算子は、箱をデクリメントする演算子です。

インクリメント演算子とデクリメント演算子は、どちらも単項演算子で、それぞれ、前置演算子と後置演算子の両方があります。つまり、次のような 4 個の演算子があるということです。

- 前置インクリメント演算子 (prefix increment operator)
- 後置インクリメント演算子 (postfix increment operator)
- 前置デクリメント演算子 (prefix decrement operator)
- 後置デクリメント演算子 (postfix decrement operator)

3.5.2　前置インクリメント演算子

前置インクリメント演算子を使う式は、

++式

と書きます。この形の式を評価すると、++の右側の式を評価することによって得られた左辺値がインクリメントされます。前置インクリメント演算子を使う式の値は、インクリメントされた後の箱の内容です。

```
>>> var a = 7
>>> ++a
8
>>> a
8
```

3.5.3 後置インクリメント演算子

後置インクリメント演算子を使う式は、

式++

と書きます。前置インクリメント演算と後置インクリメント演算とで違うのは、式の値です。後置インクリメント演算子を使う式の値は、インクリメントされる前の箱の内容です。

```
>>> var a = 7
>>> a++
7
>>> a
8
```

3.5.4 前置デクリメント演算子

前置デクリメント演算子を使う式は、

--式

と書きます。この形の式を評価すると、--の右側の式を評価することによって得られた左辺値がデクリメントされます。前置デクリメント演算子を使う式の値は、デクリメントされた後の箱の内容です。

```
>>> var a = 7
>>> --a
6
>>> a
6
```

3.5.5 後置デクリメント演算子

後置デクリメント演算子を使う式は、

式--

と書きます。後置デクリメント演算子を使う式の値は、デクリメントされる前の箱の内容です。

```
>>> var a = 7
>>> a--
7
>>> a
6
```

3.6 関数宣言

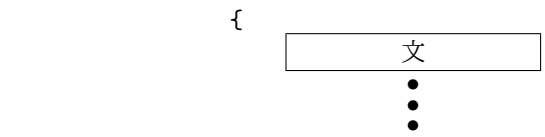
3.6.1 関数宣言の基礎

関数を生成して、識別子をその関数に名前として与えることを、関数を「宣言する」(declare)と言います。

関数は、「関数宣言」(function declaration) と呼ばれる記述を書くことによって宣言することができます。

3.6.2　ブロック

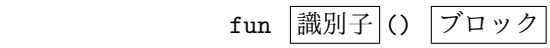
Kotlin では、



という形のものを「ブロック」(block) と呼びます。ブロックは、実行されることができて、実行されると、その中の文の列が実行されます。

3.6.3　引数を受け取らない関数の関数宣言の書き方

引数を受け取らない関数を宣言する関数宣言は、



と書きます。プログラムの中に関数宣言を書いておくと、その中に書かれたブロックを実行する関数が生成されて、「識別子」のところに書かれた識別子が、その関数に名前として与えられます。

たとえば、次のような関数宣言を書くことによって、namako という文字列を出力する関数を生成して、namako という識別子を名前としてその関数に与えることができます。

```
fun namako() {
    println("namako")
}
```

3.6.4　REPL への関数宣言の入力

REPL には、関数宣言を入力することもできます。それでは、関数宣言を REPL に入力して、宣言された関数を呼び出してみましょう。

```
>>> fun namako() {
...     println("namako")
... }
>>> namako()
namako
```

このように、REPL は、エンターキーが押されたときに、それが何かの入力の途中だった場合は、>>> ではなく ... をプロンプトとして出力します。

3.6.5　ファイルに保存された関数宣言のロード

関数は、ファイルに保存された関数宣言を REPL にロードさせることによって宣言することもできます。

それでは、関数宣言をファイルに保存しておいて、それを REPL にロードさせて、そして、宣言された関数を呼び出してみましょう。

まず、次の関数宣言を、world.kt というファイルに保存してください。

プログラムの例　world.kt

```
fun world() {
    println("この素晴らしき世界")
}
```

次に、ロードのコマンドを REPL に入力してください。

```
>>> :load world.kt
>>>
```

REPL からの応答は何もありませんが、これで関数が宣言されているはずですので、それを呼び出してみましょう。

```
>>> world()
この素晴らしき世界
```

関数宣言を REPL に入力する場合、それを直接 REPL に入力するという方法では、後日、も

う一度同じものを入力しようと思ったときに、同じ作業を繰り返す必要があります。ファイルに保存してからロードさせるという方法を使えば、同じ作業を繰り返す必要がなくなります。

練習問題 3.1 「あなたの今日の運勢は大吉です。」という文字列を出力する、`fortune`という関数を宣言してください。

実行例

```
>>> fortune()
あなたの今日の運勢は大吉です。
```

3.6.6 関数を宣言するという機能は何のためにあるのか

Kotlin は、関数を宣言するという機能を持っています。そのような、名前によって呼び出すことのできる動作を宣言するという機能は、Kotlin に限らず、ほとんどすべてのプログラミング言語が備えているものです。プログラミング言語は、いったい何のために、このような機能を備えているのでしょうか。

人間にとって、複雑なものを理解するというのは、容易なことではありません。ですから、複雑なものを作るときには、それを人間にとって理解しやすいものにする工夫をすることが、とても大切です。

複雑なものを人間にとって理解しやすいものにする上で重要なことは、少数の部品の組み合わせによって全体を構築するということです。個々の部品は、もしもそれ自体が複雑なものである場合は、それもまた、少数の部品の組み合わせによって構築される必要があります。

つまり、単純な部品を組み合わせることによって少し複雑な部品を作って、少し複雑な部品を組み合わせることによってさらに複雑な部品を作って、……というように、部品を階層的に組み合わせることによって構築されたものは、それがどれだけ複雑なものであっても人間にとって理解しやすい、ということです。

プログラムを書く場合も、もしもそれが複雑なものである場合は、それを人間にとって理解しやすいものにする工夫が必要になります。部品を階層的に組み合わせて構築することによって、人間にとって理解しやすいものを作ることができるという原理は、プログラムの場合にも有効です。プログラムを構築するための部品というのは、名前によって呼び出すことのできる動作です。

名前によって呼び出すことのできる動作を宣言するという機能がプログラミング言語に備わっているのはいったい何のためなのか、という問題の解答は、人間にとって理解しやすいプログラムを書くことができるようにするため、ということになります。

3.7 スコープ

3.7.1 スコープの基礎

識別子と、その識別子が名前として与えられたものとのあいだの関係が有効である、プログラムの上での範囲は、その識別子の「スコープ」(scope) と呼ばれます。

3.7.2 グローバルスコープ

Kotlin では、関数宣言の外で宣言された変数の名前は、ひとつのプログラムの全域というスコープを持つことになります。そのようなスコープは、「グローバルスコープ」(global scope) と呼ばれます。

関数宣言の内部もグローバルスコープの一部分ですから、関数宣言の中でグローバルスコープを持つ変数名を評価すると、その変数が参照しているオブジェクトが値として得られます。

プログラムの例 `global.kt`

```
val a = "グローバルスコープ"
function()

fun function() {
    println("function: ${a}")
}
```

実行例

```
>>> :load global.kt
function: グローバルスコープ
```

3.7.3 ローカルスコープ

Kotlin では、関数宣言の中で宣言された変数の名前は、その関数宣言の中だけというスコープを持つことになります。そのような、関数宣言の中だけという限定されたスコープは、「ローカルスコープ」(local scope) と呼ばれます。

グローバルスコープを持つ識別子と同一の識別子を、ローカルスコープを持つ識別子として使っても、問題はありません。

プログラムの例 `local.kt`

```
val a = "グローバルスコープ"
println(a)
funtion1()
println(a)

fun funtion1() {
    val a = "ローカルスコープ (functcion1)"
    println("funtion1: ${a}")
    funtion2()
    println("funtion1: ${a}")
}

fun funtion2() {
    val a = "ローカルスコープ (functcion2)"
    println("funtion2: ${a}")
}
```

実行例

```
>>> :load local.kt
グローバルスコープ
funtion1: ローカルスコープ (functcion1)
funtion2: ローカルスコープ (functcion2)
funtion1: ローカルスコープ (functcion1)
グローバルスコープ
```

3.7.4 ローカルスコープのメリット

ところで、「関数宣言の中で宣言された変数の名前は、その関数宣言の中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「ひとつの関数宣言の中で宣言された変数の名前は、それとは別の関数宣言の中でも有効である」という規則が定められていたと仮定するとどうなるか、ということについて考えてみましょう。その場合、名前として変数に識別子を与えるときには、その識別子がすでに別の関数宣言で使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の識別子を複数の関数宣言の中で使うと、思わぬ不具合が発生しかねません。

つまり、「関数宣言の中で宣言された変数の名前は、その関数宣言の中だけというスコープを持つ」という規則は、「関数宣言を書くときに、その関数宣言の外でどのような識別子が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

3.8 引数

3.8.1 仮引数

引数を受け取る関数を宣言するためには、その引数を受け取る変数を宣言する必要があります。そのような変数は、「仮引数」(parameter) と呼ばれます。関数が呼び出されたときに、その関数が受け取った引数は、仮引数に代入されることになります。

引数を受け取る関数を宣言する関数宣言は、

`func` 識別子 `(` 仮引数の宣言 `, ･･･ )` ブロック

と書きます。つまり、丸括弧の中に、仮引数の宣言を書くわけです。

仮引数の宣言は、基本的には、

識別子 : 型

と書きます。
次のプログラムの中で宣言されている argument という関数は、引数として受け取った整数を出力します。

プログラムの例 argument.kt

```
fun argument(a: Int) {
    println("引数は${a}です。")
}
```

実行例

```
>>> argument(68)
引数は 68 です。
```

仮引数は、ローカルスコープを持つことになります。つまり、仮引数のスコープは関数宣言の中だけです。

練習問題 3.2 a を整数とするとき、threetimes(a) という式で呼び出すと、a を 3 倍した結果を、

${a}の 3 倍は${a * 3}です。

という形で出力する関数を宣言してください。

実行例

```
>>> threetimes(8)
8 の 3 倍は 24 です。
```

練習問題 3.3 整数 m を、分を単位とする時間の長さとするとき、mtohm(m) という式で呼び出すと、m 分を何時間何分という形式に変換した結果を、

${m}分は${m / 60}時間${m % 60}分です。

という形で出力する関数を宣言してください。

実行例

```
>>> mtohm(160)
160 分は 2 時間 40 分です。
```

3.8.2 複数の引数を受け取る関数

複数の引数を受け取る関数を宣言したいときは、受け取る引数の個数と同じ個数の仮引数の宣言を、コンマで区切って並べます。たとえば、

```
fun namako(a: Int, b: Int, c: Int) {
    ⋮
}
```

という関数宣言で宣言された namako という関数は、3 個の引数を受け取ります。
関数宣言の中で並んでいるそれぞれの仮引数と、呼び出しの中で並んでいるそれぞれの式とは、原則としては、同じ順序で結び付けられます。したがって、先ほどの namako という関数を呼び出して、a に 24、b に 33、c に 81 を渡したいときは、

```
namako(24, 33, 81)
```

という関数呼び出しを書けばいい、ということになります。
次のプログラムの中で宣言されている three という関数は、引数として 3 個の整数を受け取って、それらの整数を出力します。

プログラムの例 three.kt

```
fun three(a: Int, b: Int, c: Int) {
    println("引数は${a}と${b}と${c}です。")
```

```
}
```

実行例

```
>>> three(24, 33, 81)
引数は 24 と 33 と 81 です。
```

練習問題 3.4 a と b を整数とするとき、divide(a, b) という式で呼び出すと、a を b で除算した結果を、

```
${a}割る${b}は${a / b}あまり${a % b}です。
```

という形で出力する関数を宣言してください。

実行例

```
>>> divide(60, 7)
60 割る 7 は 8 あまり 4 です。
```

練習問題 3.5 整数 h を、時間を単位とする時間の長さ、整数 m を、分を単位とする時間の長さとするとき、hmtom(h, m) という式で呼び出すと、h 時間 m 分を分に変換した結果を、

```
${h}時間${m}分は${h * 60 + m}分です。
```

という形で出力する関数を宣言してください。

実行例

```
>>> hmtom(2, 40)
2 時間 40 分は 160 分です。
```

3.8.3 名前付き引数

引数と仮引数とを対応させる方法としては、仮引数の宣言の順序と同じ順序で式を書くという方法のほかに、「名前付き引数」(named argument) と呼ばれるものを書くという方法もあります。

名前付き引数というのは、関数呼び出しの丸括弧の中に書かれる、

仮引数名 = 式

という形の記述のことです。この中の「仮引数名」のところに、呼び出される関数が持っている仮引数の名前を書くと、「式」のところに書かれた式の値が、その仮引数に代入されます。

名前付き引数は、どんな順序で並べても、引数と仮引数との対応は期待したとおりになります。

それでは、先ほどの three という関数を、名前付き引数を使って呼び出してみましょう。

```
>>> three(b = 33, c = 81, a = 24)
引数は 24 と 33 と 81 です。
```

3.8.4 デフォルト値

仮引数は、関数を宣言する段階で、あらかじめ特定のオブジェクトで初期化しておくことができます。仮引数に初期値として設定されているオブジェクトは、「デフォルト値」(default value) と呼ばれます。仮引数をあらかじめデフォルト値で初期化しておくと、その仮引数に引数が渡されなかった場合、その仮引数はデフォルト値で初期化されたままになりますので、引数の代わりとしてデフォルト値が使われることになります。

仮引数をデフォルト値で初期化したいときは、仮引数の宣言として、

識別子 : 型 = 式

という形のものを書きます。そうすると、イコールの右側に書かれた式の値が、仮引数のデフォルト値になります。

プログラムの例 three2.kt

```
fun three2(a: Int = 100, b: Int = 200, c: Int = 300) {
    println("引数は${a}と${b}と${c}です。")
}
```

three2という関数をこのように宣言したとすると、引数を 2 個しか渡さなかった場合にはcがデフォルト値のままになり、引数を 1 個しか渡さなかった場合にはbとcがデフォルト値のままになり、引数をまったく渡さなかった場合にはaとbとcがデフォルト値のままになります。

実行例

```
>>> three2(38, 47, 26)
引数は 38 と 47 と 26 です。
>>> three2(38, 47)
引数は 38 と 47 と 300 です。
>>> three2(38)
引数は 38 と 200 と 300 です。
>>> three2()
引数は 100 と 200 と 300 です。
```

名前付き引数を使えば、任意の仮引数をデフォルト値のままにすることができます。

実行例

```
>>> three2(b = 47, c = 26)
引数は 100 と 47 と 26 です。
>>> three2(a = 38, c = 26)
引数は 38 と 200 と 26 です。
>>> three2(c = 26)
引数は 100 と 200 と 26 です。
```

3.9 戻り値

3.9.1 戻り値の型

戻り値を返す関数を宣言するためには、その関数が返す戻り値の型を関数宣言の中に書く必要があります。

戻り値を返す関数を宣言する関数宣言は、仮引数の宣言を囲む丸括弧の右側にコロン (:) を書いて、その右側に戻り値の型を書きます。つまり、

```
func 識別子(仮引数の宣言, ··· ): 型 {
        文
        •
        •
        •
}
```

と書くわけです。

3.9.2 return 式

戻り値を返す関数を宣言するためには、戻り値の型だけではなくて、もうひとつ、書かないといけないものがあります。

それは、「return 式」(return expression) と呼ばれる式です。

return 式は、

```
return 式
```

と書きます。この中の「式」というところには、戻り値として返したいオブジェクトを求める式を書きます。return 式は、その中の式を評価して、その値を戻り値にして、関数の動作を終了させる、という動作をあらわしています。

a が 0 でない数値だとするとき、$1/a$ という数値は、a の「逆数」(reciprocal, multiplicative inverse) と呼ばれます。

次のプログラムの中で宣言されているreciproという関数は、1 個の数値を引数として受け取って、その逆数を戻り値として返します。

プログラムの例 recipro.kt

```
fun recipro(a: Double): Double {
    return 1 / a
}
```

実行例

```
>>> recipro(4.0)
0.25
```

n が自然数だとするとき、1 から n までの自然数の総和は、

$$\frac{n(n+1)}{2}$$

という計算をすることによって求めることができます。次のプログラムの中で宣言されている sum という関数は、1 個の自然数を引数として受け取って、1 からその自然数までの総和を戻り値として返します。

プログラムの例　sum.kt

```
fun sum(n: Int): Int {
    return n * (n + 1) / 2
}
```

実行例

```
>>> sum(10)
55
```

次のプログラムの中で宣言されている mtohm という関数は、分を単位とする時間の長さを引数として受け取って、それを何時間何分という形式に変換した結果（文字列）を戻り値として返します。

プログラムの例　mtohm2.kt

```
fun mtohm(m: Int): String {
    return "${m / 60}時間${m % 60}分"
}
```

実行例

```
>>> mtohm(160)
2 時間 40 分
```

練習問題 3.6　a を整数とするとき、square(a) という式で呼び出すと、a の 2 乗を戻り値として返す関数を宣言してください。

実行例

```
>>> square(7)
49
```

練習問題 3.7　整数 h を、時間を単位とする時間の長さ、整数 m を、分を単位とする時間の長さとするとき、hmtom(h, m) という式で呼び出すと、h 時間 m 分を分に変換した結果（整数）を戻り値として返す関数を宣言してください。

実行例

```
>>> hmtom(2, 40)
160
```

3.9.3　return 式を評価しないで終了した関数の戻り値

Kotlin では、あらゆる関数が戻り値を返します。戻り値の型が明記されておらず、return 式を評価しないで動作を終了する関数も、戻り値を返しています。その場合に関数が返すのは、kotlin.Unit というオブジェクトです。

```
>>> fun noreturn() {}
>>> println(noreturn())
kotlin.Unit
```

3.9.4　ブロック本体と式本体

関数宣言は、「頭部」(head) と「本体」(body) という二つの部分から構成されていると考えることができます。

関数宣言の頭部というのは、`fun` という予約語、関数に対して名前として与えられる識別子、仮引数の宣言、戻り値の型から構成される部分のことです。そして関数宣言の本体というのは、頭部よりもうしろに書かれる部分のことです。

関数宣言の本体の書き方には、二つのものがあります。第 3.6 節で説明した、ブロックを書くという書き方は、それらの二つの書き方のうちの一つです。

もう一つの本体の書き方というのは、

= 式

という形のものを書くというものです。つまり、まずイコール (`=`) を書いて、その右側に式を書く、という書き方です。この書き方で関数宣言の本体を書いた場合、その関数を呼び出すと、本体の中の式が評価されて、その値が戻り値になります。

ブロックの形の本体は、「ブロック本体」(block body) と呼ばれます。それに対して、イコールと式という形の本体は、「式本体」(expression body) と呼ばれます。ブロック本体よりも式本体のほうが記述が簡潔ですが、式本体を使うことができるのは、式を評価することだけが動作であるような関数を宣言する場合だけです。

次のプログラムは、1 から n までの自然数の総和を返す、先ほどの関数の関数宣言の本体を、ブロック本体から式本体に書き換えたものです。

プログラムの例　`sum2.kt`

```
fun sum(n: Int): Int = n * (n + 1) / 2
```

練習問題 3.8　第 3.9.2 項の練習問題で出題した、整数の 2 乗を戻り値として返す `square` という関数の宣言を、式本体を持つものに書き換えてください。

第 4 章　選択

4.1　選択の基礎

4.1.1　選択とは何か

第 1.4.2 項で説明したように、Kotlin のプログラムの中には、文を、いくつでも並べて書くことができて、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

ですから、いくつかの文を書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していく、という動作を記述することができます。しかし、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行する、ということも必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

4.1.2　真偽値

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

Kotlin では、真偽値は `Boolean` という型を持つオブジェクトです。この型は、第 3.3.2 項で説明した基本型のひとつです。

4.1.3　真偽値リテラル

真偽値のデータを生成するリテラルは、「真偽値リテラル」(Boolean literal) と呼ばれます。

真偽値リテラルとしては、次の二つのものがあります。

`true`　真。
`false`　偽。

```
>>> true
true
>>> false
false
```

4.1.4　選択を記述するための式

Kotlin で選択を記述したいときは、そのための式を書きます。
選択を記述するための式としては、次の二つのものがあります。

- `if` 式 (`if` expression)
- `when` 式 (`when` expression)

`if` 式については第 4.4 節で、 `when` 式については第 4.5 節で説明することにしたいと思います。

4.2　比較演算子

4.2.1　比較演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる二項演算子は、「比較演算子」(comparison operator) と呼ばれます。
多くのプログラミング言語では、比較演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。
比較演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、関係が成り立っているならば真、成り立っていないならば偽が、式全体の値になります。

4.2.2　大小関係

次の比較演算子を使うことによって、オブジェクトの大小関係について調べることができます。

a `>` b　　a は b よりも大きい。
a `<` b　　a は b よりも小さい。
a `>=` b　　a は b よりも大きいか、または a と b とは等しい。
a `<=` b　　a は b よりも小さいか、または a と b とは等しい。

```
>>> 8 > 5
true
>>> 5 > 8
false
>>> 5 > 5
false
>>> 5 >= 5
true
```

大小関係があるのは、数値と数値とのあいだだけではありません。文字と文字とのあいだにも、文字列と文字列とのあいだにも、大小関係があります。
文字と文字とのあいだの大小関係は、それぞれの文字の文字コードの大小関係です。

```
>>> 'b' > 'a'
true
>>> 'a' > 'b'
false
```

辞書の見出しは、「辞書式順序」(lexicographical order) と呼ばれる順序で並べられています。文字列と文字列とのあいだの大小関係は、辞書式順序で文字列を並べたときに、後ろにあるものは前にあるものよりも大きい、という関係です。

```
>>> "stay" > "star"
true
```

```
>>> "star" > "stay"
false
```

4.2.3 等しいかどうか

二つのオブジェクトが等しいかどうかということは、次の比較演算子を使うことによって調べることができます。

a == b　a と b とは等しい。

a != b　a と b とは等しくない。

```
>>> 5 == 5
true
>>> 5 == 8
false
>>> "star" == "star"
true
>>> "star" == "stay"
false
```

次のプログラムの中で宣言されている even という関数は、1 個の整数を受け取って、それが偶数ならば真を、そうでなければ偽を返します。

プログラムの例　even.kt

```
fun even(n: Int): Boolean = n % 2 == 0
```

実行例

```
>>> even(6)
true
>>> even(7)
false
```

練習問題 4.1　a と b を整数とするとき、divisible(a, b) という式で呼び出すと、a を b で除算したときに割り切れるならば真を、そうでなければ偽を返す関数を宣言してください。

実行例

```
>>> divisible(12, 4)
true
>>> divisible(12, 5)
false
```

4.3 範囲

4.3.1 範囲の基礎

Kotlin では、「範囲」(range) と呼ばれるオブジェクトを扱うことができます。

範囲は、整数または文字から構成される列の一種で、その名前のとおり、「ここからここまで」という範囲を指定することによって作られる列です。

範囲が持っている型のクラス名は、整数の範囲は IntRange で、文字の範囲は CharRange です。

範囲を生成したいときは、 .. という二項演算子を使った、

式$_1$ .. 式$_2$

という形の式を書きます。この形の式を評価すると、式$_1$ の値を下限、式$_2$ の値を上限とする範囲が生成されます（下限と上限も範囲に含まれます）。たとえば、

```
30..70
```

という式を評価すると、30 から 70 までという範囲（30 と 70 も含みます）が生成されて、その範囲が式の値になります。同じように、

```
'D'..'M'
```

という式を評価すると、大文字の D から M までという範囲（D と M も含みます）が生成されて、その範囲が式の値になります。

4.3.2 範囲の中にあるということを判定する演算子

in という二項演算子は、何らかの整数または文字が何らかの範囲の中にあるならば真、ないならば偽を求める、という動作をあらわしています。この演算子の右側には範囲を求める式を書いて、左側には、その範囲の中にあるかどうかを調べたい整数または文字を求める式を書きます。

```
>>> 50 in 30..70
true
>>> 80 in 30..70
false
>>> 'K' in 'D'..'M'
true
>>> 'S' in 'D'..'M'
false
```

4.3.3 範囲の中にないということを判定する演算子

!in という二項演算子は、何らかの整数または文字が何らかの範囲の中にないならば真、あるならば偽を求める、という動作をあらわしています。in と同じように、範囲を求める式を右側に書いて、その範囲の中にないかどうかを調べたい整数または文字を求める式を左側に書きます。

```
>>> 50 !in 30..70
false
>>> 80 !in 30..70
true
>>> 'K' !in 'D'..'M'
false
>>> 'S' !in 'D'..'M'
true
```

4.4 if 式

4.4.1 if 式の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「if 式」(if expression) と呼ばれる式を書きます。

さて、これから if 式について説明するわけですが、その前に、「条件式」という言葉を定義しておきましょう。この文章（「Kotlin 実習マニュアル」）の中では、評価すると値として真偽値が得られる式のことを、「条件式」と呼ぶことにします。

if 式は、基本的には、

if (条件式) $式_1$ else $式_2$

と書きます。この中の「条件式」のところには、先ほど定義した条件式を書きます。

if 式を評価すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、$式_1$が評価されます（その場合、$式_2$は評価されません）。条件式の値が偽だった場合は、$式_2$が評価されます（その場合、$式_1$は評価されません）。

つまり、if 式は、「もしも条件式が真ならば else の左の式を評価して、そうでなければ else の右の式を評価する」という動作を意味しているわけです。

```
>>> if (8 > 5) println("namako") else println("hitode")
namako
```

この場合、条件式の値は真ですので、namako が出力されて、hitode は出力されません。

```
>>> if (5 > 8) println("namako") else println("hitode")
hitode
```

この場合、条件式の値は偽ですので、hitode が出力されて、namako は出力されません。

4.4.2 if 式の値

if 式は、式の一種です。したがって、それを評価すると、その値が得られます。先ほど、if 式というのは基本的には、

if (条件式) $式_1$ else $式_2$

と書くと説明しましたが、この形の if 式を評価したとすると、条件式の値が真だった場合は式$_1$の値が if 式全体の値になって、条件式の値が真だった場合は式$_2$の値が if 式全体の値になります。

if 式だけを REPL に入力した場合、REPL は、それを式ではなくて文だと認識しますので、その値は出力されません。ですから、REPL を使って if 式の値を確かめるためには、REPL がその中の if 式を式だと認識するようなものを入力する必要があります。たとえば、if 式を丸括弧で囲んだものを入力すると、REPL はその中の if 式を式だと認識しますので、その値が出力されます。

```
>>> (if (8 > 5) "namako" else "hitode")
namako
>>> (if (5 > 8) "namako" else "hitode")
hitode
```

次のプログラムの中で宣言されている evenodd という関数は、1 個の整数を受け取って、それが偶数ならば「偶数」という文字列を返して、そうでなければ「奇数」という文字列を返します。

プログラムの例　evenodd.kt

```
fun evenodd(a: Int): String = if (a % 2 == 0) "偶数" else "奇数"
```

実行例

```
>>> evenodd(6)
偶数
>>> evenodd(7)
奇数
```

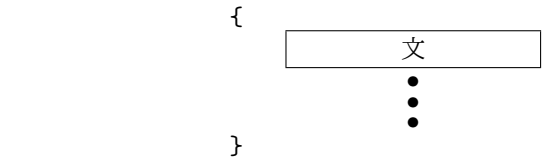
練習問題 4.2　a と b を整数とするとき、divideorzero(a, b) という式で呼び出すと、b が 0 ではないならば a を b で除算した商（整数）を、b が 0 ならば 0 を返す関数を宣言してください。

実行例

```
>>> divideorzero(33, 7)
4
>>> divideorzero(33, 0)
0
```

4.4.3　ブロック

Kotlin では、



という形のものを「ブロック」(block) と呼びます。ブロックは、実行されることができて、実行されると、その中の文の列が実行されます。

ブロックは、関数宣言の本体として使われるわけですが、ブロックが使われる場所はそれだけではありません。if 式も、その中にブロックを書くことができる場所のひとつです。

if 式の else の左側または右側には、ブロックを書くことができます。else の左側にブロックを書いたとすると、条件式の値が真だった場合に、そのブロックが実行されます。else の右側にブロックを書いたとすると、条件式の値が偽だった場合に、そのブロックが実行されます。

```
>>> if (8 > 5) {
...     println("kitsune")
...     println("tanuki")
... } else {
...     println("suzume")
...     println("karasu")
... }
kitsune
tanuki
>>> if (5 > 8) {
...     println("kitsune")
```

```
...     println("tanuki")
... } else {
...     println("suzume")
...     println("karasu")
... }
suzume
karasu
```

if 式を評価した結果としてその中のブロックが実行された場合、その if 式の値は、ブロックを実行したときに最後に評価された式の値です。

```
>>> (if (8 > 5) {
...     "kitsune"
...     "tanuki"
... } else {
...     "suzume"
...     "karasu"
... })
tanuki
>>> (if (5 > 8) {
...     "kitsune"
...     "tanuki"
... } else {
...     "suzume"
...     "karasu"
... })
karasu
```

4.4.4　else 以降を省略した if 式

しばしば、二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということがあります。そのような場合は、 else 以降を省略した if 式を書きます。つまり、

```
if (条件式) 式またはブロック
```

という形の if 式を書くわけです。この形の if 式を評価すると、条件式の値が真の場合は、式が評価されるか、またはブロックが実行されますが、条件式の値が偽の場合は何も起きません。

```
>>> if (8 > 5) println("namako")
namako
>>> if (5 > 8) println("namako")
>>>
```

else 以降を省略した if 式の値は、常に kotlin.Unit です。

```
>>> (if (8 > 5) "namako")
kotlin.Unit
>>> (if (5 > 8) "namako")
kotlin.Unit
```

次のプログラムの中で宣言されている mtohm という関数は、分を単位とする時間の長さを受け取って、それを「何時間何分」という形式であらわした文字列を出力します。ただし、「何分」という端数が出ない場合は、「何時間」という形式の文字列を出力します。

プログラムの例　mtohm3.kt

```
fun mtohm(m: Int) {
    print("${m / 60}時間")
    if (m % 60 != 0) println("${m % 60}分")
}
```

実行例

```
>>> mtohm(160)
2 時間 40 分
>>> mtohm(180)
3 時間
```

「何分」という部分を出力するという動作は、実行するかしないかを選択することになりますので、このように、 else 以降を省略した if 式を使って書くことができます。

練習問題 4.3 mtohm は、この節で紹介した宣言だと、60 分よりも短い時間を受け取った場合、「0 時間何分」という残念な形式の文字列を出力することになります。そこで、この欠点を改良して、そのような場合には「何分」という形式の文字列を出力するようにしてください。なお、0 を受け取った場合は、「0 分」という文字列を出力するようにしてください。

実行例

```
>>> mtohm(160)
2 時間 40 分
>>> mtohm(180)
3 時間
>>> mtohm(40)
40 分
>>> mtohm(0)
0 分
```

4.4.5 多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。多肢選択には、次の二つのタイプがあります。

- ひとつの式の値が何なのかということによって動作を選択するタイプ。
- いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプ。

Kotlin では、どちらのタイプの多肢選択も、「when 式」(when expression) と呼ばれる式を使うことによって記述することができます（when 式については次の節で説明します)。

いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプの多肢選択は、 when 式だけではなくて、 if 式を使って記述することも可能です。

if 式を使って、いくつかの条件による多肢選択を記述したいときは、 if 式の中に、

else if (条件式) 式またはブロック

という形のものを、必要なだけ書きます。

たとえば、3 個の条件による多肢選択は、

if (条件式$_1$) 式またはブロック$_1$

else if (条件式$_2$) 式またはブロック$_2$

else if (条件式$_3$) 式またはブロック$_3$

else 式またはブロック$_4$

という形の if 式を書くことによって記述することができます。この形の if 式を評価すると、値が真になる条件式が見つかるまで、条件式$_1$、条件式$_2$、条件式$_3$という順番で条件式が評価されていきます。値が真になる条件式が見つかった場合は、その条件式と同じ番号の「式またはブロック」が評価または実行されます（その場合、それ以降の条件式は評価されません)。すべての条件式の値が偽だった場合は、「式またはブロック$_4$」が評価または実行されます。

```
>>> (if (5 > 8) "namako"
... else if (8 > 5) "hitode"
... else "umiushi")
hitode
>>> (if (5 > 8) "namako"
... else if (5 > 10) "hitode"
... else "umiushi")
umiushi
```

次のプログラムの中で宣言されている sign という関数は、整数を受け取って、それがプラスならば「プラス」という文字列を返して、そうでなくてマイナスならば「マイナス」という文字列を返して、どちらでもないならば「ゼロ」という文字列を返します。

プログラムの例 sign.kt

```
fun sign(a: Int): String =
    if (a > 0) "プラス"
    else if (a < 0) "マイナス"
    else "ゼロ"
```

実行例

```
>>> sign(5)
プラス
>>> sign(-5)
マイナス
>>> sign(0)
ゼロ
```

練習問題 4.4　n を整数とするとき、six(n) という式で呼び出すと、n が 6 の倍数ならば「6 の倍数」という文字列を、そうでなくて 3 の倍数ならば「3 の倍数」という文字列を、そうでなくて 2 の倍数ならば「2 の倍数」という文字列を、そうでなければ「3 の倍数でも 2 の倍数でもない整数」という文字列を返す関数を宣言してください。

実行例

```
>>> six(30)
6 の倍数
>>> six(15)
3 の倍数
>>> six(10)
2 の倍数
>>> six(35)
3 の倍数でも 2 の倍数でもない整数
```

4.5　when 式

4.5.1　when 式の基礎

第 4.4.5 項で説明したように、選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。そして、多肢選択には、ひとつの式の値が何なのかということによって動作を選択するタイプと、いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプ、という二つのタイプがあります。

多肢選択は、「when 式」(when expression) と呼ばれる式を使うことによって記述することができます。

4.5.2　式の値によって動作を選択する when 式

ひとつの式の値が何なのかということによって動作を選択するタイプの多肢選択を記述する場合、 when 式は、

```
when (式) {
    選択肢
    ●
    ●
    ●
}
```

と書きます。この形の when 式を実行すると、 when の右側に書かれた式が評価されて、その値が何なのかということによって、選択肢の中からひとつが選択されて、それが実行されます。

4.5.3　式の値によって選択される選択肢の書き方

ひとつの式の値が何なのかということによって動作を選択するタイプの when 式の中には、式の値によって選択されるいくつかの選択肢を書く必要があります。この場合の選択肢は、基本的には、

```
式 -> 式またはブロック
```

と書きます。

ひとつの式の値が何なのかということによって動作を選択するタイプの when 式を構成している選択肢は、 when の右側に書かれた式の値と、 -> の左側に書かれた式の値とが一致した場合に選択されて、その中の式が評価されるか、またはブロックが実行されます。

```
>>> when (2) {
...     1 -> println("namako")
...     2 -> println("hitode")
...     3 -> println("umiushi")
... }
hitode
```

この when 式の場合、 when の右側に書かれた式の値は 2 ですので、 -> の左側に 2 という式が書かれている選択肢が選択されます。

when の右側の式の値と一致する値を持つ式が -> の左側に書かれた選択肢が見つからなかった場合は、どの選択肢も選択されません。

```
>>> when (8) {
...     1 -> println("namako")
...     2 -> println("hitode")
...     3 -> println("umiushi")
... }
>>>
```

4.5.4 選択肢の順番

when 式の選択肢は、上から下へという順番で、 when の右側の式の値と -> の左側の式の値が一致するかどうかが調べられていきます。そして、最初に見つかった一致する選択肢が選択されて、その中の式が評価されるか、またはブロックが実行されます。最初に見つかった選択肢よりも下に書かれている選択肢については、式の値が一致するかどうかを調べるということは実行されません。

ですから、上から下へ順番に調べて行って、最初に見つかった選択肢よりも下に、もしも調べられたならば式の値が一致する選択肢が存在していたとしても、その選択肢が選択される可能性はありません。

```
>>> when (2) {
...     1 -> println("namako")
...     2 -> println("hitode")
...     2 -> println("umiushi")
... }
hitode
```

この when 式の場合、

```
2 -> println("umiushi")
```

という 3 番目の選択肢も、もしも調べられたならば式の値が一致するわけですが、最初に見つかった 2 番目の選択肢よりも下に書かれていますので、一致するかどうかは調べられません。

4.5.5 when 式の値

when 式は、式の一種です。したがって、それを評価すると、その値が得られます。

when 式を評価することによって得られる値は、選択された選択肢を実行した結果です。選択された選択肢の -> の右側に式が書かれていた場合は、その式の値が when 式の全体の値になります。

if 式と同じように、 when 式も、それだけを REPL に入力した場合、REPL は、それを式ではなくて文だと認識しますので、その値は出力されません。ですから、REPL を使って when 式の値を確かめるためには、REPL がその中の when 式を式だと認識するようなもの（たとえば when 式を丸括弧で囲んだもの）を入力する必要があります。

```
>>> (when (2) {
...     1 -> "namako"
...     2 -> "hitode"
...     3 -> "umiushi"
... })
hitode
```

選択された選択肢の -> の右側にブロックが書かれていた場合は、そのブロックを実行したときに最後に評価された式の値が、 when 式の全体の値になります。

```
>>> (when (2) {
...     1 -> {
...         "suzume"
...         "karasu"
...     }
...     2 -> {
...         "kitsune"
...         "tanuki"
...     }
...     3 -> {
...         "medaka"
...         "namazu"
...     }
... })
tanuki
```

when の右側の式の値と一致する値を持つ式が -> の左側に書かれた選択肢が見つからなかった場合は、 kotlin.Unit が when 式の全体の値になります。

```
>>> (when (8) {
...     1 -> "namako"
...     2 -> "hitode"
...     3 -> "umiushi"
... })
kotlin.Unit
```

4.5.6　どの式の値も一致しなかった場合に選択される選択肢

when 式の選択肢としては、その最後のものとして、

```
else -> 式またはブロック
```

という形のものを書くこともできます。この形の選択肢を書いた場合、 when の右側に書かれた式の値と一致する値を持つ式が -> の左側に書かれた選択肢が存在しなかったならば、この形の選択肢が選択されます。

```
>>> (when (7) {
...     1 -> "namako"
...     2 -> "hitode"
...     3 -> "umiushi"
...     else -> "isoginchaku"
... })
isoginchaku
```

次のプログラムの中で宣言されている ntoweek という関数は、曜日の番号（1 が月曜日、2 が火曜日、……）を受け取って、その番号があらわしている曜日を返します。

プログラムの例　ntoweek.kt

```
fun ntoweek(n: Int): String =
    when (n) {
        1 -> "月曜日"
        2 -> "火曜日"
        3 -> "水曜日"
        4 -> "木曜日"
        5 -> "金曜日"
        6 -> "土曜日"
        7 -> "日曜日"
        else -> "未定義番号"
    }
```

実行例

```
>>> ntoweek(4)
木曜日
>>> ntoweek(8)
```

未定義番号

練習問題 4.5　m を月の番号（1 が 1 月、2 が 2 月、……）とするとき、tsuki(m) という式で呼び出すと、旧暦の m 月の名前を返す関数を宣言してください。m が 1 から 12 までの整数ではなかった場合は、「存在しない月」という文字列を返すようにしてください。

旧暦の月の名前については、次の表を参考にしてください。

1 月	睦月	4 月	卯月	7 月	文月	10 月	神無月
2 月	如月	5 月	皐月	8 月	葉月	11 月	霜月
3 月	弥生	6 月	水無月	9 月	長月	12 月	師走

実行例

```
>>> tsuki(8)
葉月
>>> tsuki(14)
存在しない月
```

4.5.7　->の左側に複数の式を持つ選択肢

式の値によって選択される選択肢は、基本的には、

式 -> 式またはブロック

と書くわけですが、-> の左側には、コンマ (,) で区切った 2 個以上の式を書くこともできます。つまり、

式 , 式 , ··· -> 式またはブロック

という形の選択肢を書くこともできるということです。

-> の左側に 2 個以上の式を持つ選択肢は、それらの式の値のうちのどれかひとつが、when の右側の式の値と一致した場合に選択されます。

```
>>> (when (5) {
...     1, 2, 3 -> "namako"
...     4, 5, 6 -> "hitode"
...     7, 8, 9 -> "umiushi"
... })
hitode
```

次のプログラムの中で宣言されている month という関数は、月の番号を受け取って、その番号があらわしている月が大の月ならば「大の月」、小の月ならば「小の月」、存在しない月ならば「存在しない月」という文字列を返します。

プログラムの例　month.kt

```
fun month(n: Int): String =
    when (n) {
        1, 3, 5, 7, 8, 10, 12 -> "大の月"
        2, 4, 6, 9, 11 -> "小の月"
        else -> "存在しない月"
    }
```

実行例

```
>>> month(8)
大の月
>>> month(9)
小の月
```

練習問題 4.6　n を整数とするとき、ordinal(n) という式で呼び出すと、n の序数詞を、1st、2nd、3rd、……というような文字列で返す関数を宣言してください。

実行例

```
>>> ordinal(1)
```

```
1st
>>> ordinal(2)
2nd
>>> ordinal(3)
3rd
>>> ordinal(4)
4th
>>> ordinal(11)
11th
>>> ordinal(21)
21st
```

4.5.8　範囲による選択肢

when 式の選択肢としては、式の値と式の値とが一致した場合に選択されるものだけではなく、式の値が、指定された範囲の中にある場合に選択されるものや、指定された範囲の中にない場合に選択されるものを作ることもできます。

範囲の中にある場合に選択される選択肢を作りたい場合は、 -> の左側に、

```
in 式
```

という形のものを書きます。この中の「式」のところには、評価すると値として範囲が値として得られる式を書きます。そうすると、 when の右側に書かれた式の値が、 in の右側に書かれた式を評価することによって得られた範囲の中にある場合に、その選択肢が選択されます。たとえば、

```
in 30..70 -> 式またはブロック
```

という選択肢を書くことによって、 when の右側に書かれた式の値が 30 から 70 までの範囲に入っている場合に選択される選択肢を作ることができます。

範囲の中にない場合に選択される選択肢を作りたい場合は、範囲を求める式の左側に、 in ではなくて、 !in と書きます。

次のプログラムの中で宣言されている agegroup という関数は、年齢を受け取って、その年齢の年齢層を出力します。

プログラムの例　agegroup.kt

```
fun agegroup(age: Int): String =
    when (age) {
        in 0..4 -> "幼年期"
        in 5..14 -> "少年期"
        in 15..24 -> "青年期"
        in 25..44 -> "壮年期"
        in 45..64 -> "中年期"
        else -> "高年期"
    }
```

実行例

```
>>> agegroup(12)
少年期
>>> agegroup(54)
中年期
```

練習問題 4.7　year を西暦の年とするとき、 era(year) という式で呼び出すと、その年を含んでいる日本史の時代名を返す関数を宣言してください。 year が 710 年よりも前か、または 1867 年よりもあとの場合は、「範囲外」という文字列を返すようにしてください。

日本史の時代名については、次の表を参考にしてください。

710〜793	奈良時代	1392〜1572	室町時代
794〜1184	平安時代	1573〜1599	安土桃山時代
1185〜1332	鎌倉時代	1600〜1867	江戸時代
1333〜1391	南北朝時代		

```
>>> era(939)
平安時代
>>> era(1582)
安土桃山時代
>>> era(2199)
範囲外
```

4.5.9 いくつかの条件によって動作を選択する when 式

いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプの多肢選択を記述する場合、 when 式は、

```
when {
    選択肢
    ●
    ●
    ●
}
```

と書きます。この形の when 式を実行すると、選択肢の中に書かれた条件式が順番に評価されていって、最初に見つかった、値が真になる条件式を持つ選択肢が選択されて、それが実行されます。

4.5.10 条件式を持つ選択肢の書き方

いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプの when 式の中には、条件式を持ついくつかの選択肢を書く必要があります。この場合の選択肢は、基本的には、

```
条件式 -> 式またはブロック
```

と書きます。

when 式を構成している選択肢は、上から下へ順番に、その中の条件式が評価されていって、値が真になる条件式を持つ選択肢が見つかったならば、その選択肢が選択されて、その中の式が評価されるか、またはブロックが実行されます。

```
>>> (when {
...     false -> "namako"
...     true -> "hitode"
...     false -> "umiushi"
... })
hitode
```

値が真になる条件式を持つ選択肢が見つからなかった場合は、 kotlin.Unit が when 式の全体の値になります。

```
>>> (when {
...     false -> "namako"
...     false -> "hitode"
...     false -> "umiushi"
... })
kotlin.Unit
```

いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプの when 式も、その最後の選択肢として、

```
else -> 式またはブロック
```

という形のものを書くことができます。この形の選択肢は、値が真になる条件式を持つ選択肢が見つからなかった場合に選択されます。

```
>>> (when {
...     false -> "namako"
...     false -> "hitode"
...     else -> "umiushi"
... })
umiushi
```

次のプログラムの中で宣言されている sign という関数は、整数を受け取って、それがプラスならば「プラス」という文字列を返して、そうでなくてマイナスならば「マイナス」という文字列を返して、どちらでもないならば「ゼロ」という文字列を返します。

プログラムの例 sign2.kt

```
fun sign(a: Int): String =
    when {
        a > 0 -> "プラス"
        a < 0 -> "マイナス"
        else -> "ゼロ"
    }
```

実行例

```
>>> sign(5)
プラス
>>> sign(-5)
マイナス
>>> sign(0)
ゼロ
```

練習問題 4.8 n を整数とするとき、six(n) という式で呼び出すと、n が 6 の倍数ならば「6 の倍数」という文字列を、そうでなくて 3 の倍数ならば「3 の倍数」という文字列を、そうでなくて 2 の倍数ならば「2 の倍数」という文字列を、そうでなければ「3 の倍数でも 2 の倍数でもない整数」という文字列を返す関数を、when 式を使って宣言してください。

実行例

```
>>> six(30)
6 の倍数
>>> six(15)
3 の倍数
>>> six(10)
2 の倍数
>>> six(35)
3 の倍数でも 2 の倍数でもない整数
```

4.6 論理演算子

4.6.1 論理演算子の基礎

何個かの真偽値が与えられたときに、それらの真偽値に対する処理を実行して、その結果として 1 個の真偽値を求める、という動作は、「論理演算」(logical operation) と呼ばれます。

論理演算をあらわしている演算子は、「論理演算子」(logical operator) と呼ばれます。

4.6.2 論理積演算子

二つの真偽値が与えられたときに、それらが両方とも真のときだけ真、そうでないときは偽を結果とする、という論理演算は、「論理積」(conjunction) と呼ばれます。

a と b を真偽値とするとき、それらの論理積を求めた結果を表にすると、次のようになります。

a	b	a と b の論理積
真	真	真
真	偽	偽
偽	真	偽
偽	偽	偽

論理積は、二つの条件が両方とも成り立っているかどうかを判断したいときに使われる論理演算です。つまり、この論理演算を使うことによって、A と B のそれぞれが何らかの条件だとするとき、

A かつ B

という条件が成り立っているかどうかを判断することができます。

&&は、論理積をあらわしている演算子で、「論理積演算子」(conjunction operator) と呼ばれます。

```
>>> true && true
true
>>> true && false
false
>>> false && true
false
>>> false && false
false
```

4.6.3 論理和演算子

二つの真偽値が与えられたときに、それらが両方とも偽のときだけ偽、そうでないときは真を結果とする、という論理演算は、「論理和」(disjunction) と呼ばれます。

a と b を真偽値とするとき、それらの論理和を求めた結果を表にすると、次のようになります。

a	b	a と b の論理和
真	真	真
真	偽	真
偽	真	真
偽	偽	偽

論理和は、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいときに使われる論理演算です。つまり、この論理演算を使うことによって、A と B のそれぞれが何らかの条件だとするとき、

A または B

という条件が成り立っているかどうかを判断することができます。

||は、論理和をあらわしている演算子で、「論理和演算子」(disjunction operator) と呼ばれます。

```
>>> true || true
true
>>> true || false
true
>>> false || true
true
>>> false || false
false
```

4.6.4 論理積演算子と論理和演算子の優先順位

&& と || は、比較演算子よりも低くて代入演算子よりも高い優先順位を持っています。そして、&& は、|| よりも高い優先順位を持っています。

次のプログラムの中で宣言されている leapyear という関数は、西暦であらわされた年を受け取って、その年がうるう年ならば真を返して、そうでなければ偽を返します。

プログラムの例 leapyear.kt

```
fun leapyear(y: Int): Boolean =
    y % 4 == 0 && y % 100 != 0 || y % 400 == 0
```

実行例

```
>>> leapyear(2080)
true
>>> leapyear(2100)
false
>>> leapyear(2400)
true
```

4.6.5　論理否定演算子

ひとつの真偽値が与えられたときに、それが真ならば偽を結果として、偽ならば真を結果とする、という論理演算は、「論理否定」(logical negation) と呼ばれます。

a を真偽値とするとき、それの論理否定を求めた結果を表にすると、次のようになります。

a	a の論理否定
真	偽
偽	真

論理否定は、条件が成り立っていないということを判断したいときに使われる論理演算です。つまり、この論理演算を使うことによって、A が何らかの条件だとするとき、

A ではない

という条件が成り立っているかどうかを判断することができます。

`!` は、論理否定をあらわしている演算子で、「論理否定演算子」(logical negation operator) と呼ばれます。

```
>>> ! true
false
>>> ! false
true
```

`!` は、比較演算子よりも高い優先順位を持っています。ですから、比較演算子が求めた真偽値の論理否定を求める式を書くためには、丸括弧が必要になります。

```
>>> ! (8 > 5)
false
```

第 5 章　繰り返しと再帰

5.1　繰り返しの基礎

5.1.1　繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」(iteration) と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

5.1.2　繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数の文を書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

Kotlin では、次の 3 種類の文のうちのどれかを使うことによって、繰り返しを簡潔に記述することができます。

- `for` 文 (`for` statement)
- `while` 文 (`while` statement)
- `do-while` 文 (`do-while` statement)

`for` 文については第 5.2 節で、`while` 文については第 5.3 節で、`do-while` 文については第 5.4 節で説明することにしたいと思います。

5.2 for 文

5.2.1 for 文の基礎

for 文 (for statement) は、何らかのひとつのオブジェクトから次々とオブジェクトを取り出して、取り出したオブジェクトに対して何かを実行する、という繰り返しを記述したいときに使われる文です。

ただし、どんなオブジェクトに対しても for 文を使って繰り返しを記述することができる、というわけではありません。for 文による繰り返しの対象にすることができるのは、「イテレーター」(iterator) と呼ばれる特殊なメソッドを持っているオブジェクトだけです。

このチュートリアルでは、イテレーターを持っているオブジェクトのことを、「繰り返し可能オブジェクト」(iterable object) と呼ぶことにしたいと思います。

5.2.2 for 文の書き方

for 文は、

for (識別子 in 式) ブロック

と書きます。この中の「式」のところには、値として繰り返し可能オブジェクトが得られる式を書きます。

for 文を実行すると、まず最初に、in の右側の式が 1 回だけ評価されます。そして、その式の値として得られた繰り返し可能オブジェクトからオブジェクトを取り出して、ブロックを実行する、ということが繰り返されます。ブロックが実行される直前には、毎回、変更不可能な変数が宣言されて、繰り返し可能オブジェクトから取り出されたオブジェクトで、その変数が初期化されます。in の左側の識別子は、その変数の名前になります。

5.2.3 文字列に対する繰り返し

文字列は、繰り返し可能オブジェクトです。したがって、for 文を使うことによって、文字列に対する繰り返しを記述することができます。

文字列に対する繰り返しというのは、文字列の中から文字を取り出して、その文字に対して何かを実行する、ということを繰り返す、ということです。たとえば、

```
for (c in "tako") {
    println(c)
}
```

という for 文を実行すると、まず最初に、"tako" という式が評価されて、文字列が値として得られます。そして、その文字列から取り出された文字に対して、

```
{
    println(c)
}
```

というブロックが実行されます。このブロックが実行される直前には、毎回、変更不可能な変数が宣言されて、t、a、k、o という文字のそれぞれで、その変数が初期化されます。c という識別子は、その変数の名前になります。

REPL を使って試してみましょう。

```
>>> for (c in "tako") {
...     println(c)
... }
t
a
k
o
```

次のプログラムの中で宣言されている reverse という関数は、文字列を受け取って、それを構成しているそれぞれの文字を逆の順序で並べ替えることによってできる文字列を返します。

プログラムの例 reverse.kt

```
fun reverse(s: String): String {
    var s2 = ""
    for (c in s) {
```

```
        s2 = c + s2
    }
    return s2
}
```

実行例

```
>>> reverse("isoginchaku")
ukahcnigosi
```

練習問題 5.1　s を文字列とするとき、space(s) という式で呼び出すと、s を構成しているそれぞれの文字の直後に 1 個の空白を挿入することによってできる文字列を返す関数を宣言してください。

実行例

```
>>> space("hitode")
h i t o d e
```

練習問題 5.2　s を文字列、a を文字とするとき、delete(s, a) という式で呼び出すと、s に含まれているすべての a を削除することによってできる文字列を返す関数を宣言してください。

実行例

```
>>> delete("asparagus", 'a')
sprgus
```

練習問題 5.3　s を文字列、a と b を文字とするとき、replace(s, a, b) という式で呼び出すと、s を構成しているすべての a を b に置き換えることによってできる文字列を返す関数を宣言してください。

実行例

```
>>> replace("asparagus", 'a', 'o')
osporogus
```

練習問題 5.4　s を文字列とするとき、reduce(s) という式で呼び出すと、s に含まれている 2 個以上の連続する空白を 1 個の空白に縮小させることによってできる文字列を返す関数を宣言してください。

実行例

```
>>> reduce("abc     def  ghi        jkl")
abc def ghi jkl
```

練習問題 5.5　s を文字列とするとき、maxchar(s) という式で呼び出すと、s を構成している文字のうちで文字コードが最も大きいものを返す関数を宣言してください。s が空文字列の場合は疑問符 (?) を返すようにしてください。

実行例

```
>>> maxchar("minokasago")
s
>>> maxchar("")
?
```

5.2.4　範囲に対する繰り返し

第 4.3.1 項で紹介した、「範囲」(range) と呼ばれるオブジェクトは、繰り返し可能オブジェクトです。したがって、for 文を使うことによって、範囲に対する繰り返しを記述することができます。

範囲に対する繰り返しというのは、範囲の中から整数または文字を取り出して、その整数または文字に対して何かを実行する、ということを繰り返す、ということです。

REPL を使って試してみましょう。

```
>>> for (i in 3..6) {
```

```
...     println(i)
... }
3
4
5
6
>>> for (c in 'D'..'G') {
...     println(c)
... }
D
E
F
G
```

次のプログラムの中で宣言されている divisor という関数は、プラスの整数を受け取って、そのすべての約数を出力します。

プログラムの例 divisor.kt

```
fun divisor(n: Int) {
    for (i in 1..n) {
        if (n % i == 0) {
            print("${i} ")
        }
    }
    println()
}
```

実行例

```
>>> divisor(96)
1 2 3 4 6 8 12 16 24 32 48 96
```

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

次のプログラムの中で宣言されている factorial という関数は、0 またはプラスの整数を受け取って、その整数の階乗を返します。

プログラムの例 factorial.kt

```
fun factorial(n: Int): Int {
    var f = 1
    for (i in 2..n) {
        f *= i
    }
    return f
}
```

実行例

```
>>> factorial(5)
120
```

練習問題 5.6 同じ数値を何回か乗算するという計算は、「べき乗」(power) と呼ばれます。a が数値、b が 0 またはプラスの整数だとするとき、a を b 回だけ乗算した結果、つまり、

$$\underbrace{a \times a \times \cdots \times a}_{b\text{ 個}}$$

という計算の結果は、a の b 乗 (bth power of a) と呼ばれて、a^b と書きあらわされます。ただし、

任意の数値 a について、a^0 は 1 だと定義します。

たとえば、3^4 は、

$$3 \times 3 \times 3 \times 3$$

という計算をすればいいわけですから、81 ということになります。

a を整数、b を 0 またはプラスの整数とするとき、power(a, b) という式で呼び出すと、a の b 乗を返す関数を宣言してください。

実行例

```
>>> power(3, 4)
81
```

練習問題 5.7　n をプラスの整数とするとき、n 自身を除いた n の約数の和が n と等しいならば、n を「完全数」(perfect number) と言います。たとえば、6、28、496、8128 などは完全数です。

n をプラスの整数とするとき、perfect(n) という式で呼び出すと、n が完全数ならば真、そうでなければ偽を返す関数を宣言してください。

実行例

```
>>> perfect(28)
true
>>> perfect(36)
false
```

5.2.5　プログレッション

Kotlin では、整数または文字を、一定の間隔で、小さいものから大きいものへという順序で、またはその逆の順序で並べることによってできるオブジェクトを、「プログレッション」(progression) と呼びます。

プログレッションは、繰り返し可能オブジェクトです。したがって、for 文を使うことによって、プログレッションに対する繰り返しを記述することができます。

プログレッションを使うことによって、大きいものから小さいものへという順序で整数や文字を処理したり、整数や文字を、一定の間隔で飛び飛びに処理したりすることができます。

プログレッションは、次のようなクラス名の型を持っています。

整数のプログレッション　IntProgression

文字のプログレッション　CharProgression

範囲というのは、小さいものから大きいものへという順序を持っていて、飛び飛びではない、という限定された性質を持つ、プログレッションの一種です。

次のプログラムの中で宣言されている pip という関数は、整数のプログレッションを受け取って、そのプログレッションの中にあるすべての整数を出力します。

プログラムの例　pip.kt

```
fun pip(p: IntProgression) {
    for (i in p) {
        print("${i} ")
    }
    println()
}
```

実行例

```
>>> pip(38..52)
38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
```

練習問題 5.8　r を文字のプログレッションとするとき、pcp(r) という式で呼び出すと、r の中にあるすべての文字を出力する関数を宣言してください。

実行例

```
>>> pcp('A'..'Z')
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

5.2.6 逆順のプログレッション

小さいものから大きいものへという順序ではなくて、それとは逆の順序を持つプログレッションを生成したいときは、

式$_1$ downTo 式$_2$

という形の式を書きます。この形の式を評価すると、式$_1$ の値を上限、式$_2$ の値を下限とする、逆の順序を持つプログレッションが生成されます（上限と下限もプログレッションに含まれます）。たとえば、

```
70 downTo 30
```

という式を評価すると、70 から 30 までの逆順のプログレッション（70 と 30 も含みます）が生成されて、そのプログレッションが式の値になります。同じように、

```
'M'..'D'
```

という式を評価すると、大文字の M から D までの逆順のプログレッション（M と D も含みます）が生成されて、そのプログレッションが式の値になります。

先ほどの pip と pcp を使って試してみましょう。

```
>>> pip(52 downTo 38)
52 51 50 49 48 47 46 45 44 43 42 41 40 39 38
>>> pcp('Z' downTo 'A')
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

5.2.7 飛び飛びのプログレッション

整数または文字が、小さいものから大きいものへという順序で、一定の間隔で飛び飛びに並んでいるプログレッションを生成したいときは、

式$_1$ .. 式$_2$ step 式$_3$

という形の式を書きます。この形の式を評価すると、式$_1$ の値から出発して、式$_2$ の値を超えない範囲で、式$_3$ の値ずつ大きくしながら、整数または文字を飛び飛びに並べたプログレッションが生成されます。たとえば、

```
0..20 step 3
```

という式を評価すると、

0、3、6、9、12、15、18

というプログレッションが生成されて、そのプログレッションが式の値になります。同じように、

```
'A'..'Z' step 4
```

という式を評価すると、

A、E、I、M、Q、U、Y

というプログレッションが生成されて、そのプログレッションが式の値になります。

pip と pcp を使って試してみましょう。

```
>>> pip(0..100 step 7)
0 7 14 21 28 35 42 49 56 63 70 77 84 91 98
>>> pcp('A'..'Z' step 3)
A D G J M P S V Y
```

整数または文字が、大きいものから小さいものへという逆の順序で、一定の間隔で飛び飛びに並んでいるプログレッションを生成したいときは、

式$_1$ downTo 式$_2$ step 式$_3$

という形の式を書きます。この形の式を評価すると、式$_1$ の値から出発して、式$_2$ の値を下回らない範囲で、式$_3$ の値ずつ小さくしながら、整数または文字を飛び飛びに並べたプログレッションが生成されます。たとえば、

```
20 downTo 0 step 3
```

という式を評価すると、

20、17、14、11、8、5、2

というプログレッションが生成されて、そのプログレッションが式の値になります。同じように、

```
'Z' downTo 'A' step 4
```

という式を評価すると、

Z、V、R、N、J、F、B

というプログレッションが生成されて、そのプログレッションが式の値になります。

pip と pcp を使って試してみましょう。

```
>>> pip(100 downTo 0 step 7)
100 93 86 79 72 65 58 51 44 37 30 23 16 9 2
>>> pcp('Z' downTo 'A' step 3)
Z W T Q N K H E B
```

5.3 while 文

5.3.1 while 文の基礎

繰り返しを記述したいときは、基本的には for 文を使えばいいわけですが、for 文では記述することが困難な繰り返しも存在します。たとえば、条件による繰り返しは、for 文では記述することが困難です。

条件による繰り返しというのは、繰り返しの対象となる動作を実行するたびに、繰り返しを続行するか終了するかということを、何らかの条件が成り立っているかどうかを判断することによって決定する、というタイプの繰り返しのことです。このようなタイプの繰り返しは、for 文では記述することが困難です。

そこで登場するのが、条件による繰り返しを表現するために存在する、「while 文」(while statement) と呼ばれる文と、「do-while 文」(do-while statement) と呼ばれる文です。

ただし、この節で説明するのは while 文だけです。do-while 文については、次の節で説明することにしたいと思います。

5.3.2 while 文の書き方

while 文は、

while (条件式) ブロック

と書きます。この中の「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

while 文は、次のような動作をあらわしています。

(1) 条件式を評価する。その値が偽だった場合、while 文の動作は終了する。
(2) 条件式の値が真だった場合は、ブロックを実行する。
(3) (1) に戻って、ふたたび同じ動作を実行する。

5.3.3 無限ループ

while 文を使うと、永遠に終わらない繰り返しというものを記述することも可能になります。永遠に終わらない繰り返しは、「無限ループ」(infinite loop) と呼ばれます。

true という真偽値リテラルを評価すると、真を意味するデータが値として得られますので、while 文の条件式としてそれを書くと、その while 文は無限ループになります。

たとえば、次の while 文は、kurage という文字列の出力を永遠に繰り返します。

```
while (true) {
    println("kurage")
}
```

この while 文の実行を終了させたいときは、Ctrl-C、つまりコントロールキーを押しながら C のキーを押す、という操作をします。

5.3.4 条件による繰り返しの例

条件による繰り返しの例として、二つの整数の最大公約数を求めるという処理について考えてみることにしましょう。

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使えば、きわめて簡単に求めることができます。ユークリッドの互除法というのは、

ステップ 1　与えられた二つの整数のそれぞれを、n と m という変数に代入する。
ステップ 2　m が 0 ならば計算を終了する。
ステップ 3　n を m で除算して、そのあまりを r という変数に代入する。
ステップ 4　m を n に代入する。
ステップ 5　r を m に代入する。
ステップ 6　ステップ 2 に戻る。

という計算を実行していけば、計算が終了したときの n が、最初に与えられた二つの整数の最大公約数になっている、というものです。ステップ 2 からステップ 6 までは、

　m が 0 ではないあいだ、ステップ 3 からステップ 5 までを繰り返す。

ということだと考えることができますので、その部分は、`while` 文を書くことによって記述することができます。

次のプログラムの中で宣言されている `gcm` という関数は、二つの整数を受け取って、それらの最大公約数を返します。

プログラムの例　`gcm.kt`

```
fun gcm(a: Int, b: Int): Int {
    var n = a
    var m = b
    var r = 0
    while (m != 0) {
        r = n % m
        n = m
        m = r
    }
    return n
}
```

実行例

```
>>> gcm(54, 36)
18
```

練習問題 5.9　2 以上の整数のうちで、1 と自分自身以外に約数を持たないものを、「素数」(prime number) と呼びます。たとえば、17 は、1 と 17 以外に約数を持っていませんので、素数だということになります。

2 以上の任意の整数は、素数の積によって求めることができます。たとえば、882 という整数は、

　2、3、3、7、7

という素数の積です。

n を 2 以上の整数とするとき、n の約数のうちで素数であるものを、n の「素因数」(prime factor) と呼びます。

n を 2 以上の整数とするとき、それらの積が n となる素因数を求めることを、n の「素因数分解」(prime factor decomposition) と呼びます。

`n` を 2 以上の整数とするとき、`decomp(n)` という式で呼び出すと、`n` を素因数分解した結果を出力する関数を宣言してください。

実行例

```
>>> decomp(882)
2 3 3 7 7
```

5.4　do-while 文

5.4.1　do-while 文の基礎

do-while 文は、

do ブロック while (条件式)

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

do-while 文は、次のような動作をあらわしています。

(1) ブロックを実行する。
(2) 条件式を評価する。その値が偽だった場合、do-while 文の動作は終了する。
(3) 条件式の値が真だった場合は、(1) に戻って、ふたたび同じ動作を実行する。

5.4.2　while 文と do-while 文の相違点

while 文と do-while 文は、どちらも、条件式の値が真であるあいだだけ文の列の実行を繰り返す、という動作をあらわしています。では、この 2 種類の文は、どこに相違点があるのでしょうか。

while 文と do-while 文の相違点は、「最初に何をするか」というところにあります。while 文の場合、最初の動作は「条件式の評価」です。それに対して、do-while 文の場合、最初の動作は「ブロックの実行」です。

その結果として、while 文と do-while 文とでは、条件式の値が最初から偽だった場合に異なる動作をすることになります。

REPL を使って、動作を調べてみましょう。まず、次のプログラムを REPL に入力してみてください。

```
var i = 10
while (i <= 5) {
    print(i)
    i++
}
```

このプログラムは、何も出力しないで終了します。その理由は、

```
i <= 5
```

という条件式の値が最初から偽だからです。

それでは、次のプログラムを REPL に入力すると、どうなるでしょうか。

```
var i = 10
do {
    print(i)
    i++
} while (i <= 5)
```

このプログラムは、条件式の値が最初から偽であるにもかかわらず、10 を出力してから終了します。その理由は、条件式の評価よりも先に文の列が実行されるからです。

5.5　break 式と continue 式

5.5.1　break 式

for 文や while 文や do-while 文を使って繰り返しを記述するとき、場合によっては、何らかの条件が成り立ったときに繰り返しを途中で終了するようにしたい、ということがあります。そのようなときに使われるのが、「break 式」(break expression) と呼ばれる式です。

break 式は、

```
break
```

と書きます。

for 文、while 文、do-while 文の中に break 式を書いておくと、それが評価されたとき、for 文、while 文、do-while 文の実行は終了します。

REPL を使って試してみましょう。

```
>>> for (i in 1..5) {
...     if (i == 4) break
...     println(i)
... }
1
2
3
>>>
```

5.5.2 continue 式

break 式は、for 文や while 文や do-while 文による繰り返しを途中で終了させたいときに使われるわけですが、繰り返しを終了させるのではなくて、繰り返しの対象となっている動作の実行をスキップしたい、ということもあります。そのようなときに使われるのが、「continue 式」(continue statement) と呼ばれる式です。

continue 式は、

```
continue
```

と書きます。

for 文、while 文、do-while 文の中に continue 式を書いておくと、それが評価された場合だけ、繰り返しの対象となっている動作のうちで、それ以降の部分の実行がスキップされます。

REPL を使って試してみましょう。

```
>>> for (i in 1..5) {
...     if (i == 3) continue
...     println(i)
... }
1
2
4
5
```

5.6 再帰

5.6.1 再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

5.6.2 基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

5.6.3　関数の再帰的な宣言

関数は、再帰的に宣言することが可能です。関数を再帰的に宣言するというのは、宣言される当の関数を使って関数を宣言するということです。再帰的な構造を持っている概念を取り扱う関数は、再帰的に宣言するほうが、再帰的ではない方法で宣言するよりもすっきりした記述になります。

関数を再帰的に宣言する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

5.6.4　階乗の再帰的な構造

第 5.2.4 項で説明したように、n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求める関数も、再帰的に宣言することができます。次のプログラムは、階乗を求める `factorial` という関数を再帰的に宣言しています。

プログラムの例　`factorial2.kt`

```
fun factorial(n: Int): Int =
    when {
        n == 0 -> 1
        n >= 1 -> n * factorial(n - 1)
        else -> 0
    }
```

実行例

```
>>> factorial(5)
120
```

5.6.5　フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

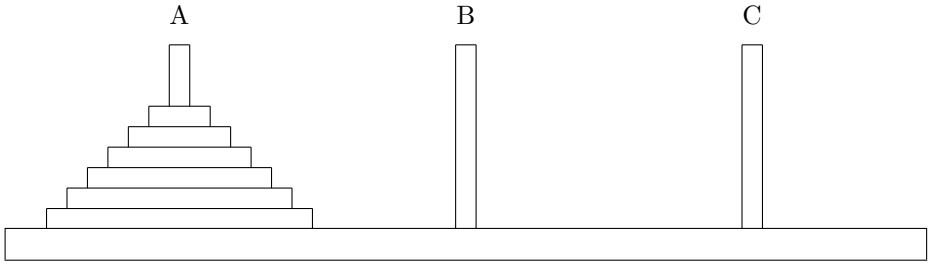
$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

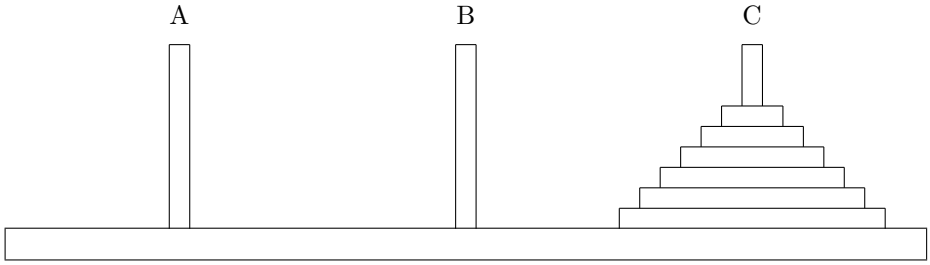
フィボナッチ数列の第 n 項を求める関数も、再帰的に宣言することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める `fibonacci` という関数を再帰的に宣言しています。

プログラムの例　`fibonacci.kt`

```
fun fibonacci(n: Int): Int =
    when {
        n == 0 -> 1
```



初期状態



最終状態

図 5.1: ハノイの塔

```
        n == 1 -> 1
        n >= 2 -> fibonacci(n - 2) + fibonacci(n - 1)
        else -> 0
    }
```

実行例

```
>>> fibonacci(7)
21
```

5.6.6 最大公約数

第 5.3.4 項で、条件による繰り返しの例として、ユークリッドの互除法を使って二つの整数の最大公約数を求めるという処理について説明しましたが、ユークリッドの互除法は、二つの整数を n と m とするとき、次のように再帰的に記述することも可能です。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、2 個のプラスの整数の最大公約数を求める gcm という関数を、ユークリッドの互除法を使って再帰的に宣言しています。

プログラムの例 gcm2.kt

```
fun gcm(n: Int, m: Int): Int =
    when {
        m == 0 -> n
        m >= 1 -> gcm(m, n % m)
        else -> 0
    }
```

実行例

```
>>> gcm(54, 36)
18
```

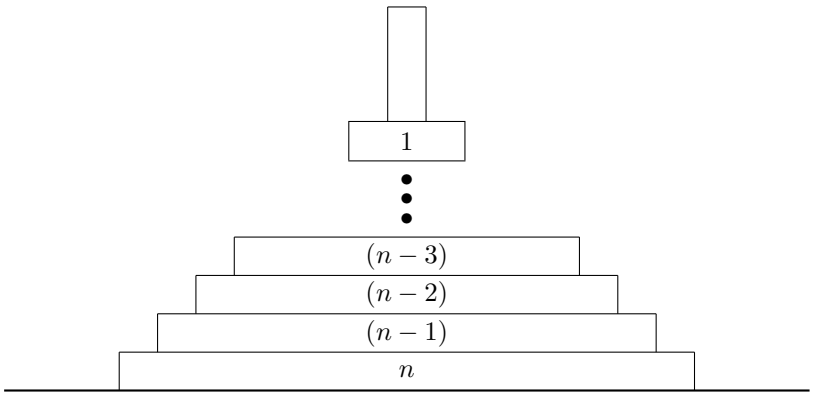


図 5.2: 円盤の番号

5.6.7　ハノイの塔

これまでこの章で紹介してきた問題は、再帰を使うことによってそのプログラムを書くことができるわけですが、再帰を使わないでプログラムを書くことも、それほど難しいことではありません。しかし、問題の中には、再帰を使えばプログラムを簡単に書くことができるけれども、再帰を使わないでプログラムを書くことはきわめて難しい、というものもあります。たとえば、「ハノイの塔」(Tower of Hanoi) と呼ばれるパズルを解くという問題は、そのような問題の一例です。

ハノイの塔では、3 本の棒が垂直に立っている台と、直径が少しずつ違う何枚かの円盤が道具として使われます。それらの円盤は、中央に穴が開いていて、台の上の棒にはめ込むことができるようになっています。

ハノイの塔は、円盤を動かしていくことによって初期状態から最終状態へ移行させるための手順を求めてください、というパズルです。初期状態と最終状態というのは、図 5.1 のような状態のことです。つまり、初期状態では、すべての円盤が棒 A にはまっていて、しかも下にある円盤ほど直径が大きいという順番になっています。そして最終状態では、すべての円盤が棒 C にはまっていて、そして初期状態と同じように、下にある円盤ほど直径が大きいという順番になっていないといけません。

円盤を動かすときには、次の二つの規則にしたがう必要があります。

- 1 回の操作で実行できるのは、どれかの棒にはめ込まれている円盤のうちで、もっとも上にある 1 枚を棒から抜き取って、それを別の棒にはめ込む、ということだけである。
- すでに棒にはめ込まれている円盤よりも直径の大きな円盤をその棒にはめ込むことはできない。

円盤の枚数が n 枚だとするとき、図 5.2 のように、直径の大きな円盤から順番に、n、$(n-1)$、$(n-2)$、$(n-3)$、……、1、という番号がそれぞれの円盤に与えられているとします。そして、n 番目から 1 番目までの円盤を、上へ行くほど小さくなるように重ねたもののことを、「n 円錐」と呼ぶことにします。そうすると、n 円錐から n 番目の円盤を取り除いた部分は、「$(n-1)$ 円錐」と呼ばれることになります。

ハノイの塔の 3 本の棒のそれぞれは、「出発点」、「待避所」、「目的地」という 3 種類の役割を持っていると考えることができます。出発点というのは、n 円錐がそこから出発していく棒のことで、目的地というのは、移動が終わったときに n 円錐がはめ込まれている棒のことです。そして待避所というのは、n 番目の円盤を移動させるために $(n-1)$ 円錐を待避させておくための棒のことです。

ただし、どの棒がどの役割なのかという関係は、固定されていません。パズルの全体としては、棒 A が出発点で、棒 B が待避所で、棒 C が目的地ですが、パズルを解いていく過程で、棒と役割の関係は変化します。

ハノイの塔は、次のような再帰的な手順を実行することによって解くことができます。

ステップ 1　$(n-1)$ 円錐を出発点から目的地経由で待避所へ移動させる。

ステップ 2　n 番目の円盤を出発点から目的地へ移動させる。

ステップ 3　$(n-1)$ 円錐を待避所から出発点経由で目的地へ移動させる。

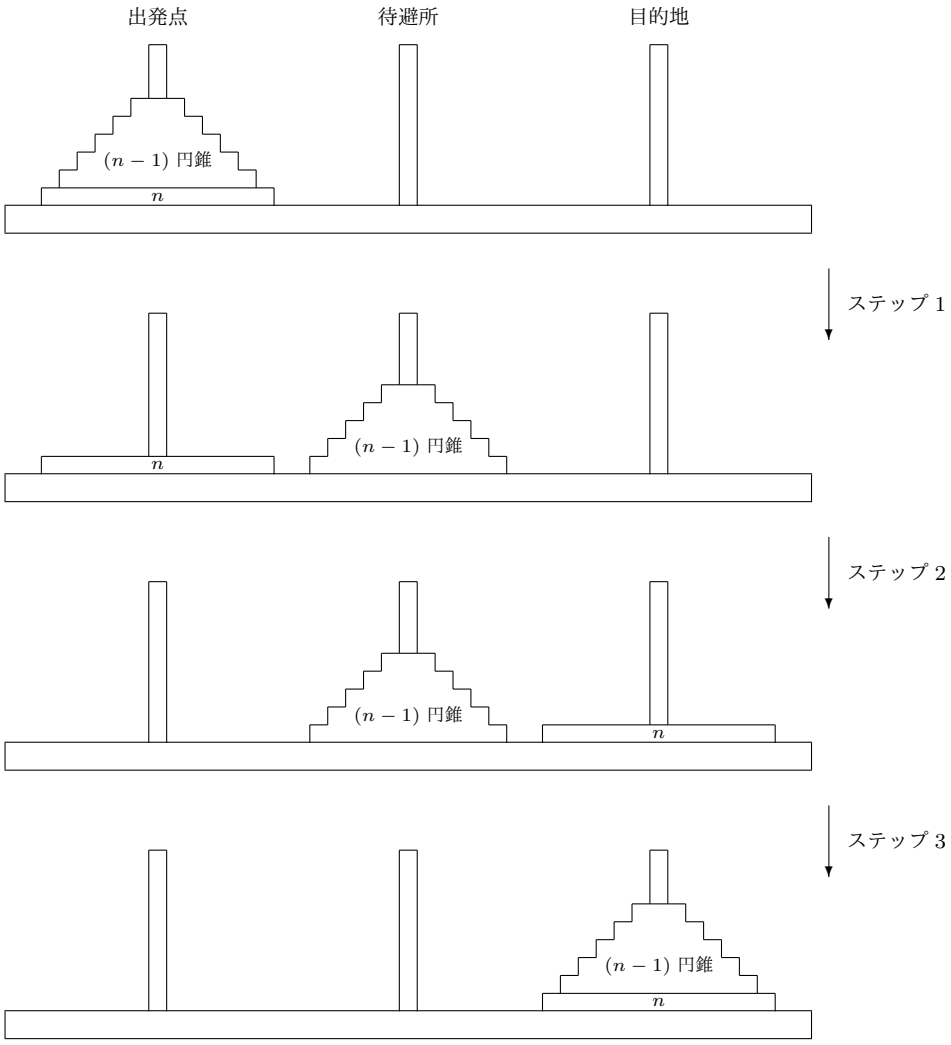


図 5.3: ハノイの塔の解法

図 5.3 は、この手順を図で示したものです。

ハノイの塔を解く再帰的な手順の基底は、n が 0 の場合です。円盤の枚数が 0 のハノイの塔は、初期状態と終了状態がまったく同じです。したがって、n が 0 の場合は、何もしないというのが、それを解く手順になります。

次のプログラムは、ハノイの塔を解く hanoi という関数を宣言しています。

プログラムの例 hanoi.kt

```
fun hanoi(n: Int) {
    hanoi2(n, 'A', 'B', 'C')
}

fun hanoi2(n: Int, start: Char, shelter: Char, goal: Char) {
    if (n > 0) {
        hanoi2(n - 1, start, goal, shelter)
        println("${start} ---> ${goal}")
        hanoi2(n - 1, shelter, start, goal)
    }
}
```

n を 0 またはプラスの整数とするとき、hanoi(n) という式で hanoi を呼び出すと、hanoi は、円盤をどの棒からどの棒へ移動させればよいかということを、

どの棒から ---> どの棒へ

という形式で出力します。

実行例

```
>>> hanoi(3)
A ---> C
A ---> B
C ---> B
A ---> C
B ---> A
B ---> C
A ---> C
```

hanoi は、再帰的に宣言された hanoi2 という関数を呼び出すだけの関数です。hanoi2 は、円盤の枚数、出発点の名前、退避所の名前、目的地の名前を受け取って、ハノイの塔を解く手順を出力します。

練習問題 5.10　第 5.2.4 項の練習問題で出題した、a を整数、b を 0 またはプラスの整数とするとき、power(a, b) という式で呼び出すと、a の b 乗を返す関数を、再帰を使って宣言してください。

実行例

```
>>> power(3, 4)
81
```

ヒント　a の b 乗は、

$$\begin{cases} & a^0 = 1 \\ b \geq 1 \text{ ならば} & a^b = a \times a^{(b-1)} \end{cases}$$

というように再帰的に定義することができます。

練習問題 5.11　「n 重の括弧列」(n-fold parenthesis sequence) という文字列を次のように定義します。

- 空文字列は 0 重の括弧列である。
- n を 1 以上の整数とするとき、$(n-1)$ 重の括弧列を丸括弧で囲んだものは n 重の括弧列である。

たとえば、() は 1 重の括弧列で、((((())))) は 4 重の括弧列です。

n を 0 またはプラスの整数とするとき、paren(n) という式で呼び出すと、n 重の括弧列を返す関数を、再帰を使って宣言してください。

実行例

```
>>> paren(4)
((((()))))
```

練習問題 5.12　「n 重の二分括弧列」(n-fold binary parenthesis sequence) という文字列を次のように定義します。

- 空文字列は 0 重の二分括弧列である。
- n を 1 以上の整数とするとき、

  ($\boxed{(n-1) \text{ 重の二分括弧列}}$ $\boxed{(n-1) \text{ 重の二分括弧列}}$)

  という文字列は n 重の二分括弧列である。

たとえば、(()()) は 2 重の二分括弧列で、(((()())(()()))) は 3 重の二分括弧列です。

n を 0 またはプラスの整数とするとき、binparen(n) という式で呼び出すと、n 重の二分括弧列を返す関数を、再帰を使って宣言してください。

実行例

```
>>> binparen(4)
((((()())(()()))(((()())(()()))))
```

練習問題 5.13　第 5.3.4 項の練習問題で出題した、n を 2 以上の整数とするとき、decomp(n) という式で呼び出すと、n を素因数分解した結果を出力する関数を、再帰を使って宣言してください。

実行例

```
>>> decomp(882)
2 3 3 7 7
```

ヒント　ハノイの塔を解く関数を宣言する場合と同じように、decomp2 という補助的な関数を宣言するとよいでしょう。

decomp2 は、n と p を 2 以上の整数とするとき、decomp2(n, p) という式で呼び出すと、n は p よりも小さい素因数を持たないと仮定して、n を素因数分解した結果を出力する関数です。

2 よりも小さい素因数は存在しませんので、decomp2(n, 2) という式で decomp2 を呼び出せば、n を素因数分解した結果が出力されることになります。

第 6 章　コレクション

6.1　コレクションの基礎

6.1.1　コレクションとは何か

Kotlin では、「コレクション」(collection) と呼ばれるオブジェクトを扱うことができます。

コレクションというのは、複数のオブジェクトから構成されるオブジェクトのことです。

コレクションを構成している個々のオブジェクトは、そのコレクションの「要素」(element) と呼ばれます。

Kotlin で扱うことができるコレクションとしては、次のようなものがあります。

- リスト (list)
- セット (set)
- マップ (map)

リストについては第 6.2 節で、セットについては第 6.3 節で、マップについては第 6.4 節で説明することにしたいと思います。

6.1.2　変更不可能なコレクションと変更可能なコレクション

コレクションには、「変更不可能な」(immutable) ものと、「変更可能な」(mutable) ものがあります。

変更不可能なコレクションというのは、生成されたのちは、それに対していかなる変更もできないコレクションのことで、変更可能なコレクションというのは、生成されたのちも、それに対して変更ができるコレクションのことです。

リスト、セット、マップのそれぞれには、変更不可能なものと変更可能なものという二つの種類のものがあります。

6.1.3　コレクションの型名

コレクションというのもオブジェクトの一種ですから、ほかのオブジェクトと同じように、型というものを持っています。

これまでの章で登場したオブジェクトの型名は、Int、Double、Char、String、Boolean、IntRange というように、クラス名という 1 個の識別子でした。しかし、コレクションの型名は、1 個の識別子ではありません。

コレクションの型名は、

識別子 < 型名 , ･･･>

という、複数の識別子を組み合わせた形の記述です。小なり (<) と大なり (>) とのあいだに書かれる型名は、「型引数」(type argument) と呼ばれます。

コレクションの型ように、小なりと大なりを使った型名によって識別される型は、「ジェネリクス」(generics) と呼ばれる仕組みによって生み出されます（日本語では、ジェネリクスは、「総

称型」と呼ばれることもあります)。ジェネリクスについては、第 8 章で詳しく説明することにしたいと思います。

6.2　リスト

6.2.1　リストの基礎

同じ型を持つ 0 個以上のオブジェクトを並べることによってできる列であるようなコレクションは、「リスト」(list) と呼ばれます。

リストの型は、次のような型名によって識別されます。

変更不可能なリスト　`List<`要素の型`>`

変更可能なリスト　`MutableList<`要素の型`>`

たとえば、整数を要素とする変更不可能なリストの型は、

```
List<Int>
```

という型名によって識別されます。同じように、文字列を要素とする変更可能なリストの型は、

```
MutableList<String>
```

という型名によって識別されます。

6.2.2　リストを生成する関数

`listOf` という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数を並べることによってできる変更不可能なリストを生成して、その変更不可能なリストを戻り値として返します。

```
>>> listOf(38, 27, 64, 52, 41, 70)
[38, 27, 64, 52, 41, 70]
```

このように、REPL は、入力された式の値がリストだった場合、

`[`要素`, `要素`, `要素`, ···]`

という形でそれを出力します。

`println` と `print` も、引数としてリストを受け取った場合、それを同じ形で出力します。

```
>>> println(listOf("namako", "umiushi", "hitode"))
[namako, umiushi, hitode]
```

`mutableListOf` という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数を並べることによってできる変更可能なリストを生成して、その変更可能なリストを戻り値として返します。

```
>>> mutableListOf(82, 36, 54, 19, 66, 43)
[82, 36, 54, 19, 66, 43]
```

6.2.3　空リスト

0 個の要素から構成されるリストは、「空リスト」(empty list) と呼ばれます。

空リストを生成するためには、型名を記述することによって、その空リストの型を明示する必要があります。

```
>>> val a: List<Int> = listOf()
>>> a
[]
>>> val b: MutableList<String> = mutableListOf()
>>> b
[]
```

6.2.4　コレクションの要素の個数

コレクションは、`size` という名前のプロパティーを持っています。このプロパティーは、コレクションの要素の個数です。

```
>>> listOf('a', 'b', 'c', 'd', 'e').size
5
```

コレクションは、isEmpty という、引数を受け取らないメソッドを持っています。このメソッドは、コレクションの要素の個数が 0 個ならば真、そうでなければ偽を返します。

```
>>> val a: List<Int> = listOf()
>>> a.isEmpty()
true
>>> listOf('a', 'b', 'c', 'd', 'e').isEmpty()
false
```

6.2.5 コレクションの要素かどうかを判定する演算子

第 4.3 節で紹介した、in と !in という二項演算子は、整数または文字が範囲の中にあるかどうかということを判定したいときに使うことができるだけではなくて、オブジェクトがコレクションの要素かどうかということを判定したいときにも使うことができます。

```
>>> val a = listOf(84, 23, 51, 72, 62)
>>> 72 in a
true
>>> 49 in a
false
>>> 72 !in a
false
>>> 49 !in a
true
```

6.2.6 リストに対する繰り返し

リストは、繰り返し可能オブジェクトです。したがって、for 文を使うことによって、リストを構成している要素をひとつずつ順番に処理する繰り返しを記述することができます。

```
>>> val a = listOf("namako", "umiushi", "hitode", "kamenote")
>>> for (s in a) {
...     println(s)
... }
namako
umiushi
hitode
kamenote
```

練習問題 6.1 a を整数の変更不可能なリストとするとき、sum(a) という式で呼び出すと、a のすべての要素の合計を返す関数を宣言してください。

実行例

```
>>> sum(listOf(50000, 7000, 300, 20, 4))
57324
```

リストに対する繰り返しでは、しばしば、処理の対象になっている要素の番号が必要になることがあります。そのような場合に便利なのが、リストが持っている withIndex というメソッドです。

withIndex を使ってリストに対する繰り返しを実行したいときは、

for ((識別子$_1$, 識別子$_2$) in 式.withIndex()) ブロック

という形の for 文を書きます。この形の for 文では、識別子$_1$ は、要素の番号で初期化された変更不可能な変数の名前になって、識別子$_2$ は、要素で初期化された変更不可能な変数の名前になります。

```
>>> val a = listOf(73, 24, 68, 52)
>>> for ((i, n) in a.withIndex()) {
...     println("${i}: ${n}")
... }
0: 73
1: 24
2: 68
```

```
3: 52
```

withIndex は、リストだけではなくて、文字列でも使うことができます。

```
>>> for ((i, c) in "kani".withIndex()) {
...     println("${i}: ${c}")
... }
0: k
1: a
2: n
3: i
```

6.2.7　添字式

リストを構成しているそれぞれの要素には、それが並んでいる順番のとおりに、0 番から始まる番号が与えられています。リストを構成しているそれぞれの要素は、その番号によって指定することができます。

番号によって要素を指定するための式は、「添字式」(subscript expression) と呼ばれます。添字式は、

式$_1$ [式$_2$]

と書きます。この中の 式$_1$ のところにはリストを求める式を書いて、式$_2$ のところには要素の番号を求める式を書きます。

添字式を評価すると、その値として、番号によって指定された要素が得られます。

```
>>> listOf("namako", "umiushi", "hitode", "isoginchaku")[2]
hitode
```

添字式の後部にある、式を角括弧で囲んだ部分は、「添字」(subscript) と呼ばれます。

次のプログラムの中で宣言されている lastInt という関数は、空リストではない整数の変更不可能なリストを受け取って、そのリストの末尾にある整数を返します。

プログラムの例　lastint.kt

```
fun lastInt(a: List<Int>): Int = a[a.size - 1]
```

実行例

```
>>> lastInt(listOf(38, 24, 67, 31, 58))
58
```

リストと同じように、文字列も、それを構成しているそれぞれの文字には、それが並んでいる順番のとおりに、0 番から始まる番号が与えられています。添字式は、文字列を構成しているそれぞれの文字を指定したいときにも使うことができます。

```
>>> "abcdefg"[3]
d
```

練習問題 6.2　s を空文字列ではない文字列とするとき、lastChar(s) という式で呼び出すと、s の末尾の文字を返す関数を宣言してください。

実行例

```
>>> lastChar("hamaguri")
i
```

6.2.8　変更可能なリストの要素の変更

変更不可能なリストを変数に代入した場合、その変数に入るものは、その変更不可能なリストを指し示す参照です。

それに対して、変更可能なリストを変数に代入した場合には、その変数に入るものは、箱を指し示す参照です。その箱は、その変更可能なリストの要素と同じ個数の箱が並んでできています。そして、それぞれの箱には、そのリストのそれぞれの要素を指し示す参照が入っています。

変数に代入された変更可能なリストの要素を指定する添字式を評価すると、指定された要素が右辺値として得られるだけではなくて、指定された要素を参照している箱が、その左辺値として

得られます。ですから、代入演算子の左側に添字式を書くことによって、変更可能なリストの要素を変更することができます。

```
>>> val a = mutableListOf('a', 'b', 'c', 'd', 'e')
>>> a[3] = 'x'
>>> a
[a, b, c, x, e]
>>> val b = mutableListOf(0, 1, 2, 3, 4, 5)
>>> b[4] += 100
>>> b
[0, 1, 2, 3, 104, 5]
```

6.2.9 変更可能なリストの末尾への要素の追加

変更可能なリストは、add というメソッドを持っています。このメソッドを呼び出すことによって、変更可能なリストの末尾に要素を追加することができます。add には、引数として、追加したい要素を渡します。

```
>>> val a = mutableListOf('a', 'b', 'c', 'd', 'e')
>>> a.add('x')
true
>>> a
[a, b, c, d, e, x]
```

6.2.10 変更可能なリストへの要素の挿入

変更可能なリストが持っている add というメソッドは、変更可能なリストの末尾への要素の追加だけではなくて、変更可能なリストの任意の位置に要素を挿入するということもできます。

変更可能なリストの任意の位置に要素を挿入したいときは、二つの引数を add に渡します。1 個目は、挿入する位置を指定する番号で、2 個目は、挿入したい要素です。2 個目の引数は、1 個目の引数を番号とする要素の直前に挿入されます。

```
>>> val a = mutableListOf('a', 'b', 'c', 'd', 'e')
>>> a.add(3, 'x')
>>> a
[a, b, c, x, d, e]
>>> a.add(0, 'y')
>>> a
[y, a, b, c, x, d, e]
```

6.2.11 変更可能なリストの要素の削除

変更可能なリストは、remove というメソッドを持っています。このメソッドは、1 個のオブジェクトを受け取って、変更可能なリストの先頭から末尾に向かってそのオブジェクトと等しい要素を探索して、最初に発見された要素を削除します。このメソッドの戻り値は、要素を発見できた場合は真で、発見できなかった場合は偽です。

```
>>> val a = mutableListOf(3, 2, 3, 1, 2, 3, 1, 3, 2)
>>> a.remove(1)
true
>>> a
[3, 2, 3, 2, 3, 1, 3, 2]
>>> a.remove(4)
false
```

変更可能なリストから要素を削除するメソッドとしては、remove のほかに、removeAt というものもあります。このメソッドは、1 個の整数を受け取って、その整数を番号とする要素を削除して、その要素を戻り値として返します。

```
>>> val a = mutableListOf('a', 'b', 'c', 'd', 'e')
>>> a.removeAt(3)
d
>>> a
[a, b, c, e]
```

6.2.12 リストの連結

+という二項演算子は、左右の式の値が同じ型のリストの場合は、左のリストの右側に右のリストを連結することによってできるリストを求める、という動作をあらわします。

```
>>> listOf(35, 83, 27, 64) + listOf(51, 72, 33, 49)
[35, 83, 27, 64, 51, 72, 33, 49]
```

+は、左側の式の値がリストで、右側の式の値が、左のリストの要素と同じ型のオブジェクトの場合は、左のリストの末尾に右のオブジェクトを追加することによってできるリストを求める、という動作をあらわします。

```
>>> listOf(35, 83, 27, 64) + 51
[35, 83, 27, 64, 51]
```

6.2.13 プログレッションからリストへの変換

プログレッションは、toListという名前のメソッドを持っています。このメソッドは、プログレッションを変更不可能なリストに変換して、そのリストを戻り値として返します。

```
>>> (7..20).toList()
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> (20 downTo 7).toList()
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7]
>>> (100 downTo 0 step 10).toList()
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

同じように、プログレッションが持っているtoMutableListというメソッドを呼び出すことによって、プログレッションを変更可能なリストに変換することもできます。

6.2.14 文字列からリストへの変換

プログレッションと同じように、文字列も、toListという名前のメソッドを持っています。このメソッドは、文字列を文字の変更不可能なリストに変換して、そのリストを戻り値として返します。

```
>>> "namako".toList()
[n, a, m, a, k, o]
```

同じように、文字列が持っているtoMutableListというメソッドを呼び出すことによって、文字列を文字の変更可能なリストに変換することもできます。

6.2.15 変更不可能なリストから変更可能なリストへの変換

変更不可能なリストは、toMutableListという名前のメソッドを持っています。このメソッドは、変更不可能なリストを変更可能なリストに変換して、その変更可能なリストを戻り値として返します。

```
>>> val a = listOf(3, 5, 2, 7, 4)
>>> val b = a.toMutableList()
>>> b.add(8)
true
>>> b
[3, 5, 2, 7, 4, 8]
```

6.2.16 変更可能なリストから変更不可能なリストへの変換

変更可能なリストは、toListという名前のメソッドを持っています。このメソッドは、変更可能なリストを変更不可能なリストに変換して、その変更不可能なリストを戻り値として返します。

```
>>> val a = mutableListOf(3, 5, 2, 7, 4)
>>> val b = a.toList()
>>> b.add(8)
error: unresolved reference: add
b.add(8)
  ^
```

6.2.17 配列

Kotlinでは、「配列」(array)と呼ばれるオブジェクトを扱うことができます。

配列は、変更可能なリストによく似たオブジェクトです。どちらも、同じ型の 0 個以上のオブジェクトを並べることによってできる列ですし、添字式を使うことによって、要素を求めたり要素を変更したりすることができます。

しかし、配列と変更可能なリストとのあいだには、違っているところもあります。最大の相違点は、配列に対しては、要素を追加したり挿入したり削除したりすることができない、ということです。

arrayOf という標準ライブラリー関数は、0 個以上の任意の個数の引数を受け取って、それらの引数を並べることによってできる配列を生成して、その配列を戻り値として返します。

```
>>> val a = arrayOf('a', 'b', 'c', 'd', 'e')
>>> a[3] = 'x'
>>> a[3]
x
```

配列の型は、

Array<要素の型>

という型名によって識別されます。

6.2.18 コマンドライン引数

プログラムを起動するコマンドは、そのプログラムの名前と、それに続く、空白で区切られた 0 個以上の引数から構成されています。それらの引数は、「コマンドライン引数」(command line argument) と呼ばれます。

第 1.2.7 項で説明したように、Kotlin で書かれたプログラムをコンパイルすることによってできた JVM のバイナリーコードは、

kotlin クラス名

というコマンドによって実行することができます。このコマンドには、

kotlin クラス名 引数 引数 ···

というように、コマンドライン引数を何個でも書くことができます。

JVM のバイナリーコードにコンパイルすることのできる Kotlin のプログラムを書く場合には、main という名前の関数を宣言する必要があります。第 1.2.5 項で紹介したプログラムも、main という名前の関数を宣言しています。

Kotlin で書かれたプログラムをコンパイルすることによってできた JVM のバイナリーコードを実行すると、main が呼び出されて、その実行が終了すると、プログラムの実行も終了します。

main という関数は、Array<String> という型の配列を引数として受け取るように宣言する必要があります。main が引数として受け取るのは、コマンドライン引数のそれぞれを要素とする配列です。

次のプログラムは、受け取ったコマンドライン引数をすべて出力します。

プログラムの例 echo.kt

```
fun main(args: Array<String>) {
    for (s in args) {
        print("${s} ")
    }
    println()
}
```

実行例

```
$ kotlin EchoKt namako umiushi kurage hitode isoginchaku
namako umiushi kurage hitode isoginchaku
```

6.3　セット

6.3.1　セットの基礎

同じ型を持つ 0 個以上のオブジェクトをひとつにまとめることによってできるコレクションは、「セット」(set) と呼ばれます。

リストを構成しているそれぞれの要素は、一列に並んでいます。それに対して、セットを構成しているそれぞれの要素は、列を作っていません。つまり、要素と要素とのあいだに位置関係はない、ということです。したがって、リストとは違って、セットの要素を先頭からの番号で指定するということはできません。

また、リストは、等しい要素を何個でも持つことができます。それに対して、セットは、2 個以上の等しい要素を持つことができません。言い換えると、いかなるセットも、その要素は互いに異なっている、ということです。

セットの型は、次のような型名によって識別されます。

変更不可能なセット　`Set<`要素の型`>`

変更可能なセット　`MutableSet<`要素の型`>`

たとえば、整数を要素とする変更不可能なセットの型は、

```
Set<Int>
```

という型名によって識別されます。同じように、文字列を要素とする変更可能なセットの型は、

```
MutableSet<String>
```

という型名によって識別されます。

6.3.2　セットを生成する関数

`setOf` という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数をひとつにまとめることによってできる変更不可能なセットを生成して、その変更不可能なセットを戻り値として返します。

```
>>> setOf(38, 27, 64, 52, 41, 70)
[38, 27, 64, 52, 41, 70]
```

このように、REPL は、リストの場合と同じように、入力された式の値がセットだった場合も、

`[`要素`, `要素`, `要素`, ···]`

という形でそれを出力します。

`println` と `print` も、引数としてセットを受け取った場合、それを同じ形で出力します。

```
>>> println(setOf("namako", "umiushi", "hitode"))
[namako, umiushi, hitode]
```

`mutableSetOf` という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数をひとつにまとめることによってできる変更可能なセットを生成して、その変更可能なセットを戻り値として返します。

```
>>> mutableSetOf(82, 36, 54, 19, 66, 43)
[82, 36, 54, 19, 66, 43]
```

`setOf` と `mutableSetOf` は、受け取った引数の中に 2 個以上の等しいオブジェクトがあった場合は、それらのうちのひとつだけを残して、それ以外の等しいオブジェクトは要素にしません。

```
>>> setOf(8, 1, 3, 1, 4, 1, 7, 1, 5, 1, 6)
[8, 1, 3, 4, 7, 5, 6]
```

6.3.3　空セット

0 個の要素から構成されるセットは、「空セット」(empty set) と呼ばれます。

空セットを生成するためには、型名を記述することによって、その空セットの型を明示する必要があります。

```
>>> val a: Set<Int> = setOf()
>>> a
```

```
[]
>>> val b: MutableSet<String> = mutableSetOf()
>>> b
[]
```

6.3.4 コレクションに共通するセットの機能

セットはコレクションの一種ですので、コレクションに共通する機能は、セットでも使うことができます。

つまり、セットが持っている size という名前のプロパティーは、そのセットの要素の個数ですし、 in または !in という二項演算子を使うことによって、オブジェクトがセットの要素かどうかということを判定することができますし、 isEmpty というメソッドは、セットが空セットならば真、そうでなければ偽を返します。。

```
>>> val a = setOf(84, 23, 51, 72, 62)
>>> a.size
5
>>> 72 in a
true
>>> 49 in a
false
>>> val b: Set<Int> = setOf()
>>> b.isEmpty()
true
>>> a.isEmpty()
false
```

6.3.5 セットに対する繰り返し

セットは、繰り返し可能オブジェクトです。したがって、 for 文を使うことによって、セットを構成している要素をひとつずつ処理する繰り返しを記述することができます。ただし、処理される順番は、期待したとおりになるとは限りません。

```
>>> val a = setOf("namako", "umiushi", "hitode", "kamenote")
>>> for (s in a) {
...     println(s)
... }
namako
umiushi
hitode
kamenote
```

6.3.6 変更可能なセットへの要素の追加

変更可能なセットは、 add というメソッドを持っています。このメソッドを呼び出すことによって、変更可能なセットに要素を追加することができます。 add には、引数として、追加したい要素を渡します。

```
>>> val a = mutableSetOf('a', 'b', 'c', 'd', 'e')
>>> a.add('x')
true
>>> a
[a, b, c, d, e, x]
```

6.3.7 変更可能なセットの要素の削除

変更可能なセットは、 remove というメソッドを持っています。このメソッドは、1 個のオブジェクトを受け取って、もしもそのオブジェクトと等しい要素が存在するならば、その要素を削除します。このメソッドの戻り値は、引数と等しい要素が存在していた場合は真で、存在していなかった場合は偽です。

```
>>> val a = mutableSetOf(8, 4, 2, 3, 6, 5)
>>> a.remove(6)
true
>>> a
[8, 4, 2, 3, 5]
>>> a.remove(7)
```

```
false
```

6.3.8 セットの併合

+という二項演算子は、左右の式の値が同じ型のセットの場合は、左右のセットを併合することによってできるセットを求める、という動作をあらわします。

```
>>> setOf(43, 25, 67, 59) + setOf(85, 67, 43, 38)
[43, 25, 67, 59, 85, 38]
```

+は、左側の式の値がセットで、右側の式の値が、左のセットの要素と同じ型のオブジェクトの場合は、左のセットに右のオブジェクトを追加することによってできるセットを求める、という動作をあらわします。

```
>>> setOf(43, 25, 67, 59) + 85
[43, 25, 67, 59, 85]
```

6.3.9 プログレッションからセットへの変換

プログレッションは、toSetという名前のメソッドを持っています。このメソッドは、プログレッションを変更不可能なセットに変換して、そのセットを戻り値として返します。

```
>>> (7..20).toSet()
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> (20 downTo 7).toSet()
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7]
>>> (100 downTo 0 step 10).toSet()
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

同じように、プログレッションが持っているtoMutableSetというメソッドを呼び出すことによって、プログレッションを変更可能なセットに変換することもできます。

6.3.10 文字列からセットへの変換

プログレッションと同じように、文字列も、toSetという名前のメソッドを持っています。このメソッドは、文字列を文字の変更不可能なセットに変換して、そのセットを戻り値として返します。

```
>>> "namako".toSet()
[n, a, m, k, o]
```

同じように、文字列が持っているtoMutableSetというメソッドを呼び出すことによって、文字列を文字の変更可能なセットに変換することもできます。

6.3.11 リストからセットへの変換

プログレッションや文字列と同じように、リストも、toSetという名前のメソッドを持っています。このメソッドは、リストを変更不可能なセットに変換して、そのセットを戻り値として返します。

```
>>> val a = listOf(3, 1, 2, 1, 4, 1, 7, 1, 6)
>>> a
[3, 1, 2, 1, 4, 1, 7, 1, 6]
>>> a.toSet()
[3, 1, 2, 4, 7, 6]
```

同じように、リストが持っているtoMutableSetというメソッドを呼び出すことによって、リストを文字の変更可能なセットに変換することもできます。

6.3.12 セットからリストへの変換

セットは、toListという名前のメソッドを持っています。このメソッドは、セットを変更不可能なリストに変換して、そのリストを戻り値として返します。ただし、リストの中で要素が並ぶ順番は、期待したとおりになるとは限りません。

```
>>> setOf(3, 2, 1, 2, 1, 4, 2, 3).toList()
[3, 2, 1, 4]
```

同じように、セットが持っている toMutableList というメソッドを呼び出すことによって、セットを文字の変更可能なリストに変換することもできます。

6.3.13 変更不可能なセットから変更可能なセットへの変換

変更不可能なセットは、toMutableSet という名前のメソッドを持っています。このメソッドは、変更不可能なセットを変更可能なセットに変換して、その変更可能なセットを戻り値として返します。

```
>>> val a = setOf(3, 5, 2, 7, 4)
>>> val b = a.toMutableSet()
>>> b.add(8)
true
>>> b
[3, 5, 2, 7, 4, 8]
```

6.3.14 変更可能なセットから変更不可能なセットへの変換

変更可能なセットは、toSet という名前のメソッドを持っています。このメソッドは、変更可能なセットを変更不可能なセットに変換して、その変更不可能なセットを戻り値として返します。

```
>>> val a = mutableSetOf(3, 5, 2, 7, 4)
>>> val b = a.toSet()
>>> b.add(8)
error: unresolved reference: add
b.add(8)
  ^
```

6.3.15 エラトステネスのふるい

2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

エラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数から構成される集合を作る。
(2) 2 を i とする。
(3) 次の (4) と (5) を、i^2 が n 以下であるあいだ繰り返す。
(4) i が集合の要素ならば、その倍数のうちで集合の要素であるものをすべて削除する。ただし、i 自身は削除しない。
(5) i に 1 を加算した整数を i とする。

この手順が終了すると、素数だけを要素とする集合ができます。

次のプログラムの中で宣言されている eratosthenes という関数は、プラスの整数 n を受け取って、エラトステネスのふるいを使って 2 から n までの範囲にあるすべての素数を求めて、それらの素数から構成される変更不可能なセットを返します。

プログラムの例 eratos.kt

```
fun eratosthenes(n: Int): Set<Int> {
    val sieve = (2..n).toMutableSet()
    var i = 2
    while (i * i <= n) {
        if (i in sieve) {
            var j = i + i
            while (j <= n) {
                if (j in sieve) sieve.remove(j)
                j += i
            }
        }
        i += 1
    }
    return sieve.toSet()
}
```

実行例

```
>>> eratosthenes(50)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

6.4 マップ

6.4.1 マップの基礎

「マップ」(dictionary) と呼ばれるオブジェクトは、セットと同じように、順序を持たないコレクションです。

セットとマップとの相違点は、要素の構造です。セットの個々の要素は単独のオブジェクトですが、マップの要素は、2 個のオブジェクトがペアになったものです。

マップの要素を構成しているペアのオブジェクトのそれぞれは、「キー」(key) と「値」(value) と呼ばれます。

キーは、個々の要素を識別するために使われるオブジェクトです。したがって、1 個のマップの中に、等しいキーを持つ要素が 2 個以上存在することはできません。それに対して、値は、要素の識別には使われませんので、1 個のマップの中に、同じ値を持つ要素が 2 個以上存在することも可能です。

マップは、辞書に似ています。辞書は、多くの項目から構成されています。それぞれの項目は、見出し語と、その見出し語についての説明とをペアにしたものです。そして、それぞれの項目は、見出し語によって識別されます。マップというのは、キーというのが見出し語に相当して、値というのが見出し語についての説明に相当する、辞書のようなものだと考えることができます。

マップの型は、次のような型名によって識別されます。

変更不可能なマップ　`Map<`[キーの型]`,` [値の型]`>`

変更可能なマップ　`MutableMap<`[キーの型]`,` [値の型]`>`

たとえば、文字をキー、真偽値を値とする要素から構成される変更不可能なマップの型は、

```
Map<Char, Boolean>
```

という型名によって識別されます。同じように、文字列をキー、整数を値とする要素から構成される変更可能なマップの型は、

```
MutableMap<String, Int>
```

という型名によって識別されます。

6.4.2 マップを生成する関数

`mapOf` という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数をひとつにまとめることによってできる変更不可能なマップを生成して、その変更不可能なマップを戻り値として返します。

`mapOf` に渡す引数は、キーと値をペアにしたものです。キーと値のペアは、

[キー] `to` [値]

という式を書くことによって生成することができます。たとえば、

```
48 to "namako"
```

という式を書くことによって、48 という整数をキー、`namako` という文字列を値とするペアを生成することができます。

```
>>> mapOf(48 to "namako", 73 to "umiushi", 95 to "hitode")
{48=namako, 73=umiushi, 95=hitode}
```

このように、REPL は、入力された式の値がマップだった場合は、

{[キー]=[値]`,` [キー]=[値]`, ···}`

という形でそれを出力します。

`println` と `print` も、引数としてマップを受け取った場合、それを同じ形で出力します。

```
>>> println(mapOf('a' to 24, 'b' to 38, 'c' to 21))
{a=24, b=38, c=21}
```

mutableMapOf という標準ライブラリー関数は、同じ型を持つ 0 個以上の任意の個数の引数を受け取って、それらの引数をひとつにまとめることによってできる変更可能なマップを生成して、その変更可能なマップを戻り値として返します。

```
>>> mutableMapOf("namako" to 81, "umiushi" to 53)
{namako=81, umiushi=53}
```

mapOf と mutableMapOf は、受け取った引数の中に 2 個以上の等しいキーを持つペアがあった場合は、それらのうちのひとつだけを残して、それ以外の等しいキーを持つペアは要素にしません。

```
>>> mapOf(5 to 7, 4 to 2, 5 to 0, 3 to 8, 5 to 9, 3 to 6)
{5=9, 4=2, 3=6}
```

6.4.3 空マップ

0 個の要素から構成されるマップは、「空マップ」(empty map) と呼ばれます。

空マップを生成するためには、型名を記述することによって、その空マップの型を明示する必要があります。

```
>>> val a: Map<Int, Char> = mapOf()
>>> a
{}
>>> val b: MutableMap<String, Int> = mutableMapOf()
>>> b
{}
```

6.4.4 コレクションに共通するマップの機能

マップはコレクションの一種ですので、コレクションに共通する機能は、マップでも使うことができます。

つまり、マップが持っている size という名前のプロパティーは、そのマップの要素の個数ですし、isEmpty というメソッドは、マップが空マップならば真、そうでなければ偽を返します。。

```
>>> val a = mapOf(3 to 4, 2 to 8, 5 to 1, 7 to 6, 4 to 3)
>>> a.size
5
>>> val b: Map<String, Int> = mapOf()
>>> b.isEmpty()
true
>>> a.isEmpty()
false
```

マップの場合、 in と !in は、左側の式の値をキーとして持つ要素がマップの中に存在するかどうかということを判定します。

```
>>> val a = mapOf('a' to 35, 'b' to 47, 'c' to 86)
>>> 'b' in a
true
>>> 'x' in a
false
```

6.4.5 マップに対する繰り返し

マップは、繰り返し可能オブジェクトです。したがって、for 文を使うことによって、マップを構成している要素をひとつずつ処理する繰り返しを記述することができます。ただし、処理される順番は、期待したとおりになるとは限りません。

マップに対する繰り返しを実行する for 文は、

for ((識別子$_1$, 識別子$_2$) in 式) ブロック

と書きます。そうすると、識別子$_1$ は、要素のキーで初期化された変更不可能な変数の名前になって、識別子$_2$ は、要素の値で初期化された変更不可能な変数の名前になります。

```
>>> val a = mapOf('a' to 38, 'b' to 52, 'c' to 96)
>>> for ((k, v) in a) {
...     println("${k}: ${v}")
... }
```

```
a: 38
b: 52
c: 96
```

6.4.6 マップの要素を指定する添字式

リストの場合と同じように、マップも、添字式を書くことによって、それぞれの要素を指定することができます。

ただし、マップの要素は、番号ではなくて、キーによって識別されますので、添字の中には、指定したい要素のキーを求める式を書くことになります。

マップの要素を指定する添字式を評価すると、その値として、キーによって指定された要素の値が得られます。

```
>>> mapOf("namako" to 35, "umiushi" to 53)["namako"]
35
>>> mapOf(7 to 's', 8 to 'e', 5 to 'f', 2 to 't')[8]
e
```

添字式を評価したときに、添字の中の式の値をキーとして持つ要素がマップの中に存在しなかった場合、その添字式の値として、null というオブジェクトが得られます。

```
>>> mapOf('a' to 83, 'b' to 41, 'c' to 26, 'd' to 57)['x']
null
```

null は、「何もない」ということを意味しているオブジェクトです。null という式を評価すると、その値として、このオブジェクトが得られます。

```
>>> null
null
```

6.4.7 変更可能なマップの要素の変更

変数に代入された変更可能なマップの要素を指定する添字式を評価すると、指定された要素の値が右辺値として得られるだけではなくて、指定された要素の値を参照している箱が、その左辺値として得られます。ですから、代入演算子の左側に添字式を書くことによって、変更可能なマップの要素を変更することができます（ただし、拡張代入演算子を使うことはできません）。

```
>>> val a = mutableMapOf('a' to 43, 'b' to 81, 'c' to 27)
>>> a['b'] = 200
>>> a
{a=43, b=200, c=27}
```

変更可能なマップの要素を変更する方法としては、添字式を使うという方法のほかに、変更可能なマップが持っている put というメソッドを使うという方法もあります。

put は、2 個の引数を受け取ります。1 個目は変更したい要素を指定するキーで、2 個目は要素の変更後の値です。put の戻り値は、変更した要素の変更前の値です。

```
>>> val a = mutableMapOf('a' to 47, 'b' to 92, 'c' to 75)
>>> a.put('b', 333)
92
>>> a
{a=47, b=333, c=75}
```

6.4.8 変更可能なマップへの要素の追加

代入演算子の左側に書かれた、マップの要素を指定する添字式を評価したときに、もしも、添字によって指定されたキーを持つ要素が存在しなかった場合は、そのキーを持つ要素がマップに追加されて、代入演算子の右側に書かれた式の値が、追加された要素の値になります。

```
>>> val a = mutableMapOf('a' to 81, 'b' to 45, 'c' to 30)
>>> a['x'] = 800
>>> a
{a=81, b=45, c=30, x=800}
```

変更可能なマップが持っている put も、1 個目の引数をキーとして持つ要素が存在しなかった場合は、そのキーを持つ要素をマップに追加して、2 個目の引数を追加した要素の値にします。その場合、put は戻り値として null を返します。

```
>>> val a = mutableMapOf('a' to 37, 'b' to 64, 'c' to 58)
>>> a.put('x', 900)
null
>>> a
{a=37, b=64, c=58, x=900}
```

6.4.9　変更可能なマップの要素の削除

変更可能なマップは、remove というメソッドを持っています。このメソッドは、1 個のオブジェクトを受け取って、もしもそのオブジェクトをキーとして持つ要素が存在するならば、その要素を削除します。このメソッドの戻り値は、引数をキーとして持つ要素が存在していた場合はその要素の値で、存在していなかった場合は null です。

```
>>> val a = mutableMapOf('a' to 87, 'b' to 23, 'c' to 64)
>>> a.remove('b')
23
>>> a.remove('x')
null
```

6.4.10　マップの併合

+ という二項演算子は、左右の式の値が同じ型のマップの場合は、左右のマップを併合することによってできるマップを求める、という動作をあらわします。

```
>>> mapOf('a' to 43, 'b' to 24) + mapOf('c' to 53, 'a' to 44)
{a=44, b=24, c=53}
```

+ は、左側の式の値がマップで、右側の式の値が、左のマップの要素と同じ型を持つキーと値のペアである場合は、左のマップに右のペアを追加することによってできるマップを求める、という動作をあらわします。

```
>>> mapOf('a' to 66, 'b' to 84) + ('x' to 47)
{a=66, b=84, x=47}
```

6.4.11　変更不可能なマップから変更可能なマップへの変換

変更不可能なマップは、toMutableMap という名前のメソッドを持っています。このメソッドは、変更不可能なマップを変更可能なマップに変換して、その変更可能なマップを戻り値として返します。

```
>>> val a = mapOf('a' to 44, 'b' to 86, 'c' to 73)
>>> val b = a.toMutableMap()
>>> b['x'] = 555
>>> b
{a=44, b=86, c=73, x=555}
```

6.4.12　変更可能なマップから変更不可能なマップへの変換

変更可能なマップは、toMap という名前のメソッドを持っています。このメソッドは、変更可能なマップを変更不可能なマップに変換して、その変更不可能なマップを戻り値として返します。

第 7 章　クラス

7.1　クラスの基礎

7.1.1　クラスとは何か

第 3.3 節で説明したように、数値や文字や文字列や真偽値などが持っている型を識別する、Int や Char や String や Boolean などの識別子は、「クラス名」(class name) と呼ばれます。

クラス名というのは、その名前のとおり、「クラス」(class) というものを識別する名前のことです。クラスというのは、オブジェクトの仕様を決定するもののことです。

クラスとオブジェクトとの関係は、鋳型と鋳物との関係に似ています。

「鋳物」というのは、高熱で溶けた金属を冷やして固めることによって製造される金属製品のことです。鋳物を製造するとき、溶けた金属は、「鋳型」と呼ばれるものの中にある空洞に流し

込まれます。そうすることによって、鋳物は、鋳型の中の空洞と同じ形を持つことになります。そして、ひとつの鋳型からは、同じ形を持つ鋳物をいくつでも製造することができます。

鋳物が鋳型から製造されるのと同じように、オブジェクトはクラスから作られます。ひとつの鋳型から製造されたすべての鋳物が同じ形を持つのと同じように、ひとつのクラスから作られたすべてのオブジェクトは、同じプロパティーとメソッドを持つことになります。

7.1.2　ユーザー定義クラスと標準ライブラリークラス

クラスを作って、そのクラスに名前を与えることを、クラスを「宣言する」(declare) と言います。

Kotlin では、「クラス宣言」(class declaration) と呼ばれる記述をプログラムの中に書くことによって、クラスを自由に宣言することができます。プログラムの中にクラス宣言を書くことによって宣言されたクラスは、「ユーザー定義クラス」(user-defined class) と呼ばれます。

Kotlin の標準ライブラリーには、さまざまなクラスも含まれています。標準ライブラリーに含まれているクラスは、「標準ライブラリークラス」(standard library class) と呼ばれます。標準ライブラリークラスは、プログラムの中にクラス宣言を書かなくても利用することができます。`Int` や `Char` や `String` や `Boolean` などは、標準ライブラリークラスです。

7.1.3　コンストラクタ

クラスからオブジェクトを作ることを、オブジェクトを「生成する」(generate) と言います。

クラスは、そのクラスからオブジェクトを生成して、そのオブジェクトを戻り値として返す関数を持っています。そのような、クラスが持っている、そのクラスからオブジェクトを生成する関数は、「コンストラクタ」(constructor) と呼ばれます。

コンストラクタは、それを持っているクラスの名前と同じ識別子によって識別されます。ですから、クラスと同じ名前を持つ関数を呼び出す、

クラス名(式, ･･･)

という形の関数呼び出しを書くことによって、そのクラスが持っているコンストラクタを呼び出すことができます。コンストラクタに渡した引数は、生成されたオブジェクトを初期化するために使われます。

たとえば、

IntRange(式$_1$, 式$_2$)

という関数呼び出しを評価すると、`IntRange` というクラスが持っているコンストラクタが呼び出されて、そのコンストラクタが、式$_1$ の値を下限、式$_2$ の値を上限とする整数の範囲のオブジェクトを生成して、そのオブジェクトを戻り値として返します。

```
>>> IntRange(30, 70)
30..70
```

ユーザー定義クラスと、大多数の標準ライブラリークラスについては、コンストラクタを呼び出すことによってオブジェクトを生成することができます。しかし、標準ライブラリークラスの中には、コンストラクタを呼び出すことができないものもあります。たとえば、`Int` は、コンストラクタを呼び出すことができない標準ライブラリークラスのひとつです。

```
>>> Int()
error: cannot access '<init>': it is private in 'Int'
Int()
^
```

7.2　クラス宣言

7.2.1　クラス宣言の基礎

第 7.1.2 節で説明したように、クラスは、「クラス宣言」(class declaration) と呼ばれる記述を書くことによって宣言することができます。

最も単純な形のクラス宣言は、

`class` 識別子

と書きます。この形のクラス宣言を書くことによって、新しいクラスが宣言されて、「識別子」のところに書かれた識別子が、そのクラスに名前として与えられます。たとえば、

```
class Namako
```

というクラス宣言を書くことによって、新しいクラスが宣言されて、それに対して Namako という名前が与えられます。

この形のクラス宣言は、コンストラクタの明示的な宣言を含んでいません。しかし、この形のクラス宣言を書いた場合、最低限の動作をするコンストラクタが暗黙のうちに宣言されます。暗黙のうちに宣言されるコンストラクタは、引数を受け取らず、クラスからオブジェクトを生成して、そのオブジェクトを戻り値として返します。

それでは、クラス宣言を REPL に入力して、そのクラスからオブジェクトを生成してみましょう。

```
>>> class Namako
>>> Namako()
Line_0$Namako@710787
```

REPL は、生成されたオブジェクトを文字列に変換した結果を出力しているのですが、このクラスはオブジェクトを文字列に変換する機能がデフォルトのままですので、出力される文字列は、このような意味不明なものになります。オブジェクトを文字列に変換する機能を変更する方法については、第 7.6.11 項で説明することにしたいと思います。

なお、クラスに名前として与える識別子は、Namako のように、その先頭の文字を大文字にするという慣習があります。

7.2.2 バッキングフィールド

オブジェクトを自分の中に保持する必要のあるオブジェクトは、自分の中に変数を持つ必要があります。オブジェクトの中にある変数は、「フィールド」(field) と呼ばれます。

第 2.5.2 項で説明したように、プロパティーというのは、オブジェクトという箱の中と外とのあいだで、オブジェクトの出し入れをする窓口です。

Kotlin では、プロパティーは明示的に宣言することができます。それに対して、フィールドを明示的に宣言することはできません。フィールドは、プロパティーを宣言したときに、もしも必要ならば、暗黙のうちに宣言されます。プロパティーを宣言したときに暗黙のうちに宣言されたフィールドは、そのプロパティーの「バッキングフィールド」(backing field) と呼ばれます。

7.2.3 プロパティー宣言

クラス宣言の中には、「プロパティー宣言」(property declaration) と呼ばれるものを書くことができます。プロパティー宣言というのは、その名前のとおり、プロパティーを宣言する記述のことです。

プロパティー宣言は、基本的には、変数宣言と同じように、

```
val 識別子 : 型名 = 式
```

と書くか、または、

```
var 識別子 : 型名 = 式
```

と書きます。この中の「識別子」のところにはプロパティーに与える名前を書いて、「型名」のところにはプロパティーが扱うオブジェクトの型の名前を書きます。

val を使ったプロパティー宣言は、初期化したのちはオブジェクトを変更することができないプロパティー（つまり、取り出すことしかできない窓口）を宣言します。それに対して、var を使ったプロパティー宣言は、オブジェクトを変更することが可能なプロパティー（つまり、取り出すことも入れることもできる窓口）を宣言します。

この項で説明した書き方でプロパティーを宣言した場合は、バッキングフィールドが暗黙のうちに宣言されます。バッキングフィールドを持たないプロパティーの作り方については、第 7.4.6 項で説明します。

7.2.4 プライマリーコンストラクタ宣言

プロパティー宣言を書くことができる場所としては、二つの場所があります。そのひとつは、「プライマリーコンストラクタ宣言」(primary constructor declaration) と呼ばれる記述の中です。

プライマリーコンストラクタ宣言というのは、「プライマリーコンストラクタ」(primary constructor) と呼ばれるコンストラクタを宣言する、クラス宣言の一部分のことです。それを書く場所は、クラス名になる識別子の直後です。

プライマリーコンストラクタ宣言は、プロパティー宣言をコンマで区切って並べて、その全体を丸括弧で囲んだ記述です。つまり、クラス名になる識別子の直後に、

(プロパティー宣言 , ･･･)

という形の記述を書けば、それがプライマリーコンストラクタ宣言になるということです。

プライマリーコンストラクタ宣言を書いた場合、その中に書かれたプロパティー宣言によって宣言されたプロパティーは、プライマリーコンストラクタが受け取った引数をバッキングフィールドに代入します。ですから、プライマリーコンストラクタは、宣言されたプロパティーと同じ個数の引数を受け取ることになります。

プライマリーコンストラクタ宣言の中にプロパティー宣言を書く場合、

= 式

という部分は、省略してもかまいません。

```
>>> class Umiushi(val a: Int, val b: Char, val c: String)
>>> val u = Umiushi(87, 'D', "hoge")
>>> u.a
87
>>> u.b
D
>>> u.c
hoge
```

var を使ってプロパティーを宣言した場合、代入演算子の左側に、そのプロパティーを指定する式を書くことによって、そのプロパティーのバッキングフィールドに、代入演算子の右側の式の値を代入することができます。

```
>>> class Hitode(var a: Int)
>>> val h = Hitode(62)
>>> h.a
62
>>> h.a = 347
>>> h.a
347
```

プライマリーコンストラクタ宣言の中にプロパティー宣言を書くときに、

= 式

という部分を省略しないで書いたとすると、プライマリーコンストラクタに渡す引数を省略した場合に、その式の値がバッキングフィールドに代入されます。

```
>>> class Kurage(val a: String = "default")
>>> val k1 = Kurage("hoge")
>>> k1.a
hoge
>>> val k2 = Kurage()
>>> k2.a
default
```

7.2.5　有理数のクラス

整数を整数で除算した商という形によって数値を表現したものは、「分数」(fraction) と呼ばれます。a を整数、b を 0 ではない整数とするとき、a/b は、a を b で除算した商という数値を表現している分数です。このとき、a は、この分数の「分子」(numerator) と呼ばれ、b は、この分数の「分母」(denominator) と呼ばれます。

分数によって表現することができる数値は、「有理数」(rational number) と呼ばれます。異なる分数は、必ずしも異なる有理数を表現しているとは限りません。たとえば、2/3、4/6、6/9、10/15 などは、分数としては異なっていますが、どれも同じ有理数を表現しています。

次のプログラムの中で宣言されている Rational というクラスは、有理数のオブジェクトを生成します。

プログラムの例 rational.kt

```
class Rational(val numerator: Int, val denominator: Int)
```

実行例

```
>>> val a = Rational(17, 30)
>>> a.numerator
17
>>> a.denominator
30
```

7.2.6 中括弧を持つクラス宣言

プロパティー宣言を書くことができる、もうひとつの場所は、中括弧の中です。

クラス宣言は、

```
class 識別子 {
    プロパティー宣言
        ⋮
}
```

というように、クラス名になる識別子の後ろに中括弧を書くことができて、その中にもプロパティー宣言を書くことができます。

プライマリーコンストラクタによってバッキングフィールドを初期化する必要がないプロパティーを宣言する場合は、このように、中括弧の中にそのプロパティーの宣言を書きます。

中括弧の中にプロパティー宣言を書く場合、バッキングフィールドを初期化するための式から型を特定することができるならば、型名は省略することができます。

```
>>> class Fujitsubo {
...     val a = false
...     val b = listOf(38, 72, 65, 44)
... }
>>> val f = Fujitsubo()
>>> f.a
false
>>> f.b
[38, 72, 65, 44]
```

クラス宣言の中括弧は、プライマリーコンストラクタ宣言の後ろに書くこともできます。つまり、クラス宣言は、

```
class 識別子 プライマリーコンストラクタ宣言 {
    プロパティー宣言
        ⋮
}
```

というように書くこともできるということです。

中括弧の中で宣言されたプロパティーのバッキングフィールドの初期化は、プライマリーコンストラクタ宣言の中で宣言されたプロパティーのバッキングフィールドの初期化が終わったあとで実行されます。ですから、中括弧の中で宣言されたプロパティーのバッキングフィールドを、プライマリーコンストラクタ宣言の中で宣言されたプロパティーのバッキングフィールドを使って初期化する、ということが可能です。

```
>>> class Sango(val a: Char) {
...     val b = a.toInt()
... }
>>> val s = Sango('M')
>>> s.a
```

```
M
>>> s.b
77
```

7.2.7 イニシャライザーブロック

コンストラクタは、デフォルトでは、クラスからオブジェクトを生成して、受け取った引数をバッキングフィールドに代入する、という動作しかしません。

デフォルトの動作に加えて、さらに別の動作もするようなコンストラクタを宣言したい場合は、「イニシャライザーブロック」(initializer block) と呼ばれる、ブロックを含む記述をクラス宣言の中に書きます。

クラス宣言の中にイニシャライザーブロックを書いておくと、コンストラクタは、オブジェクトを生成して、バッキングフィールドに引数を代入したのち、イニシャライザーブロックの中に書かれた動作を実行します。

プライマリーコンストラクタが実行するイニシャライザーブロックは、

```
init ブロック
```

と書きます。これを書く場所は、クラス宣言の中括弧の中です。

次のプログラムの中で宣言されている Speak というクラスのプライマリーコンストラクタは、「初期値は○○です。」という形で、受け取った引数を出力します。

プログラムの例 speak.kt

```
class Speak(val s: String) {
    init {
        println("初期値は${s}です。")
    }
}
```

実行例

```
>>> val a = Speak("namako")
初期値は namako です。
>>> a.s
namako
```

プライマリーコンストラクタを宣言しなかった場合に暗黙のうちに宣言されるコンストラクタも、 init で始まるイニシャライザーブロックを実行します。

プログラムの例 talk.kt

```
class Talk {
    init {
        println("私はイニシャライザーブロックです。")
    }
}
```

実行例

```
>>> val a = Talk()
私はイニシャライザーブロックです。
```

7.3 メソッド宣言

7.3.1 メソッドの宣言の基礎

第 2.5.1 項で説明したように、オブジェクトの中にある関数は、「メソッド」(method) と呼ばれます。

メソッドを宣言する記述は、「メソッド宣言」(method declaration) と呼ばれます。メソッド宣言は、クラス宣言の中括弧の中に書きます。

クラス宣言の中括弧の中にメソッド宣言を書くと、そのクラス宣言によって宣言されたクラスから生成されたオブジェクトは、そのメソッド宣言によって宣言されたメソッドを持つことになります。

メソッドは関数ですから、メソッド宣言の書き方は、関数宣言と同じです。

次のプログラムの中で宣言されている Greet というクラスは、hello というメソッドを持つオブジェクトを生成します。

プログラムの例 greet.kt

```
class Greet {
    fun hello() {
        println("こんにちは。")
    }
}
```

hello は、「こんにちは。」という文字列を出力するメソッドです。

実行例

```
>>> val a = Greet()
>>> a.hello()
こんにちは。
```

練習問題 7.1 divine というメソッドを持つオブジェクトを生成する、Fortune というクラスを宣言してください。divine は、「あなたの今日の運勢は大吉です。」という文字列を出力するメソッドです。

実行例

```
>>> val a = Fortune()
>>> a.divine()
あなたの今日の運勢は大吉です。
```

7.3.2 メンバー

プロパティーとメソッドは、総称して「メンバー」(member) と呼ばれます。そして、メンバーの名前は、「メンバー名」(member name) と呼ばれます。

オブジェクトの外から、そのオブジェクトのメンバーを指定するためには、

式 . メンバー名

という形のものを書く必要がありますが、メソッド宣言の中で、自分と同じオブジェクトの中にあるメンバーを指定する場合、

式 .

という部分は、書く必要がありません。つまり、その場合はメンバー名だけを書けばいいわけです。

次のプログラムの中で宣言されている Account というクラスは、預金口座のオブジェクトを生成します。

プログラムの例 account.kt

```
class Account(var money: Int) {
    fun deposit(m: Int) {
        money += m
    }
}
```

money というプロパティーのバッキングフィールドが、預金の金額です。deposit というメソッドを呼び出して、引数として金額を渡すと、その金額が預金に加算されます。

実行例

```
>>> val a = Account(300)
>>> a.money
300
>>> a.deposit(500)
>>> a.money
800
>>> a.deposit(-200)
>>> a.money
600
```

次のプログラムは、第 7.2.5 項で紹介した有理数のクラス宣言に、print というメソッドの宣言を追加したものです。

プログラムの例　rational2.kt

```
class Rational(val numerator: Int, val denominator: Int) {
    fun print() {
        println("${numerator}/${denominator}")
    }
}
```

print は、「分子/分母」という形で有理数を出力します。

実行例

```
>>> val a = Rational(17, 30)
>>> a.print()
17/30
```

練習問題 7.2　次のプロパティーとメソッドを持つオブジェクトを生成する、Counter というクラスを宣言してください。

- オブジェクトが生成されたときに 0 で初期化されるバッキングフィールドを持つ n というプロパティー。
- n をインクリメントする、count というメソッド。

実行例

```
>>> val a = Counter()
>>> a.n
0
>>> a.count()
>>> a.n
1
>>> a.count()
>>> a.n
2
```

7.4　アクセサー

7.4.1　アクセサーの基礎

第 2.5.2 項で説明したように、プロパティーというのは、オブジェクトという箱の中と外とのあいだで、オブジェクトの出し入れをする窓口のことです。

オブジェクトという箱の中と外とのあいだでのオブジェクトの出し入れには、「アクセサー」(accessor) と呼ばれるメソッドが使われます。

プロパティーを宣言すると、そのプロパティーが持つアクセサーが暗黙のうちに宣言されます。

7.4.2　ゲッターとセッター

アクセサーには、オブジェクトという箱からオブジェクトを取り出すものと、オブジェクトという箱にオブジェクトを入れるものの 2 種類のものがあります。オブジェクトからオブジェクトを取り出すアクセサーは「ゲッター」(getter) と呼ばれ、オブジェクトにオブジェクトを入れるアクセサーは「セッター」(setter) と呼ばれます。

ゲッターは、オブジェクトという箱から取り出したオブジェクトを戻り値として返すメソッドで、引数は受け取りません。セッターは、1 個の引数を受け取って、それをバッキングフィールドに代入するメソッドで、戻り値は返しません。

val を使ってプロパティーを宣言した場合は、暗黙のうちに宣言されるアクセサーはゲッターだけです。それに対して、var を使ってプロパティーを宣言した場合は、ゲッターだけではなくて、セッターも暗黙のうちに宣言されます。

7.4.3 カスタムアクセサー

アクセサーは、暗黙のうちに宣言されるものだけではなくて、明示的に宣言することも可能です。明示的に宣言されたゲッターは「カスタムゲッター」(custom getter) と呼ばれ、明示的に宣言されたセッターは「カスタムセッター」(custom setter) と呼ばれます。カスタムゲッターとカスタムセッターは、総称して「カスタムアクセサー」(sustom accessor) と呼ばれます。

7.4.4 カスタムアクセサーの宣言

カスタムアクセサーの宣言は、プロパティー宣言の一部分です。

第 7.2.3 項で説明した書き方でプロパティー宣言をまず書いて、その直後にカスタムアクセサーの宣言を書くと、その全体は、カスタムアクセサーを持つプロパティーを宣言するプロパティー宣言になります。

カスタムゲッターの宣言とカスタムセッターの宣言は、どちらが先でどちらが後でもかまいません。

カスタムアクセサーの宣言の中では、そのカスタムアクセサーによって扱われるバッキングフィールドは、`field` という識別子によって識別されます。

7.4.5 カスタムゲッターの宣言

カスタムゲッターは、

`get() =` 式

または、

`get()` ブロック

という形の宣言、つまり、`get` という名前を持つ、引数が 0 個のメソッドを宣言するメソッド宣言から、先頭の `fun` と戻り値の型名を取り除いた形の宣言を書くことによって宣言することができます（戻り値の型名を書くことは可能ですが、プロパティーの型とは異なる型のオブジェクトを返すゲッターを宣言することはできませんので、書く必要性がありません）。

次のプログラムの中で宣言されている `Twice` というクラスは、`n` というプロパティーを持つオブジェクトを生成します。

プログラムの例 `twice.kt`

```
class Twice {
    var n = 0
        get() = field * 2
}
```

`n` は、カスタムゲッターを持っています。`n` のカスタムゲッターは、バッキングフィールドに代入されている整数を 2 倍した結果を返します。

実行例

```
>>> val a = Twice()
>>> a.n = 60
>>> a.n
120
```

練習問題 7.3 `s` というプロパティーを持つオブジェクトを生成する、`Bracket` というクラスを宣言してください。`s` については、バッキングフィールドに代入されている文字列を角括弧で囲んだものを返すカスタムゲッターを宣言してください。

実行例

```
>>> val a = Bracket()
>>> a.s = "namako"
>>> a.s
[namako]
```

7.4.6 バッキングフィールドを持たないプロパティー

プロパティーを宣言すると、多くの場合、それに伴って暗黙のうちにバッキングフィールドが宣言されますが、バッキングフィールドが宣言されるのは、あくまでそれが必要とされる場合だけです。バッキングフィールドを必要としないプロパティーを宣言した場合、バッキングフィールドは宣言されません。

次のプログラムの中で宣言されている Hunred というクラスから生成されたオブジェクトは、n というプロパティーを持っています。

プログラムの例 hundred.kt

```
class Hundred {
    val n: Int
        get() = 100
}
```

実行例

```
>>> val a = Hundred()
>>> a.n
100
```

このクラスが持っている n というプロパティーは、バッキングフィールドを持っていません。なぜなら、このプロパティーが持っているカスタムゲッターは常に 100 を返すだけですので、バッキングフィールドは必要ではないからです。

次のプログラムは、第 7.2.5 項で紹介した有理数のクラス宣言に、 isInteger というプロパティーの宣言を追加したものです。

プログラムの例 rational3.kt

```
class Rational(val numerator: Int, val denominator: Int) {
    val isInteger: Boolean
        get() = numerator % denominator == 0
}
```

isInteger は、有理数のオブジェクトから、その有理数が整数ならば真、そうでなければ偽を取り出します。

実行例

```
>>> val a = Rational(6, 3)
>>> a.isInteger
true
>>> val b = Rational(6, 4)
>>> b.isInteger
false
```

この isInteger も、バッキングフィールドを持たないプロパティーです。

練習問題 7.4 次のプロパティーを持つオブジェクトを生成する、Rectangle というクラスを宣言してください。

- オブジェクトが生成されたときに、長方形の横の長さ（整数）で初期化されるバッキングフィールドを持つ width というプロパティー。
- オブジェクトが生成されたときに、長方形の縦の長さ（整数）で初期化されるバッキングフィールドを持つ height というプロパティー。
- width と height が等しいならば真、そうでなければ偽を取り出す、バッキングフィールドを持たない isSquare というプロパティー。

実行例

```
>>> val a = Rectangle(61, 61)
>>> val b = Rectangle(72, 58)
>>> a.isSquare
true
>>> b.isSquare
false
```

7.4.7 カスタムセッターの宣言

カスタムセッターは、

```
set(value) ブロック
```

という形の宣言、つまり、set という名前を持つ、value という名前の仮引数に引数を受け取るメソッドを宣言するメソッド宣言から、先頭の fun と仮引数の型名を取り除いた形の宣言を書くことによって宣言することができます（仮引数の型名を書くことは可能ですが、プロパティーの型とは異なる型のオブジェクトを受け取るセッターを宣言することはできませんので、書く必要性がありません）。

カスタムセッターの仮引数に与える名前としては、どんな識別子を使ってもかまわないのですが、value という識別子を使うという慣習があります。

次のプログラムの中で宣言されている TenTimes というクラスは、n というプロパティーを持つオブジェクトを生成します。

プログラムの例 tentimes.kt

```
class TenTimes {
    var n = 0
        set(value) {
            field = value * 10
        }
}
```

n は、カスタムセッターを持っています。n のカスタムセッターは、受け取った整数を 10 倍した結果をバッキングフィールドに代入します。

実行例

```
>>> val a = TenTimes()
>>> a.n = 60
>>> a.n
600
```

練習問題 7.5 s というプロパティーを持つオブジェクトを生成する、Shorten というクラスを宣言してください。s については、受け取った文字列の長さが 10 よりも長いならば文字列の先頭から取り出した長さが 10 の部分文字列をバッキングフィールドに代入して、そうでなければ文字列をそのままバッキングフィールドに代入するカスタムセッターを宣言してください。

実行例

```
>>> val a = Shorten()
>>> a.s = "tatsunootoshigo"
>>> a.s
tatsunooto
>>> a.s = "namako"
>>> a.s
namako
```

カスタムセッターを持つプロパティーを宣言した場合も、もしもそのプロパティーにバッキングフィールドが必要ではないならば、バッキングフィールドは宣言されません。

次のプログラムの中で宣言されている CharCode というクラスから生成されたオブジェクトは、c というプロパティーと code というプロパティーを持っています。

プログラムの例 charcode.kt

```
class CharCode {
    var c = '\u0000'
    var code: Int
        get() = c.toInt()
        set(value) {
            c = value.toChar()
        }
}
```

このクラスが持っている二つのプロパティーのうちで、c のほうはバッキングフィールドを持っていますが、code のほうはバッキングフィールドを持っていません。なぜなら、code が持っているカスタムゲッターは、c を文字コードに変換して返すだけですし、カスタムセッターも、受け取った文字コードを文字に変換して c に代入するだけですので、バッキングフィールドは必要ではないからです。

実行例

```
>>> val a = CharCode()
>>> a.c = 'M'
>>> a.c
M
>>> a.code
77
>>> a.code = 78
>>> a.code
78
>>> a.c
N
```

7.5 セカンダリーコンストラクタ

7.5.1 セカンダリーコンストラクタの基礎

第 7.1.3 項で説明したように、クラスが持っている、そのクラスからオブジェクトを生成する関数は、「コンストラクタ」(constructor) と呼ばれます。

そして、第 7.2.4 項で説明したように、クラス宣言の中に「プライマリーコンストラクタ宣言」(primary constructor declaration) と呼ばれる記述を書くことによって、「プライマリーコンストラクタ」(primary constructor) と呼ばれるコンストラクタを宣言することができます。

1 個のクラス宣言の中に書くことができるプライマリーコンストラクタ宣言は、1 個だけです。したがって、2 個以上のプライマリーコンストラクタを持つクラスを宣言することはできません。

しかし、クラスは、コンストラクタを何個でも持つことができます。プライマリーコンストラクタ以外のコンストラクタは、「セカンダリーコンストラクタ」(secondary constructor) と呼ばれます。

ただし、セカンダリーコンストラクタは、受け取る引数の型または個数が、プライマリーコンストラクタとは異なっている必要があります。

7.5.2 セカンダリーコンストラクタ宣言

セカンダリーコンストラクタは、「セカンダリーコンストラクタ宣言」(secondary constructor declaration) と呼ばれる記述を書くことによって宣言することができます。それを書く場所は、クラス宣言の中括弧の中です。

セカンダリーコンストラクタ宣言は、

```
constructor(仮引数の宣言, ・・・) : this(式, ・・・)
```

と書きます。この中の「仮引数の宣言」のところには、関数宣言を書く場合と同じ書き方で、仮引数の宣言を書きます。コロンの右側は、プライマリーコンストラクタを呼び出す式です。このように、セカンダリーコンストラクタからプライマリーコンストラクタを呼び出す式では、クラス名の代わりに this と書きます。たとえば、

```
constructor(n: Int) : this(n.toChar())
```

というセカンダリーコンストラクタ宣言によって宣言されたセカンダリーコンストラクタは、引数として整数を受け取って、その整数を文字コードとする文字を引数にしてプライマリーコンストラクタを呼び出します。

次のプログラムの中で宣言されている Number というクラスは、1 個のセカンダリーコンストラクタを持っています。

プログラムの例 number.kt

```
class Number(val n: Int) {
    constructor(s: String) : this(s.toInt())
}
```

このクラスのセカンダリーコンストラクタは、引数として文字列を受け取って、その文字列を整数に変換した結果を引数にしてプライマリーコンストラクタを呼び出します。

実行例

```
>>> val a = Number(437)
>>> a.n
437
>>> val b = Number("806")
>>> b.n
806
```

クラス宣言の中括弧の中には、セカンダリーコンストラクタ宣言を何個でも書くことができます。ただし、2 個以上のセカンダリーコンストラクタを宣言する場合、それぞれのセカンダリーコンストラクタは、受け取る引数の型または個数が異なっている必要があります。

次のプログラムの中で宣言されている Person というクラスは、2 個のセカンダリーコンストラクタを持っています。

プログラムの例 person.kt

```
class Person(val name: String) {
    constructor(first: String, last: String) :
        this("${first} ${last}")
    constructor(id: Int) : this("id:${id}")
}
```

実行例

```
>>> val a = Person("Tanaka")
>>> a.name
Tanaka
>>> val b = Person("Kumiko", "Yamamoto")
>>> b.name
Kumiko Yamamoto
>>> val c = Person(6803245)
>>> c.name
id:6803245
```

7.5.3 セカンダリーコンストラクタが実行するイニシャライザーブロック

セカンダリーコンストラクタは、デフォルトでは、引数を受け取って、プライマリーコンストラクタを呼び出す式を評価する、という動作しかしません。

プライマリーコンストラクタと同じように、セカンダリーコンストラクタも、イニシャライザーブロックを書くことによって、デフォルトの動作に加えて、さらに別の動作をするように宣言することができます。

セカンダリーコンストラクタ宣言の直後にブロックを書くと、そのブロックが、セカンダリーコンストラクタが実行するイニシャライザーブロックになります。プライマリーコンストラクタとは違って、セカンダリーコンストラクタが実行するイニシャライザーブロックは、その先頭に init は書きません。

セカンダリーコンストラクタのイニシャライザーブロックは、プライマリーコンストラクタが呼び出されて、バッキングフィールドの初期化が終わったあとで実行されます。

次のプログラムの中で宣言されている Seblock というクラスのプライマリーコンストラクタは、「引数は○○で、s の初期値は○○です。」という形で、受け取った引数と、バッキングフィールドの初期値を出力します。

プログラムの例 seblock.kt

```
class Seblock(val s: String) {
    constructor(n: Int) : this("[${n}]") {
        println("引数は${n}で、s の初期値は${s}です。")
    }
}
```

実行例

```
>>> val a = Seblock(617)
引数は 617 で、s の初期値は [617] です。
```

ところで、プライマリーコンストラクタとセカンダリーコンストラクタの両方がイニシャライザーブロックを持っているクラスのオブジェクトを、セカンダリーコンストラクタを呼び出すことによって生成した場合、それぞれのイニシャライザーブロックは、どちらが先に実行されて、どちらがあとに実行されるのでしょうか。

次のプログラムで宣言されている Seblock2 というクラスのセカンダリーコンストラクタを呼び出すことによって、イニシャライザーブロックが実行される順序を確かめてみましょう。

プログラムの例　seblock2.kt

```
class Seblock2(val s: String) {
    init {
        println("s の初期値は${s}です。")
    }
    constructor(n: Int) : this("[${n}]") {
        println("引数として${n}を受け取りました。")
    }
}
```

実行例

```
>>> val a = Seblock2(362)
s に [362] が代入されました。
引数として 362 を受け取りました。
```

この実行例から分かるように、まずプライマリーコンストラクタのイニシャライザーブロックが実行されて、そののちにセカンダリーコンストラクタのイニシャライザーブロックが実行されます。ちなみに、バッキングフィールドの初期化は、プライマリーコンストラクタのイニシャライザーブロックの実行よりも先に実行されます。

7.5.4　プライマリーコンストラクタを持たないクラスのセカンダリーコンストラクタ

セカンダリーコンストラクタを持つことができるのは、プライマリーコンストラクタを持っているクラスだけではありません。

つまり、プライマリーコンストラクタを宣言していないクラスでも、セカンダリーコンストラクタを宣言することができる、ということです。

プライマリーコンストラクタ宣言を持たないクラス宣言の中にセカンダリーコンストラクタ宣言を書く場合、コロンと、プライマリーコンストラクタを呼び出す式は、書く必要がありません。

プライマリーコンストラクタ宣言を持たないクラス宣言の中に、init で始まるイニシャライザーブロックと、セカンダリーコンストラクタのイニシャライザーブロックの両方が書かれていた場合は、まず init で始まるイニシャライザーブロックが実行されて、そのあとでセカンダリーコンストラクタのイニシャライザーブロックが実行されます。

次のプログラムの中で宣言されている Seblock3 というクラスは、プライマリーコンストラクタを持っていませんが、セカンダリーコンストラクタは持っています。

プログラムの例　Seblock3.kt

```
class Seblock3 {
    init {
        println("コンストラクタが呼び出されました。")
    }
    constructor(n: Int) {
        println("引数は${n}です。")
    }
}
```

実行例

```
>>> val a = Seblock3(421)
コンストラクタが呼び出されました。
引数は 421 です。
```

7.6 サブクラス

7.6.1 スーパークラスとサブクラス

第 7.1.1 項で説明したように、クラスというのはオブジェクトを生成する鋳型のようなものです。しかし、鋳型のようなものというのは、クラスというものの側面のひとつにすぎません。クラスには、もうひとつの別の側面があります。それは、「分類体系の部品」という側面です。

さまざまなクラスの全体は、生物などの分類体系と同じように、木の形をした構造を持っています。この木は、本物の木とは上と下とが逆になっていて、いちばん上に根があって、下に向かって枝が伸びています。木を構成するそれぞれのクラスは、上にあるものほど一般的で、下にあるものほど特殊です。

A と B という二つのクラスがあって、それらのあいだに、A は B よりも一般的なクラス、つまり B は A よりも特殊なクラス、という関係があるとき、A は、B の「スーパークラス」(superclass) と呼ばれ、B は、A の「サブクラス」(subclass) と呼ばれます。

7.6.2 継承

Kotlin では、すでに存在するクラスを特殊化することによって新しいクラスを宣言する、ということができます。特殊化したクラスを宣言するというのは、プロパティーやメソッドを追加することによって、用途を絞ったクラスを宣言するということです。

すでに存在するクラスを特殊化することによって新しいクラスを宣言した場合、つまりスーパークラスからサブクラスを宣言した場合、スーパークラスで宣言されているプロパティーやメソッドは、サブクラスに自動的に受け継がれます。この機能は、「継承」(inheritance) と呼ばれます。そして、継承によってプロパティーやメソッドを受け継ぐことを、プロパティーやメソッドを「継承する」(inherit) と言います。

継承という機能があるおかげで、スーパークラスを特殊化したサブクラスを宣言するとき、スーパークラスで宣言されているプロパティーやメソッドの宣言を繰り返す必要はありません。宣言する必要があるのは、サブクラスに追加するプロパティーやメソッドの宣言だけです。

7.6.3 スーパークラスになることができるクラスの宣言

Kotlin では、どんなクラスについても、そのクラスからサブクラスを宣言することができる、というわけではありません。

スーパークラスになることができるクラスを宣言するためには、その宣言の先頭に、`open` と書いておく必要があります。たとえば、

```
class Namako
```

というクラス宣言で宣言された `Namako` というクラスは、スーパークラスになることができませんが、

```
open class Sakana
```

というクラス宣言で宣言された `Sakana` というクラスは、スーパークラスになることができます。

7.6.4 サブクラスの宣言

すでに存在するクラスを特殊化した新しいクラスを宣言したいとき、つまり、サブクラスを宣言したいときは、基本的には、

`class` 識別子 `:` コンストラクタ呼び出し

という形のクラス宣言を書きます。この中の「識別子」のところには、サブクラスに名前として与える識別子を書きます。そして「コンストラクタ呼び出し」のところには、スーパークラスにしたいクラスのコンストラクタを呼び出す式を書きます。たとえば、`Sakana` というクラスが、

```
open class Sakana
```

と宣言されているとすると、

```
class Maguro : Sakana()
```

というクラス宣言を書くことによって、`Sakana` をスーパークラスとする `Maguro` というサブクラスを宣言することができます。

```
>>> open class Sakana
>>> class Maguro : Sakana()
>>> Maguro()
Line_1$Maguro@981784
```

7.6.5 サブクラスのプライマリーコンストラクタ

サブクラスのプライマリーコンストラクタ宣言の中には、2 種類のものを書くことができます。仮引数の宣言とプロパティー宣言です。引数を受け取りたいだけの場合は、仮引数の宣言を書きます。そして、スーパークラスから継承したものではない、サブクラスの独自のプロパティーを宣言したい場合は、プロパティー宣言を書きます。

次のプログラムの中で宣言されている Student というクラスは、 Human というクラスのサブクラスとして宣言されています。

プログラムの例 student.kt

```
open class Human(val age: Int)

class Student(a: Int, val grade: Int) : Human(a)
```

Human のクラス宣言は、age というプロパティーを宣言するプライマリーコンストラクタ宣言を持っています。この age というプロパティーは、 Human のサブクラスとして宣言された Student に継承されます。

Student のクラス宣言は、 a という仮引数と、 grade というプロパティーを宣言するプライマリーコンストラクタ宣言を持っています。

Student のプライマリーコンストラクタは、 a という仮引数に受け取った整数を、 Human のプライマリーコンストラクタに引数として渡します。 Human のプライマリーコンストラクタは、 Student のオブジェクトが持っている age のバッキングフィールドを初期化します。

実行例

```
>>> val s = Student(18, 3)
val s = Student(18, 3)>>>
>>> s.age
18
>>> s.grade
3
```

7.6.6 サブクラスのメソッド

サブクラスは、スーパークラスから継承したメソッドに加えて、独自のメソッドを持つことができます。

次のプログラムの中で宣言されている Student というクラスは、 Human というクラスのサブクラスとして宣言されています。

プログラムの例 student2.kt

```
open class Human {
    fun eat() {
        println("食事をしました。")
    }
}

class Student : Human() {
    fun study() {
        println("勉強をしました。")
    }
}
```

Human のクラス宣言は、 eat というメソッドを宣言するメソッド宣言を持っています。この eat というメソッドは、 Human のサブクラスとして宣言された Student に継承されます。

Student のクラス宣言は、 study というメソッドを宣言するメソッド宣言を持っています。これは、 Student というクラスの独自のメソッドです。

実行例

```
>>> val s = Student()
>>> s.eat()
食事をしました。
>>> s.study()
勉強をしました。
```

7.6.7 メソッドのオーバーライド

新しいクラスを宣言するとき、新しいメソッドを追加するだけではなくて、スーパークラスから継承したメソッドの動作を変更したい、ということがしばしばあります。スーパークラスから継承したメソッドの動作をサブクラスで変更したいときは、スーパークラスから継承したメソッドと同じ名前のメソッドをサブクラスで宣言します。このことを、スーパークラスから継承したメソッドを「オーバーライドする」(override) と言います。

Kotlin では、どんなメソッドについても、それをオーバーライドすることができる、というわけではありません。

オーバーライドすることができるメソッドを宣言するためには、その宣言の先頭に、 open と書いておく必要があります。たとえば、

```
fun namako() { ··· }
```

というメソッド宣言で宣言された namako というメソッドは、オーバーライドすることができませんが、

```
open fun umiushi() { ··· }
```

というメソッド宣言で宣言された umiushi というメソッドは、オーバーライドすることができます。

スーパークラスから継承したメソッドをオーバーライドしたいときは、サブクラスの宣言の中に、継承したメソッドと同じ名前のメソッドを宣言するメソッド宣言を書けばいいわけですが、ただし、その先頭に、 override と書く必要があります。

次のプログラムの中で宣言されている Student というクラスは、 Human というクラスのサブクラスとして宣言されています。

プログラムの例 student3.kt

```
open class Human {
    open fun what() {
        println("私は人間です。")
    }
}

class Student : Human() {
    override fun what() {
        println("私は学生です。")
    }
}
```

Human のクラス宣言は、 what というメソッドを宣言するメソッド宣言を持っています。このメソッド宣言は、先頭に open と書かれていますので、このメソッドはサブクラスでオーバーライドすることができます。

Student のクラス宣言は、 Human から継承した what というメソッドをオーバーライドするメソッド宣言を持っています。

実行例

```
>>> val h = Human()
>>> val s = Student()
>>> h.what()
私は人間です。
>>> s.what()
私は学生です。
```

7.6.8 プロパティーのオーバーライド

メソッドだけではなくて、プロパティーも、オーバーライドすることができます。

メソッドをオーバーライドする場合と同じように、プロパティーをオーバーライドする場合も、オーバーライドされるプロパティーの宣言には、先頭に open と書く必要があります。そして、オーバーライドするプロパティーの宣言には、先頭に override と書く必要があります。

次のプログラムの中で宣言されている Student というクラスは、Human というクラスのサブクラスとして宣言されています。

プログラムの例 student4.kt

```
open class Human(val name: String) {
    open val annotatedName get() = "${name} (human)"
}

class Student(n: String) : Human(n) {
    override val annotatedName get() = "${name} (student)"
}
```

Human のクラス宣言は、annotatedName というプロパティーを宣言するプロパティー宣言を持っています。このプロパティー宣言は、先頭に open と書かれていますので、このプロパティーはサブクラスでオーバーライドすることができます。

Student のクラス宣言は、Human から継承した annotatedName というプロパティーをオーバーライドするプロパティー宣言を持っています。

実行例

```
>>> val h = Human("Yamamoto")
>>> val s = Student("Tanaka")
>>> h.annotatedName
Yamamoto (human)
>>> s.annotatedName
Tanaka (student)
```

7.6.9 オーバーライドによって隠されたメンバー

オーバーライドされたスーパークラスのメンバーは、サブクラスのメンバーによって隠されることになりますが、サブクラスの宣言の中でそれを扱うことは可能です。

オーバーライドされて隠されているスーパークラスのメンバーをサブクラスの宣言の中で扱いたいときは、

```
super.メンバー名
```

という形のものを書きます。たとえば、

```
super.namako()
```

という式を書くことによって、オーバーライドによって隠されている、スーパークラスから継承した namako を呼び出すことができます。

次のプログラムの中で宣言されている Student というクラスは、Human というクラスのサブクラスとして宣言されています。

プログラムの例 student5.kt

```
open class Human {
    open fun what() {
        println("私は人間です。")
    }
}

class Student : Human() {
    override fun what() {
        super.what()
        println("私は学生です。")
    }
}
```

Student は、Human から継承した what というメソッドをオーバーライドしています。オーバーライドしている what は、オーバーライドされている what を呼び出したのちに、「私は学生です。」と出力します。

実行例

```
>>> val h = Human()
>>> val s = Student()
>>> h.what()
私は人間です。
>>> s.what()
私は人間です。
私は学生です。
```

7.6.10 Any

Kotlin では、スーパークラスを明示しないでクラスを宣言したとしても、そのクラスは暗黙のうちにスーパークラスを持つことになります。

スーパークラスを明示しないでクラスを宣言した場合に暗黙のうちにスーパークラスとして指定されるのは、Any という名前を持つクラスです。

いかなるクラスも、そのスーパークラスをたどっていくと、必ず Any に到達します。

Any では、toString、equals、hashCode という三つのメソッドが宣言されています。あらゆるクラスはこれらのメソッドを継承していて、必要に応じてオーバーライドすることができます。

7.6.11 オブジェクトを文字列へ変換するメソッド

Any で宣言されている toString というメソッドは、オブジェクトを文字列に変換する必要が起きたときに暗黙のうちに呼び出されて、そのオブジェクトをあらわす文字列を戻り値として返します。

Any で宣言されている toString は、オブジェクトを、

```
Human@1b5b4b0
```

というような意味不明な文字列に変換します。しかし、このメソッドをサブクラスでオーバーライドすることによって、人間にとって分かりやすい文字列に変換されるように、その機能を変更することができます。

次のプログラムの中で宣言されている Human というクラスは、Any から継承した toString をオーバーライドしています。

プログラムの例 human.kt

```
open class Human {
    override fun toString(): String = "私は人間です。"
}
```

Human のオブジェクトは、文字列に変換する必要が起きると、「私は人間です。」という文字列に変換されます。

実行例

```
>>> Human()
私は人間です。
```

次のプログラムは、第 7.2.5 項で紹介した有理数のクラス宣言に、toString をオーバーライドするメソッド宣言を追加したものです。

プログラムの例 rational4.kt

```
class Rational(val numerator: Int, val denominator: Int) {
    override fun toString(): String =
        "${numerator}/${denominator}"
}
```

Rational のオブジェクトは、文字列に変換する必要が起きると、「分子/分母」という形の文字列に変換されます。

実行例

```
>>> Rational(17, 30)
17/30
```

7.6.12　スーパータイプとサブタイプ

クラスから生成されたオブジェクトは、そのクラスの名前によって識別される型を持っています。

クラスとクラスとのあいだにスーパークラスとサブクラスという関係がある場合は、その関係が、それらのクラスの名前によって識別される型にも反映されます。

スーパークラスのオブジェクトが持つ型は、サブクラスのオブジェクトが持つ型の「スーパータイプ」(supertype) と呼ばれ、サブクラスのオブジェクトが持つ型は、スーパークラスのオブジェクトが持つ型の「サブタイプ」(subtype) と呼ばれます。

オブジェクトは、それが持っている型のスーパータイプのオブジェクトとして扱うことができます。たとえば、スーパータイプを持つ変数には、そのサブタイプのオブジェクトを代入することもできます。また、スーパータイプのオブジェクトを返す関数は、そのサブタイプのオブジェクトを返すこともできます。また、スーパータイプのオブジェクトのリストには、そのサブタイプのオブジェクトを要素として混在させることができます。

次のプログラムの中で宣言されている Student というクラスは、Human というクラスのサブクラスとして宣言されています。そして、couple という関数は、Human という型のオブジェクトを二つ受け取って、それらから構成されるリストを返します。

プログラムの例　student6.kt

```
open class Human {
    override fun toString(): String = "人間"
}

class Student : Human() {
    override fun toString(): String = "学生"
}

fun couple(a: Human, b: Human): List<Human> = listOf(a, b)
```

実行例

```
>>> couple(Human(), Student())
[人間, 学生]
```

この実行例から分かるように、Student は Human のサブタイプですので、couple は、引数として Student のオブジェクトを受け取って、そのオブジェクトを要素とするリストを返すことができます。

7.7　抽象クラス

7.7.1　ポリモーフィズム

次のプログラムは、Fish というクラスを宣言しています。そして、Fish のサブクラスとして、Tuna、Eel、Shark という三つのクラスを宣言しています。そして、それらのサブクラスは、Fish で宣言されている what というメソッドをオーバーライドしています。

プログラムの例　fish.kt

```
open class Fish {
    open fun what(): String = "魚"
}

class Tuna : Fish() {
    override fun what(): String = "マグロ"
}

class Eel : Fish() {
    override fun what(): String = "ウナギ"
}

class Shark : Fish() {
    override fun what(): String = "サメ"
}
```

```
fun printFish() {
    val list = listOf(Fish(), Tuna(), Eel(), Shark())
    for (f in list) {
        println(f.what())
    }
}
```

それでは、このプログラムの中で宣言されている printFish という関数を呼び出してみましょう。

```
>>> printFish()
魚
マグロ
ウナギ
サメ
```

この実行結果から分かるように、for 文の中に書かれている、f.what() というメソッド呼び出しによって呼び出されるメソッドは、f に代入されているオブジェクトのクラスに応じて決定されています。

このような、プログラムの中に書かれているメソッド呼び出しが同じであるにもかかわらず、オブジェクトのクラスに応じたメソッドが呼び出されるという性質は、「ポリモーフィズム」(polymorphism) または「多態性」と呼ばれます。

7.7.2 抽象メソッド

ポリモーフィズムが成り立つためには、スーパークラスで宣言されているメソッドをサブクラスでオーバーライドするということが必要です。しかし、場合によっては、スーパークラスでは具体的な動作を記述することができないメソッドをポリモーフィズムの対象にしたい、ということがあります。

例として、さまざまな多角形のクラスでポリモーフィズムが成り立つようにする、ということについて考えてみましょう。三角形や四角形や五角形などの具体的な多角形のクラスでは、辺の本数を返す side というメソッドの具体的な動作を記述することができます。たとえば、三角形のクラスでは、

```
override fun side(): Int = 3
```

と宣言すればいいわけです。しかし、具体的な多角形のクラスのスーパークラスとなる一般的な多角形のクラスでは、side の具体的な動作は、記述することができません。

ポリモーフィズムを性質として持っているプログラミング言語には、具体的な動作が記述されていないメソッドを宣言することができるという機能があります。具体的な動作が記述されていないメソッドは、「抽象メソッド」(abstract method) と呼ばれます。

Kotlin では、抽象メソッドのメソッド宣言は、

```
abstract fun 識別子(仮引数の宣言, ···): 型名
```

と書きます。先頭に abstract と書くということと、本体は書かないということが、普通のメソッド宣言との違いです。たとえば、

```
abstract fun side(): Int
```

というメソッド宣言を書くことによって、side というメソッドを抽象メソッドとして宣言することができます。

抽象メソッドは、サブクラスでオーバーライドされることを前提としたメソッドですので、open と書かなくても、オーバーライドすることができます。

7.7.3 抽象クラスとは何か

抽象メソッドの宣言は、どんなクラス宣言の中にでも書くことができるというわけではありません。それを書くことができるのは、特殊なクラスの宣言の中だけです。

抽象メソッドの宣言をその宣言の中に書くことができる特殊なクラスは、「抽象クラス」(abstract class) と呼ばれます。

クラスを抽象クラスとして宣言すると、そのクラスからはオブジェクトを生成することができなくなります。

7.7.4 抽象クラスの宣言

クラスを宣言するときに、class の左側に abstract と書くと、そのクラスは抽象クラスになります。

抽象クラスは、それのサブクラスが宣言されることを前提としたクラスですので、open と書かなくても、スーパークラスになることができます。

次のプログラムは、Polygon というクラスを宣言しています。そして、Polygon のサブクラスとして、Triangle、Quadrangle、Pentagon という三つのクラスを宣言しています。

プログラムの例 polygon.kt

```
abstract class Polygon {
    abstract fun side(): Int
}

class Triangle : Polygon() {
    override fun side(): Int = 3
}

class Quadrangle : Polygon() {
    override fun side(): Int = 4
}

class Pentagon : Polygon() {
    override fun side(): Int = 5
}

fun printPolygon() {
    val list = listOf(Triangle(), Quadrangle(), Pentagon())
    for (p in list) {
        println(p.side())
    }
}
```

Polygon は、抽象クラスとして宣言されています。その理由は、その宣言の中で、side というメソッドを抽象メソッドとして宣言する必要があるからです。

Polygon のサブクラスとして宣言されている、Triangle、Quadrangle、Pentagon という三つのクラスのそれぞれは、Polygon の side をオーバーライドしています。ですから、side の呼び出しでは、ポリモーフィズムが成り立ちます。

printPolygon という関数の宣言の中に書かれている p.side() というメソッド呼び出しは、p に代入されているオブジェクトのクラスに応じたメソッドを呼び出します。

```
>>> printPolygon()
3
4
5
```

第 7.7.3 項で述べたように、抽象クラスからはオブジェクトを生成することができません。先ほどのプログラムの中で宣言されている Polygon というクラスは抽象クラスですから、それからオブジェクトを生成しようとすると、次のようなエラーメッセージが出力されます。

```
>>> Polygon()
error: cannot create an instance of an abstract class
Polygon()
^
```

7.8 インターフェース

7.8.1 多重継承

第 7.6.1 項で説明したように、クラスには、「分類体系の部品」という側面があります。

分類体系というのは、常にひとつだけとは限りません。たとえば、人間は、ミミズやトンボと同じように、「動物」というカテゴリーに分類することができますが、それだけではなくて、自動車や自転車と同じように、「走るもの」というカテゴリーにも分類することができます。

人間のクラスを宣言するときに、動物のクラスと走るもののクラスの二つをスーパークラスとして指定することができれば、動物としての人間と、走るものとしての人間の両方について、ポリモーフィズムが成り立つことになります。

プログラミング言語のうちには、2 個以上のスーパークラスを指定して、それらのサブクラスを宣言することができる、という機能を持っているものもあります。そのようなプログラミング言語で、2 個以上のスーパークラスを指定してサブクラスを宣言した場合、そのサブクラスは、それらのスーパークラスからメンバーを継承することになります。そのような、2 個以上のスーパークラスからメンバーを継承するという機能は、「多重継承」(multiple inheritance) と呼ばれます。

プログラミング言語の多くは、多重継承の機能を持っていません。Kotlin も、多重継承の機能を持っていないプログラミング言語のひとつです。

7.8.2 インターフェースとは何か

Kotlin は多重継承の機能を持っていないわけですが、しかし、それを代替する機能を持っています。その機能を使えば、共通するスーパークラスを持たない複数のクラスに関しても、ポリモーフィズムを成り立たせることができます。

Kotlin では、共通するスーパークラスを持たない複数のクラスに関してポリモーフィズムが成り立つようにしたい、というときには、「インターフェース」(interface) と呼ばれるものを使います。

インターフェースは、見かけ上は、抽象メソッドだけから構成される抽象クラスのように見えます。しかし、インターフェースはクラスではありませんので、インターフェースのサブクラスを宣言するということはできません。

インターフェースの中で宣言されている抽象メソッドは、クラス宣言の中でオーバーライドすることができます。複数のクラスが、共通するインターフェースの抽象メソッドをオーバーライドすれば、それらのメソッドについて、ポリモーフィズムが成り立つことになります。

ただし、インターフェースの中で宣言されている抽象メソッドをオーバーライドするクラスを宣言する場合は、そのインターフェースの中で宣言されているすべての抽象メソッドをオーバーライドする必要があります。

7.8.3 インターフェース宣言

インターフェースは、「インターフェース宣言」(interface declaration) と呼ばれる記述を書くことによって宣言することができます。

インターフェース宣言は、

```
interface 識別子 {
    抽象メソッドの宣言
        ●
        ●
        ●
}
```

と書きます。この中の「識別子」のところには、インターフェースに名前として与える識別子を書きます。

インターフェース宣言の中には抽象メソッドの宣言を書くわけですが、抽象クラスの場合とは違って、`abstract` と書く必要はありません。

たとえば、`run` という抽象メソッドを持つ `Runner` というインターフェースを宣言したいときは、次のようなインターフェース宣言を書きます。

```
interface Runner {
    fun run(): String
}
```

7.8.4 インターフェースの実装

インターフェースの中で宣言されているすべての抽象メソッドをオーバーライドすることを、インターフェースを「実装する」(implement) と言います。

インターフェースを実装するクラスを宣言する場合には、クラス名になる識別子の右側にコロンを書いて、その右側に、実装する対象となるインターフェースの名前を書く必要があります。

次のプログラムは、Runner というインターフェースを宣言しています。そして、Runner を実装する、Human、Bicycle、Car という三つのクラスを宣言しています。

プログラムの例 runner.kt

```
interface Runner {
    fun run(): String
}

class Human : Runner {
    override fun run(): String = "走る（人間）"
}

class Bicycle : Runner {
    override fun run(): String = "走る（自転車）"
}

class Car : Runner {
    override fun run(): String = "走る（自動車）"
}

fun printRunner() {
    val list = listOf(Human(), Bicycle(), Car())
    for (r in list) {
        println(r.run())
    }
}
```

ポリモーフィズムが成り立っていますので、printRunner という関数の宣言の中に書かれている r.run() というメソッド呼び出しは、r に代入されているオブジェクトのクラスに応じたメソッドを呼び出します。

実行例

```
>>> printRunner()
走る（人間）
走る（自転車）
走る（自動車）
```

1 個のクラスを宣言するとき、スーパークラスとして指定することができるクラスは 1 個だけですが、実装することができるインターフェースは 1 個だけではありません。ですから、スーパークラスとなるクラスを宣言する代わりにインターフェースを宣言することによって、多重継承によって実現できることとほぼ同じことを実現することができます。

コロンの右側に、コンストラクタを呼び出す式とインターフェースの名前を書きたいとき、あるいは 2 個以上のインターフェースの名前を書きたいときは、それらをコンマで区切って並べます。

たとえば、A というクラスをスーパークラスとして指定して、B、C、D という三つのインターフェースを実装する、E というクラスを宣言するクラス宣言の先頭部分は、

```
class E : A(), B, C, D
```

というように書きます。

7.9 拡張

7.9.1 拡張の基礎

クラスにメンバーを追加したい、というときは、そのクラスをスーパークラスとするサブクラスを宣言すればいいわけですが、しかし、Kotlin では、スーパークラスにすることができるのは、その宣言に open と書かれているクラスだけです。たとえば、String というクラスは、スーパークラスにすることができるクラスではありませんので、それのサブクラスを宣言するということはできません。

それでは、スーパークラスにすることができないクラスには、メンバーを追加することはできないのでしょうか。

Kotlin には、関数またはプロパティーを型に追加することができるという機能があって、この

機能は「拡張」(extension) と呼ばれます。拡張を使えば、スーパークラスにすることができないクラスについても、それに関数またはプロパティーを追加することができます。さらに、拡張を使うことによって、`List<Int>` のような、クラスではない型に関数またはプロパティーを追加するということもできます。

拡張を使って型に追加された関数は「拡張関数」(extension function) と呼ばれ、拡張を使って型に追加されたプロパティーは「拡張プロパティー」(extension property) と呼ばれます。

拡張を使うことによって、プロパティーを型に追加することができるわけですが、追加することができるプロパティーは、バッキングフィールドを持たないものだけです。

7.9.2 拡張関数の宣言

拡張関数の宣言の書き方は、普通の関数の宣言の書き方とほとんど同じです。普通の関数の宣言は、

`fun` 識別子 `(` ･･･

と書くわけですが、拡張関数の場合は、この部分が、

`fun` 型名 `.` 識別子 `(` ･･･

となります。この中の「型名」のところには、拡張関数を追加したい型を識別する型名を書きます。たとえば、

```
fun String.namako( ･･･
```

という宣言を書くことによって、`namako` という拡張関数を `String` に追加することができます。同じように、

```
fun List<Int>.umiushi( ･･･
```

という宣言を書くことによって、`umiushi` という拡張関数を `List<Int>` に追加することができます。

7.9.3 拡張関数を呼び出す式

拡張関数を呼び出す式の書き方は、メソッド呼び出しと同じです。つまり、

式$_1$ `.` 関数名 `(` 式$_2$ `,` ･･･`)`

と書けばいいわけです。この中の「式$_1$」のところには、拡張の対象となった型を持つオブジェクトを求める式を書きます。たとえば、

```
"hoge".namako(7)
```

という式を書くことによって、`String` に追加された `namako` という拡張関数を呼び出して、引数として 7 を渡すことができます。

7.9.4 レシーバー

メソッドまたは拡張関数を呼び出す、

式$_1$ `.` 関数名 `(` 式$_2$ `,` ･･･`)`

という形の式を評価したときに、式$_1$ の値として得られたオブジェクトは、呼び出されたメソッドまたは拡張関数の「レシーバー」(receiver) と呼ばれます。

メソッドまたは拡張関数の宣言の中では、レシーバーは、`this` という式を評価することによって求めることができます。

次のプログラムの中で宣言されている、`String` に追加された `countChar` という拡張関数は、文字を受け取って、レシーバーの中に含まれているその文字の個数を返します。

プログラムの例 `countchar.kt`

```
fun String.countChar(c: Char): Int {
    var n = 0
    for (cc in this) {
        if (cc == c) n++
    }
```

```
    return n
}
```

実行例

```
>>> "namamuginamagomenamatamago".countChar('m')
6
```

次のプログラムの中で宣言されている、List<Int> に追加された sublist という拡張関数は、整数 i と整数 j を受け取って、レシーバーの i 番目から $(j-1)$ 番目までの要素から構成されるリストを返します。

プログラムの例 sublist.kt

```
fun List<Int>.sublist(i: Int, j: Int): List<Int> {
    var sub: MutableList<Int> = mutableListOf()
    for (n in i..(j - 1)) {
        sub.add(this[n])
    }
    return sub.toList()
}
```

実行例

```
>>> listOf(30, 31, 32, 33, 34, 35, 36, 37, 38).sublist(2, 7)
[32, 33, 34, 35, 36]
```

練習問題 7.6 n と r を整数とするとき、n.countFigure(r) という式で呼び出すと、n を r 進法で表記した文字列の長さを返す拡張関数を宣言してください。

実行例

```
>>> 365.toString(2)
101101101
>>> 365.countFigure(2)
9
```

練習問題 7.7 x を整数の変更不可能なセット、s を文字列とするとき、x.toMap(s) という式で呼び出すと、x のそれぞれの要素をキー、s を値とする要素から構成される変更不可能なマップを返す拡張関数を宣言してください。

実行例

```
>>> setOf(48, 27, 64, 58, 39).toMap("hoge")
{48=hoge, 27=hoge, 64=hoge, 58=hoge, 39=hoge}
```

7.9.5 拡張プロパティー

拡張プロパティーの宣言の書き方は、拡張関数の宣言の書き方と同じように、識別子の左側に型名とドットを書くという点を除いて、普通のプロパティーの宣言と同じです。ただし、バッキングフィールドを持つ拡張プロパティーを宣言することはできませんので、カスタムゲッターの宣言が常に必要です。

拡張プロパティーが取り出したオブジェクトを求める式は、普通のプロパティーの場合と同じように、

式 . プロパティー名

と書きます。この中の「式」のところには、拡張の対象となった型を持つオブジェクトを求める式を書きます。その式の値は、拡張関数やメソッドの場合と同じように、「レシーバー」と呼ばれます。カスタムゲッターの宣言の中では、this という式を評価することによってレシーバーを求めることができます。

次のプログラムの中で宣言されている、String に追加された countSpace という拡張プロパティーは、レシーバーの中に含まれている空白の個数を取り出します。

プログラムの例 countspace.kt

```
val String.countSpace: Int
```

```
    get() {
        var n = 0
        for (c in this) {
            if (c == ' ') n++
        }
        return n
    }
```

実行例

```
>>> "a b c d e f g h".countSpace
7
```

次のプログラムの中で宣言されている、List<Int>に追加されたduplicateという拡張プロパティーは、レシーバーの中に重複した要素が存在するならば真、そうでなければ偽を取り出します。

プログラムの例 duplicate.kt

```
val List<Int>.duplicate: Boolean
    get() = this.size > this.toSet().size
```

実行例

```
>>> listOf(3, 8, 7, 6, 2, 8, 4).duplicate
true
>>> listOf(3, 8, 7, 6, 2, 5, 4).duplicate
false
```

練習問題 7.8 bを真偽値とするとき、b.intという式を評価すると、bが真ならば1、偽ならば0を取り出す拡張プロパティーを宣言してください。

実行例

```
>>> true.int
1
>>> false.int
0
```

練習問題 7.9 xを真偽値の変更不可能なリストとするとき、x.andという式を評価すると、xを構成しているすべての要素が真ならば真、そうでないならば偽を取り出す拡張プロパティーを宣言してください。

実行例

```
>>> listOf(true, true, true, true, true).and
true
>>> listOf(true, true, true, false, true).and
false
>>> listOf(false, false, false, false, false).and
false
```

7.10 メンバーの可視性

7.10.1 可視性の基礎

プログラムの中で宣言されているものが持っている、別の場所からそれにアクセスすることができるかどうかという性質、つまり別の場所でそれを使うことができるかどうかという性質は、「可視性」(visibility) と呼ばれます。

可視性は、「可視性修飾子」(visibility modifier) と呼ばれる単語を宣言の先頭に書くことによって、指定することができます。

クラスのメンバーは、クラス宣言の外からそれにアクセスすることができるかどうかという可視性を、可視性修飾子を書くことによって指定することができます。

7.10.2 public

public という可視性修飾子は、「公開されている」、つまり「どこからでもアクセスすることができる」という可視性を指定します。可視性のデフォルトは、この public です。可視性修飾子を書かなかった場合は、 public が指定されたと解釈されます。

7.10.3 private

private という可視性修飾子は、最も強く可視性を制限します。

クラスのメンバーを宣言するときに、その先頭に private という可視性修飾子を書くと、そのメンバーは、そのクラスの宣言の中からはアクセスすることができますが、そのクラスの宣言の外からはアクセスすることができなくなります。

次のプログラムは、 Human というクラスを宣言しています。

プログラムの例 private.kt

```
class Human {
    val family = "田中"
    private val first = "景子"

    fun niceToMeetYou() {
        println("はじめまして。")
        sayName()
    }

    private fun sayName() {
        println("私は${family}${first}です。")
    }
}
```

Human の宣言の中には、 family 、 first 、 niceToMeetYou 、 sayName という四つのメンバーの宣言があります。これらのうち、 first と sayName という二つのメンバーの宣言は、それらの先頭に private という可視性修飾子が書かれています。ですから、それらの二つのメンバーは、クラス宣言の外からはアクセスすることができません。

実行例

```
>>> val h = Human()
>>> h.family
田中
>>> h.first
error: cannot access 'first': it is private in 'Human'
h.first
  ^
>>> h.niceToMeetYou()
はじめまして。
私は田中景子です。
>>> h.sayName()
error: cannot access 'sayName': it is private in 'Human'
h.sayName()
  ^
```

プログラムというのは、必要に応じて修正が加えられる文書です。したがって、プログラムを書くときには、修正が加わったことによって不具合が発生する危険性を最小限に抑えるということに注意を払う必要があります。

クラスを宣言する場合には、外部に公開するメンバーを必要最小限に留めておくということが重要です。なぜなら、外部に公開する必要がないメンバーまで外部に公開することは、クラス宣言の内部を修正したときに不具合が発生する危険性を増大させるからです。

したがって、クラス宣言の内部だけで使われるメンバーの宣言には private という可視性修飾子を書いておくということは、きわめて重要な原則です。

7.10.4 protected

クラスの宣言の中で private という可視性修飾子を付けて宣言されたメンバーは、そのクラスの宣言の外からはアクセスすることができません。そのクラスのサブクラスの宣言も、そのクラスの外にあるわけですから、アクセスすることができないのは、サブクラスの宣言の中も同じです。

次のプログラムは、Human というクラスと、それをスーパークラスとする Student というクラスを宣言しています。

プログラムの例 private2.kt

```
open class Human {
    private fun sayName() {
        println("私は田中景子です。")
    }
}

class Student : Human() {
    fun niceToMeetYou() {
        println("はじめまして。")
        sayName()
        println("私は学生です。")
    }
}
```

このプログラムを REPL にロードさせると、エラーメッセージが出力されます。その理由は、スーパークラスの宣言の中で、private という可視性修飾子を付けて宣言されているメソッドに、サブクラスの宣言の中からアクセスしようとしているからです。

クラスのメンバーを宣言するときに、その先頭に protected という可視性修飾子を書くと、そのメンバーは、そのクラスの宣言の中と、そのクラスのサブクラスの宣言の中からはアクセスすることができますが、それ以外の場所からはアクセスすることができなくなります。ですから、「このメンバーは、サブクラスの宣言の中からはアクセスできないと困るけれども、全面的に公開することは避けたい」というときには、protected が使われることになります。

次のプログラムは、先ほどのプログラムの中に書かれていた private を、protected に書き換えたものです。

プログラムの例 protected.kt

```
open class Human {
    protected fun sayName() {
        println("私は田中景子です。")
    }
}

class Student : Human() {
    fun niceToMeetYou() {
        println("はじめまして。")
        sayName()
        println("私は学生です。")
    }
}
```

実行例

```
>>> val s = Student()
>>> s.niceToMeetYou()
はじめまして。
私は田中景子です。
私は学生です。
>>> s.sayName()
error: cannot access 'sayName': it is protected in 'Student'
s.sayName()
  ^
```

7.11 データクラス

7.11.1 オブジェクトの等価性

第 7.6.10 項で説明したように、あらゆるクラスは、Any というクラスで宣言されているメソッドを継承しています。

Any で宣言されている equals というメソッドは、1 個のオブジェクトを受け取って、レシー

バーとそれとが等しいならば真、そうでなければ偽を返します。このメソッドは、 == を使ってオブジェクトとオブジェクトとを比較したときに、それらが等しいかどうかを判定するために呼び出されます。

equals は、もしもそれをオーバーライドしていないならば、ただ単に、比較の対象となっているオブジェクトが同一のものかどうかを調べて、同一のものならば、それらは等しいと判定します。

```
>>> class Namako(val n: Int)
>>> val a = Namako(30)
>>> val b = Namako(30)
>>> a == a
true
>>> a == b
false
```

Namako のオブジェクトが二つあって、どちらのオブジェクトも n というプロパティーに代入されている整数が等しいとするとき、それらを == で比較した場合に、それらのオブジェクトは等しいと判定してほしいならば、 equals をオーバーライドする必要があります。

7.11.2 ハッシュコード

Any で宣言されている hashCode というメソッドは、「ハッシュコード」(hash code) と呼ばれる整数を返します。

ハッシュコードは、セットやマップなどで使われる、二つのオブジェクトが等しいかどうかを判定するための整数です。セットやマップなどでは、オブジェクトのハッシュコードが等しいかどうかということによって、要素やキーが等しいかどうかを判定します。ですから、 equals をオーバーライドした場合は、 hashCode も適切にオーバーライドする必要があります。

7.11.3 データクラスとは何か

Any から継承した toString 、 equals 、 hashCode というメソッドをオーバーライドするためには、普通のクラスの場合、明示的にメソッドの宣言を書く必要があります。

それに対して、「データクラス」(data class) と呼ばれるクラスの場合は、明示的にメソッドの宣言を書かなくても、 Any から継承したメソッドが自動的にオーバーライドされます。

データクラスの equals は、比較の対象となった二つのオブジェクトについて、対応するすべてのプロパティーを比較して、それらが等しいならば真、そうでなければ偽を返します。同じように、データクラスのオブジェクトは、対応するすべてのプロパティーが等しいならば、どのオブジェクトの hashCode も、等しいハッシュコードを返します。

このように、データクラスは、 Any から継承したメソッドを自動的にオーバーライドしますので、データを保持することを目的とするオブジェクトを生成するクラスとして、かなり便利です。

ただし、普通のクラスとは違って、データクラスは、それをスーパークラスとするサブクラスを宣言することができません。

7.11.4 データクラスの宣言

クラス宣言を書くときに、 class の左側に data と書くと、そのクラスはデータクラスになります。

次のプログラムは、 User というデータクラスを宣言しています。

プログラムの例 user.kt

```
data class User(val id: Int, val name: String)
```

User はデータクラスですので、このクラスの二つのオブジェクトを == で比較した場合、それらが同一のオブジェクトではなかったとしても、 id と name が等しいならば、それらのオブジェクトは等しいと判定されます。

実行例

```
>>> val a = User(73145, "Hikari")
>>> val b = User(73145, "Hikari")
>>> a == b
true
```

また、User では、toString も、人間にとって意味が分かりやすい文字列に変換するメソッドによって自動的にオーバーライドされています。

実行例

```
>>> User(62089, "Nozomi")
User(id=62089, name=Nozomi)
```

7.12 型のチェック

7.12.1 型が Any の変数

第 7.6.10 項で説明したように、いかなるクラスも、そのスーパークラスをたどっていくと、必ず Any というクラスに到達します。このことは、型が Any の変数にはどんなオブジェクトでも代入することができる、ということを意味しています。

```
>>> var a: Any = 35
>>> a = "namako"
>>> a = true
>>> a = listOf(47, 22, 16, 83)
```

型が Any の仮引数を持つ関数には、どんなオブジェクトでも引数として渡すことができます。

次のプログラムの中で宣言されている printAny という関数は、オブジェクトを受け取って、「引数は○○です。」という形で、そのオブジェクトを出力します。

プログラムの例 printany.kt

```
fun printAny(a: Any) {
    println("引数は${a}です。")
}
```

実行例

```
>>> printAny(87)
引数は 87 です。
>>> printAny("umiushi")
引数は umiushi です。
>>> printAny(listOf(53, 12, 48, 61))
引数は [53, 12, 48, 61] です。
```

型が Any の変数にはどんなオブジェクトでも代入することができるわけですが、型が Any の変数にオブジェクトを代入した場合、原則として、そのオブジェクトが持っている、そのオブジェクトの本来の型に固有の機能は、使うことができません。

```
>>> val a: Any = "kamenote"
>>> a.length
error: unresolved reference: length
a.length
  ^
```

7.12.2 キャスト

本来の型ではない変数にオブジェクトが代入されているときに、そのオブジェクトを本来の型に戻すことを、オブジェクトを「キャストする」(cast) と言います。オブジェクトをキャストすると、そのオブジェクトの本来の型に固有の機能を使うことができるようになります。

オブジェクトは、as という演算子を使うことによってキャストすることができます。

as を使ってオブジェクトをキャストしたいときは、

変数名 as 型名

という形の式を書きます。この中の「変数名」のところには、キャストしたいオブジェクトが代入されている変数の名前を書きます。そして「型名」のところには、そのオブジェクトの本来の型の名前を書きます。そうすると、変数に代入されているオブジェクトをキャストした結果が式全体の値になります。たとえば、

```
a as String
```

という式を評価すると、 a という変数に代入されている文字列をキャストした結果が値として得られます。

```
>>> val a: Any = "kamenote"
>>> (a as String).length
8
```

7.12.3 型をチェックする演算子

変数にオブジェクトが代入されていて、そのオブジェクトの本来の型として複数の可能性がある場合、そのオブジェクトの本来の型に固有の機能を使うためには、そのオブジェクトの型をチェックする必要があります。

変数に代入されているオブジェクトの型は、 is または !is という演算子を使うことによってチェックすることができます。

is を使ってオブジェクトの型をチェックしたいときは、

変数名 is 型名

という形の式を書きます。この中の「変数名」のところには、型をチェックしたいオブジェクトが代入されている変数の名前を書きます。そして「型名」のところには、チェックしたい型の名前を書きます。そうすると、オブジェクトの本来の型が、型名で指定された型と一致しているならば、式全体の値として真が得られて、一致していないならば偽が得られます。たとえば、

```
a is String
```

という式を評価すると、 a という変数に代入されているオブジェクトが文字列ならば真、そうでなければ偽が得られます。

```
>>> val a: Any = "isoginchaku"
>>> a is String
true
>>> a is Int
false
```

!is も、それを使う式の書き方は is と同じです。 !is の結果は、 is とは逆に、型が一致していないならば真、一致しているならば偽です。

```
>>> val a: Any = "kurage"
>>> a !is Int
true
>>> a !is String
false
```

7.12.4 スマートキャスト

Kotlin のコンパイラは、 is または !is によって型がチェックされていて、変数に代入されているオブジェクトの本来の型を特定することができる場合は、 as による明示的なキャストが書かれていなかったとしても、キャストを自動的に補います。

そのような、明示的に記述されていないにもかかわらずコンパイラによって自動的に補われるキャストは、「スマートキャスト」(smart cast) と呼ばれます。

7.12.5 if 式によるスマートキャスト

is または !is を使った式が if 式の条件式として書かれていて、変数に代入されているオブジェクトの本来の型をコンパイラが特定することができる場合は、キャストが自動的に補われますので、キャストを明示的に書くことは必要ではありません。

次のプログラムの中で宣言されている lenAny という関数は、オブジェクトを受け取って、それが文字列ならばその長さを返して、文字列ではないならば −1 を返します。

プログラムの例 lenany.kt

```
fun lenAny(a: Any): Int = if (a is String) a.length else -1
```

実行例

```
>>> lenAny("namako")
6
```

```
>>> lenAny(380)
-1
>>> lenAny(true)
-1
```

7.12.6 when 式によるスマートキャスト

if 式と同じように、when 式でも、変数に代入されているオブジェクトの本来の型をコンパイラが特定することができる場合は、キャストが自動的に補われます。

when 式の選択肢としては、

is 型名 -> 式またはブロック

という形のものを書くことができます。この形の選択肢は、when の右側に書かれた変数名で指定された変数に代入されているオブジェクトの本来の型が、「型名」のところに書かれた型名で指定された型と一致している場合に選択されます。

次のプログラムの中で宣言されている half という関数は、オブジェクトを受け取って、それが整数ならばそれを 2 で除算した商を返して、それが文字列ならばそれの前半（長さが奇数の場合は中央の文字の手前まで）を返して、整数でも文字列でもないならばそれをそのまま返します。

プログラムの例　half.kt

```
fun half(a: Any): Any =
    when (a) {
        is Int -> a / 2
        is String -> a.substring(0, a.length / 2)
        else -> a
    }
```

実行例

```
>>> half(34)
17
>>> half("isoginchaku")
isogi
>>> half(true)
true
```

7.12.7 論理演算子によるスマートキャスト

if 式や when 式と同じように、論理演算子の&&と||についても、それらの演算子の左側で型をチェックした場合、右側の式ではキャストが自動的に補われます。たとえば、

```
a is String && a.length == 0
```

という式では、&&の右側の a は、String に自動的にキャストされます。同じように、

```
a !is String || a.length == 0
```

という式も、||の右側の a は、String に自動的にキャストされます。

第 8 章　ジェネリクス

8.1　ジェネリクスの基礎

8.1.1　ジェネリクスとは何か

「ジェネリック」(generic) という英語の形容詞は、「総称的な」という意味を持っています。Kotlin には、ジェネリックな型、つまり特定の型に限定されない総称的な型を扱うことができる仕組みがあります。この仕組みは、「ジェネリクス」(generics) と呼ばれます。日本語では、ジェネリクスは、「総称型」と呼ばれることもあります。

第 6.1.3 項で説明したように、コレクションの型は、

識別子<型名, ･･･>

という形の型名であらわされます。このような形の型名は、ジェネリックな型を特定の型に限定した型をあらわしています。

ジェネリックな型を特定の型に限定した型の型名を書くときに、小なりと大なりとのあいだに書かれる型名は、「型引数」(type argument) と呼ばれます。

8.1.2　ジェネリッククラスとジェネリック関数

クラスと関数のそれぞれは、ジェネリックな型を扱うことができます。

ジェネリックな型を扱うクラスは「ジェネリッククラス」(generic class) と呼ばれ、ジェネリックな型を扱う関数は「ジェネリック関数」(generic function) と呼ばれます。

ジェネリッククラスについては第 8.2 節で、ジェネリック関数については第 8.3 節で説明することにしたいと思います。

8.2　ジェネリッククラス

8.2.1　ジェネリッククラスの宣言

ジェネリッククラスの宣言の書き方は、普通のクラス宣言の書き方とほとんど同じです。違うところは、ジェネリッククラスの宣言の場合、クラスの名前になる識別子の右側に、

<識別子, ･･･>

という形のものを書く、ということだけです。この中に書かれた識別子は、「型パラメーター」(type parameter) と呼ばれます。型パラメーターにする識別子に関しては、1 文字の英字の大文字を使うという慣習があります。

型パラメーターがあらわしているのは、ジェネリックな型です。それがあらわしているジェネリックな型は、型引数によって特定の型に限定されることになります。たとえば、

```
class Namako<T>(val a: T)
```

というクラス宣言で宣言された Namako というジェネリッククラスの場合、T という型パラメーターがあらわしているジェネリックな型を Int で限定したとすると、a というプロパティーは、型が Int だと宣言されることになります。

8.2.2　ジェネリッククラスのプライマリーコンストラクタ

ジェネリッククラスのプライマリーコンストラクタは、

クラス名<型名>(式, ･･･)

という形の式を書くことによって呼び出すことができます。この中の「型名」のところには、T という型パラメーターがあらわしているジェネリックな型を限定する型引数を書きます。たとえば、

```
Namako<Int>(370)
```

という式でプライマリーコンストラクタを呼び出すことによって、ジェネリックな型を Int で限定した Namako というジェネリッククラスのオブジェクトを生成して、プロパティーのバッキングフィールドに 370 を代入することができます。

プライマリーコンストラクタを呼び出す式の中の型引数は、ジェネリックな型を限定する型をコンパイラが推論することができる場合は、省略することができます。たとえば、Namako のプライマリーコンストラクタを呼び出す先ほどの式は、

```
Namako(370)
```

と書くこともできます。

8.2.3　ジェネリッククラスの例

次のプログラムは、Container というジェネリッククラスを宣言しています。

プログラムの例　container.kt

```
class Container<T>(val content: T)
```

Containerのオブジェクトは、Tという型パラメーターがあらわしているジェネリックな型を限定した型のオブジェクトを1個だけ保持することができます。

実行例

```
>>> val a = Container<Int>(802)
>>> val b = Container<String>("namako")
>>> a.content
802
>>> b.content
namako
```

この場合、型引数は省略することができます。

実行例

```
>>> val a = Container(617)
>>> val b = Container("umiushi")
>>> a.content
617
>>> b.content
umiushi
```

次のプログラムは、TwoObjectsというジェネリッククラスを宣言しています。

プログラムの例 twoobjects.kt

```
class TwoObjects<F, S>(val first: F, val second: S)
```

TwoObjectsのオブジェクトは、FとSという二つの型パラメーターのそれぞれによってあらわされるジェネリックな型を限定したそれぞれの型を持つ2個のオブジェクトを保持することができます。

実行例

```
>>> val a = TwoObjects<Char, Boolean>('Q', false)
>>> a.first
Q
>>> a.second
false
```

この場合も、型引数は省略することができます。

実行例

```
>>> val a = TwoObjects('R', true)
>>> a.first
R
>>> a.second
true
```

8.3 ジェネリック関数

8.3.1 ジェネリック関数の宣言

ジェネリッククラスの宣言の場合と同じように、ジェネリック関数の宣言も、普通の関数宣言の書き方とほとんど同じです。違うところは、ジェネリック関数の場合、関数名になる識別子の左側に（つまりfunと関数名の中間に）、

<識別子, ···>

という形のものを書く、ということだけです。この中に書かれた識別子も、ジェネリッククラスの場合と同じように、型パラメーターになります。

8.3.2 ジェネリック関数の呼び出し

ジェネリック関数を呼び出す関数呼び出しは、

関数名<型名, ···>(式, ···)

と書きます。たとえば、2 個の型パラメーターを持つ hoge というジェネリック関数があるとするとき、

```
hoge<String, Int>(a)
```

というような関数呼び出しを書くことによって、hoge を呼び出すことができます。

ジェネリック関数を呼び出す関数呼び出しの中の型引数は、ジェネリックな型を限定する型をコンパイラが推論することができる場合は、省略することができます。ですから、hoge を呼び出す先ほどの関数呼び出しは、コンパイラによる推論が可能ならば、

```
hoge(a)
```

と書くこともできます。

8.3.3　ジェネリック関数の例

次のプログラムは、reverse というジェネリック関数を宣言しています。

プログラムの例　reverse.kt

```
class TwoObjects<F, S>(val first: F, val second: S)

fun <F, S> reverse(t: TwoObjects<F, S>): TwoObjects<S, F> =
    TwoObjects(t.second, t.first)
```

このプログラムの中で宣言されている TwoObjects というジェネリッククラスは、第 8.2.3 項で紹介したものと同じです。reverse は、TwoObjects のオブジェクトを受け取って、その中のオブジェクトの順番が入れ替わった TwoObjects のオブジェクトを返します。

実行例

```
>>> val a = TwoObjects(485, "kurage")
>>> val b = reverse(a)
>>> b.first
kurage
>>> b.second
485
```

第 9 章　高階関数

9.1　高階関数の基礎

9.1.1　高階関数とは何か

プログラミング言語のうちには、引数として関数を受け取る関数を宣言したり、戻り値として関数を返す関数を宣言したりすることができる、という機能を持っているものがあります。

そのようなプログラミング言語においては、引数として関数を受け取る関数や、戻り値として関数を返す関数は、「高階関数」(higher-order function) と呼ばれます。

9.1.2　関数オブジェクト

Kotlin では、関数それ自体を引数として受け取ったり、関数それ自体を戻り値として返したりする関数を宣言することはできません。しかし、関数として動作させることができるオブジェクトを引数として受け取ったり戻り値として返したりする関数を宣言する、ということは可能です。

Kotlin の関数が引数として受け取ったり戻り値として返したりすることができる、関数として動作させることができるオブジェクトは、通常、「関数オブジェクト」(function object) と呼ばれます。

関数と同じように、関数オブジェクトを動作させることを、その関数オブジェクトを「呼び出す」(call) と言います。

関数オブジェクトが持っている型は、「関数型」(function type) と呼ばれます。

関数から、その関数の動作をする関数オブジェクトを求めたいときは、

::関数名

という形の式を書きます。そうすると、関数名で指定された関数の動作をする関数オブジェクトが、その式の値として得られます。たとえば、

```
::namako
```

という宣言を書くことによって、namako という名前の関数の動作をする関数オブジェクトを求めることができます。

関数オブジェクトを呼び出したいときは、

式$_1$(式$_2$, ･･･)

という形の式を書きます。この中の「式$_1$」のところには、関数オブジェクトを求める式を書きます。この形の式を評価すると、式$_1$ の値として得られた関数オブジェクトが呼び出されて、丸括弧の中に書かれた式の値が、その関数オブジェクトに引数として渡されます。たとえば、関数オブジェクトが f という変数に代入されているとするとき、

```
f(83)
```

という式を評価することによって、f に代入されている関数オブジェクトを呼び出して、83 という引数をそれに渡すことができます。

それでは、整数の 2 乗を求める square という関数を宣言して、その関数から関数オブジェクトを求めて、その関数オブジェクトを呼び出してみましょう。

```
>>> fun square(n: Int) = n * n
>>> val f = ::square
>>> f(7)
49
```

9.1.3 関数型の型名

関数型の型名は、

(型名, ･･･) -> 型名

という型名によってあらわされます。-> の左側の「型名」のところには、引数の型をあらわす型名を書きます。そして、-> の右側の「型名」のところには、戻り値の型をあらわす型名を書きます。たとえば、1 個目の引数として文字列、2 個目の引数として整数を受け取って、戻り値として真偽値を返す関数オブジェクトの型名は、

```
(Strint, Int) -> Boolean
```

と書きます。

引数を受け取らない関数オブジェクトの型名は、

() -> 型名

というように、-> の左側に丸括弧だけを書きます。

戻り値を返さない関数オブジェクトは、実際には、kotlin.Unit というオブジェクトを戻り値として返します。このオブジェクトは、Unit という型を持っていますので、戻り値を返さない関数オブジェクトの型は、

(型名, ･･･) -> Unit

という型名であらわされることになります。

9.2 ラムダ式

9.2.1 ラムダ式の基礎

関数オブジェクトを求める方法は、第 9.1.2 項で説明した、関数名の左側に :: と書くという方法だけではありません。「ラムダ式」(lambda expression) と呼ばれるものを書くというのも、関数オブジェクトを求める方法のひとつです。

ラムダ式は、式の一種です。ラムダ式を評価すると、関数オブジェクトが生成されて、その関数オブジェクトがラムダ式の値になります。

関数名の左側に :: を書くという方法で関数オブジェクトを求めるためには、その関数があらかじめ宣言されている必要があります。それに対して、ラムダ式は、関数から関数オブジェクトを求めるのではなくて、何もないところから関数オブジェクトを生成します。

引数として関数オブジェクトを受け取る高階関数を呼び出す場合、あらかじめ関数を宣言しておいて、その関数から関数オブジェクトを求めてもいいのですが、ラムダ式を使えば、あらかじめ関数を宣言しておくという手間を省くことができます。

9.2.2　ラムダ式の書き方

ラムダ式は、

{ 仮引数の宣言 , ··· -> 式 }

と書きます。この中の「仮引数の宣言」のところには、関数宣言を書く場合と同じ書き方で、関数オブジェクトが引数を受け取るための仮引数の宣言を書きます。そして、この中の「式」のところには、関数オブジェクトが戻り値として返すオブジェクトを求める式を書きます。たとえば、

```
{ n: Int -> n * n }
```

というラムダ式を評価すると、その値として、引数の 2 乗を求める関数オブジェクトが得られます。

```
>>> val f = { n: Int -> n * n }
>>> f(7)
49
```

ラムダ式の値として得られた関数オブジェクトを、変数に代入するのではなくて、その場所でいきなり呼び出す、ということも可能です。たとえば、

```
{ a: Int, b: Int -> a * 10000 + b }(38, 74)
```

という式を評価すると、ラムダ式の値として得られた関数オブジェクトが呼び出されて、38 と 74 という整数がその関数オブジェクトに引数として渡されます。

```
>>> { a: Int, b: Int -> a * 10000 + b }(38, 74)
380074
```

コンパイラが、ラムダ式によって生成される関数オブジェクトが受け取る引数の型を推論することができる場合、そのラムダ式は、仮引数の型を指定する型名を省略することができます。

```
>>> val f: (Int, Int) -> Int = { a, b -> a * 10000 + b }
>>> f(38, 74)
380074
```

ラムダ式の -> の右側には、文の列を書くこともできます。ラムダ式の -> の右側に文の列を書いた場合、生成される関数オブジェクトの戻り値は、最後に評価された式の値です。

```
>>> val f = { n: Int ->
...     for (i in 1..n) {
...         print("${i} ")
...     }
...     println()
...     n * n
... }
>>> f(20)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
400
```

ラムダ式によって生成される関数オブジェクトが受け取る引数が 1 個だけで、コンパイラがその引数の型を推論することができる場合、そのラムダ式は、仮引数の宣言と -> を省略することができます。それらを省略した場合、 it という名前の仮引数が暗黙のうちに宣言されます。

```
>>> val f: (Int) -> Int = { it * it }
>>> f(7)
49
```

引数を受け取らない関数オブジェクトを生成するラムダ式を書く場合、 -> は省略することができます。

```
>>> { -> "namako" }()
namako
>>> { "umiushi" }()
umiushi
```

引数を受け取るけれども、その引数を使わない関数オブジェクトを生成するラムダ式を書く場合、仮引数名を書く代わりに、1 個のアンダースコア (_) を書くことができます。

```
>>> { _: Int, a: Int, _: Int -> a * 10000 }(62, 59, 37)
590000
```

9.3 関数オブジェクトを受け取る関数

9.3.1 関数型の仮引数の宣言

引数として関数オブジェクトを受け取る関数を宣言するためには、関数型を持つ仮引数を宣言する必要があります。たとえば、(Int, String) -> Boolean という型の関数オブジェクトを引数として受け取る関数を宣言するためには、

```
f: (Int, String) -> Boolean
```

というような仮引数の宣言を書く必要があります。

次のプログラムの中で宣言されている fifty という関数は、(Int) -> Int という型の関数オブジェクトを受け取って、それを呼び出して、引数として 50 を渡して、その戻り値を返します。

プログラムの例 fifty.kt

```
fun fifty(f: (Int) -> Int): Int = f(50)
```

実行例

```
>>> fifty({ it + 10000 })
10050
>>> fifty({ it * it })
2500
```

9.3.2 関数呼び出しの中のラムダ式

関数オブジェクトだけを引数として受け取る関数を呼び出して、ラムダ式の値をその関数に渡す場合、そのラムダ式を囲む丸括弧は、省略することができます。

先ほど紹介した fifty という関数を使って、試してみましょう。

実行例

```
>>> fifty { it * 100 + 5 }
5005
```

次のプログラムの中で宣言されている times という関数は、init と n を整数、f を (Int) -> Int という型の関数オブジェクトとするとき、times(init, n, f) という式で呼び出すと、init を初期値として、f の戻り値を引数にして f を呼び出す、ということを n 回だけ繰り返して、その結果を返します。

プログラムの例 times.kt

```
fun times(init: Int, n: Int, f: (Int) -> Int): Int {
    var a = init
    for (i in 1..n) {
        a = f(a)
    }
    return a
}
```

実行例

```
>>> times(2, 1, { it * it })
4
>>> times(2, 2, { it * it })
16
>>> times(2, 3, { it * it })
```

```
256
>>> times(2, 4, { it * it })
65536
```

関数オブジェクトを最後の引数として受け取る関数を呼び出して、ラムダ式の値をその関数に渡す場合、そのラムダ式は、丸括弧の外に出すことができます。

実行例

```
>>> times(3, 2) { it * it }
81
```

9.4　関数オブジェクトを返す関数

9.4.1　加算をする関数オブジェクトを返す関数

戻り値として関数オブジェクトを返す関数の例として、加算をする関数オブジェクトを返す関数を宣言してみましょう。

次のプログラムの中で宣言されている add という関数は、n という仮引数に整数を受け取って、戻り値として関数オブジェクトを返します。この関数が返す関数オブジェクトは、引数として整数を受け取って、n と引数を加算した結果を返します。

プログラムの例　add.kt

```
fun add(n: Int): (Int) -> Int = { n + it }
```

実行例

```
>>> add(50)(7)
57
```

9.4.2　関数オブジェクトを合成する関数

1 個の引数を受け取って戻り値を返す、f と g という 2 個の関数が与えられたとき、1 個の引数を受け取って、それを引数にして g を呼び出して、その戻り値を引数にして f を呼び出して、その戻り値を返す、という動作をする関数を宣言することを、f と g を「合成する」(compose) と言います。

次のプログラムの中で宣言されている compose という関数は、1 個の引数を受け取ってそれと同じ型の戻り値を返す二つの関数オブジェクトを受け取って、それらの関数オブジェクトを合成した関数オブジェクトを返します。

プログラムの例　compose.kt

```
fun <T> compose(f: (T) -> T, g: (T) -> T): (T) -> T =
    { f(g(it)) }
```

実行例

```
>>> compose({ n: Int -> n + 37 }, { n: Int -> n * 10000 })(45)
450037
>>> compose({ n: Int -> n * 100 }, { n: Int -> n * n })(7)
4900
```

9.5　コレクションを扱う高階関数

9.5.1　コレクションを扱う高階関数の基礎

Kotlin の標準ライブラリーには、コレクションを扱うさまざまな高階関数が含まれています。この節では、標準ライブラリーに含まれている、コレクションを扱う高階関数を紹介したいと思います。

標準ライブラリーに含まれている、コレクションを扱う高階関数は、通常、

式 . 関数名　ラムダ式

という形の式を書くことによって呼び出されます。この中の「式」のところには、処理の対象となるコレクションを求める式を書きます。

9.5.2 map

map という関数は、引数として関数オブジェクトを受け取って、それによってリストまたはセットのそれぞれの要素を処理して、それらの処理の結果から構成されるリストを戻り値として返します。

要素の型が T であるリストまたはセットを処理する場合に map に引数として渡す関数オブジェクトは、引数の型が T である必要があります。(T) -> U という型の関数オブジェクトを引数として渡した場合、map の戻り値は、List<U> という型のリストになります。

```
>>> listOf(8, 25, 6, 3, 12).map { it.toString(2) }
[1000, 11001, 110, 11, 1100]
```

map は、文字列を構成しているそれぞれの文字を処理して、その結果から構成されるリストを求めることもできます。

```
>>> "namako".map { it.toInt() }
[110, 97, 109, 97, 107, 111]
```

9.5.3 forEach

forEach という関数は、map と同じように、引数として関数オブジェクトを受け取って、それによってそれぞれの要素を処理します。

map と forEach との相違点は、前者は関数オブジェクトの戻り値から構成されるリストを生成するのに対して、後者はそれをしないというところにあります。ですから、関数オブジェクトの戻り値から構成されるリストが必要ではない場合は、map よりも forEach を使うほうがいいでしょう。

```
>>> listOf(38, 74, 59, 21).forEach { println(it) }
38
74
59
21
```

forEach は、リストとセットだけではなくて、マップを処理することもできます。forEach を使ってマップを処理する場合は、引数として、2 個の引数を受け取る関数オブジェクトを渡します。その関数オブジェクトは、1 個目の引数としてキー、2 個目の引数として値を受け取ります。

```
>>> mapOf('a' to 63, 'b' to 72, 'c' to 40).forEach {
...     k, v -> println("${k}: ${v}")
... }
a: 63
b: 72
c: 40
```

9.5.4 filter

filter という関数は、戻り値として真偽値を返す関数オブジェクトを引数として受け取って、それによってリストまたはセットのそれぞれの要素をテストして、戻り値として真が得られた要素から構成されるリストを戻り値として返します。

要素の型が T であるリストまたはセットを処理する場合に filter に引数として渡す関数オブジェクトは、

```
T -> Boolean
```

という型である必要があります。filter の戻り値は、List<T> という型のリストになります。

```
>>> listOf(3, 8, 7, 6, 4, 1, 0, 5).filter { it % 2 == 0 }
[8, 6, 4, 0]
```

filter は、文字列を構成しているそれぞれの文字をテストして、関数オブジェクトが真を返した文字から構成される文字列を求めることもできます。

```
>>> "Unidentified Flying Object".filter { it.toInt() <= 90 }
U F O
```

9.5.5　all と any

all という関数は、戻り値として真偽値を返す関数オブジェクトを引数として受け取って、それによってリストまたはセットのそれぞれの要素をテストして、すべての要素について戻り値として真が得られたならば真を、そうでなければ偽を戻り値として返します。

```
>>> listOf(72, 84, 56, 40, 22, 38, 64).all { it % 2 == 0 }
true
>>> listOf(72, 84, 56, 40, 22, 33, 64).all { it % 2 == 0 }
false
```

any という関数は、戻り値として真偽値を返す関数オブジェクトを引数として受け取って、それによってリストまたはセットのそれぞれの要素をテストして、戻り値として真が得られる要素が少なくともひとつ存在していたならば真を、そうでなければ偽を戻り値として返します。

```
>>> listOf(72, 84, 56, 40, 22, 33, 64).any { it % 2 == 0 }
true
>>> listOf(69, 23, 81, 59, 45, 77, 23).any { it % 2 == 0 }
false
```

all は、文字列についても、それを構成しているすべての文字が何らかの条件を満足しているかどうかを調べることができます。 any も同様です。

```
>>> "abcdefg".all { it.toInt() >= 97 }
true
>>> "abcdEfg".all { it.toInt() >= 97 }
false
>>> "abcdEfg".any { it.toInt() >= 97 }
true
>>> "ABCDEFG".any { it.toInt() >= 97 }
false
```

9.5.6　count

count という関数は、戻り値として真偽値を返す関数オブジェクトを引数として受け取って、それによってリストまたはセットのそれぞれの要素をテストして、戻り値として真が得られた要素の個数を戻り値として返します。

```
>>> listOf(48, 72, 33, 24, 59, 86, 34).count { it % 2 == 0 }
5
```

count は、文字列についても、それを構成している文字のうちで何らかの条件を満足しているものの個数を求めることができます。

```
>>> "abcDefGhijkLmnOpQrstUvwxYz".count { it.toInt() <= 90 }
7
```

第 10 章　null 許容型

10.1　null 許容型の基礎

10.1.1　Java と null

Java というプログラミング言語があります。この言語で書かれたプログラムは、しばしば実行中にトラブルを起こします。その原因の多くは、「何もない」ということを意味している null と呼ばれるものに対して何らかの処理を実行しようとしたことです。

Kotlin の設計においては、null がトラブルを起こしやすいという Java の問題点を改善するための注意が払われています。すなわち、Kotlin では、null がトラブルを起こす可能性があるプログラムをコンパイルしようとすると、可能な限り、コンパイラがそれを指摘してくれるのです。

この章では、Kotlin においては null はどのように扱われるのか、ということについて説明したいと思います。

10.1.2 Kotlin における null

Kotlin では、「何もない」ということをあらわすために、null というオブジェクトが使われます。null という式を評価すると、その値として、このオブジェクトが得られます。

== と != は、通常は異なる型のオブジェクトを比較することができませんが、null については、どんな型のオブジェクトとも比較することができます。

```
>>> null == null
true
>>> null != null
false
>>> null == 350
false
>>> null == "namako"
false
>>> null != "namako"
true
```

10.1.3 null 許容型と null 非許容型

null は、「何もない」ということを意味しているオブジェクトです。このオブジェクトを扱うためには、何らかの型のオブジェクトを代入することができて、null を代入することもできる変数を宣言したり、何らかの型のオブジェクトを戻り値として返すことができて、null を返すこともできる関数を宣言したりすることが必要になります。そして、そのためには、何らかの型のオブジェクトであってもいいし、null であってもいいという型、つまり null を許容する型というものが必要になります。

Kotlin では、普通の型は null を許容しません。たとえば、型が Int の変数には、null を代入することができません。同じように、戻り値の型が String の関数は、戻り値として null を返すことができません。このような、null を許容しない普通の型は、「null 非許容型」(non-null type) と呼ばれます。

それに対して、null を許容する型は、「null 許容型」(nullable type) と呼ばれます。たとえば、null 許容型の Int の変数には、整数を代入することもできますし、null を代入することもできます。同じように、戻り値の型が null 許容型の String の関数は、戻り値として文字列を返すこともできますし、null を返すこともできます。

null 非許容型の型名の右側に疑問符 (?) を書いたものは、その null 非許容型を null 許容型にしたものの型名になります。たとえば、Int? は null 許容型の Int の型名で、String? は null 許容型の String の型名です。

10.1.4 null 許容型に対する制限

オブジェクトの型を null 許容型にすると、そのオブジェクトに対する操作が制限されます。たとえば、null 許容型のオブジェクトが持っているメンバーには、通常の方法ではアクセスすることができません。

```
>>> fun strToInt(s: String?): Int = s.toInt()
error: only safe (?.) or non-null asserted (!!.) calls are
allowed on a nullable receiver of type String?
fun strToInt(s: String?): Int = s.toInt()
                                ^

>>> fun strLen(s: String?): Int = s.length
error: only safe (?.) or non-null asserted (!!.) calls are
allowed on a nullable receiver of type String?
fun strLen(s: String?): Int = s.length
                              ^
```

また、null 非許容型の変数には、null 許容型のオブジェクトを代入することができません。

```
>>> val a: Int? = 434
>>> val b: Int = a
error: type mismatch: inferred type is Int? but Int was expected
val b: Int = a
             ^
```

10.2 安全呼び出し演算子

10.2.1 if 式による null 許容型の制限解除

第 10.1.4 項で述べたように、null 許容型のオブジェクトが持っているメンバーには、通常の方法ではアクセスすることができません。

null 許容型のオブジェクトが持っているメンバーにアクセスするための方法のひとつは、if 式を使うというものです。null 許容型のオブジェクトが null かどうかを調べて、それが null ではない場合だけ、そのオブジェクトのメンバーにアクセスする、という if 式は、正常にコンパイルされます。

次のプログラムの中で宣言されている strLen という関数は、文字列または null を受け取って、それが文字列ならばその長さを返して、そうでなくて null ならば null を返します。

プログラムの例 strlen.kt

```
fun strLen(s: String?): Int? =
    if (s != null) s.length else null
```

実行例

```
>>> strLen("tatsunootoshigo")
15
>>> strLen(null)
null
```

10.2.2 安全呼び出し演算子とは何か

null 許容型のオブジェクトが持っているメンバーにアクセスするための方法は、if 式を使うというものだけではありません。もうひとつの方法は、「安全呼び出し演算子」(safe call operator) と呼ばれる、?. という演算子を使うというものです。

安全呼び出し演算子があらわしている動作は、「null 許容型のオブジェクトが null かどうかを調べて、それが null ではない場合だけ、そのオブジェクトのメンバーにアクセスする」という、if 式で書くことができる動作と同じです。

10.2.3 安全呼び出し演算子の使い方

オブジェクトが持っているメンバーにアクセスしたいとき、通常は、オブジェクトを求める式とメンバー名とのあいだにドット (.) を書くわけですが、そのドットの代わりとして、

式?.メンバー名

というように安全呼び出し演算子を書くことによって、null 許容型のオブジェクトのメンバーにアクセスすることができます。

安全呼び出し演算子を使った式の値は、null 許容型のオブジェクトが null ではなかった場合はメソッドの戻り値またはプロパティーによって取り出されたオブジェクトで、null だった場合は null です。たとえば、

```
s?.length
```

という式を評価すると、

```
if (s != null) s.length else null
```

という if 式と同じ値が得られます。

10.2.4 let

安全呼び出し演算子と組み合わせて使われることが多い、let という関数があります。

let を呼び出す式は、

式.let ラムダ式

と書きます。

let は、引数として受け取った関数オブジェクト（ラムダ式の値）を呼び出して、レシーバーをその関数オブジェクトに引数として渡します。そして、呼び出した関数オブジェクトの戻り値

を返します。

```
>>> 6.let { it * it }
36
```

aという変数にnull許容型のオブジェクトが代入されているとするとき、

```
if (a != null) 何らかの処理 else null
```

というif式があらわしている動作は、安全呼び出し演算子とletを組み合わせることによって、if式よりも短い式で書くことができます。安全呼び出し演算子とletを組み合わせた、

```
式?.let ラムダ式
```

という式を書くことによって、式の値がnullではない場合だけ、ラムダ式の値として得られた関数オブジェクトを呼び出す、という動作を記述することができるのです。

```
>>> val a = listOf("namako", null, "umiushi", null, "uni")
>>> for (s in a) {
...     s?.let { println(it) }
... }
namako
umiushi
uni
```

10.3 エルビス演算子

10.3.1 エルビス演算子とは何か

第10.2節で説明したように、null許容型のオブジェクトがnullではない場合はそれのメンバーにアクセスして、nullの場合はnullを求めるという動作は、安全呼び出し演算子を使うことによって、if式よりも短い式で記述することができます。たとえば、

```
if (s != null) s.length else null
```

というif式であらわされる動作は、安全呼び出し演算子を使うことによって、

```
s?.length
```

という短い式で記述することができます。

場合によっては、nullの場合はnullを求めるのではなくて、nullの場合はnullではない何らかのオブジェクトを求めるという動作を記述したい、ということもあります。たとえば、

```
if (s != null) s.length else -1
```

というような動作です。そのような動作は、「エルビス演算子」(elvis operator)と呼ばれる、?:という演算子を使うと、if式よりも短い式で記述することができます。

「エルビス演算子」という名前の由来は、エルビス・プレスリー(Elvis Presley)というアメリカのミュージシャンです。エルビス演算子を時計回りに90度だけ回転させると、それがプレスリーの顔に見えるのです（疑問符が髪の毛で、コロンが目）。

10.3.2 エルビス演算子の使い方

エルビス演算子を使いたいときは、

```
式1 ?: 式2
```

という形の式を書きます。この形の式を評価すると、まず式$_1$が評価されて、その値がnullではないならば、その値が式全体の値になります。その場合、式$_2$は評価されません。もしも式$_1$の値がnullだった場合は、式$_2$が評価されて、その値が式全体の値になります。

```
>>> "namako" ?: "umiushi"
namako
>>> null ?: "umiushi"
umiushi
```

null許容型のオブジェクトがnullではないならばそれのメンバーにアクセスして、nullならばnullではない何らかのオブジェクトを求める、という動作は、安全呼び出し演算子とエルビ

ス演算子を組み合わせることによって記述することができます。たとえば、

```
if (s != null) s.length else -1
```

という if 式によってあらわされる動作は、

```
s?.length ?: -1
```

という短い式で記述することができます。

```
>>> val a = listOf("namako", null, "umiushi", null, "uni")
>>> a.map { it?.length ?: -1 }
[6, -1, 7, -1, 3]
```

第 11 章　例外

11.1　例外の基礎

11.1.1　例外とは何か

Kotlin のプログラムは、実行中に不都合な事象が起きた場合、「例外」(exception) と呼ばれるオブジェクトを発生させます。

たとえば、/ という演算子によってあらわされる除算という演算を実行したとき、右側の数値（割る数）が 0 だった場合、プログラムは例外を発生させます。

例外は、「例外クラス」(exception class) と呼ばれるクラスから生成されるオブジェクトです。例外クラスは、ユーザーが宣言することもできますが、Kotlin の標準ライブラリーにも、さまざまな例外クラスが含まれています。

たとえば、0 による除算を実行した場合にプログラムが発生させる例外は、

```
ArithmeticException
```

という例外クラスから生成されます。

また、文字列が持っている toInt というメソッドは、文字列があらわしている整数を求めるわけですが、その文字列が整数をあらわしていない場合、プログラムは、

```
NumberFormatException
```

というクラスの例外を発生させます。

11.1.2　例外による関数の終了

プログラムが例外を発生させると、その例外は、原則的には実行中の関数をすべて終了させます。

それでは、次のプログラムを使って、例外は実行中の関数を終了させるということを確認してみましょう。

プログラムの例　exception.kt

```
fun func1(a: Int, b: Int) {
    println("func1 の開始")
    func2(a, b)
    println("func1 の終了")
}

fun func2(a: Int, b: Int) {
    println("func2 の開始")
    func3(a, b)
    println("func2 の終了")
}

fun func3(a: Int, b: Int) {
    println("func3 の開始")
    println(a / b)
    println("func3 の終了")
}
```

このプログラムは、func1、func2、func3という三つの関数から構成されています。func1はfunc2を呼び出して、func2はfunc3を呼び出します。func3は、その中で除算を実行しますので、そこで例外が発生する可能性があります。

このプログラムのfunc1を、2回、呼び出してみましょう。1回目は例外を発生させない引数を、2回目は例外を発生させる引数を与えてみます。

実行例

```
>>> func1(20, 3)
func1の開始
func2の開始
func3の開始
6
func3の終了
func2の終了
func1の終了
>>> func1(20, 0)
func1の開始
func2の開始
func3の開始
java.lang.ArithmeticException: / by zero
        at Line_0.func3(Line_0.kts:15)
        at Line_0.func2(Line_0.kts:9)
        at Line_0.func1(Line_0.kts:3)
>>>
```

このように、例外が発生した場合は、func3もfunc2もfunc1も、その時点で実行が中断されるということが分かります。

例外が発生した場合、原則的には、発生した例外のクラス名、例外が発生した理由、そして「スタックトレース」(stack trace) と呼ばれるものが出力されます。スタックトレースは、プログラムのどこを実行しているときに例外が発生したのかということを示していますので、プログラムを修正する上で有益な情報になります。

11.2 例外処理

11.2.1 例外処理の基礎

第11.1.2項で説明したように、例外が発生すると、原則的には実行中のすべての関数が終了させられます。しかし、場合によっては、ただ単に終了させられるのではなくて、発生した例外に対応した何らかの処理を実行したい、ということもあります。そのような、例外が発生した場合に実行される処理は、「例外処理」(exception processing) と呼ばれます。

例外処理を実行するためには、発生した例外を「捕獲する」(catch) ということが必要になります。「捕獲する」というのは、関数を終了させるという例外の機能を停止させるということです。

11.2.2 try式

例外を捕獲するためには、「try式」(try expression) と呼ばれる式を書く必要があります。

try式は、基本的には、

try ブロック$_1$ catch (識別子 : 例外クラス名) ブロック$_2$

と書きます。この中の「例外クラス名」のところには、捕獲したい例外のクラス名を書きます。「識別子」のところには、変数に名前として与える識別子を書きます。その変数には、捕獲された例外が代入されます。

この形のtry式を評価すると、まずブロック$_1$が実行されます。その実行中に例外が発生しなかった場合、ブロック$_2$は実行されないで、try式の評価は終了します。

もしも、ブロック$_1$の実行中に例外が発生して、その例外のクラスが、「例外クラス名」のところに書かれたものだった場合、その例外は捕獲されて、例外処理としてブロック$_2$が実行されます。

例外が発生しなかった場合、try式の値は、ブロック$_1$の実行で最後に評価された式の値です。それに対して、例外が発生した場合、try式の値は、ブロック$_2$の実行で最後に評価された式の値です。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を文字列に変換した結果を返します。ただし、割る数が 0 だった場合は、「割る数が 0 です。」という文字列を返します。

プログラムの例 divide.kt

```
fun divide(a: Int, b: Int): String =
    try {
        (a / b).toString()
    } catch (e: ArithmeticException) {
        "割る数が 0 です。"
    }
```

実行例

```
>>> divide(20, 3)
6
>>> divide(20, 0)
割る数が 0 です。
```

11.2.3 複数の catch ブロック

try 式の中にある、

catch (識別子 : 例外クラス名) ブロック

という部分は、「catch ブロック」(catch block) と呼ばれます。

catch ブロックは、1 個の try 式の中に何個でも書くことができます。それぞれの catch ブロックは、異なる例外クラスを指定することができますので、発生した例外のクラスごとに、異なる例外処理を実行することができます。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を整数に変換した結果を、2 個目の引数を整数に変換した結果で除算して、その商を文字列に変換した結果を返します。割る数が 0 だった場合は「割る数が 0 です。」という文字列を、文字列を整数に変換することができなかった場合は「整数に変換できない文字列です。」という文字列を返します。

プログラムの例 divide2.kt

```
fun divide(a: String, b: String): String =
    try {
        (a.toInt() / b.toInt()).toString()
    } catch (e: ArithmeticException) {
        "割る数が 0 です。"
    } catch (e: NumberFormatException) {
        "整数に変換できない文字列です。"
    }
```

実行例

```
>>> divide("20", "3")
6
>>> divide("20", "0")
割る数が 0 です。
>>> divide("20", "namako")
整数に変換できない文字列です。
```

11.2.4 例外クラスの階層

サブクラスというのは、スーパークラスを特殊化することによって作られたクラスのことです。つまり、サブクラスというのはスーパークラスの一種だということです。したがって、catch ブロックの中に名前を書くことによって指定される例外クラスは、その名前を持つクラスだけではありません。それを特殊化したすべてのサブクラスも、指定されたことになります。

標準ライブラリーに含まれている例外クラスの多くは、Exception というクラスを特殊化することによって作られています。ですから、catch ブロックでこのクラスを指定すると、例外の多くを一網打尽にすることができます。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を整数に変換した

結果を、2 個目の引数を整数に変換した結果で除算して、その商を文字列に変換した結果を返します。ただし、Exception というクラス、またはそのクラスを特殊化したクラスの例外が発生した場合は、「例外が発生しました。」という文字列を返します。

プログラムの例 divide3.kt

```
fun divide(a: String, b: String): String =
    try {
        (a.toInt() / b.toInt()).toString()
    } catch (e: Exception) {
        "例外が発生しました。"
    }
```

実行例

```
>>> divide("20", "3")
6
>>> divide("20", "0")
例外が発生しました。
>>> divide("20", "namako")
例外が発生しました。
```

11.2.5 finally ブロック

try 式の末尾には、「finally ブロック」(finally block) と呼ばれるものを 1 個だけ書くことができます。

finally ブロックは、

finally [ブロック]

と書きます。

finally ブロックの中のブロックは、例外が発生した場合も、例外が発生しなかった場合も、最後に必ず実行されます。ですから、例外が発生したかどうかにかかわらず、必ず実行されないといけない後始末などの処理は、finally ブロックの中に書いておくといいでしょう。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を返します。ただし、割る数が 0 だった場合は、「割る数が 0 です。」という文字列を返します。そして、例外が発生した場合も、発生しなかった場合も、「除算が終了しました。」という文字列を最後に出力します。

プログラムの例 divide4.kt

```
fun divide(a: Int, b: Int): String =
    try {
        (a / b).toString()
    } catch (e: ArithmeticException) {
        "割る数が 0 です。"
    } finally {
        println("除算が終了しました。")
    }
```

実行例

```
>>> divide(20, 3)
除算が終了しました。
6
>>> divide(20, 0)
除算が終了しました。
割る数が 0 です。
```

11.2.6 例外から文字列への変換

例外が捕獲された場合、その例外は、catch ブロックの丸括弧の中に書かれた識別子を名前とする変数に代入されます。

例外というのはオブジェクトですので、toString というメソッドを持っていて、それを呼び出すことによって文字列に変換することができます。

例外が持っている toString は、その例外のクラス名と、その例外が発生した理由についてのメッセージから構成される文字列を返します。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を返します。ただし、割る数が 0 だった場合は、例外を捕獲して、その例外を文字列に変換した結果を返します。

プログラムの例 divide5.kt

```
fun divide(a: Int, b: Int): String =
    try {
        (a / b).toString()
    } catch (e: ArithmeticException) {
        e.toString()
    }
```

実行例

```
>>> divide(20, 3)
6
>>> divide(20, 0)
java.lang.ArithmeticException: / by zero
```

11.2.7 例外のメッセージ

例外は、それが発生した理由についてのメッセージを持っています。このメッセージは、例外が持っている message というプロパティーから取り出すことができます（このプロパティーの型は String? です）。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を返します。ただし、割る数が 0 だった場合は、捕獲した例外の message から取り出したメッセージを返します。

プログラムの例 divide6.kt

```
fun divide(a: Int, b: Int): String? =
    try {
        (a / b).toString()
    } catch (e: ArithmeticException) {
        e.message
    }
```

実行例

```
>>> divide(20, 3)
6
>>> divide(20, 0)
/ by zero
```

11.3 throw 式

11.3.1 例外を投げるということ

第 11.1.1 項で説明したように、「例外」というのは、「例外クラス」と呼ばれるクラスから生成されるオブジェクトのことです。したがって、「例外が発生する」というのは、「例外クラスからオブジェクトが生成される」ということを意味しています。

第 11.1.2 項で説明したように、プログラムが例外を発生させると、その例外は、原則的には実行中の関数をすべて終了させます。しかし、例外というのは、ただ単に生成されただけでは関数を終了させません。例外に関数を終了させるためには、関数を終了させるという、それが持っている機能を発現させる必要があるのです。

関数を終了させるという例外が持っている機能を発現させることを、例外を「投げる」(throw)、または例外を「スローする」と言います。

11.3.2 throw 式の書き方

例外を投げたいときは、「throw 式」と呼ばれる式を評価します。

throw 式は、

throw 式

と書きます。この中の「式」のところには、値として例外が得られる式を書きます。

throw 式を評価すると、「式」のところに書かれた式の値として得られた例外が投げられます。

11.3.3 例外を再び投げる例外処理

throw 式を使うことによって、例外を捕獲して何らかの処理を実行したのち、捕獲した例外を再び投げる、という例外処理を記述することができます。

次のプログラムの中で宣言されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を返します。ただし、割る数が 0 だった場合は、例外を捕獲して、「割る数が 0 です。」という文字列を出力して、そののち捕獲した例外を再び投げます。

プログラムの例 divide7.kt

```
fun divide(a: Int, b: Int): Int =
    try {
        a / b
    } catch (e: ArithmeticException) {
        println("割る数が 0 です。")
        throw e
    }
```

実行例

```
>>> divide(20, 3)
6
>>> divide(20, 0)
割る数が 0 です。
java.lang.ArithmeticException: / by zero
        at Line_0.divide(Line_0.kts:3)
```

11.3.4 例外の生成

例外は、例外クラスのコンストラクタを呼び出すことによって生成することができます。例外クラスのコンストラクタは、引数として文字列を受け取ります。例外は、コンストラクタが受け取った文字列をメッセージとして保持します。

第 5.6.4 項で、再帰を使って階乗を求める関数の宣言を紹介しましたが、そのときに紹介した宣言は、もしも引数としてマイナスの整数を受け取った場合は戻り値として 0 を返すというものでした。

次のプログラムは、引数がマイナスだった場合は ArithmeticException というクラスの例外を投げるように、階乗を求める関数の宣言を改良したものです。

プログラムの例 factorial3.kt

```
fun factorial(n: Int): Int =
    when {
        n == 0 -> 1
        n >= 1 -> n * factorial(n - 1)
        else -> throw ArithmeticException("minus argument")
    }
```

実行例

```
>>> factorial(5)
120
>>> factorial(-5)
java.lang.ArithmeticException: minus argument
        at Line_0.factorial(Unknown Source)
```

11.3.5 独自の例外クラスの宣言

例外クラスは、プログラムを書く人が独自のものを自由に宣言することができます。既存の例外クラスのサブクラスを宣言すると、それは新しい例外クラスになります。

次のプログラムは、引数がマイナスだった場合は FactorialException という独自の例外クラスの例外を投げるように、階乗を求める関数の宣言を改良したものです。

プログラムの例　factorial4.kt

```
class FactorialException(s: String) : ArithmeticException(s)

fun factorial(n: Int): Int =
    when {
        n == 0 -> 1
        n >= 1 -> n * factorial(n - 1)
        else -> throw FactorialException("minus argument")
    }
```

実行例

```
>>> factorial(5)
120
>>> factorial(-5)
Line_0$FactorialException: minus argument
        at Line_0.factorial(Unknown Source)
```

付録 A　練習問題の解答例

第 3 章の解答例

3.1
```
fun fortune() {
    println("あなたの今日の運勢は大吉です。")
}
```

3.2
```
fun threetimes(a: Int) {
    println("${a}の 3 倍は${a * 3}です。")
}
```

3.3
```
fun mtohm(m: Int) {
    println("${m}分は${m / 60}時間${m % 60}分です。")
}
```

3.4
```
fun divide(a: Int, b: Int) {
    println("${a}割る${b}は${a / b}あまり${a % b}です。")
}
```

3.5
```
fun hmtom(h: Int, m: Int) {
    println("${h}時間${m}分は${h * 60 + m}分です。")
}
```

3.6
```
fun square(a: Int): Int {
    return a * a
}
```

3.7
```
fun hmtom(h: Int, m: Int): Int {
    return h * 60 + m
}
```

3.8
```
fun square(a: Int): Int = a * a
```

第 4 章の解答例

4.1
```
fun divisible(a: Int, b: Int): Boolean = a % b == 0
```

4.2
```
fun divideorzero(a: Int, b: Int): Int = if (b != 0) a / b else 0
```

4.3
```
fun mtohm(m: Int) {
```

```
        if (m != 0) {
            if (m >= 60) print("${m / 60}時間")
            if (m % 60 != 0) print("${m % 60}分")
        } else
            print("0 分")
        println()
    }
```

4.4
```
    fun six(n: Int): String =
        if (n % 6 == 0) "6 の倍数"
        else if (n % 3 == 0) "3 の倍数"
        else if (n % 2 == 0) "2 の倍数"
        else "3 の倍数でも 2 の倍数でもない整数"
```

4.5
```
    fun tsuki(m: Int): String =
        when (m) {
            1 -> "睦月"
            2 -> "如月"
            3 -> "弥生"
            4 -> "卯月"
            5 -> "皐月"
            6 -> "水無月"
            7 -> "文月"
            8 -> "葉月"
            9 -> "長月"
            10 -> "神無月"
            11 -> "霜月"
            12 -> "師走"
            else -> "存在しない月"
        }
```

4.6
```
    fun ordinal(n: Int): String =
        when (n % 100) {
            11, 12, 13 -> "${n}th"
            else ->
                when (n % 10) {
                    1 -> "${n}st"
                    2 -> "${n}nd"
                    3 -> "${n}rd"
                    else -> "${n}th"
                }
        }
```

4.7
```
    fun era(year: Int): String =
        when (year) {
            in 710..793 -> "奈良時代"
            in 794..1184 -> "平安時代"
            in 1185..1332 -> "鎌倉時代"
            in 1333..1391 -> "南北朝時代"
            in 1392..1572 -> "室町時代"
            in 1573..1599 -> "安土桃山時代"
            in 1600..1867 -> "江戸時代"
            else -> "範囲外"
        }
```

4.8
```
    fun six(n: Int): String =
        when {
            n % 6 == 0 -> "6 の倍数"
            n % 3 == 0 -> "3 の倍数"
            n % 2 == 0 -> "2 の倍数"
            else -> "3 の倍数でも 2 の倍数でもない整数"
        }
```

第 5 章の解答例

5.1
```
    fun space(s: String): String {
```

```
    var s2 = ""
    for (c in s) {
        s2 = s2 + c + ' '
    }
    return s2
}
```

5.2
```
fun delete(s: String, a: Char): String {
    var s2 = ""
    for (c in s) {
        if (c != a) {
            s2 += c
        }
    }
    return s2
}
```

5.3
```
fun replace(s: String, a: Char, b: Char): String {
    var s2 = ""
    for (c in s) {
        s2 += if (c == a) b else c
    }
    return s2
}
```

5.4
```
fun reduce(s: String): String {
    var s2 = ""
    var spaceflag = false
    for (c in s) {
        if (c == ' ') {
            if (! spaceflag) {
                spaceflag = true
                s2 += c
            }
        } else {
            if (spaceflag) {
                spaceflag = false
            }
            s2 += c
        }
    }
    return s2
}
```

5.5
```
fun maxchar(s: String): Char {
    if (s == "") {
        return '?'
    } else {
        var max = '\u0000'
        for (c in s) {
            if (c > max) max = c
        }
        return max
    }
}
```

5.6
```
fun power(a: Int, b: Int): Int {
    var p = 1
    for (i in 1..b) {
        p *= a
    }
    return p
}
```

5.7
```
fun perfect(n: Int): Boolean {
```

```
    var sum = 0
    for (i in 1..(n - 1)) {
        if (n % i == 0) {
            sum += i
        }
    }
    return n == sum
}
```

5.8
```
fun pcp(p: CharProgression) {
    for (c in p) {
        print("${c} ")
    }
    println()
}
```

5.9
```
fun decomp(n: Int) {
    var m = n
    var d = 2
    while (m > 1) {
        if (m % d == 0) {
            print("${d} ")
            m /= d
        } else
            d++
    }
    println()
}
```

5.10
```
fun power(a: Int, b: Int): Int =
    when {
        b == 0 -> 1
        b >= 0 -> a * power(a, b - 1)
        else -> 0
    }
```

5.11
```
fun paren(n: Int): String =
    if (n > 0) '(' + paren(n - 1) + ')' else ""
```

5.12
```
fun binparen(n: Int): String =
    if (n > 0)
        '(' + binparen(n - 1) + binparen(n - 1) + ')'
    else
        ""
```

5.13
```
fun decomp(n: Int) {
    decomp2(n, 2)
    println()
}

fun decomp2(n: Int, p: Int) {
    if (n >= 2)
        if (n % p == 0) {
            print("${p} ")
            decomp2(n / p, p)
        } else
            decomp2(n, p + 1)
}
```

第 6 章の解答例

6.1
```
fun sum(a: List<Int>): Int {
    var s = 0
    for (n in a) {
        s += n
```

```
        }
        return s
    }
```

6.2
```
fun lastChar(s: String): Char = s[s.length - 1]
```

第 7 章の解答例

7.1
```
class Fortune {
    fun divine() {
        println("あなたの今日の運勢は大吉です。")
    }
}
```

7.2
```
class Counter {
    var n = 0
    fun count() {
        n++
    }
}
```

7.3
```
class Bracket {
    var s = ""
        get() = "[${field}]"
}
```

7.4
```
class Rectangle(val width: Int, val height: Int) {
    val isSquare: Boolean
        get() = width == height
}
```

7.5
```
class Shorten {
    var s = ""
        set(value) {
            field = if (value.length > 10)
                        value.substring(0, 10)
                    else
                        value
        }
}
```

7.6
```
fun Int.countFigure(r: Int): Int = this.toString(r).length
```

7.7
```
fun Set<Int>.toMap(s: String): Map<Int, String> {
    val m: MutableMap<Int, String> = mutableMapOf()
    for (n in this) {
        m.put(n, s)
    }
    return m.toMap()
}
```

7.8
```
val Boolean.int: Int
    get() = if (this) 1 else 0
```

7.9
```
val List<Boolean>.and: Boolean
    get() = this.toSet() == setOf(true)
```

参考文献

[Jemerov,2016] Dmitry Jemerov and Svetlana Isakova, *Kotlin in Action*, Manning Publications, 2016, ISBN 978-1-6172-9329-0. 邦訳（長澤太郎、藤原聖、山本純平、yy_yank）、『Kotlin イン・アクション』、マイナビ出版、2017、ISBN 978-4-8399-6174-9。

[Kotlin,2018] JetBrains, “Kotlin Language Documentation,” JetBrains, 2018.

[金田,2018] 金田浩明、『初めての Android プログラミング・第 3 版』、SB クリエイティブ、2018、ISBN 978-4-7973-9581-5。

[長澤,2016] 長澤太郎、『Kotlin スタートブック：新しい Android プログラミング』、リックテレコム、2016、ISBN 978-4-86594-039-8。

[野崎,2018] 野崎英一、『やさしい Kotlin 入門』、カットシステム、2018、ISBN 978-4-87783-427-2。

[森,2018] 森洋之、『基本からしっかり身につく Android アプリ開発入門：ヤフーの黒帯が本気で伝える大切な基本技』、SB クリエイティブ、2018、ISBN 978-4-7973-9580-8。

索引