■基本的な構文

・変数の宣言

変数を宣言するには、**var命令**を用いる

書き方

```
var 変数名: 変数の型 = 初期値
```

「var」は「variable（変数）」の略

例：Int型の変数nを定義する（初期値10）

```
var n: Int = 10
```

Kotlinの基本のデータ型（よく使う型）は以下のようになっている

型	**概要**
Double	64bit浮動小数点数型
Float	32bit浮動小数点数型
Long	64bit整数型
Int	32bit整数型
Short	16bit整数型
Byte	8bit整数型
Boolean	真偽値
Char	文字型
String	文字列型

初期値があればデータ型を省略できる
以下のように書いてもnはInt型の変数として定義される
浮動小数点数はDouble型とみなされる

```
var n = 10
var m = 1.5
```

このように初期値から型を類推する仕組みを**型推定・型推論**と呼ぶ
初期値とデータ型の**両方を省略することはできない**ことに注意

変数に定義したデータ型以外の型の変数を代入することは出来ない

```
var n = 10
n = 11 //OK
n = "foo" //NG
```

変数に任意の型を持たせたい場合は**Any型**を指定する

```
var n: Any
n = 10 //OK
n = "foo" //OK
```

Kotlinでは**すべてのデータがオブジェクト**である
そしてすべてのデータ型はAny型を継承している

・リテラル表現

リテラルとは？
⇒　ソースコードに書かれた値のこと

リテラル表現とは？
⇒　ソースコード中の値の書き方・表現方法

ここではよく使う数値リテラルと文字列リテラルについて扱う

主な数値リテラルとして以下のようなものがある

```
var a = 100     //10進数
var b = 0x0F    //16進数、「0x」で始める
var c = 0b1010  //2進数、「0b（0B）」で始める
var d = 1.23    //浮動小数点数
var e = 1.23e+5 //浮動小数点数（指数）
```

桁区切り文字としてアンダースコア「_」を利用することもできる

```
var a = 123_456_789
var b = 0xAA_BB_CC
```

以下3行のコードは意味的にすべて同じ
なので、**最も見やすい書き方を採用する**のが重要

```
var a = 1000
var b = 1_000
var c = 0x3E8
println(a)
println(b)
println(c)
```

Kotlinのプログラムにおいて最初に実行される関数がmain関数である
main関数の書き方は定形で、**fun main(args: Array<String>) {...}** というように書く
引数はコマンドライン引数を受け取るための配列

型サフィックスを用いてデータ型を指定することもできる

サフィックスとは？
⇒　接尾辞のこと、文字や数値の後ろに着けて意味を付与する

```
var a = 10L //Long型として定義
var b = 10F //Float型として定義
println(a::class)
println(b::class)
```

サフィックスは大文字小文字を区別しないので「10f」というように書いてもOK
ただし、小文字の「l」は「1」と判別しづらいので禁止されている

Kotlinは2種類の文字列リテラルを持つ
①エスケープ文字を含む文字列（Escaped String）
②改行やタブなど任意の文字列を含む文字列（Raw String）

こんにちは。
私はKotlinを学んでいます。

という2行の文章を出力するときに上記2種類の書き方は以下のようになる

```
var msg1 = "こんにちは。\n私はKotlinを学んでいます。"
var msg2 = """こんにちは。
私はKotlinを学んでいます。"""
println(msg1)
println(msg2)
```

改行を意味する「\n」などを**エスケープ文字（シーケンス）**という
他にもタブを意味する「\t」やバックスペースを意味する「\b」などがある

Raw StringのtrimMargin()メソッドを使うと、
行頭のインデント（字下げ）を除去することができる

```
var msg = """こんにちは。
                    |私はKotlinを学んでいます。""".trimMargin()
println(msg)
```

Kotlinの文字列リテラルの便利な機能として**文字列テンプレート**がある

文字列テンプレートとは？
⇒　文字列に任意の式を埋め込むことのできる機能のこと

```
var data = arrayOf(1, 2, 3)
println("配列dataの先頭の値は${data[0]}で、要素数は${data.size}です。
")
println("1+1は、${1+1}です。")
```

・nullとnull許容型

nullとは？
⇒　値（オブジェクトの参照）を持たないことを意味する値

オブジェクトの参照とは？
値（オブジェクト）はコンピュータのメモリに格納されることで状態を保持できる
オブジェクトごとにメモリ上の住所（メモリアドレス）が与えられる
ひとつの住所にひとつのオブジェクトが入っている状態

変数はメモリアドレスを参照して格納されている値を引き出すための仕組み
変数にオブジェクトの参照が設定されていない状態が**null**である

Javaでは明示的に初期化しない限りnullが初期値になっていたので、
nullの変数のメソッドやフィールドにアクセスして、
NullPointerExceptionを引き起こすことがとても多かった
⇒　そこで、Kotlinでは原則「**nullを許容しない**」ように作られている

```
var foo: String = "foo"
foo = null //エラー、nullを代入できない
```

あえてnullを許容したいときは、型名の後ろに「**?**」を付ける

```
var foo: String? = "foo"
foo = null //OK
```

このような型を**Nullable型（null許容型）**と呼ぶ
Stringに限らずNullable型として定義することができる

Nullable型はnullが入っている可能性があるので取り扱いには注意が必要
特に以下の3点に注意する必要がある
①非null型をNullable型に代入すると暗黙的な型変換（ボクシング）が発生する
②Nullable型を非null型に代入することはできない
③Nullable型のメンバにアクセスする際には「?.（セーフコール演算子）」を使う

①非null型をNullable型に代入すると暗黙的な型変換（ボクシング）が発生する

```
var a: Int = 10000
var b: Int = a
var c: Int? = a
println(a == b) //結果：true
println(a === b) //結果：true
println(a == c) //結果：true
println(a === c) //結果：false
```

変数aを、非null型の変数bと、Nullable型の変数cにそれぞれ代入している

同じ値を持つことを**同値**、同じオブジェクトであることを**同一**という
同値性の比較には「==」、同一性の比較には「===」という演算子を用いる

aとbは同じオブジェクトなので**同値かつ同一**
一方で、aとcは下二行からもわかるように、**同値だが同一ではない**
⇒　暗黙的な型変換（ボクシング）が発生して別のオブジェクトになっている

②Nullable型を非null型に代入することはできない
以下のコードはエラーになる

```
var foo1: String? = "foo"
var foo2: String = foo1 //エラー
```

では以下のコードはどうなるか？

```
var a1: String? = "a"
var a2: Any = a1
```

実行してみるとエラーになる
Any型もnullを許容しないことに注意
Nullable型である「Any?」に変更するとエラーが解消される

Any型を指定すると任意の型を代入できるという話をしたが、
厳密には「Any型を指定すると任意の**非null**型を代入できる」、
かつ「**Any?型**を指定すると**null型を含む**任意の型を代入できる」と言える

③Nullable型のメンバにアクセスする際には「?.（セーフコール演算子）」を使う

```
var a: String? = "foo"
println(a?.length) //結果：3

var b: String? = null
println(b?.length) //結果：null
```

nullの場合に返す既定値を設定するには「**?:**」演算子を使う

```
var b: String? = null
println(b?.length ?: 0) //結果：0
```

Nullable型を非null型に変換するには「**!!**」演算子を使う

```
var a: String? = "foo"
println(a!!.length) //結果：3
```

ただし、上記の例でaがnullだった場合にはNullPointerExceptionが発生するので、
Nullable型の値がnullでないことが保証されている際にしか利用してはいけない

・型変換

型同士に互換性があるのであれば、必要に応じて型を変換することもできる

Javaでは**ワイドニング**が自動で行われるが、Kotlinでは明示的に変換する必要がある

ワイドニングとは？
⇒　互換性がある型において、値範囲の狭い型から広い型への変換のこと

例えば、Float型とDouble型を比べたときにDouble型の方が範囲が広いため、
Float型の数値をDouble型の数値として扱っても問題がないので変換が可能

Javaではワイドニングを自動的に行ってくれたが、
Kotlinでは明示的に変換する必要がある

以下のように書くとどちらもエラーになる

```
var a: Float = 1.2
var b: Double = 10
```

以下のように書く必要がある

```
var a: Float = 1.2f
var b: Double = 10.0
```

上記のようなサフィックス、もしくは**toデータ型()メソッド**を用いて変換する

```
var a = 10
var b: Long = a.toLong()
```

toLong()以外にも、toByte()やtoFloat()など変換先の型ごとにメソッドがある

・配列とコレクション

Kotlinで複数の要素をまとめて扱うには、Javaと同様に配列とコレクションを用いる

配列と代表的なコレクション（List・Set・Map）は以下のような特徴がある
・配列
最もシンプルな複数の要素を扱う仕組みだが、宣言の段階でサイズを決める必要があるなど柔軟性に欠ける
・List
配列と同様に複数の要素を扱うことができる、インデックスを指定して値を取得・設定ができる
・Set
Listと似ているが重複した要素は登録しないので、重複のない複数の要素を扱うことが出来る、順序性はないのでインデックスを指定した値の取得・設定は出来ない
・Map
キーと値を利用して要素を扱う、他の言語では連想配列・辞書（ディクショナリ）と呼ばれることもある

配列

```
var a = arrayOf(1, 2, 3) // [1, 2, 3]
var b = intArrayOf(1, 2, 3) // [1, 2, 3] 要素はInt型
var c: Array<String?> = arrayOfNulls(3) // [null, null, null]
var d = Array(3, {i -> i * 2}) // [0, 2, 4]
println(a[1]) //結果：2
```

arrayOfNulls()関数を指定した要素数の空の配列を作成できる（3行目）
Kotlinの配列はArrayクラスで表現される
「Array<String?>」は、「要素がString?型の配列」を意味する
要素にnullが入るので明示的にNullable型を指定する必要がある

Arrayクラスのコンストラクタを使った書き方もある（4行目）
第1引数は要素数、第2引数は要素の式（ラムダ式、後ほど説明）
iがインデックスを表すので、インデックスがiの要素の値はi*2という宣言をしている

コレクション
各種コレクションは以下のように生成できる

```
var list = listOf("あ", "い", "う")
var set = setOf("A", "B", "A", "C", "D", "B")
var map = mapOf("First" to 1, "Second" to 2, "Third" to 3)
println(list) //結果：[あ, い, う]
```

```
println(set) //結果：[A, B, C, D]
println(map) //結果：{First=1, Second=2, Third=3}
```

このようにコレクションを作成すると読み取り専用になるので、
変更可能なコレクションを作成するには、mutableコレクション名Of()関数を使う

```
var list = mutableListOf(1, 2, 3)
list[2] = 4
println(list) //結果：[1, 2, 4]
```

通常のListがimutable（変更不可能）なわけではないことに注意
以下のような記述をすれば、Listも変更される

```
var list1 = mutableListOf(1, 2, 3)
var list2: List<Int> = list1
list1[2] = 4
println(list2) //結果：[1, 2, 4]
```

・定数

変数のように値に名前を与えるが、一度定義したら値が変わらないもの
定数を宣言するには、**val命令**を用いる（変数は、var命令）

定数を変更しようとするとエラーになる

```
val a = 10
a = 11 //エラー
```

配列の場合は以下のような振る舞いになる

```
val a = arrayOf(1, 2, 3)
a = arrayOf(4, 5, 6) //エラー
a[2] = 4 //OK
```

2行目で配列そのものを変更しようとするとエラーになるが、
3行目のように配列の**要素を変更することは可能**
⇒　再代入が禁止されているイメージ

varとvalの使い分けは？
再代入する予定のないものは、**valを利用するのが基本**
変化する可能性があるものが増えると、管理が難しくなり可読性を損なう

■演算子・制御構文

・演算子

四則演算や比較などの、値の計算処理を行うための記号
四則演算など基本的なものは他のプログラミング言語と同様だが、
Kotlinの特徴的なところについて見ていく

比較演算子
null許容型のところで登場した「==」と「===」など

範囲演算子
「m..n」というように書くと、m~nの範囲を表現することができる

```
val i = 10
println(i in 1..20) //結果：true
```

・制御構文

条件分岐や繰り返し処理などの制御構文について見ていく

if式
条件分岐は「**if式**」を使って表現する

基本的な書き方は以下のようになる

```
val a = 10
if (a <= 5) {
  println("aは5以下です")
} else if (a <= 10) {
  println("aは10以下です")
} else {
  println("aは10より大きいです")
}
```

elseとelse ifは省略可能、{ }の中には複数行の記述も可能

Kotlinにおいて**「if」は「式」であるので、値を返すことができる**

```
val a = 10
var msg = if (a <= 5) {
  "aは5以下です"
} else if (a <= 10) {
  "aは10以下です"
} else {
  "aは10より大きいです"
}
println(msg) //結果：aは10以下です
```

ifを式として用いた場合は何らかの値を返す必要があるので、**elseは省略できない**
{ }の中には複数行の記述も可能、その場合最後に書いた値を戻り値とみなす
処理を一文しか書かないならば、{ }は省略可能なので以下のようにも書ける

```
val a = 10
var msg = if (a <= 5) "aは5以下です" else if (a <= 10) "aは10以下です" else  "aは10より大きいです"
```

when式
式の値に応じて処理を分岐させるには「**when式**」を使う

```
val x = 1
when (x) {
  1 -> println("xは1です")
  2 -> println("xは2です")
  else -> {  // ブロックとして書くと複数行処理を書ける
    println("xは1でも2でもないです")
  }
}
```

一般化すると以下のようになる

```
when（チェックする変数・式）{
  条件1 -> 条件1を満たしたときの処理
  条件2 -> 条件2を満たしたときの処理
  else -> どの条件も満たさなかった時の処理
  }
}
```

when式もif式と同様に式として扱うことが出来る（この場合elseは必須）

条件の箇所はいろいろな書き方ができる

```
val score = 80
val msg = when (score) {
  100,99 -> "素晴らしい" // 複数の値
  in 60..98 -> "合格です" // 範囲を指定
  !in 60..100 -> "不合格です" // 指定範囲にない
  else -> "不正な点数です"
}
```

is演算子を利用することによって、
チェックする変数（式）の**型で処理を分岐**させることもできる

```
val obj: Any = "あいうえお"
when (obj) {
  is String -> println("文字数は${obj.length}です")
  else -> println("String型ではない型です")
} //結果：文字数は5です
```

変数objはAny型の定数であるのに、lengthにアクセスできていることに注目
以下のように書くとAny型はメンバにlengthを持たないのでエラーになってしまう

```
val obj: Any = "あいうえお"
println("文字数は${obj.length}です")
```

スマートキャストとは？
型チェックが行われたあとのブロックでは、該当する変数をその型として扱うことができる機能のこと、暗黙的にキャスト（型変換）を行う機能ともいえる

上記の例では、is演算子によってobjは型チェックを行われたので、
その後のobjはString型として扱えるようになった
⇒　String型のプロパティである、lengthにアクセスすることが出来ている

引数を取らずに宣言すれば、if式の代替としても使える

```
val a = 10
when {
  a <= 5 -> println("aは5以下です")
  a <= 10 -> println("aは10以下です")
  else ->  println("aは10より大きいです")
}
```

このほかにも関数なども条件として書くことができる

forループ
反復処理を行うにはforループを用いる
以下のように書くと配列arrの先頭からひとつずつ要素を取り出していく

```
val arr = arrayOf(1, 2, 3)
for (item in arr) {
  println(item)
}
```

Mapにも使用できる

```
val map = mapOf("First" to 1, "Second" to 2, "Third" to 3)
for ((key, value) in map) {
  println("${key}:${value}")
}
```

配列でも**withIndex()メソッド**を用いて、インデックスと値をセットで取得できる

```
val list = listOf("あ", "い", "う")
for ((index, value) in list.withIndex()){
  println("${index}:${value}")
}
```

範囲演算子を用いれば指定した回数繰り返すことも可能

```
for (i in 1..3) {
 println(i)
} //結果：1、2、3
```

範囲演算子を使うと3が含まれる（1 <= i <= 3）が、
3を含めたくない場合（1 <= i < 3）は、**until**を用いる

```
for (i in 1 until 3) {
 println(i)
} //結果：1、2
```

範囲演算子やuntilを使うとインクリメント（1ずつ増加）されるが、
デクリメント（1ずつ減少）させるには**downTo**を用いる

```
for (i in 3 downTo 1) {
 println(i)
} //結果：3、2、1
```

デフォルトでは1ずつ増加・減少するが、**step**で増分・減分を指定することもできる

```
for (i in 1..10 step 3) {
 println(i)
} //結果：1、4、7、10
```

ループを中断するには**break**、周回をスキップするには**continue**を用いる

```
for (i in 1..10) {
  if (i % 3 == 0) continue
  println(i)
  if (i == 9) break
} //結果：1、2、4、5、7、8
```

breakとcontinueは現在のループと周回を脱出するので、以下のようになる

```
for (i in 1..3) {
  for (j in 1..3) {
    if (i * j > 5) break
    print("${i * j} ")
  }
  println()
}

結果
1 2 3
2 4
3
```

「i * j」が5より大きくなった時点で、内側のループだけでなく、
外側のループも終了させるには、**ラベル構文**を用いる

```
outer@ for (i in 1..3) {
  for (j in 1..3) {
    if (i * j > 5) break@outer
    print("${i * j} ")
  }
  println()
}

結果
1 2 3
2 4
```

①「ラベル名@」をforの前に付ける、ラベル名は任意
②break/continueの後ろに「break@ラベル名」というように指定する

while / do while ループ
条件式を満たす間は実行するwhileループもある

```
var i = 1
while (i < 2) {
  println(i)
  i ++
}
// 結果：1
```

do whileループは条件式の判定が処理の後になる

```
var j = 2
do {
  println(j)
  j ++
} while (j < 2)
// 結果：2
```

do whileループはパスワードの入力などに使える

```
do {
  パスワードの入力をしてもらう
} while（パスワードのチェック）
```

■関数

KotlinではJavaとは異なり、クラス内だけでなくトップレベルでも宣言できる

・関数の基本

関数を定義するには**fun**を使用する、基本的な書き方は以下のようになる

```
//関数の定義
fun 関数名（仮引数: 引数の型,...): 戻り値の型 {処理}

//関数の呼び出し
関数名(実引数,...)
```

例：三角形の面積を求めるgetTriangleArea関数

```
fun getTriangleArea (width: Double, height: Double): Double {
  return width * height / 2
}
println(getTriangleArea(10.0, 11.1)) //結果：55.5
```

仮引数と戻り値の型宣言は必須であることに注意
戻り値がない場合は戻り値の型として、**Unit**を指定する

```
fun sayHello (name: String): Unit {
  println("Hello, ${name}.")
}
sayHello("Masao")
```

Unit型だけは例外的に省略が可能なので、以下のように書いてもOK

```
fun sayHello (name: String) {...}
```

Kotlinでは関数内に関数を定義することもできる（ローカル関数）

単一式関数
関数が単一の式を返す場合は以下のように書くことができる

```
fun getTriangleArea (width: Double, height: Double): Double =
width * height / 2
```

コンパイラが戻り値の型を推論できる場合には、戻り値の型を省略可能

```
fun getTriangleArea (width: Double, height: Double) = width *
height / 2
```

・引数と戻り値の表現方法

引数のデフォルト値
引数が省略されたときに使用されるデフォルト値を設定することもできる
デフォルト値を設定するには「仮引数=デフォルト値」の形で記述する

```
fun getTriangleArea(width:Double=1.0, height:Double=1.5): Double {
  return width * height / 2
}
println(getTriangleArea()) //結果：0.75（= 1.0 * 1.5 / 2）
println(getTriangleArea(3.0)) //結果：2.25（= 3.0 * 1.5 / 2）
```

名前付き引数
名前付き引数を使えば、どの引数にどの値を渡すかを明示的に指定できる

```
fun getTriangleArea(width:Double=1.0, height:Double=1.5): Double {
  return width * height / 2
}
// 引数の順番を入れ替え
println(getTriangleArea(height = 3.0, width = 2.0)) //結果：3.0
// 二個目の引数にだけ値を渡す
println(getTriangleArea(height = 3.0)) //結果：1.5
```

名前付き引数には以下のようなメリットがある
・引数の数が増えてもどの引数になにを渡しているかがわかりやすい
・定義時の引数の順番による縛りがないので、呼び出しやすい・使いやすい

名前付き引数を扱うときの注意点・間違えやすいところ
名前付き引数と通常の実引数は混在させることができるが、扱いには注意が必要
名前付き引数は通常の実引数の後方に記述する必要がある

```
fun getTriangleArea(width:Double=1.0, height:Double=1.5): Double {
  return width * height / 2
}
println(getTriangleArea(3.0, height = 2.0)) //結果：3.0
// heightにわたすつもりで以下のように書くとエラー、1.0はwidthに渡されてしまう
println(getTriangleArea(width = 3.0, 1.0)) //結果：エラー
```

また、以下のようなwidthだけは省略可能な関数を考える

```
fun getSquareArea (width: Double = 1.0, height: Double): Double {
  return width * height
}
// このように呼び出すとエラー
println(getSquareArea(2.0)) //結果：エラー
```

必須の引数（必須引数）と省略可能な引数（任意引数）があったときは、
必須引数を先頭に、**任意引数を後方**に並べて定義するのが良い

```
fun getSquareArea (width: Double, height: Double = 1.0): Double {
  return width * height
}
```

Kotlinで**可変長引数**を表現するには**varargキーワード**を用いる
可変長引数は内部的には配列とみなされるので、forループによる処理が可能

```
fun allSum(vararg values: Int): Int {
  var result = 0
  for (value in values) {
    result += value
  }
  return result
}
println(allSum(2, 3, 1, 7)) //結果：13
```

スプレッド演算子（*）を使えば可変長引数に配列を渡すこともできる

```
fun allSum(vararg values: Int): Int {
  var result = 0
  for (value in values) {
    result += value
  }
  return result
}

val arr = intArrayOf(2, 3, 1, 7)
//スプレッド演算子を使えば配列を可変長引数に渡せる
println(allSum(*arr)) //結果：13
// 配列をそのまま渡すことはできないの以下のように書くとエラー
println(arr)
// 可変長引数の一部として配列を渡すことも可能
println(allSum(4, *arr, 3)) //結果：20
```

関数から**複数の戻り値**を返すことも可能
Kotlinでは**Pair（2値）**と**Triple（3値）**を利用することで実現できる

```
// Pair型の戻り値を定義した関数
fun getSumAverage(vararg values: Int): Pair<Int, Double> {
  var result = 0
  var count = 0.0
  for (value in values) {
    result += value
    count ++ //インクリメント
  }
  return Pair(result, result / count) // Pairクラスのコンストラクタに2値を渡す
}

fun main(args: Array<String>) {
    val (sum, average) = getSumAverage(3, 4, 8, 1)
    println(sum)
    println(average)
}
```

①戻り値の型を、Pair<Int, Double>としている
2つある戻り値の1つ目の値がInt型、2つ目の値がDouble型であることを表している

②戻り値を、Pair(result, result / count)としている
Pairクラスのコンストラクタに、戻り値とする2値を渡している

③getSumAverage関数の戻り値を(sum, average)という形で受け取っている
Pairの2値をsum、averageにそれぞれ代入・定義している（**分解宣言**）

Tripleの場合も同様にして3つの戻り値を返すことができる

複数の戻り値のうち一部だけ使用できれば良い場合は以下のように書く

```
val (sum, _ ) = getSumAverage(3, 4, 8, 1)
```

このように書くと平均値は使われず、合計値だけを使うことができる
「_」という名前の変数が定義されるわけではないことに注意

Kotlinにおいて割り算の結果をDouble型にするには、
割る数・割られる数のどちらかをDouble型にする必要がある

・高階関数

引数として関数を取ったり、戻り値として関数を返す関数を**高階関数**と呼ぶ
高階関数の一例としてforEachメソッドなどがある
配列の要素を一つずつ取り出して引数の関数に渡していくメソッドである

```
fun hyoji(n: Int) {
  println(n)
}

val arr = arrayOf(1, 2, 3, 4)
arr.forEach(::hyoji) //結果：1、2、3、4
```

関数（メソッド）の引数に関数を渡すには、
「::」演算子を用いて、「**::関数名**」というように渡す

このように書くと、配列arrの先頭から一つずつ要素が取り出され、
forEachメソッドの引数であるhyoji関数の引数として渡されていく
hyoji(1)→hyoji(2)→hyoji(3)→hyoji(4)というように順々に呼び出されている

高階関数の引数として渡される関数（上記ではhyoji関数）は、
この場でしか使用されないケースが多い
高階関数に渡すためだけの使い捨ての関数を宣言するのは無駄が多いので、
そういった場合は**匿名関数**（名前のない関数）を使うほうが便利
匿名関数の書き方として、**ラムダ式**を使うのが主流

ラムダ式の基本的な書き方は以下のようになる

```
{ 引数 -> 関数の本体(処理) }
```

上記の例をラムダ式で書き換えると以下のようになる

```
val arr = arrayOf(1, 2, 3, 4)
arr.forEach({ n: Int -> println(n) })
```

引数の型が推論できる場合は省略して以下のようにも書ける

```
arr.forEach({ n -> println(n) })
```

高階関数の最後の引数がラムダ式なら、ラムダ式を高階関数のカッコの外に出せる

```
arr.forEach() { n -> println(n) }
```

また、高階関数の唯一の引数がラムダ式なら、高階関数のカッコを省略可能

```
arr.forEach { n -> println(n) }
```

ラムダ式の引数が単一なら、その引数を暗黙的な引数itで受け取ることができる

```
arr.forEach { println(it) }
```

省略形はAndroid Studioのリファレンスなどでも多用されているので要理解

ラムダ式中でreturnを呼び出すと、
ラムダ式ではなく直上の関数を抜けてしまうことに注意

以下のように書くとmain関数を抜けてしまうので、
「ラムダ式を終了しました」と表示されない

```
fun main(args: Array<String>) {
  val arr = arrayOf(1, 2, 3, 4)
  arr.forEach {
    if (it == 3) return
    println(it)
  }
  println("ラムダ式を終了しました")
}

結果
1
2
```

ラムダ式だけを抜けるにはラベル構文を用いる

```
val arr = arrayOf(1, 2, 3, 4)
arr.forEach loop@ {
  if (it == 3) return@loop
  println(it)
}

結果
1
2
4
終了しました
```

このようにラベル構文は、break/continue/returnなどにおいて、
Kotlinのデフォルトの設定とは違う場所へジャンプしたいときに使える

ラムダ式を引数にとる高階関数の名前をラベルに指定しても良い

```
val arr = arrayOf(1, 2, 3, 4)
arr.forEach {
  if (it == 3) return@forEach
  println(it)
}
```

高階関数はもちろん自分で定義することも可能
関数を引数にするときの型宣言は以下のように書く

```
仮引数:（受け取る関数の引数の型, ...） -> 受け取る関数の戻り値の型
```

例：処理時間を計測するbenchmark関数

```
fun benchmark(unitStr: String, func: ()-> Unit): String {
  val start = System.currentTimeMillis()
  func()
  val end = System.currentTimeMillis()
  return (end - start).toString() + unitStr
}

val time = benchmark("ミリ秒") {
  var x = 0
  for (i in 1..1_000_000_000) {
    x++
  }
}

println("処理時間：" + time)
```

■オブジェクト指向構文

・クラスの基本

Kotlinでクラスを定義するには以下のように書く

```
class Human {
  var name = "名無し"
  var age = 20

  fun intro() {
    println("私の名前は${name}です。${age}歳です。")
  }
}

val human = Human()
human.intro() //結果：私の名前は名無しです。20歳です。
```

インスタンスの生成は関数を呼び出すような形で「クラス名()」と書く
インスタンスのメンバーにアクセスするには、「.」を使えばアクセスできる

アクセス修飾子を利用することもできる
⇒　変数やメソッドのスコープを定義する機能

```
class Human {
  private var name = "名無し"
  var age = 20

  protected fun intro() {
    println("私の名前は${name}です。${age}歳です。")
  }
}
```

基本的なアクセス修飾子としては以下がある
Javaのアクセス修飾子とは似て非なるものなので注意！

public：すべてのクラスからアクセス可能
protected：定義されたクラスとそのサブクラスからのみアクセス可能
internal：同じモジュール内のクラスからのみアクセス可能
private：定義されたクラスからのみアクセス可能

モジュールとは？
⇒　Kotlin のファイルが一体としてコンパイルされるまとまりのこと
　　EclipseやAndroid Studioなどのアプリ開発であればプロジェクト単位

Javaで書かれたプログラムが実行されるまでの流れ
①複数のJavaファイルがまとめてコンパイルされてクラスファイルになる
②Java Virtual Machine（JVM）がクラスファイルに記載されたコードを解釈・実行

クラスファイルとは？
⇒　ソースコードを実行する過程で生成される中間コード、OSに依存しない

JVMとは？
⇒　クラスファイルをOSごとに実行可能なコードに変換・実行するソフトウェア

コンパイルして生成されたクラスファイルがJVM上で動く言語がJava以外にもあり
それらのことをJVM言語と呼んだりする、もちろんKotlinもJVM言語のひとつ

・プロパティ

Kotlinでは他の言語のようなフィールド（クラス内で定義される変数）が使えない
フィールドの代わりにKotlinには、**プロパティ**という機能がある
内部的にはフィールドと異なるが、基本的にはフィールドのように使える（前例）

プロパティはただ単にクラスが持つ変数ではなく、それ自体が**アクセサーを持つ**

アクセサーとは？
⇒　値の取得・設定を行うシンプルなメソッドのこと。getter(取得)/setter(設定)

プロパティの定義の基本的な書き方は以下

```
var プロパティ名: データ型 = 初期値
  getter関数
  setter関数
```

getterとsetterはいずれも省略可能
上記の例ではどちらも省略された形だった

アクセサーを定義すると以下のようになる

```
class Human {
  var name = "名無し"
  var age = 20
    set(value) {
      if (value < 0) {
        println("年齢が不正です。")
      } else {
      field = value
      }
    }

  fun intro() {
    println("私の名前は${name}です。${age}歳です。")
  }
}

val human = Human()
human.intro() //結果：私の名前は名無しです。20歳です。
human.age = -1 // 年齢が不正です。
println(human.name) //結果：名無し
```

「human.age = -1」というように書くと、ageプロパティのsetterが呼び出され
「human.name」というように書くと、nameプロパティのgetterが呼び出される
⇒　変数にアクセスしているようだが、アクセサー（メソッド）を呼び出している
⇒　デフォルトの実装によりフィールドにアクセスするような振る舞いになる

アクセサーの引数は慣例的にvalueを用いる
fieldは**バッキングフィールド**と呼ばれる
⇒　プロパティの値を格納するために裏側で自動生成されるフィールド
⇒　アクセサー内部でのみ使用可能なフィールド

valを使ってプロパティを宣言すると**読み取り専用のプロパティ**になる
⇒　setterを使用することはできない、getterは可能

private set とするとクラス外では読み取り専用、クラス内からは変更可能というように、アクセサのスコープを変更することもできる（setterのみ可能）

・コンストラクタ

Kotlinのクラスは、**プライマリコンストラクタ**と
0個以上の**セカンダリコンストラクタ**を持つことができる

プライマリコンストラクタ
クラスにひとつだけ記述できるコンストラクタで、以下のように書く

```
class クラス名 constructor(引数: データ型) {}
```

先ほどのHumanクラスにプライマリコンストラクタを使うと以下のようになる

```
class Human constructor(name: String, age: Int) {
  var name: String
  var age: Int

  init {
    this.name = name
    this.age = age
  }

  fun intro() {
    println("私の名前は${this.name}です。${this.age}歳です。")
  }
}

val taro = Human("太郎", 10)
taro.intro() //結果：私の名前は太郎です。10歳です。
```

initブロックに具体的な初期化処理を記述する
initブロックではプライマリコンストラクタの引数にアクセスできる

プロパティの名前と引数の名前が同じ場合、プロパティ名の前に「**this.**」をつけて
プロパティを指定していることを明確にする必要がある

constructorキーワードは、プライマリコンストラクタがアノテーションや
アクセス修飾子を持たない場合は以下のように省略可能、こちらの方が一般的

```
class Human(name: String, age: Int) { }
```

コンストラクタの引数にvar/valを付けることで、
プロパティの宣言と初期化を同時に行うこともできる

```
class Human(var name: String, var age: Int) {
  fun intro() {
    println("私の名前は${this.name}です。${this.age}歳です。")
  }
}
```

このように書いた場合アクセス修飾子も付与できる

```
class Human(private var name: String, private var age: Int) { }
```

セカンダリコンストラクタ
ひとつのクラスに2つ以上コンストラクタを定義する際、
ただ一つのプライマリコンストラクタ以降は**セカンダリコンストラクタ**として定義

```
class Human(var name: String, var age: Int) {
  constructor(name: String):this(name, 10) {} // 一つ目
  constructor():this("太郎") {} // 二つ目
  fun intro() {
    println("私の名前は${this.name}です。${this.age}歳です。")
  }
}

val human1 = Human("三郎", 5)
val human2 = Human("次郎")
val human3 = Human()
human1.intro() // 結果：私の名前は三郎です。5歳です。
human2.intro() // 結果：私の名前は次郎です。10歳です。
human3.intro() // 結果：私の名前は太郎です。10歳です。
```

一つ目のセカンダリコンストラクタでは、ageが省略された場合、
二つ目のセカンダリコンストラクタでは、nameとageのどちらも省略された場合の
初期化の処理を記述している

「**:this(実引数)**」という記述がセカンダリコンストラクタにあるが、
これは**他のコンストラクタを呼び出す**ための記述

処理の流れ
①「Human()」というように引数をすべて省略してインスタンス生成
②引数がないので二つ目のセカンダリコンストラクタが呼び出される
③「:this("太郎")」から、一つ目のセカンダリコンストラクタが呼び出される
④nameに「"太郎"」が渡され、「:this(name, 10)」からプライマリコンストラクタが呼び出される
⑤プライマリコンストラクタの引数nameには「"太郎"」、ageには「10」という値が渡されて、プロパティが初期化される

プライマリコンストラクタが存在する場合、セカンダリコンストラクタは
最終的に必ずプライマリコンストラクタを呼び出す必要があることに注意

関数と同じくコンストラクタにおいても、引数に既定値を設定できる
上記の例は既定値を使えば以下のようにシンプルに書ける

```
class Human(var name: String = "太郎", var age: Int = 10) {
  fun intro() {
    println("私の名前は${this.name}です。${this.age}歳です。")
  }
}

val human1 = Human("三郎", 5)
val human2 = Human("次郎")
val human3 = Human()
human1.intro() // 結果：私の名前は三郎です。5歳です。
human2.intro() // 結果：私の名前は次郎です。10歳です。
human3.intro() // 結果：私の名前は太郎です。10歳です。
```

セカンダリコンストラクタの使いどころとしては、
より柔軟な記述を可能にしたり、Javaとの相互運用性を高める際に使える

■継承・インターフェース

・継承

継承とメソッドのオーバーライドを使ったコードの例は以下のようになる

```
open class Human(var name: String) {
  open fun intro() {
    println("私の名前は${this.name}です。")
  }
}

class PerfectHuman(name: String, var place: String): Human(name) {
  override fun intro() {
    println("${this.name}! ${this.name}! I'm a perfect human.")
    super.intro()
  }
  fun liveIn() {
    println("We live in ${this.place}.")
  }
}

val nakata = PerfectHuman("Nakata", "Tokyo")
nakata.liveIn()
nakata.intro()
```

継承とオーバーライドを可能にするには、**open修飾子**を使う必要がある

継承を行うには「: 基底クラス(基底クラスのコンストラクタに渡す引数)」と書く
PerfectHumanクラスは引数としてnameとplaceを受け取る
placeはPerfectHumanクラスのプライマリコンストラクタで初期化され、
nameはHumanクラスのプライマリコンストラクタに渡されてそこで初期化される

メソッドをオーバーライドするには**override**を使う
基底クラスのメソッドを派生クラスから呼び出すには、
「super.メソッド名()」というように呼び出す

オーバーライドを強制するためには**抽象クラス**と**抽象メソッド**を用いる
抽象メソッド　⇒　派生クラスでのオーバーライドを強制するメソッド
抽象クラス　　⇒　抽象メソッドを定義できるクラス
ポリモーフィズムを実現するためには必須の機能（拙講座参照）

```
abstract class Human(var name: String) {
  abstract fun intro()
}

class TekitoHuman(name: String): Human(name) {
  override fun intro() {
    println("どうも、レオナルド・ディカプリオです。")
  }
}

val takada = TekitoHuman("高田")
takada.intro()  //結果：どうも、レオナルド・ディカプリオです。
```

上記のように書くと、Humanクラスの派生クラスにintro()メソッドを
オーバーライドすることを強制することができる

抽象メソッドには、**具体的な実装を行うことが出来ない**ことに注意
空のブロックを書いているだけでエラーになる

・インターフェース

Kotlinでは多重継承（複数のクラスを継承した派生クラスの定義）が禁止されている
この性質を**単一継承**という

単一継承が問題となるケースもある、以下のようなケースについて考えてみよう

HogeAとHogeBという2つのクラスがある
それぞれのクラスにおいて抽象メソッドによって、
HogeAクラスはhoge1メソッドとhoge2メソッド、
HogeBクラスはhoge2メソッドとhoge3メソッドの実装を強制したいとする

hoge2メソッドが共通なのでこれらをまとめようとすると、
単一継承のしばりによって以下のような構成にする必要がある

```
abstract class Hoge {
  abstract fun hoge1()
  abstract fun hoge2()
  abstract fun hoge3()
}
class HogeA: Hoge() {
  override fun hoge1() {}
```

```
  override fun hoge2() {}
  override fun hoge3() {}
}
class HogeB: Hoge() {
  override fun hoge1() {}
  override fun hoge2() {}
  override fun hoge3() {}
}
```

だが上記のコードだと本来HogeAクラスには必要なかったhoge3メソッドと、
HogeBクラスに必要のないhoge1メソッドまで実装を強制されている
このように単一継承だとサブクラスに**必要なメソッドすべてをまとめておく**必要があり、サブクラスは**不要なメソッドの実装も強制されてしまう**という問題がある
⇒　**インターフェース**を使う

インターフェースは抽象クラスと同様に抽象メソッドを持たせることができる
インターフェースは「継承」ではなくて、「**実装**」するが意味はほとんど同じ

インターフェースを用いて上記のコードを書き直すと以下のようになる

```
interface Hoge1 {
  fun hoge1()
}

interface Hoge2 {
  fun hoge2()
}

interface Hoge3 {
  fun hoge3()
}

class HogeA: Hoge1, Hoge2 {
  override fun hoge1() {}
  override fun hoge2() {}
}

class HogeB: Hoge2, Hoge3 {
  override fun hoge2() {}
  override fun hoge3() {}
}
```

インターフェース定義のための基本的な書き方は以下

```
interface インターフェース名 {定義}
```

インターフェースを実装するには継承と同じように「**:インターフェース名**」と書く
クラスの場合はひとつのクラスしか継承できなかったが、
インターフェースは複数を実装することが出来る（カンマで区切って続けて書く）
これで、メソッドの実装を強制しつつ不要なメソッドは実装する必要がなくなる
Pythonでは多重継承が許されているので、インターフェースは用意されていない

継承と実装を同時に行うことも可能

```
class Child : Parent(), MyInterface {...}
```

インターフェースは抽象クラスとは以下のような違いがある
・コンストラクタが持てない
・プロパティはカスタムアクセサーを持つものか、抽象プロパティしか持てない
・abstractやopen修飾子は不要（自明なので、あってもエラーにはならない）

インターフェースのメソッドがデフォルトの実装を持つ場合、
オーバーライドは強制されないことに注意

```
interface MyInterface {
  fun bar()
  fun foo() = println("foo")
}

class Bar: MyInterface {
  override fun bar() = println("bar")
}
```

インターフェースは多重継承（実装）が可能なため、実装するインターフェースが同名のメソッド、かつデフォルトの実装が異なるメソッドを持つ場合が起こりうる
このように名前が衝突した場合、実装クラスでどのような呼び出すかを明示的に指定する必要がある

```
interface Hoge1 {
  fun hoge() = println("hoge1")
}

interface Hoge2 {
  fun hoge() = println("hoge2")
```

```
}

class Myclass: Hoge1, Hoge2 {
  override fun hoge() {
    super<Hoge1>.hoge()
    super<Hoge2>.hoge()
  }
}
```

「**super<インターフェース名>**」と書くことで、
特定のインターフェースを指定できる

Kotlinではインターフェース中にプロパティを定義できる

```
interface FooInterface {
  var foo: String
  fun foo() = println(foo)
}

class Foo: FooInterface {
  override var foo = "foo"
}

val foo = Foo()
foo.foo() //結果：foo
```

インターフェースは、プロパティは定義できるが状態は持てないので、以下に注意
・abstractであるかアクセサーを持つ必要がある
・バッキングフィールド（field）を持てない

上記ではabstractな変数fooをFooクラスでオーバーライドしている

カスタムアクセサーを定義する際バッキングフィールドを持てないので、
他のプロパティを参照したりすることでアクセサーを記述することになる

```
interface MyInterface {
  var price: Int
  val taxIncludePrice: Int
    get() = (this.price * 1.1).toInt()
  fun checkPrice() {
    println("税込み${taxIncludePrice}円です")
```

```
  }
}

class Price: MyInterface {
  override var price = 100
}

val price = Price()
price.checkPrice()
```

・型変換

型同士が継承・実装の関係にある場合は、相互に型変換（キャスト）が可能

```
open class Person {}
class BusinessPerson: Person() {
  fun work() = println("Working")
}

val p: Person = BusinessPerson() // アップキャスト
if (p is BusinessPerson) { // ダウンキャスト
  p.work() // スマートキャスト
}
```

派生クラス⇒基底クラスのキャストを**アップキャスト**、
基底クラス⇒派生クラスのキャストを**ダウンキャスト**と呼ぶ

アップキャストはいつでも可能なのに対して、ダウンキャストはできない場合あり
⇒　派生クラス独自のメンバがある場合はできない
⇒　is演算子によるチェックが必要

■特殊なクラス・オブジェクト

・データクラス

オブジェクト指向で設計・開発をしていると、
処理を持たないデータだけを保持するクラスが生まれることもある

そういったクラスを表現するための仕組みが**データクラス**である
データクラスはclassの前に「**data**」というキーワードをつけて定義する

以下、プロパティにnameとageを持つMemberというデータクラスを定義している

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

データだけを保持するクラスは、必要な記述がプライマリコンストラクタで
事足りるため、本体ブロックは省略できることに注目

データクラスを定義する際には以下の条件を満たす必要がある
・プライマリコンストラクタが一つ以上の引数を持つ
・コンストラクタの引数はすべてvar/valを付与してプロパティの宣言とする
・abstract / open / sealed / innerにすることはできない

Kotlinのデータクラスには、データだけを保持するクラスを定義する際に
よく使われるメソッドがいくつか予め用意されている
equals / toString / componentN / copy という4つのメソッドを紹介する

equalsメソッド（同値性のチェック）
⇒　プライマリコンストラクタで定義されたプロパティが等しいかを確認する

```
data class Member(var name: String = "") {
  var age: Int = 0
}

val m1 = Member("佐藤")
m1.age = 30
val m2 = Member("佐藤")
m2.age = 22
println(m1 == m2)  //結果：true
println(m1)  //結果：Member(name=佐藤)
```

ageプロパティはプライマリコンストラクタで定義されていないので、
ageプロパティの値が同じでなくてもtrueを返している

toStringメソッド（文字列化）
⇒　文字列表現、println関数を呼び出したときにも暗黙的に呼び出されている

データクラスのオブジェクトを文字列化すると、
「**データクラス名(プロパティ1=値, ...)**」というように出力される

equalsメソッドと同様に、プライマリコンストラクタで定義されていない
プロパティは出力されていないことに注目
⇒　データクラスに用意されるメソッドは、
　　プライマリコンストラクタで定義されたプロパティだけを対象にしている

componentN（分割代入）
⇒　プロパティの値を個々の変数に分解する

複数の戻り値を持つ関数で紹介した、分割代入を使って以下のように書ける

```
val t = Triple("海崎", "フリーター", 27)
val (name, job, age) = t
```

このコードは内部的には以下のようになっている

```
val t = Triple("海崎", "フリーター", 27)
val name = t.component1()
val job = t.component2()
val age = t.component3()
```

分割代入はcomponentNのシンタックスシュガー
→　データクラスにはcomponentNが実装されている
→　データクラスのオブジェクトは分割代入が可能

```
data class Member(val name: String, val job: String, val age: Int)

val m = Member("海崎", "フリーター", 27)
val (name, job, age) = m
println(age) //結果：27
```

copy（オブジェクトの複製）
⇒　特定のプロパティの値だけを変更した上でオブジェクトの複製を可能に

```
data class Member(val name: String, val job: String, val age: Int)

val m1 = Member("海崎", "フリーター", 27)
val m2 = m1.copy(age = 28)
println(m2) //結果：Member(name=海崎, job=フリーター, age=28)
```

・オブジェクト式

Android開発では再利用を目的としないクラスを定義することがよくある
そういったときに便利なのがオブジェクト式という機能

基本的な書き方は以下

```
object {クラス本体}
```

基底クラスの継承やインターフェースの実装がある場合は以下のように書く

```
object: 基底クラスorインターフェース {クラス本体}
```

```
btn.setOnClickListener(object: View.OnClickListener {
  override fun onClick(view: View) {
    ボタンがクリックされたときの処理
  }
})
```

setOnClickListenerメソッドのカッコの中にオブジェクト式が使われている
この例ではView.OnClickListenerインターフェースを実装・onClickメソッドをオーバーライドしている

View.OnClickListenerインターフェースが持つ抽象メソッドはonClickメソッドだけ
⇒　ひとつしか抽象メソッドを持たないインターフェースのことを、
SAM（Single Abstract Method）インターフェースと呼ぶ

オブジェクト式で実装するインターフェースがSAMインターフェースの場合、
ラムダ式で置き換えてよりシンプルに表現できる　⇒　**SAM変換**

先程のコードはSAM変換により以下のように書くこともできる

```
btn.setOnClickListener({ view: View -> クリックされたときの処理 })
```

setOnClickListenerメソッドの引数にはView.OnClickListenerインターフェースを実装したクラスが入ることは自明なのでインターフェースの指定も省略可能

また、ラムダ式においては以下も成り立つ
・引数の型が明らかならば省略可能
・引数を利用しないなら引数そのものも省略可能
・ラムダ式をくくる丸カッコは省略可能

なので、もっとシンプルに以下のようにも書ける

```
btn.setOnClickListener { ボタンがクリックされたときの処理 }
```

Androidの公式リファレンスでもよく登場する記述なので、
どういう意味なのかは理解できるようにしておく必要あり

・オブジェクト宣言

開発の場面でひとつのインスタンスしか持たないようなクラスを用意したいことがある
Javaなどの言語ではデザインパターンのひとつのSingletonパターンを使うのが一般的だが
Kotlinではオブジェクト宣言という仕組みによって実現できる

基本的な書き方は以下のような形

```
object オブジェクト名 {オブジェクト本体}
```

クラスを宣言するかのようにオブジェクトを宣言できる

```
object TanakaTaro {
    val name = "田中太郎"
    var clothe = "Tシャツ"

    fun intro() {
      println("${name}は${clothe}を着ている。")
    }
}

TanakaTaro.clothe = "Yシャツ"
TanakaTaro.intro()
```

既存のクラスを継承することも可能
その場合は基底クラスのコンストラクタに値を渡して上げる必要がある

```
object Foo: Parent("foo") {}
```

・コンパニオンオブジェクト

Kotlinでクラス変数・クラスメソッド（Staticメンバ）を利用するには、
クラス内部でのオブジェクト宣言である、**コンパニオンオブジェクト**を利用する

```
class MyClass {
  companion object Factory {
    fun create(): MyClass = MyClass()
  }
}

val my = MyClass.create()
println(my::class) //結果：class MyClass
```

クラス内部でのオブジェクト宣言にcompanionキーワードをつける
コンパニオンオブジェクト内で定義された変数やメソッドは、
「クラス名.変数」「クラス名.メソッド()」というように呼び出せる

コンパニオンオブジェクトの名前（上記ではFactory）は省略可能

```
class MyClass {
  companion object {
    fun create(): MyClass = MyClass()
  }
}
```

再利用しないクラスの定義　　⇒　オブジェクト式
シングルトンの定義　　⇒　オブジェクト宣言
staticメンバの（代替の）定義　⇒　コンパニオンオブジェクト

というように使い分けるが、プロパティの初期化のタイミングもそれぞれ異なる
オブジェクト式　　⇒　利用時に初期化
オブジェクト宣言　　⇒　メンバにアクセスする際に初期化（遅延初期化）
コンパニオンオブジェクト　⇒　上位のクラスの利用時に初期化

基本的にプロパティは実行時にそれぞれのタイミングで初期化されるが、
実行よりも前の**コンパイル時に初期化**される**コンパイル時定数**というものもある

コンパイル時定数を定義するには、**const修飾子**を付与する

```
const val DATABASE_VRESION = 1
```

const修飾子を利用できるのは以下の条件を満たしている場合だけであることに注意
・トップレベルかオブジェクト式、コンパニオンオブジェクトのメンバである
・整数、浮動小数点数、真偽値、文字型、文字列型のいずれかで初期化されている
・カスタムのgetterを持たない

・ジェネリック型

ジェネリック（プログラミング）とは？
具体的なデータ型に依存しない、抽象的かつ汎用的なコード記述を可能にする手法

例えば配列やコレクションには**要素としてどんな型でも入れることができた**
これはジェネリックプログラミングによるもの（**具体的なデータ型に依存しない**）

だが、入れることのできる方を制限できたほうが安全なプログラムを書けるので
val arr: mutableList<Int> = mutableListOf(1, 2, 3)
というように「<データの型>」として要素のデータ型を指定することもできた

このように**定義時は具体的なデータ型に依存しないようにしておいて、**
利用時にデータ型を設定することで、汎用的なコード記述を可能にするのが
ジェネリックプログラミングの思想

```
class genericExample<T>(var value: T) {
  fun getProp(): T {
    return value
}

val g1 = genericExample<String>("foo")
val g2 = genericExample<Int>(1)
val g3 = genericExample("hoge") //型推論ができるなら省略可能
println(g1.getProp()) // 結果：foo
println(g2.getProp()) // 結果：1
g1.value = 1 // 結果：エラー
```

「<T>」という記述がクラス名の後ろにあるが、これは**型引数**という機能
型引数はその名の通り「**型を受け取るための仮引数**」のこと
あくまでも仮引数なので名前は任意だが、T（Type）やE（Element）が一般的
ここで受け取った型がvalueプロパティの型とgetPropメソッドの戻り値になる

ジェネリック型のクラスをインスタンス化する際に、型引数にデータ型を渡す

型引数は複数与えることも可能、例えばMapは以下のように指定する

```
var map: Map<String, Int> = mapOf("First" to 1, "Second" to 2)
```

定義時は「class クラス名<K, V>(){..}」というように記述すればよい

先程見たgenericExampleクラスではどんな型も渡すことができたが、
渡せる型を制限したい場面もある　⇒　「**<T: クラス名>**」の形で記述する

このように書くと、「**型引数のあとに書いたクラス、もしくはその派生クラス**」に
型を制限することができるようになる

```
open class Hoge() {}
class Foo(): Hoge() {
  val foo = "foo"
}

class Generic<T: Hoge>(var obj: T) {
  fun getProp(): T {
    return obj
  }
}

val g1 = Generic<Foo>(Foo())
println(g1.getProp().foo) // 結果：foo
val g2 = Generic<String>("foo") // エラー
```

ジェネリック型のクラスのほかに、**ジェネリック関数**というものもある
ジェネリック関数とは？
⇒　引数と戻り値の型を関数の呼び出し時に決めることのできる関数

```
fun <T> tail(arr: Array<T>): T = arr[arr.size - 1]

val arr1 = arrayOf(1, 2, 3)
val arr2 = arrayOf("あ", "い", "う")
println(tail<Int>(arr1)) // 結果：3
println(tail(arr1)) // 結果：3
println(tail<String>(arr2)) // 結果：う
```

ジェネリック関数の定義は関数名の前に型引数を記述することで可能
呼び出しは関数名のあとに型を指定するか、型推論が可能なら普通に呼び出せる

ジェネリック関数は関数を定義できる場所ならばどこでも定義できることに注意
ジェネリック型のクラス内部などに限定されていない

・ネストクラス

Kotlinでは、classの中にclassを定義することができる
これを**ネストクラス**（入れ子のクラス）と呼ぶ

例えば、2つのクラス「Outer」と「Nested」があり、この2つのクラスは
NestedがOuterに強く依存しており、NestedはOuterからしか呼ばれないとする

このような場合、プログラム上でも仕様に準じて、NestedはOuter以外からは
アクセスできないようにするのが望ましい　⇒　ネストクラスの利用

```
class Outer {
  private class Nested {
    fun show() = println("Nested is running")
  }

  fun run(){
    val nested = Nested()
    nested.show()
  }
}

val o = Outer()
o.run() // 結果：Nested is running
```

入れ子になっているクラスのうち、外側のクラスを**アウタークラス**と呼ぶ
内側のクラスにはprivate修飾子を付けることでOuter以外が使えないクラスになる

入れ子のクラスからアウタークラスのメンバーにアクセスしたいケースもある
その場合は**inner修飾子**を付与して**インナークラス**として内部のクラスを定義する

```
class Outer(val name: String = "Outer") {
  inner class Nested(val name: String = "Nested") {
    fun show() {
      println("${this@Outer.name} called show()")
      println("${name} is running")
    }
  }

  fun run(){
    val nested = Nested()
    nested.show()
  }
}

val o = Outer()
o.run()

/* 結果
Outer called show()
Nested is running
*/
```

Outer/Nestedともにnameプロパティを持っている
showメソッドの中でnameと指定すると、インナークラスのプロパティが優先して参照されるので「${name} is running」は「Nested is running」になる

アウタークラスのメンバにアクセスしたいときは、「**this@アウタークラス名**」というように、thisキーワードにラベル構文を付与した形で記述する
なので「${this@Outer.name} called show()」は「Outer called show()」になる

■その他のKotlinの便利機能

・拡張関数

継承を用いずに既存のクラスにメソッドを追加できる**拡張関数**という機能がある
拡張関数を使えばopenでないクラスに対してもメソッドを追加できるようになる

拡張関数の基本的な書き方は以下

```
fun 拡張するクラス名.追加メソッド名(仮引数:型, ..): 戻り値の型 {処理}
```

```
fun String.repeat(n: Int): String {
  var builder = StringBuilder()
  for (i in 1..n) {
    builder.append(this)
  }
  return builder.toString()
}

val hoge = "ほげ"
println(hoge.repeat(3)) // 結果：ほげほげほげ
```

「this」は拡張するクラスのオブジェクトを表すので、上記の場合は「"ほげ"」の意
「this.クラスのメンバ」とすればもともと備わっているメンバにアクセスもできる
ただし、privateなメンバにアクセスすることはできない

・演算子のオーバーロード

「+」や「==」などいろいろな演算子があるが、これらはすべてメソッドである
例えば以下のコードはどちらも同じ意味になる

```
println(1 + 2) // 結果：3
println(1.plus(2)) // 結果：3
```

演算子は以下のようにそれぞれ対応するメソッドがある
メソッドの簡単・直感的な書き方として演算子があるというイメージ
「a + b」⇒「a.plus(b)」
「a == b」⇒「a.equals(b)」

「+」という演算子が使える型（クラス）には「plusメソッド」が実装されている

演算子はメソッドであるので再定義することができる
これを**演算子のオーバーロード**という

オーバーロードとは？
⇒　引数の型や数が異なっていれば同じ名前のメソッドを定義することができる
　　これをメソッドのオーバーロードという、**オーバーライドとの違いに注意**

```
fun show() = println("OK")
fun show(value: String) = println(value)

show() // 結果：OK
show("OK") // 結果：OK
```

Kotlinでは引数にデフォルト値をもたせることができるので、
オーバーロードは演算子以外では基本的に使用しない

```
fun show(value: String = "OK") = println(value)

show()
show("OK")
```

例えばx,y座標を表すCoordinateクラスを考える
plusメソッドをオーバーロードして座標計算ができるようにする
⇒　「(1.0, 2.0) + (0.5, 1.5) = (1.5, 3.5)」というようになるようにしたい

```
class Coordinate(val x: Double, val y: Double) {
  operator fun plus(c: Coordinate): Coordinate {
    return Coordinate(this.x + c.x, this.y + c.y)
  }
}
val c1 = Coordinate(1.0, 2.0)
val c2 = Coordinate(0.5, 1.5)
val c3 = c1 + c2
println("Coordinate:(${c3.x}, ${c3.y})") // 結果：Coordinate:(1.5,
3.5)
```

plusメソッドの頭に演算子を表す、**operatorキーワード**をつける
「c1 + c2」というように記述すると「c1.plus(c2)」となるので、
「this」は「c1」、引数の「c」は「c2」に対応することになる

■演習サイトと実行環境

演習サイト
Kotlin Koans

実行環境
Kotlin Playground