Kotlin®
专业人士笔记

80+ 页
专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明
这是一本非官方的免费书籍，旨在教育用途，
与官方Kotlin®组织或公司无关。
所有商标和注册商标均为其各自所有者的财产

Kotlin®
Notes for Professionals

80+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Kotlin® group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

目录

Contents

关于

欢迎随意免费分享此PDF，
本书最新版本可从以下网址下载：
https://goalkicker.com/KotlinBook

本*Kotlin® 专业人士笔记*一书汇编自Stack Overflow Documentation，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费书籍，旨在教育用途，与官方Kotlin®组织或公司及Stack Overflow无关。所有商标及注册商标均为其各自公司所有。

本书所提供信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/KotlinBook

This *Kotlin® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Kotlin® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Kotlin入门

版本发布日期

1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30
1.0.4	2016-09-22
1.0.5	2016-11-08
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23
1.1.6	2017-11-13
1.2.2	2018-01-17
1.2.3	2018-03-01
1.2.4	2018-04-19

第1.1节：你好，世界

所有 Kotlin 程序都从main函数开始。以下是一个简单的 Kotlin“Hello World”程序示例：

```
package my.program

fun main(args: Array<String>) {
    println("Hello, world!")
}
```

将上述代码放入名为Main.kt的文件中（此文件名完全可自定义）当目标为 JVM 时，

该函数将被编译为一个静态方法，所在类的名称由文件名派生。在上述示例中，运行的主类将是my.program.MainKt。

要更改包含特定文件顶层函数的类名，请在文件顶部的 package 语句上方添加以下注解：

```
@file:JvmName("MyApp")
```

在此示例中，运行的主类现在将是my.program.MyApp。

另见：

- 包级函数包括@JvmName注解。
- 注解使用位置目标

第 1.2 节：使用伴生对象的 Hello World

类似于使用对象声明，您可以使用类的Companion

Object来定义Kotlin程序的main函数。

Chapter 1: Getting started with Kotlin

Version Release Date

1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30
1.0.4	2016-09-22
1.0.5	2016-11-08
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23
1.1.6	2017-11-13
1.2.2	2018-01-17
1.2.3	2018-03-01
1.2.4	2018-04-19

Section 1.1: Hello World

All Kotlin programs start at the `main` function. Here is an example of a simple Kotlin "Hello World" program:

```
package my.program

fun main(args: Array<String>) {
    println("Hello, world!")
}
```

Place the above code into a file named `Main.kt` (this filename is entirely arbitrary)

When targeting the JVM, the function will be compiled as a static method in a class with a name derived from the filename. In the above example, the main class to run would be `my.program.MainKt`.

To change the name of the class that contains top-level functions for a particular file, place the following annotation at the top of the file above the package statement:

```
@file:JvmName("MyApp")
```

In this example, the main class to run would now be `my.program.MyApp`.

See also:

- Package level functions including `@JvmName` annotation.
- Annotation use-site targets

Section 1.2: Hello World using a Companion Object

Similar to using an Object Declaration, you can define the `main` function of a Kotlin program using a Companion Object of a class.

```
package my.program

class App {
    companion object {
@JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

你将运行的类名是你的类名，在本例中是my.program.App。

这种方法相比顶层函数的优点是，运行的类名更直观，且你添加的任何其他函数都被限定在App类中。这类似于Object声明的例子，区别在于你可以控制实例化任何类以进行进一步操作。

一个稍微不同的变体，实例化该类以执行实际的“hello”：

```
class App {
    companion object {
@JvmStatic fun main(args: Array<String>) {
            App().run()
        }
    }

    fun run() {
println("Hello World")
    }
}
```

另见：

- 包括 @JvmStatic 注解的静态方法

第1.3节：使用对象声明的 Hello World

你也可以使用包含 Kotlin 程序主函数的对象声明。

```
package my.program

object App {
@JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
}
```

你运行的类名是对象的名称，在本例中是my.program.App。

这种方法相比顶层函数的优点是运行的类名更直观，且你添加的任何其他函数都被限定在App类中。你还拥有App的单例实例来存储状态和执行其他工作。

另见：

- 包括 @JvmStatic 注解的静态方法


```
package my.program

class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

The class name that you will run is the name of your class, in this case is `my.program.App`.

The advantage to this method over a top-level function is that the class name to run is more self-evident, and any other functions you add are scoped into the class `App`. This is similar to the `Object Declaration` example, other than you are in control of instantiating any classes to do further work.

A slight variation that instantiates the class to do the actual "hello":

```
class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            App().run()
        }
    }

    fun run() {
        println("Hello World")
    }
}
```

See also:

- Static Methods including the @JvmStatic annotation

Section 1.3: Hello World using an Object Declaration

You can alternatively use an Object Declaration that contains the main function for a Kotlin program.

```
package my.program

object App {
    @JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
}
```

The class name that you will run is the name of your object, in this case is `my.program.App`.

The advantage to this method over a top-level function is that the class name to run is more self-evident, and any other functions you add are scoped into the class `App`. You then also have a singleton instance of `App` to store state and do other work.

See also:

- Static Methods including the `@JvmStatic` annotation

第1.4节：使用可变参数的主要方法

所有这些主方法风格也可以与varargs一起使用：

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

第1.5节：在命令行中编译和运行Kotlin代码

正如Java提供了两个不同的命令来编译和运行Java代码，Kotlin同样为你提供了不同的命令。

javac 用于编译Java文件。 java 用于运行Java文件。

同样，kotlinc 用于编译Kotlin文件，kotlin 用于运行Kotlin文件。

第1.6节：从命令行读取输入

从控制台传递的参数可以在Kotlin程序中接收，并可用作输入。你可以从命令提示符传递N个（1 2 3 等）参数。

Kotlin中命令行参数的一个简单示例。

```
fun main(args: Array<String>) {

    println("请输入两个数字")
                        var (a, b) = readLine()!!.split(' ') // !! 该操作符用于防止空指针异常（NPE）。

println("最大数是 : ${maxNum(a.toInt(), b.toInt())}")
}



fun maxNum(a: Int, b: Int): Int {

    var max = if (a > b) {
        println("a 的值是 $a");
          a
    } else {
println("b 的值是 $b")
          b
    }

    return max;

}
```

这里，从命令行输入两个数字以找出最大数。输出：

```
请输入两个数字
71 89 // 从命令行输入两个数字

b 的值是 89
最大数是：89
```


Section 1.4: Main methods using varargs

All of these main method styles can also be used with varargs:

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

Section 1.5: Compile and Run Kotlin Code in Command Line

As java provide two different commands to compile and run Java code. Same as Kotlin also provide you different commands.

`javac` to compile java files. `java` to run java files.

Same as `kotlinc` to compile kotlin files `kotlin` to run kotlin files.

Section 1.6: Reading input from Command Line

The arguments passed from the console can be received in the Kotlin program and it can be used as an input. You can pass N (1 2 3 and so on) numbers of arguments from the command prompt.

A simple example of a command-line argument in Kotlin.

```
fun main(args: Array<String>) {

    println("Enter Two number")
    var (a, b) = readLine()!!.split(' ') // !! this operator use for NPE(NullPointerException).

    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
}



fun maxNum(a: Int, b: Int): Int {

    var max = if (a > b) {
        println("The value of a is $a");
        a
    } else {
        println("The value of b is $b")
        b
    }

    return max;

}
```

Here, Enter two number from the command line to find the maximum number. Output:

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

对于 !! 操作符，请检查空安全。

注意：上述示例可在 Intellij 上编译并运行。

For !! Operator Please check Null Safety.

Note: Above example compile and run on Intellij.

第2章：Kotlin基础

本主题涵盖Kotlin初学者的基础知识。

第2.1节：基本示例

1. 函数的 Unit 返回类型声明是可选的。以下代码是等价的。

```
    fun printHello(name: String?): Unit {
        if (name != null)
println("Hello ${name}")
    }

    fun printHello(name: String?) {
       ...
    }
```

2. 单表达式函数：当函数返回单个表达式时，可以省略大括号，函数体写在=符号后面

```
fun double(x: Int): Int = x * 2
```

当编译器能够推断返回类型时，显式声明返回类型是可选的

```
fun double(x: Int) = x * 2
```

3. 字符串插值：使用字符串值很简单。

```
在 Java 中:
int num=10
     String s  = "i =" + i;

在 Kotlin 中
     val num = 10
     val s = "i = $num"
```

4. 在 Kotlin 中，类型系统区分可以持有 null 的引用（可空引用）和不能持有 null 的引用（非空引用）。例如，普通的 String 类型变量不能持有 null：

```
var a: String = "abc"
a = null // 编译错误
```

为了允许空值，我们可以将变量声明为可空字符串，写作 String?:

```
var b: String? = "abc"
b = null // 可以
```

5. 在 Kotlin 中，== 实际上检查的是值的相等性。按照惯例，表达式 a == b 会被转换为

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


Chapter 2: Basics of Kotlin

This topic covers the basics of Kotlin for beginners.

Section 2.1: Basic examples

1. The Unit return type declaration is optional for functions. The following codes are equivalent.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}

fun printHello(name: String?) {
    ...
}
```

2. Single-Expression functions:When a function returns a single expression, the curly braces can be omitted and the body is specified after = symbol

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler

```
fun double(x: Int) = x * 2
```

3. String interpolation: Using string values is easy.

```
In java:
    int num=10
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4. In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). For example, a regular variable of type String can not hold null:

```
var a: String = "abc"
a = null // compilation error
```

To allow nulls, we can declare a variable as nullable string, written String?:

```
var b: String? = "abc"
b = null // ok
```

5. In Kotlin,== actually checks for equality of values. By convention, an expression like a == b is translated to

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

第3章：字符串

第3.1节：字符串相等性

在 Kotlin 中，字符串使用 == 运算符进行比较，该运算符检查它们的结构相等性。

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // 输出 true
```

引用相等性使用 === 运算符进行检查。

```
val str1 = """
    |Hello, World!
""".trimMargin()

val str2 = """
    #Hello, World!
""".trimMargin("#")

val str3 = str1

println(str1 == str2) // 输出 true
println(str1 === str2) // 输出 false
println(str1 === str3) // 输出 true
```

第3.2节：字符串字面量

Kotlin 有两种类型的字符串字面量：

- 转义字符串
- 原始字符串

转义字符串通过转义处理特殊字符。转义使用反斜杠完成。支持以下转义序列：\t, \b, \n, \r, \', \", \\ 和 \$。要编码其他字符，使用 Unicode转义序列语法：\uFF00。

```
val s = "Hello, world!"
```

原始字符串由三重引号 """界定，不包含转义，可以包含换行和任何其他字符

```
val text = """
    对于 (c 在 "foo" 中)
        打印(c)
"""
```

前导空白可以使用trimMargin()函数去除。

```
val text = """
    |告诉我，我会忘记。
|教我，我会记住。
|让我参与，我会学习。
|  （本杰明·富兰克林）
    """.trimMargin()
```


Chapter 3: Strings

Section 3.1: String Equality

In Kotlin strings are compared with `==` operator which check for their structural equality.

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // Prints true
```

Referential equality is checked with `===` operator.

```
val str1 = """
    |Hello, World!
    """.trimMargin()

val str2 = """
    #Hello, World!
    """.trimMargin("#")

val str3 = str1

println(str1 == str2) // Prints true
println(str1 === str2) // Prints false
println(str1 === str3) // Prints true
```

Section 3.2: String Literals

Kotlin has two types of string literals:

- Escaped string
- Raw string

Escaped string handles special characters by escaping them. Escaping is done with a backslash. The following escape sequences are supported: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` and `\$`. To encode any other character, use the Unicode escape sequence syntax: `\uFF00`.

```
val s = "Hello, world!\n"
```

Raw string delimited by a triple quote `"""`, contains no escaping and can contain newlines and any other characters

```
val text = """
    for (c in "foo")
        print(c)
"""
```

Leading whitespace can be removed with trimMargin() function.

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

默认的边距前缀是管道符号|，这可以作为参数传递给trimMargin；例如trimMargin(">")。

第3.3节：字符串的元素

字符串的元素是可以通过索引操作string[index]访问的字符。

```
val str = "Hello, World!"
println(str[1]) // 输出 e
```

字符串元素可以通过for循环进行迭代。

```
for (c in str) {
    println(c)
}
```

第3.4节：字符串模板

转义字符串和原始字符串都可以包含模板表达式。模板表达式是一段代码，会被计算并将结果连接到字符串中。它以美元符号$开头，由以下两种形式之一组成：

```
val i = 10
val s = "i = $i" // 计算结果为 "i = 10"
```

或者是用大括号括起来的任意表达式：

```
val s = "abc"
val str = "$s.length is ${s.length}" // 计算结果为 "abc.length is 3"
```

要在字符串中包含字面上的美元符号，可以使用反斜杠进行转义：

```
val str = "\$foo" // 计算结果为 "$foo"
```

例外情况是原始字符串，它们不支持转义。在原始字符串中，你可以使用以下语法来表示美元符号。

```
val price = """
${'$'}9.99
"""
```


Default margin prefix is pipe character `|`, this can be set as a parameter to trimMargin; e.g. `trimMargin(">")`.

Section 3.3: Elements of String

Elements of String are characters that can be accessed by the indexing operation `string[index]`.

```
val str = "Hello, World!"
println(str[1]) // Prints e
```

String elements can be iterated with a for-loop.

```
for (c in str) {
    println(c)
}
```

Section 3.4: String Templates

Both escaped strings and raw strings can contain template expressions. Template expression is a piece of code which is evaluated and its result is concatenated into string. It starts with a dollar sign $ and consists of either a variable name:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

Or an arbitrary expression in curly braces:

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

To include a literal dollar sign in a string, escape it using a backslash:

```
val str = "\$foo" // evaluates to "$foo"
```

The exception is raw strings, which do not support escaping. In raw strings you can use the following syntax to represent a dollar sign.

```
val price = """
${'$'}9.99
"""
```

第4章：数组

第4.1节：泛型数组

Kotlin中的泛型数组用Array<T>表示。

要创建一个空数组，使用emptyArray<T>()工厂函数：

```
val empty = emptyArray<String>()
```

要创建一个指定大小和初始值的数组，使用构造函数：

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // 输出 "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // 输出 5
```

数组有get(index: Int): T和set(index: Int, value: T)函数：

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // 输出 "ChangedItem"

// 你也可以使用订阅方式：
strings[2] = "ChangedItem"
print(strings[2]) // 输出 "ChangedItem"
```

第4.2节：原始类型数组

这些类型不继承自Array<T>以避免装箱，但它们具有相同的属性和方法。

Kotlin类型	工厂函数	JVM类型
BooleanArray	booleanArrayOf(true, false)	boolean[]
ByteArray	byteArrayOf(1, 2, 3)	byte[]
CharArray	charArrayOf('a', 'b', 'c')	char[]
DoubleArray	doubleArrayOf(1.2, 5.0)	double[]
FloatArray	floatArrayOf(1.2, 5.0)	float[]
IntArray	intArrayOf(1, 2, 3)	int[]
LongArray	longArrayOf(1, 2, 3)	long[]
ShortArray	shortArrayOf(1, 2, 3)	short[]

第4.3节：创建数组

```
val a = arrayOf(1, 2, 3) // 创建一个包含 [1, 2, 3] 的大小为3的 Array<Int>。
```

第4.4节：使用闭包创建数组

```
val a = Array(3) { i -> i * 2 } // 创建一个包含 [0, 2, 4] 的大小为3的 Array<Int>
```

第4.5节：创建未初始化数组

```
val a = arrayOfNulls<Int>(3) // 创建一个包含 [null, null, null] 的 Array<Int?>。
```

返回的数组类型始终是可空类型。无法创建未初始化的非空类型数组。

Chapter 4: Arrays

Section 4.1: Generic Arrays

Generic arrays in Kotlin are represented by `Array<T>`.

To create an empty array, use `emptyArray<T>()` factory function:

```
val empty = emptyArray<String>()
```

To create an array with given size and initial values, use the constructor:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Arrays have `get(index: Int): T` and `set(index: Int, value: T)` functions:

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Section 4.2: Arrays of Primitives

These types **do not** inherit from `Array<T>` to avoid boxing, however, they have the same attributes and methods.

Kotlin type	Factory function	JVM type
BooleanArray	booleanArrayOf(true, false)	boolean[]
ByteArray	byteArrayOf(1, 2, 3)	byte[]
CharArray	charArrayOf('a', 'b', 'c')	char[]
DoubleArray	doubleArrayOf(1.2, 5.0)	double[]
FloatArray	floatArrayOf(1.2, 5.0)	float[]
IntArray	intArrayOf(1, 2, 3)	int[]
LongArray	longArrayOf(1, 2, 3)	long[]
ShortArray	shortArrayOf(1, 2, 3)	short[]

Section 4.3: Create an array

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Section 4.4: Create an array using a closure

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Section 4.5: Create an uninitialized array

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

The returned array will always have a nullable type. Arrays of non-nullable items can't be created uninitialized.

第4.6节：扩展

average() 定义于 Byte、Int、Long、Short、Double、Float 类型上，且始终返回 Double：

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // 输出 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // 输出 2.5
```

component1(), component2(), ... component5() 返回数组的一个元素

getOrNull(index: Int) 如果索引越界则返回 null，否则返回数组的一个元素

first(), last()

toHashSet() 返回一个包含所有元素的 HashSet<T>

sortedArray(), sortedArrayDescending() 创建并返回一个包含当前元素排序后的新数组

sort(), sortDescending 就地排序数组

min(), max()

第4.7节：遍历数组

你可以使用与 Java 增强型循环相同的方式打印数组元素，但需要将关键字
从 : 改为 in。

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

你也可以在 for 循环中更改数据类型。

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```


Section 4.6: Extensions

average() is defined for **Byte**, **Int**, **Long**, **Short**, **Double**, **Float** and always returns **Double**:

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

component1(), component2(), ... component5() return an item of the array

getOrNull(index: Int) returns null if index is out of bounds, otherwise an item of the array

first(), last()

toHashSet() returns a HashSet<T> of all elements

sortedArray(), sortedArrayDescending() creates and returns a new array with sorted elements of current

sort(), sortDescending sort the array in-place

min(), max()

Section 4.7: Iterate Array

You can print the array elements using the loop same as the Java enhanced loop, but you need to change keyword from : to in.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

You can also change data type in for loop.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

第5章：集合

与许多语言不同，Kotlin区分可变和不可变集合（列表、集合、映射等）。
精确控制集合何时可以被编辑对于消除错误和设计良好的

API非常有用。

第5.1节：使用列表

```
// 创建一个新的只读List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // 输出 "[Item 1, Item 2, Item 3]"
```

第5.2节：使用映射

```
// 创建一个新的只读Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // 输出 "{1=Item 1, 2=Item 2, 3=Item 3}"
```

第5.3节：使用集合

```
// 创建一个新的只读 Set<String>
val set = setOf(1, 3, 5)
println(set) // 输出 "[1, 3, 5]"
```


Chapter 5: Collections

Unlike many languages, Kotlin distinguishes between mutable and immutable collections (lists, sets, maps, etc). Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

Section 5.1: Using list

```
// Create a new read-only List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // prints "[Item 1, Item 2, Item 3]"
```

Section 5.2: Using map

```
// Create a new read-only Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // prints "{1=Item 1, 2=Item 2, 3=Item 3}"
```

Section 5.3: Using set

```
// Create a new read-only Set<String>
val set = setOf(1, 3, 5)
println(set) // prints "[1, 3, 5]"
```

第6章：枚举

第6.1节：初始化

枚举类和其他类一样，可以有构造函数并被初始化

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

第6.2节：枚举中的函数和属性

枚举类也可以声明成员（即属性和函数）。必须在最后一个枚举对象和第一个成员声明之间放置分号（;）。

如果成员是abstract，枚举对象必须实现它。

```
枚举类 颜色 {
    红色 {
        重写 val rgb: Int = 0xFF0000
    },
绿色 {
        重写 val rgb: Int = 0x00FF00
    },
蓝色 {
        重写 val rgb: Int = 0x0000FF
    }

    ;

    抽象 val rgb: Int

    函数 colorString() = "#%06X".格式化(0xFFFFFF 和 rgb)
}
```

第6.3节：简单枚举

```
enum class Color {
  RED, GREEN, BLUE
}
```

每个枚举常量都是一个对象。枚举常量之间用逗号分隔。

第6.4节：可变性

枚举可以是可变的，这是一种实现单例行为的另一种方式：

```
enum class 行星(var 人口: Int = 0) {
    地球(7 * 100000000),
火星();

    override fun toString() = "$name[population=$population]"
}
```


Chapter 6: Enum

Section 6.1: Initialization

Enum classes as any other classes can have a constructor and be initialized

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Section 6.2: Functions and Properties in enums

Enum classes can also declare members (i.e. properties and functions). A semicolon (;) must be placed between the last enum object and the first member declaration.

If a member is **abstract**, the enum objects must implement it.

```
enum class Color {
    RED {
        override val rgb: Int = 0xFF0000
    },
    GREEN {
        override val rgb: Int = 0x00FF00
    },
    BLUE {
        override val rgb: Int = 0x0000FF
    }

    ;

    abstract val rgb: Int

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
}
```

Section 6.3: Simple enum

```
enum class Color {
  RED, GREEN, BLUE
}
```

Each enum constant is an object. Enum constants are separated with commas.

Section 6.4: Mutability

Enums can be mutable, this is another way to obtain a singleton behavior:

```
enum class Planet(var population: Int = 0) {
    EARTH(7 * 100000000),
    MARS();

    override fun toString() = "$name[population=$population]"
}
```

```
println(Planet.火星) // 火星[population=0]
 Planet.火星.人口 = 3
println(Planet.火星) // 火星[population=3]
```



```
println(Planet.MARS) // MARS[population=0]
Planet.MARS.population = 3
println(Planet.MARS) // MARS[population=3]
```

第7章：函数

参数	详情
名称	函数名称
参数	传递给函数的带名称和类型的值：名称：类型
类型	函数的返回类型
类型	泛型编程中使用的类型参数（不一定是返回类型）
参数名	传递给函数的值的名称
参数类型	参数名的类型说明符ArgName
参数名	由逗号分隔的参数名列表

第7.1节：函数引用

我们可以通过在函数名前加上::来引用函数而不实际调用它。然后可以将其传递给接受其他函数作为参数的函数。

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

无接收者的函数将被转换为(ParamTypeA, ParamTypeB, ...) -> ReturnType，其中ParamTypeA, ParamTypeB ... 是函数参数的类型，`ReturnType` 是函数返回值的类型。

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// 可读类型：(Foo0, Foo1, Foo2) -> Bar
```

带有接收者的函数（无论是扩展函数还是成员函数）语法不同。你必须在双冒号前添加接收者的类型名称：

```
类 Foo
函数 Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
值 ref = Foo::foo
打印(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// 可读类型: (Foo, Foo0, Foo1, Foo2) -> Bar
// 接受4个参数，接收者作为第一个参数，实际参数按顺序跟随

// 但这个函数不能像扩展函数那样调用
值 ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // 编译错误

类 Bar {
    函数 bar()
}
打印(Bar::bar) // 成员函数也适用。
```

然而，当函数的接收者是一个对象时，接收者会从参数列表中省略，因为这种类型只有且仅有一个实例。

Chapter 7: Functions

Parameter	Details
Name	Name of the function
Params	Values given to the function with a name and type: ***Name*** : ***Type***
Type	Return type of the function
Type Argument	Type parameter used in generic programming (not necessarily return type)
ArgName	Name of value given to the function
ArgType	Type specifier for ***ArgName***
ArgNames	List of ArgName separated by commas

Section 7.1: Function References

We can reference a function without actually calling it by prefixing the function's name with ::. This can then be passed to a function which accepts some other function as a parameter.

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

Functions without a receiver will be converted to `(ParamTypeA, ParamTypeB, ...) -> ReturnType` where `ParamTypeA`, `ParamTypeB` ... are the type of the function parameters and `ReturnType1 is the type of function return value.

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

Functions with a receiver (be it an extension function or a member function) has a different syntax. You have to add the type name of the receiver before the double colon:

```
class Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo, Foo0, Foo1, Foo2) -> Bar
// takes 4 parameters, with receiver as first and actual parameters following, in their order

// this function can't be called like an extension function, though
val ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // compile error

class Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

However, when a function's receiver is an object, the receiver is omitted from parameter list, because these is and only is one instance of such type.

```
对象 Foo
函数 Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
值 ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// 可读类型：(Foo0, Foo1, Foo2) -> Bar
// 接受3个参数，不需要接收者

对象 Bar {
    函数 bar()
}
打印(Bar::bar) // 成员函数也适用。
```

自 kotlin 1.1 起，函数引用也可以绑定到变量上，这种变量称为绑定函数引用。

版本 ≥ 1.1.0

```
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
}
```

注意，此示例仅用于展示绑定函数引用的工作原理。在其他所有方面，这都是不好的实践。

不过，有一个特殊情况。作为成员声明的扩展函数不能被引用。

```
类 Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // 编译错误
}
```

第7.2节：基本函数

函数使用fun关键字声明，后跟函数名和参数。你也可以指定函数的返回类型，默认是Unit。函数体用大括号{}括起来。如果返回类型不是Unit，函数体内的每个终止分支必须有返回语句。

```
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

同样功能的简写版本：

```
fun sayMyName(name: String): String = "Your name is $name"
```

类型可以省略，因为可以推断：

```
fun sayMyName(name: String) = "Your name is $name"
```



```
object Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
// takes 3 parameters, receiver not needed

object Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Since kotlin 1.1, function reference can also be *bounded* to a variable, which is then called a *bounded function reference*.

Version ≥ 1.1.0

```
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
}
```

Note this example is given only to show how bounded function reference works. It's bad practice in all other senses.

There is a special case, though. An extension function declared as a member can't be referenced.

```
class Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // compile error
}
```

Section 7.2: Basic Functions

Functions are declared using the **`fun`** keyword, followed by a function name and any parameters. You can also specify the return type of a function, which defaults to **`Unit`**. The body of the function is enclosed in braces {}. If the return type is other than **`Unit`**, the body must issue a return statement for every terminating branch within the body.

```
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

A shorthand version of the same:

```
fun sayMyName(name: String): String = "Your name is $name"
```

And the type can be omitted since it can be inferred:

```
fun sayMyName(name: String) = "Your name is $name"
```

第7.3节：内联函数

函数可以使用inline前缀声明为内联函数，在这种情况下它们的行为类似于C语言中的宏——不是被调用，而是在编译时被函数体代码替换。这在某些情况下可以带来性能提升，主要是在使用lambda作为函数参数时。

```
inline fun sayMyName(name: String) = "Your name is $name"
```

与C宏的一个区别是内联函数不能访问调用它们的作用域：

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
sayMyName() # => 未解析的引用: name
}
```

第7.4节：Lambda函数

Lambda函数是匿名函数，通常在函数调用时创建，用作函数参数。它们通过用{大括号}包围表达式来声明——如果需要参数，则放在箭头->之前。

```
{ name: String ->
    "你的名字是 $name" //这是返回值
}
```

lambda 函数中的最后一个语句会自动作为返回值。

类型是可选的，如果你把 lambda 放在编译器可以推断类型的位置。

多个参数：

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

如果 lambda 函数只需要一个参数，那么参数列表可以省略，单个参数可以用it来引用。

```
{ "你的名字是 $it" }
```

如果函数的唯一参数是 lambda 函数，那么函数调用时可以完全省略括号。

```
# 这两者是相同的
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

第7.5节：操作符函数

Kotlin 允许我们为一组预定义的具有固定符号表示（如 + 或 *）和固定优先级的运算符提供实现。要实现一个运算符，我们需要为相应类型提供一个具有固定名称的成员函数或扩展函数。重载运算符的函数需要标记为

Section 7.3: Inline Functions

Functions can be declared inline using the **`inline`** prefix, and in this case they act like macros in C - rather than being called, they are replaced by the function's body code at compile time. This can lead to performance benefits in some circumstances, mainly where lambdas are used as function parameters.

```
inline fun sayMyName(name: String) = "Your name is $name"
```

One difference from C macros is that inline functions can't access the scope from which they're called:

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
    sayMyName() # => Unresolved reference: name
}
```

Section 7.4: Lambda Functions

Lambda functions are anonymous functions which are usually created during a function call to act as a function parameter. They are declared by surrounding expressions with {braces} - if arguments are needed, these are put before an arrow `->`.

```
{ name: String ->
    "Your name is $name" //This is returned
}
```

The last statement inside a lambda function is automatically the return value.

The type's are optional, if you put the lambda on a place where the compiler can infer the types.

Multiple arguments:

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

If the lambda function only needs one argument, then the argument list can be omitted and the single argument be referred to using `it` instead.

```
{ "Your name is $it" }
```

If the only argument to a function is a lambda function, then parentheses can be completely omitted from the function call.

```
# These are identical
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

Section 7.5: Operator functions

Kotlin allows us to provide implementations for a predefined set of operators with fixed symbolic representation (like `+` or `*`) and fixed precedence. To implement an operator, we provide a member function or an extension function with a fixed name, for the corresponding type. Functions that overload operators need to be marked with

操作符修饰符：

```
数据类 IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}

val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

更多操作符函数可以在 here 找到

第7.6节：接受其他函数的函数

如“Lambda函数”中所示，函数可以将其他函数作为参数。声明接受其他函数的函数时需要的“函数类型”如下：

```
# 不接受参数，返回任意类型
() -> Any?

# 接受一个字符串和一个整数，返回ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

例如，你可以使用最宽泛的类型() -> Any?来声明一个执行lambda函数两次的函数：

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
# => Foo
}
```

第7.7节：简写函数

如果函数只包含一个表达式，我们可以省略大括号，改用等号，就像变量赋值一样。表达式的结果会自动返回。

```
fun sayMyName(name: String): String = "Your name is $name"
```


the **operator** modifier:

```
data class IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}

val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

More operator functions can be found in here

Section 7.6: Functions Taking Other Functions

As seen in "Lambda Functions", functions can take other functions as a parameter. The "function type" which you'll need to declare functions which take other functions is as follows:

```
# Takes no parameters and returns anything
() -> Any?

# Takes a string and an integer and returns ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

For example, you could use the vaguest type, `() -> Any?`, to declare a function which executes a lambda function twice:

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
      # => Foo
}
```

Section 7.7: Shorthand Functions

If a function contains just one expression, we can omit the brace brackets and use an equals instead, like a variable assignment. The result of the expression is returned automatically.

```
fun sayMyName(name: String): String = "Your name is $name"
```

第8章：函数中的可变参数

第8.1节：基础：使用vararg关键字

使用vararg关键字定义函数。

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
println(number)
    }
}
```

现在你可以向函数传入任意数量的（正确类型的）参数。

```
printNumbers(0, 1)              // 打印 "0" "1"
printNumbers(10, 20, 30, 500)   // 打印 "10" "20" "30" "500"
```

> 注意： 可变参数 必须 是参数列表中的最后一个参数。

第8.2节：扩展运算符：将数组传递给可变参数函数

可以使用扩展运算符，*将数组传递给可变参数函数。

假设存在以下函数...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
println(number)
    }
}
```

你可以像这样传递一个数组给函数...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// 这与传入 (1, 2, 3) 是一样的
```

扩展运算符也可以用于参数的中间位置...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)

// 这与传入 (10, 20, 1, 2, 3, 30, 40) 是一样的
```


Chapter 8: Vararg Parameters in Functions

Section 8.1: Basics: Using the vararg keyword

Define the function using the `vararg` keyword.

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Now you can pass as many parameters (of the correct type) into the function as you want.

```
printNumbers(0, 1)              // Prints "0" "1"
printNumbers(10, 20, 30, 500)   // Prints "10" "20" "30" "500"
```

> **Notes:** Vararg parameters *must* be the last parameter in the parameter list.

Section 8.2: Spread Operator: Passing arrays into vararg functions

Arrays can be passed into vararg functions using the **Spread Operator**, `*`.

Assuming the following function exists...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

You can **pass an array** into the function like so...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// This is the same as passing in (1, 2, 3)
```

The spread operator can also be used **in the middle** of the parameters...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)

// This is the same as passing in (10, 20, 1, 2, 3, 30, 40)
```

第9章：条件语句

第9.1节：when语句的参数匹配

当给定一个参数时，**when**语句会依次将该参数与各分支进行匹配。
匹配是使用==操作符完成的，该操作符会进行空值检查并使用equals函数比较操作数。第一个匹配成功的分支将被执行。

```
when (x) {
    "English" -> print("你好吗？")
    "German" -> print("你怎么样？")
    else -> print("我还不懂那种语言 :(")
}
```

when语句还支持一些更高级的匹配选项：

```
val names = listOf("约翰", "莎拉", "蒂姆", "玛吉")
when (x) {
    in names -> print("我认识这个名字！")
    !in 1..10 -> print("参数不在1到10的范围内")
    is String -> print(x.length) // 由于智能类型转换，这里可以使用String的函数
}
```

第9.2节：作为表达式的when语句

就像 if 一样，when 也可以用作表达式：

```
val greeting = when (x) {
    "English" -> "你好吗？"
    "German" -> "你好吗？"（德语："Wie geht es dir?"）
    else -> "我还不会那种语言 :("
}
print(greeting)
```

要作为表达式使用，when 语句必须是穷尽的，即要么有 else 分支，要么以其他方式覆盖所有可能性。

第9.3节：标准 if 语句

```
val str = "你好！"
if (str.length == 0) {
    print("字符串为空！")
} else if (str.length > 5) {
    print("字符串较短！")
} else {
print("字符串较长！")
}
```

普通的 if 语句中 else 分支是可选的。

第9.4节：作为表达式的 if 语句

if 语句可以是表达式：

Chapter 9: Conditional Statements

Section 9.1: When-statement argument matching

When given an argument, the **when**-statement matches the argument against the branches in sequence. The matching is done using the == operator which performs null checks and compares the operands using the equals function. The first matching one will be executed.

```
when (x) {
    "English" -> print("How are you?")
    "German" -> print("Wie geht es dir?")
    else -> print("I don't know that language yet :(")
}
```

The when statement also knows some more advanced matching options:

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
when (x) {
    in names -> print("I know that name!")
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // Due to smart casting, you can use String-functions here
}
```

Section 9.2: When-statement as expression

Like if, when can also be used as an expression:

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

To be used as an expression, the when-statement must be exhaustive, i.e. either have an else branch or cover all possibilities with the branches in another way.

Section 9.3: Standard if-statement

```
val str = "Hello!"
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

The else-branches are optional in normal if-statements.

Section 9.4: If-statement as an expression

If-statements can be expressions:

```
val str = if (condition) "条件满足！" else "条件不满足！"
```

注意，如果将 `if` 语句用作表达式，则 `else` 分支不是可选的。

这也可以通过带有大括号和多个 `else if` 语句的多行变体来实现。

```
val str = if (condition1){
     "条件1满足！"
    } else if (condition2) {
     "条件2满足！"
    } else {
     "条件不满足！"
    }
```

> 提示：Kotlin 可以为你推断变量的类型，但如果你想确定类型，可以在变量上注解类型，如：val str: String = 这样可以强制类型，并使代码更易读。

第9.5节：使用 when 语句替代 if-else-if 链

when 语句是带有多个 else-if 分支的 if 语句的替代方案：

```
when {
str.length == 0 -> print("字符串为空！")
    str.length > 5  -> print("字符串较短！")
    else            -> print("字符串很长！")
}
```

使用if-else-if链编写的相同代码：

```
if (str.length == 0) {
    print("字符串为空！")
} else if (str.length > 5) {
    print("字符串很短！")
} else {
print("字符串较长！")
}
```

就像if语句一样，else分支是可选的，你可以添加任意多或任意少的分支。

你也可以有多行分支：

```
when {
condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

第9.6节：带枚举的when语句

when 可用于匹配 enum 值：

```
enum class Day {
    Sunday,
星期一，
```



```
val str = if (condition) "Condition met!" else "Condition not met!"
```

Note that the `else`-branch is not optional if the `if`-statement is used as an expression.

This can also been done with a multi-line variant with curly brackets and multiple `else if` statements.

```
val str = if (condition1){
    "Condition1 met!"
   } else if (condition2) {
    "Condition2 met!"
   } else {
    "Conditions not met!"
   }
```

> TIP: Kotlin can infer the type of the variable for you but if you want to be sure of the type just annotate it on the variable like: `val str: String =` this will enforce the type and will make it easier to read.

Section 9.5: When-statement instead of if-else-if chains

The when-statement is an alternative to an if-statement with multiple else-if-branches:

```
when {
    str.length == 0 -> print("The string is empty!")
    str.length > 5  -> print("The string is short!")
    else            -> print("The string is long!")
}
```

Same code written using an *if-else-if* chain:

```
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

Just like with the if-statement, the else-branch is optional, and you can add as many or as few branches as you like.
You can also have multiline-branches:

```
when {
    condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

Section 9.6: When-statement with enums

`when` can be used to match `enum` values:

```
enum class Day {
    Sunday,
    Monday,
```

```
星期二，
    星期三，
    星期四，
    星期五，
    星期六
}

fun doOnDay(day: Day) {
    when(day) {
Day.星期日 ->      // 执行某些操作
        Day.星期一, Day.星期二 ->      // 执行其他操作
        Day.星期三 ->  // ...
Day.星期四 ->   // ...
Day.星期五 ->     // ...
Day.星期六 ->   // ...
    }
}
```

正如你在第二种情况（星期一 和 星期二）中看到的，也可以组合两个或更多的枚举值。

如果你的情况不完整，编译器会显示错误。你可以使用else来处理默认情况：

```
fun doOnDay(day: Day) {
    when(day) {
Day.星期一 ->      // 工作
        Day.星期二 ->     // 努力工作
        Day.星期三 ->  // ...
Day.Thursday ->   //
        Day.Friday ->      //
        else ->          // 周末派对
    }
}
```

虽然同样可以使用if-then-else结构来实现，when处理缺失的enum值，使其更自然。

有关kotlin enum的更多信息，请查看这里


```
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

fun doOnDay(day: Day) {
    when(day) {
        Day.Sunday ->      // Do something
        Day.Monday, Day.Tuesday ->      // Do other thing
        Day.Wednesday ->  // ...
        Day.Thursday ->   // ...
        Day.Friday ->     // ...
        Day.Saturday ->   // ...
    }
}
```

As you can see in second case line (`Monday` and `Tuesday`) it is also possible to combine two or more **`enum`** values.

If your cases are not exhaustive the compile will show an error. You can use **`else`** to handle default cases:

```
fun doOnDay(day: Day) {
    when(day) {
        Day.Monday ->      // Work
        Day.Tuesday ->     // Work hard
        Day.Wednesday ->  // ...
        Day.Thursday ->   //
        Day.Friday ->     //
        else ->           // Party on weekend
    }
}
```

Though the same can be done using `if-then-`**`else`** construct, **`when`** takes care of missing **`enum`** values and makes it more natural.

Check here for more information about kotlin **`enum`**

第10章：Kotlin中的循环

第10.1节：遍历可迭代对象

你可以使用标准的for循环遍历任何可迭代对象：

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Kotlin中许多东西都是可迭代的，比如数字范围：

```
for(i in 0..9) {
    print(i)
}
```

如果在迭代时需要索引：

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

标准库中还包含一种函数式的迭代方法，没有明显的语言结构，使用forEach函数：

```
iterable.forEach {
    print(it.toString())
}
```

此示例中的it隐式表示当前元素，参见Lambda函数

第10.2节：重复执行某个操作x次

```
repeat(10) { i ->
println("这行将被打印10次")
    println("我们现在处于第${i + 1}次循环")
}
```

第10.3节：break和continue

break 和 continue 关键字的作用与其他语言中的作用相同。

```
while(true) {
    if(condition1) {
        continue // 将立即开始下一次迭代，不执行循环体的其余部分

    }
    if(condition2) {
        break // 将完全退出循环
    }
}
```

如果你有嵌套循环，可以给循环语句加标签，并限定 break 和 continue 语句，以指定你想继续或跳出哪个循环：

Chapter 10: Loops in Kotlin

Section 10.1: Looping over iterables

You can loop over any iterable by using the standard for-loop:

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Lots of things in Kotlin are iterable, like number ranges:

```
for(i in 0..9) {
    print(i)
}
```

If you need an index while iterating:

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

There is also a functional approach to iterating included in the standard library, without apparent language constructs, using the forEach function:

```
iterable.forEach {
    print(it.toString())
}
```

`it` in this example implicitly holds the current element, see Lambda Functions

Section 10.2: Repeat an action x times

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

Section 10.3: Break and continue

Break and continue keywords work like they do in other languages.

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of the
loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

If you have nested loops, you can label the loop statements and qualify the break and continue statements to specify which loop you want to continue or break:

```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // 将跳出内层循环
        break@inner // 将跳出内层循环
        break@outer // 将跳出外层循环
    }
}
```

不过，这种方法不适用于函数式的 forEach 结构。

第10.4节：在 kotlin 中遍历 Map

```
//遍历一个 map，同时获取键和值

var map = hashMapOf(1 到 "foo", 2 到 "bar", 3 到 "baz")

for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

第10.5节：递归

在Kotlin中，像大多数编程语言一样，也可以通过递归实现循环。

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)

println(factorial(10)) // 3628800
```

在上面的例子中，factorial 函数会不断地自我调用，直到满足给定的条件。

第10.6节：while循环

while和do-while循环的工作方式与其他语言相同：

```
while(condition) {
    doSomething()
}

do {
doSomething()
} while (condition)
```

在 do-while 循环中，条件块可以访问循环体中声明的值和变量。

第10.7节：用于迭代的函数构造

Kotlin 标准库 也提供了许多有用的函数，用于对集合进行迭代操作。

例如， map 函数可以用来转换一个项目列表。

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

这种风格的众多优点之一是它允许以类似的方式链式操作。比如，如果需要对上述列表进行偶数过滤，只需稍作修改即可。可以使用 filter 函数。


```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // Will break the inner loop
        break@inner // Will break the inner loop
        break@outer // Will break the outer loop
    }
}
```

This approach won't work for the functional `forEach` construct, though.

Section 10.4: Iterating over a Map in kotlin

```
//iterates over a map, getting the key and value at once

var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")

for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

Section 10.5: Recursion

Looping via recursion is also possible in Kotlin as in most programming languages.

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)

println(factorial(10)) // 3628800
```

In the example above, the `factorial` function will be called repeatedly by itself until the given condition is met.

Section 10.6: While Loops

While and do-while loops work like they do in other languages:

```
while(condition) {
    doSomething()
}

do {
    doSomething()
} while (condition)
```

In the do-while loop, the condition block has access to values and variables declared in the loop body.

Section 10.7: Functional constructs for iteration

The Kotlin Standard Library also provides numerous useful functions to iteratively work upon collections.

For example, the `map` function can be used to transform a list of items.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

One of the many advantages of this style is it allows to chain operations in a similar fashion. Only a minor modification would be required if say, the list above were needed to be filtered for even numbers. The `filter` function can be used.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```



```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```

第11章：区间

范围表达式是通过具有操作符形式 .. 的 rangeTo 函数构成的，并辅以 in 和 !in。范围适用于任何可比较类型，但对于整型原始类型，它有一个优化的实现

第11.1节：整型范围

整型范围（IntRange、LongRange、CharRange）有一个额外的特性：它们可以被迭代。编译器会负责将其类似于 Java 的索引 for 循环进行转换，而不会产生额外开销

```
for (i in 1..4) print(i) // 输出 "1234"
for (i in 4..1) print(i) // 不输出任何内容
```

第11.2节：downTo() 函数

想要逆序迭代数字吗？很简单。你可以使用标准库中定义的 downTo() 函数

```
for (i in 4 downTo 1) print(i) // 输出 "4321"
```

第11.3节：step() 函数

可以用任意步长（不等于1）迭代数字吗？当然可以，step() 函数会帮你实现

```
for (i in 1..4 step 2) print(i) // 输出 "13"
for (i in 4 downTo 1 step 2) print(i) // 输出 "42"
```

第11.4节：until函数

要创建一个不包含其结束元素的范围，可以使用 until 函数：

```
for (i in 1 until 10) { // i 在 [1, 10) 范围内，10 被排除
println(i)
}
```


Chapter 11: Ranges

Range expressions are formed with rangeTo functions that have the operator form .. which is complemented by in and !in. Range is defined for any comparable type, but for integral primitive types it has an optimized implementation

Section 11.1: Integral Type Ranges

Integral type ranges (IntRange , LongRange , CharRange) have an extra feature: they can be iterated over. The compiler takes care of converting this analogously to Java's indexed for-loop, without extra overhead

```
for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

Section 11.2: downTo() function

if you want to iterate over numbers in reverse order? It's simple. You can use the downTo() function defined in the standard library

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

Section 11.3: step() function

Is it possible to iterate over numbers with arbitrary step, not equal to 1? Sure, the step() function will help you

```
for (i in 1..4 step 2) print(i) // prints "13"
for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

Section 11.4: until function

To create a range which does not include its end element, you can use the until function:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
println(i)
}
```

第12章：正则表达式

第12.1节：When表达式中正则匹配的惯用法

使用不可变局部变量：

比“匿名临时变量”模板使用更少的水平空间，但占用更多的垂直空间。如果 when 表达式在循环中，优先使用此方法——在这种情况下，正则表达式定义应放在循环外部。

```
import kotlin.text.regex

var string = /* 某个字符串 */

val regex1 = Regex( /* 模式 */ )
val regex2 = Regex( /* 模式 */ )
/* 等等 */

when {
regex1.matches(string) -> /* 执行操作 */
    regex2.matches(string) -> /* 执行操作 */
    /* 等等 */
}
```

使用匿名临时变量：

比“不可变局部变量”模板使用更少的垂直空间，但占用更多的水平空间。如果 then 条件不满足，则不应使用此方法当表达式在循环中时。

```
导入 kotlin.text.regex

变量 string = /* 某个字符串 */

when {
Regex( /* 模式 */ ).匹配(string) -> /* 执行操作 */
    Regex( /* 模式 */ ).匹配(string) -> /* 执行操作 */
    /* 等等 */
}
```

使用访问者模式：

具有紧密模拟带参数的 when 语法的优点。这是有益的，因为它更清楚地指示了 when 表达式的参数，同时也避免了程序员在每个 whenEntry 中重复 when 参数时可能出现的某些错误。此实现的访问者模式可以使用“不可变局部变量”或“匿名临时变量”模板。

```
导入 kotlin.text.regex

变量 string = /* 某个字符串 */

when (RegexWhenArgument(string)) {
    Regex( /* 模式 */ ) -> /* 执行操作 */
    Regex( /* 模式 */ ) -> /* 执行操作 */
    /* 等等 */
}
```


Chapter 12: Regex

Section 12.1: Idioms for Regex Matching in When Expression

Using immutable locals:

Uses less horizontal space but more vertical space than the "anonymous temporaries" template. Preferable over the "anonymous temporaries" template if the **when** expression is in a loop--in that case, regex definitions should be placed outside the loop.

```
import kotlin.text.regex

var string = /* some string */

val regex1 = Regex( /* pattern */ )
val regex2 = Regex( /* pattern */ )
/* etc */

when {
    regex1.matches(string) -> /* do stuff */
    regex2.matches(string) -> /* do stuff */
    /* etc */
}
```

Using anonymous temporaries:

Uses less vertical space but more horizontal space than the "immutable locals" template. Should not be used if then **when** expression is in a loop.

```
import kotlin.text.regex

var string = /* some string */

when {
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    /* etc */
}
```

Using the visitor pattern:

Has the benefit of closely emulating the "argument-ful" **when** syntax. This is beneficial because it more clearly indicates the argument of the **when** expression, and also precludes certain programmer mistakes that could arise from having to repeat the **when** argument in every `whenEntry`. Either the "immutable locals" or the "anonymous temporaries" template may be used with this implementation the visitor pattern.

```
import kotlin.text.regex

var string = /* some string */

when (RegexWhenArgument(string)) {
    Regex( /* pattern */ ) -> /* do stuff */
    Regex( /* pattern */ ) -> /* do stuff */
    /* etc */
}
```

以及 when 表达式参数包装类的最小定义：

```
类 RegexWhenArgument (val whenArgument: CharSequence) {
    操作符函数 equals(whenEntry: Regex) = whenEntry.匹配(whenArgument)
    重写操作符函数 equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

第12.2节：Kotlin中的正则表达式简介

本文展示了如何使用 Regex 类中的大多数函数，如何安全地处理与 Regex 函数相关的空值，以及原始字符串如何使编写和阅读正则表达式模式更容易。

正则表达式类

在 Kotlin 中使用正则表达式，需要使用 Regex(pattern: String) 类，并在该正则对象上调用 find(..) 或 replace(..) 等函数。

一个使用 Regex 类的示例，如果 input 字符串包含 c 或 d，则返回 true：

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")     // matched: true
```

理解所有 Regex 函数的关键是结果基于正则表达式pattern 和 input 字符串的匹配。有些函数要求完全匹配，而其他函数只要求部分匹配。示例中使用的 containsMatchIn(..) 函数要求部分匹配，后文将对此进行解释。

正则表达式的空安全

find(..) 和 matchEntire(..) 都会返回一个 MatchResult? 对象。 ? 字符出现在 MatchResult 后，是为了让 Kotlin 能够安全地处理 null。

一个示例演示了当 find(..) 函数返回 null 时，Kotlin 如何安全地处理来自 Regex 函数的 null：

```
val matchResult =
Regex("c|d").find("efg")            // matchResult: null
val a = matchResult?.value              // a: null
val b = matchResult?.value.orEmpty()   // b: ""
a?.toUpperCase()                       // 仍然需要问号。=> null
b.toUpperCase()                        // 直接访问函数。=> ""
```

使用 orEmpty() 函数后，b 不能为 null，调用 b 上的函数时 ? 字符是不必要的。

如果你不在意这种对 null 值的安全处理，Kotlin 允许你像在 Java 中那样使用 null 值，使用 !! 字符：

```
a!!.toUpperCase()                      // => KotlinNullPointerException
```

正则表达式中的原始字符串

Kotlin 相较于 Java 提供了改进，使用 原始字符串 可以编写纯正则表达式模式，无需像 Java 字符串那样使用双反斜杠。原始字符串用三重引号表示：

```
"""\d{3}-\d{3}-\d{4}"""  // Kotlin 原始字符串
"\\d{3}-\\d{3}-\\d{4}"   // 标准 Java 字符串
```


And the minimal definition of the wrapper class for the **when** expression argument:

```
class RegexWhenArgument (val whenArgument: CharSequence) {
    operator fun equals(whenEntry: Regex) = whenEntry.matches(whenArgument)
    override operator fun equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

Section 12.2: Introduction to regular expressions in Kotlin

This post shows how to use most of the functions in the Regex class, work with null safely related to the Regex functions, and how raw strings makes it easier to write and read regex patterns.

The RegEx class

To work with regular expressions in Kotlin, you need to use the `Regex(pattern: String)` class and invoke functions like `find(..)` or `replace(..)` on that regex object.

An example on how to use the Regex class that returns true if the `input` string contains c or d:

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")     // matched: true
```

The essential thing to understand with all the Regex functions is that the result is based on matching the regex `pattern` and the `input` string. Some of the functions requires a full match, while the rest requires only a partial match. The `containsMatchIn(..)` function used in the example requires a partial match and is explained later in this post.

Null safety with regular expressions

Both `find(..)` and `matchEntire(..)` will return a `MatchResult?` object. The ? character after `MatchResult` is necessary for Kotlin to handle null safely.

An example that demonstrates how Kotlin handles null safely from a `Regex` function, when the `find(..)` function returns null:

```
val matchResult =
    Regex("c|d").find("efg")            // matchResult: null
val a = matchResult?.value              // a: null
val b = matchResult?.value.orEmpty()    // b: ""
a?.toUpperCase()                        // Still needs question mark. => null
b.toUpperCase()                         // Accesses the function directly. => ""
```

With the `orEmpty()` function, `b` can't be null and the ? character is unnecessary when you call functions on `b`.

If you don't care about this safe handling of null values, Kotlin allows you to work with null values like in Java with the `!!` characters:

```
a!!.toUpperCase()                       // => KotlinNullPointerException
```

Raw strings in regex patterns

Kotlin provides an improvement over Java with a raw string that makes it possible to write pure regex patterns without double backslashes, that are necessary with a Java string. A raw string is represented with a triple quote:

```
"""\d{3}-\d{3}-\d{4}"""  // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"   // standard Java string
```

find(input: CharSequence, startIndex: Int): MatchResult?

输入字符串将与 Regex 对象中的模式进行匹配。它返回一个 MatchResult? 对象，该对象包含从 startIndex 开始的第一个匹配文本，如果模式未匹配输入字符串，则返回 null。结果字符串从 MatchResult? 对象的 value 属性中获取。startIndex 参数是可选的，默认值为 0。

从包含联系信息的字符串中提取第一个有效的电话号码：

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

如果input字符串中没有有效的电话号码，变量phoneNumber将为null。

findAll(input: CharSequence, startIndex: Int): Sequence

返回所有匹配正则表达式pattern的input字符串中的匹配项。

从包含字母和数字的文本中打印出所有用空格分隔的数字：

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

变量matchedResults是一个包含MatchResult对象的序列。如果input字符串中没有数字，findAll(..)函数将返回一个空序列。

matchEntire(input: CharSequence): MatchResult?

如果input字符串中的所有字符都匹配正则表达式pattern，则返回与input相等的字符串。否则，**返回null**。

如果整个输入字符串是数字，则返回该输入字符串：

```
val a = Regex("""\d+""").matchEntire("100")?.value            // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value    // b: null
```

matches(input: CharSequence): Boolean

如果整个输入字符串匹配正则表达式，则返回true。否则返回false。

测试两个字符串是否只包含数字：

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

containsMatchIn(input: CharSequence): Boolean

如果输入字符串的部分内容匹配正则表达式，则返回true。否则返回false。

测试两个字符串是否至少包含一个数字：

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
```


find(input: CharSequence, startIndex: Int): MatchResult?

The input string will be matched against the pattern in the Regex object. It returns a Matchresult? object with the first matched text after the startIndex, or **null** if the pattern didn't match the input string. The result string is retrieved from the MatchResult? object's value property. The startIndex parameter is optional with the default value 0.

To extract the first valid phone number from a string with contact details:

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

With no valid phone number in the input string, the variable phoneNumber will be **null**.

findAll(input: CharSequence, startIndex: Int): Sequence

Returns all the matches from the input string that matches the regex pattern.

To print out all numbers separated with space, from a text with letters and digits:

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

The matchedResults variable is a sequence with MatchResult objects. With an input string without digits, the findAll(..) function will return an empty sequence.

matchEntire(input: CharSequence): MatchResult?

If all the characters in the input string matches the regex pattern, a string equal to the input will be returned. Else, **null** will be returned.

Returns the input string if the whole input string is a number:

```
val a = Regex("""\d+""").matchEntire("100")?.value            // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value    // b: null
```

matches(input: CharSequence): Boolean

Returns true if the whole input string matches the regex pattern. False otherwise.

Tests if two strings contains only digits:

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

containsMatchIn(input: CharSequence): Boolean

Returns true if part of the input string matches the regex pattern. False otherwise.

Test if two strings contains at least one digit:

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
```

```
Regex("""\d+""").containsMatchIn("Fifty dollars")     // => false
```

split(input: CharSequence, limit: Int): List

返回一个不包含所有正则表达式匹配项的新列表。

返回不包含数字的列表：

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

列表中每个分割对应一个元素。第一个input字符串中有三个数字，因此结果列表有
三个元素。

replace(input: CharSequence, replacement: String): String

将input字符串中所有匹配正则表达式pattern的部分替换为替换字符串。

将字符串中的所有数字替换为 x：

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // 结果: abxcdxef
```



```
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

split(input: CharSequence, limit: Int): List

Returns a new list without all the regex matches.

To return lists without digits:

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

There is one element in the list for each split. The first `input` string has three numbers. That results in a list with three elements.

replace(input: CharSequence, replacement: String): String

Replaces all matches of the regex `pattern` in the `input` string with the replacement string.

To replace all digits in a string with an x:

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

第13章：基础Lambda表达式

第13.1节：作为filter函数参数的Lambda

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

第13.2节：用于基准测试函数调用的Lambda

通用秒表，用于计时函数运行所需时间：

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
body()
        } 最终 {
            val 结束 = Instant.now()
            return Duration.between(开始, 结束)
        }
    }
}
```

用法：

```
val 时间 = Benchmark.实时({
    // 这里放一些长时间运行的代码 ...
})
println("代码执行时间为 $时间")
```

第13.3节：作为变量传递的Lambda表达式

```
val 是允许年龄 = { 用户: 用户 -> 用户.年龄 > 最小年龄 }
val allowedUsers = users.filter(isOfAllowedAge)
```


Chapter 13: Basic Lambdas

Section 13.1: Lambda as parameter to filter function

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Section 13.2: Lambda for benchmarking a function call

General-purpose stopwatch for timing how long a function takes to run:

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

Usage:

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

Section 13.3: Lambda passed as a variable

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

第14章：空安全

第14.1节：智能类型转换

如果编译器能够推断出某个对象在某个点上不可能为null，则你不必再使用特殊操作符：

```
var string: String? = "Hello!"
print(string.length)  // 编译错误
if(string != null) {
    // 编译器现在知道string不可能为null
print(string.length)  // 现在可以正常工作！
}
```

> 注意： 编译器不会允许你对可能在null检查和预期使用之间被修改的可变变量进行智能类型转换。
>
> 如果一个变量可以从当前代码块作用域外访问（例如，因为它是非本地对象的成员），你需要创建一个新的本地引用，然后对其进行智能类型转换并使用。

第14.2节：断言

!! 后缀忽略可空性并返回该类型的非空版本。如果对象为 null，将抛出 KotlinNullPointerException。

```
val message: String? = null
println(message!!) //抛出 KotlinNullPointerException，应用崩溃
```

第14.3节：从 Iterable 和数组中消除 null

有时我们需要将类型从 Collection<T?> 转换为 Collection<T>。在这种情况下，filterNotNull 是我们的解决方案。

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

第14.4节：空合并 / Elvis 操作符

有时希望以 if-else 方式计算可空表达式。Kotlin 中的 Elvis 操作符 ?: 可用于这种情况。

例如：

```
val value: String = data?.first() ?: "Nothing here."
```

上述表达式如果 data?.first() 或 data 本身返回 null，则返回 "Nothing here"，否则返回 data?.first() 的结果。

也可以使用相同的语法抛出异常以中止代码执行。

Chapter 14: Null Safety

Section 14.1: Smart casts

If the compiler can infer that an object can't be null at a certain point, you don't have to use the special operators anymore:

```
var string: String? = "Hello!"
print(string.length)  // Compile error
if(string != null) {
    // The compiler now knows that string can't be null
    print(string.length)  // It works now!
}
```

> **Note:** The compiler won't allow you to smart cast mutable variables that could potentially be modified between the null-check and the intended usage.
>
> If a variable is accessible from outside the scope of the current block (because they are members of a non-local object, for example), you need to create a new, local reference which you can then smart cast and use.

Section 14.2: Assertion

`!!` suffixes ignore nullability and returns a non-null version of that type. `KotlinNullPointerException` will be thrown if the object is a **`null`**.

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

Section 14.3: Eliminate nulls from an Iterable and array

Sometimes we need to change type from `Collection<T?>` to `Collections<T>`. In that case, `filterNotNull` is our solution.

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Section 14.4: Null Coalescing / Elvis Operator

Sometimes it is desirable to evaluate a nullable expression in an if-else fashion. The elvis operator, `?:`, can be used in Kotlin for such a situation.

For instance:

```
val value: String = data?.first() ?: "Nothing here."
```

The expression above returns `"Nothing here"` if **`data`**`?.first()` or **`data`** itself yield a **`null`** value else the result of **`data`**`?.first()`.

It is also possible to throw exceptions using the same syntax to abort code execution.

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> 提醒：可以使用断言操作符（例如 data!!.second()!!）抛出空指针异常（NullPointerException）

第14.5节：可空类型和非可空类型

普通类型，如String，默认不可为空。要使其能够保存null值，必须在类型后面显式加上?：String?

var string : String = "Hello World!" var nullableString: String? = null string = nullableString // 编译错误：不能将可空类型赋值给非可空类型。 nullableString = string // 但这将可以正常工作！

第14.6节：Elvis操作符(?:)

在Kotlin中，我们可以声明可以保存null引用的变量。假设我们有一个可空引用 a，我们可以说“如果 a不为null，则使用它，否则使用某个非空值 x”

```
var a: String? = "Nullable String Value"
```

现在，a可以为null。因此，当我们需要访问a的值时，就需要进行安全检查，判断它是否包含值。我们可以通过传统的if...else语句来执行此安全检查。

```
val b: Int = if (a != null) a.length else -1
```

但是这里出现了高级运算符Elvis（Elvis运算符：?:）。上面的if...else可以用Elvis运算符表示如下：

```
val b = a?.length ?: -1
```

如果?:左边的表达式（这里是：a?.length）不为null，Elvis运算符返回该表达式，否则返回右边的表达式（这里是：-1）。只有当左边表达式为null时，右边的表达式才会被计算。

第14.7节：安全调用运算符

要访问可空类型的函数和属性，必须使用特殊运算符。

第一个是?.，它会返回你试图访问的属性或函数，如果对象为null，则返回null：

```
val string: String? = "Hello World!"
print(string.length)   // 编译错误：不能直接访问可空类型的属性。
print(string?.length)  // 如果字符串不为null，则打印字符串长度，否则打印“null”。
```

惯用法：在同一个经过null检查的对象上调用多个方法

调用经过空值检查对象的多个方法的优雅方式是使用 Kotlin 的apply，如下所示：

```
obj?.apply {
    foo()
    bar()
}
```

这将在obj非空时调用foo和bar（在apply代码块中this指代obj），否则跳过整个代码块。


```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```


> Reminder: NullPointerExceptions can be thrown using the assertion operator (e.g. `data!!.second()!!`)

Section 14.5: Nullable and Non-Nullable types

Normal types, like `String`, are not nullable. To make them able to hold null values, you have to explicitly denote that by putting a ? behind them: `String?`

var string : String = "Hello World!" var nullableString: String? = null string = nullableString // Compiler error: Can't assign nullable to non-nullable type. nullableString = string // This will work however!

Section 14.6: Elvis Operator (?:)

In Kotlin, we can declare variable which can hold **`null reference`**. Suppose we have a nullable reference a, we can say "if a is not null, use it, otherwise use some non-null value x"

```
var a: String? = "Nullable String Value"
```

Now, a can be null. So when we need to access value of a, then we need to perform safety check, whether it contains value or not. We can perform this safety check by conventional **`if...else`** statement.

```
val b: Int = if (a != null) a.length else -1
```

But here comes advance operator `Elvis`(Operator Elvis : `?:`). Above **`if...else`** can be expressed with the Elvis operator as below:

```
val b = a?.length ?: -1
```

If the expression to the left of `?:` (here : `a?.length`) is not null, the elvis operator returns it, otherwise it returns the expression to the right (here: `-1`). Right-hand side expression is evaluated only if the left-hand side is null.

Section 14.7: Safe call operator

To access functions and properties of nullable types, you have to use special operators.

The first one, `?.`, gives you the property or function you're trying to access, or it gives you null if the object is null:

```
val string: String? = "Hello World!"
print(string.length)   // Compile error: Can't directly access property of nullable type.
print(string?.length)  // Will print the string's length, or "null" if the string is null.
```

Idiom: calling multiple methods on the same, null-checked object

An elegant way to call multiple methods of a null-checked object is using Kotlin's `apply` like this:

```
obj?.apply {
    foo()
    bar()
}
```

This will call `foo` and `bar` on `obj` (which is **`this`** in the `apply` block) only if `obj` is non-null, skipping the entire block

否则。

要将可空变量引入作用域作为非空引用，同时不使其成为函数和属性调用的隐式接收者，可以使用let代替apply：

```
nullable?.let { notnull ->
    notnull.foo()
notnull.bar()
}
```

notnull可以命名为任意名称，或者省略，直接通过隐式的 lambda 参数it来使用。

otherwise.

To bring a nullable variable into scope as a non-nullable reference without making it the implicit receiver of function and property calls, you can use `let` instead of `apply`:

```
nullable?.let { notnull ->
    notnull.foo()
    notnull.bar()
}
```

`notnull` could be named anything, or even left out and used through the implicit lambda parameter `it`.

第15章：类委托

Kotlin 类可以通过将其方法和属性委托给另一个实现该接口的对象来实现接口。这提供了一种使用关联而非继承来组合行为的方式。

第15.1节：将方法委托给另一个类

```
接口 Foo {
    函数 example()
}

class Bar {
    函数 example() {
println("Hello, world!")
    }
}

类 Baz(b : Bar) : Foo 由 b 委托

Baz(Bar()).example()
```

该示例打印Hello，world！

Chapter 15: Class Delegation

A Kotlin class may implement an interface by delegating its methods and properties to another object that implements that interface. This provides a way to compose behavior using association rather than inheritance.

Section 15.1: Delegate a method to another class

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

The example prints `Hello, world!`

第16章：类继承

参数	详情
基类	被继承的类
派生类	继承自基类的类
初始化参数	传递给基类构造函数的参数
函数定义	派生类中的函数，其代码与基类中相同函数不同
DC-对象	“派生类-对象”具有派生类类型的对象

任何面向对象编程语言都有某种形式的类继承。让我来复习一下：

想象你需要编程表示一堆水果：苹果、橙子和梨。它们在大小、形状和颜色上都有所不同，
这就是为什么我们有不同的类。

但假设它们的差异暂时不重要，你只想要一个水果，不管具体是哪种？那么getFruit()的返回类型会是什么？

答案是类水果。我们创建一个新类，让所有水果都继承自它！

第16.1节：基础知识：“open”关键字

在 Kotlin 中，类默认是final的，这意味着它们不能被继承。

要允许类被继承，使用open关键字。

```
open class Thing {
    // 我现在可以被继承了！
}
```

> 注意：抽象类、密封类和接口默认都是open的。

第16.2节：从类继承字段

定义基类：

```
open class BaseClass {
    val x = 10
}
```

定义派生类：

```
class DerivedClass: BaseClass() {
    fun foo() {
println("x 等于 " + x)
    }
}
```

使用子类：

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // 输出：'x 等于 10'
}
```


Chapter 16: Class Inheritance

Parameter	Details
Base Class	Class that is inherited from
Derived Class	Class that inherits from Base Class
Init Arguments	Arguments passed to constructor of Base Class
Function Definition	Function in Derived Class that has different code than the same in the Base Class
DC-Object	”Derived Class-Object“ Object that has the type of the Derived Class

Any object-oriented programming language has some form of class inheritance. Let me revise:

Imagine you had to program a bunch of fruit: `Apples`, `Oranges` and `Pears`. They all differ in size, shape and color, that's why we have different classes.

But let's say their differences don't matter for a second and you just want a `Fruit`, no matter which exactly? What return type would `getFruit()` have?

The answer is class `Fruit`. We create a new class and make all fruits inherit from it!

Section 16.1: Basics: the 'open' keyword

In Kotlin, classes are **final by default** which means they cannot be inherited from.

To allow inheritance on a class, use the **`open`** keyword.

```
open class Thing {
    // I can now be extended!
}
```

> **Note:** abstract classes, sealed classes and interfaces will be **`open`** by default.

Section 16.2: Inheriting fields from a class

Defining the base class:

```
open class BaseClass {
    val x = 10
}
```

Defining the derived class:

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

Using the subclass:

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

第16.3节：从类继承方法

定义基类：

```
open class Person {
    fun jump() {
println("跳跃中...")
    }
}
```

定义派生类：

```
class Ninja: Person() {
    fun sneak() {
println("偷偷摸摸...")
    }
}
```

Ninja 可以访问 Person 中的所有方法

```
fun main(args: Array<String>) {
    val ninja = Ninja()
ninja.jump()  // 输出: 'Jumping...'
    ninja.sneak() // 输出: 'Sneaking around...'
}
```

第16.4节：重写属性和方法

重写属性（包括只读和可变）：

```
抽象类 车 {
    抽象属性 名称: 字符串;
    开放变量 速度: 整数 = 0;
}

类 坏车(重写属性 名称: 字符串) : 车() {
    重写变量 速度: 整数
        获取() = 0
        设置(值) {
            抛出 不支持的操作异常("汽车坏了")
        }
}

fun main(args: Array<String>) {
    属性 车: 车 = 坏车("Lada")
    车.速度 = 10
}
```

重写方法：

```
    接口 船 {
    函数 航行()
    函数 沉没()
}

对象 泰坦尼克号 : 船 {

    变量 能否航行 = 真

    override 函数 航行() {
        沉没()
    }

    重写函数 沉没() {
```


Section 16.3: Inheriting methods from a class

Defining the base class:

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

Defining the derived class:

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

The Ninja has access to all of the methods in Person

```
fun main(args: Array<String>) {
    val ninja = Ninja()
    ninja.jump()  // prints: 'Jumping...'
    ninja.sneak() // prints: 'Sneaking around...'
}
```

Section 16.4: Overriding properties and methods

Overriding properties (both read-only and mutable):

```
abstract class Car {
    abstract val name: String;
    open var speed: Int = 0;
}

class BrokenCar(override val name: String) : Car() {
    override var speed: Int
        get() = 0
        set(value) {
            throw UnsupportedOperationException("The car is broken")
        }
}

fun main(args: Array<String>) {
    val car: Car = BrokenCar("Lada")
    car.speed = 10
}
```

Overriding methods:

```
interface Ship {
    fun sail()
    fun sink()
}

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

    override fun sink() {
```

```
能否航行 = 假
    }
}
```



```
        canSail = false
    }
}
```

第17章：可见性修饰符

在Kotlin中，有4种可见性修饰符可用。

公共的: 任何地方都可以访问。

私有： 只能从模块代码中访问。

受保护的： 只能从定义它的类及其派生类中访问。

内部的： 只能从定义它的类的作用域中访问。

第17.1节：代码示例

公共的： `public val name = "Avijit"`

私有的： `private val name = "Avijit"`

受保护的： `protected val name = "Avijit"`

内部的： `internal val name = "Avijit"`

Chapter 17: Visibility Modifiers

In Kotlin, there are 4 types of visibility modifiers are available.

Public: This can be accessed from anywhere.

Private: This can only be accessed from the module code.

Protected: This can only be accessed from the class defining it and any derived classes.

Internal: This can only be accessed from the scope of the class defining it.

Section 17.1: Code Sample

Public: `public val name = "Avijit"`

Private: `private val name = "Avijit"`

Protected: `protected val name = "Avijit"`

Internal: `internal val name = "Avijit"`

第18章：泛型

参数	详情
类型名称	泛型参数的类型名称
上界	协变类型
下界	逆变类型
类名	类的名称

列表可以包含数字、单词或任何东西。这就是为什么我们称列表为泛型。

泛型基本上用于定义一个类可以包含哪些类型以及一个对象当前包含哪种类型。

第18.1节：声明处变异

声明处变异可以被视为对所有使用处的使用处变异的一次性声明。

```
类 Consumer<in T> { 函数 consume(t: T) { ... } }

函数 charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // 可以，因为是in投影
val anyConsumer : Consumer<Any> = charSequenceConsumer() // 错误，Any不能传递

val outConsumer : Consumer<out CharSequence> = ... // 错误，T是`in`参数
```

声明处变异的广泛示例是List<out T>，它是不可变的，因此T只作为返回值类型出现，以及Comparator<in T>，它只接收T作为参数。

第18.2节：使用处变异

使用处变异类似于Java的通配符：

出投影：

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // 可以，因为上界是SomeType

takeList.add(takenValue) // 错误，泛型的下界未指定
```

投影内：

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // 可以，因为下界是 SomeType

putList[0] // 该表达式类型为 Any，因为未指定上界
```

星投影

```
val starList : MutableList<*> = ... // Java: List<?>

starList[0] // 该表达式类型为 Any，因为未指定上界
```


Chapter 18: Generics

Parameter	Details
TypeName	Type Name of generic parameter
UpperBound	Covariant Type
LowerBound	Contravariant Type
ClassName	Name of the class

A List can hold numbers, words or really anything. That's why we call the List *generic*.

Generics are basically used to define which types a class can hold and which type an object currently holds.

Section 18.1: Declaration-site variance

Declaration-site variance can be thought of as declaration of use-site variance once and for all the use-sites.

```
class Consumer<in T> { fun consume(t: T) { ... } }

fun charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // OK since in-projection
val anyConsumer : Consumer<Any> = charSequenceConsumer() // Error, Any cannot be passed

val outConsumer : Consumer<out CharSequence> = ... // Error, T is `in`-parameter
```

Widespread examples of declaration-site variance are `List<out T>`, which is immutable so that `T` only appears as the return value type, and `Comparator<in T>`, which only receives `T` as argument.

Section 18.2: Use-site variance

Use-site variance is similar to Java wildcards:

Out-projection:

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // OK, since upper bound is SomeType

takeList.add(takenValue) // Error, lower bound for generic is not specified
```

In-projection:

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // OK, since lower bound is SomeType

putList[0] // This expression has type Any, since no upper bound is specified
```

Star-projection

```
val starList : MutableList<*> = ... // Java: List<?>

starList[0] // This expression has type Any, since no upper bound is specified
```

```
starList.add(someValue) // 错误，泛型的下界未指定
```

另见：

- 变体泛型 在从 Java 调用 Kotlin 时的互操作性。


```
starList.add(someValue) // Error, lower bound for generic is not specified
```

See also:

- Variant Generics interoperability when calling Kotlin from Java.

第19章：接口

第19.1节：带有默认实现的接口

Kotlin中的接口可以为函数提供默认实现：

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

实现此类接口的类可以直接使用这些函数，无需重新实现

```
class MyClass: MyInterface {
    // 这里无需重新实现
}
val instance = MyClass()
instance.withImplementation()
```

属性

默认实现同样适用于属性的getter和setter：

```
interface MyInterface2 {
    val helloWorld
        get() = "Hello World!"
}
```

接口访问器的实现不能使用支持字段

```
接口 MyInterface3 {
    // 这个属性无法编译！
    变量 helloWorld: 整数
        获取() = 字段
        设置(值) { 字段 = 值 }
}
```

多重实现

当多个接口实现相同的函数，或者它们都定义了一个或多个实现时，派生类需要手动解决正确的调用

```
接口 A {
    函数 未实现()
    函数 仅在A中实现() { 打印("仅A") }
    函数 在两者中实现() { 打印("两者，A") }
    函数 在一个中实现() { 打印("在A中实现") }
}

接口 B {
    函数 在两者中实现() { 打印("两者，B") }
    函数 在一个中实现() // 仅定义
}

class MyClass: A, B {
```


Chapter 19: Interfaces

Section 19.1: Interface with default implementations

An interface in Kotlin can have default implementations for functions:

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

Classes implementing such interfaces will be able to use those functions without reimplementing

```
class MyClass: MyInterface {
    // No need to reimplement here
}
val instance = MyClass()
instance.withImplementation()
```

Properties

Default implementations also work for property getters and setters:

```
interface MyInterface2 {
    val helloWorld
        get() = "Hello World!"
}
```

Interface accessors implementations can't use backing fields

```
interface MyInterface3 {
    // this property won't compile!
    var helloWorld: Int
        get() = field
        set(value) { field = value }
}
```

Multiple implementations

When multiple interfaces implement the same function, or all of them define with one or more implementing, the derived class needs to manually resolve proper call

```
interface A {
    fun notImplemented()
    fun implementedOnlyInA() { print("only A") }
    fun implementedInBoth() { print("both, A") }
    fun implementedInOne() { print("implemented in A") }
}

interface B {
    fun implementedInBoth() { print("both, B") }
    fun implementedInOne() // only defined
}

class MyClass: A, B {
```

```
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() 可以在实例中正常使用

    // 类需要定义如何使用接口函数
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // 即使只有一个实现，也有多个定义
    override fun implementedInOne() {
       super<A>.implementedInOne()
       print("implementedInOne 类实现")
    }
}
```

第19.2节：接口中的属性

你可以在接口中声明属性。由于接口不能有状态，你只能将属性声明为抽象的，或者为访问器提供默认实现。

```
interface MyInterface {
    val property: Int // 抽象的

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

第19.3节：super关键字

```
interface MyInterface {
    fun funcOne() {
        //可选的函数体
print("具有默认实现的函数")
    }
}
```

如果接口中的方法有自己的默认实现，我们可以使用super关键字来访问它。

```
super.funcOne()
```

第19.4节：基本接口

Kotlin接口包含抽象方法的声明和默认方法的实现，尽管它们不能存储状态。

```
interface MyInterface {
    fun bar()
```



```
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() can by normally used in instances

    // class needs to define how to use interface functions
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // even if there's only one implementation, there multiple definitions
    override fun implementedInOne() {
        super<A>.implementedInOne()
        print("implementedInOne class implementation")
    }
}
```

Section 19.2: Properties in Interfaces

You can declare properties in interfaces. Since an interface cannot have state you can only declare a property as abstract or by providing default implementation for the accessors.

```
interface MyInterface {
    val property: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

Section 19.3: super keyword

```
interface MyInterface {
    fun funcOne() {
        //optional body
        print("Function with default implementation")
    }
}
```

If the method in the interface has its own default implementation, we can use super keyword to access it.

```
super.funcOne()
```

Section 19.4: Basic Interface

A Kotlin interface contains declarations of abstract methods, and default method implementations although they cannot store state.

```
interface MyInterface {
    fun bar()
```

```
}
```

该接口现在可以被类如下实现：

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

第19.5节：实现多个带有默认实现的接口时的冲突

当实现多个具有相同名称且包含默认实现的方法的接口时，编译器无法确定应使用哪个实现。在发生冲突的情况下，开发者必须重写冲突的方法并提供自定义实现。该实现可以选择是否委托给默认实现。

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
super<FirstTrait>.foo()  // 委托给 FirstTrait 的默认实现
        super<SecondTrait>.foo() // 委托给 SecondTrait 的默认实现
    }

    // 函数 bar() 只在一个接口中有默认实现，因此是可以的。
}
```



```
}
```

This interface can now be implemented by a class as follows:

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

Section 19.5: Conflicts when Implementing Multiple Interfaces with Default Implementations

When implementing more than one interface that have methods of the same name that include default implementations, it is ambiguous to the compiler which implementation should be used. In the case of a conflict, the developer must override the conflicting method and provide a custom implementation. That implementation may choose to delegate to the default implementations or not.

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
        super<FirstTrait>.foo()  // delegate to the default implementation of FirstTrait
        super<SecondTrait>.foo() // delegate to the default implementation of SecondTrait
    }

    // function bar() only has a default implementation in one interface and therefore is ok.
}
```

第20章：单例对象

对象（object）是一种特殊的类，可以使用 object 关键字声明。对象类似于 Java 中的单例（设计模式）。它也充当 Java 的静态部分。对于从 Java 转向 Kotlin 的初学者来说，可以大量使用此功能，代替静态或单例。

第20.1节：用作替代 Java 的静态方法/字段

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

在任何其他类中，只需这样调用变量和函数：

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

第20.2节：作为单例使用

Kotlin 对象实际上就是单例。它的主要优点是你不必使用 SomeSingleton.INSTANCE 来获取单例的实例。

在 Java 中，你的单例看起来是这样的：

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

在 Kotlin 中，等效的代码是

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

使用起来显然更简洁。

Chapter 20: Singleton objects

An *object* is a special kind of class, which can be declared using **`object`** keyword. Objects are similar to Singletons (a design pattern) in java. It also functions as the static part of java. Beginners who are switching from java to kotlin can vastly use this feature, in place of static, or singletons.

Section 20.1: Use as replacement of static methods/fields of java

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

From any other class, just invoke the variable and functions in this way:

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

Section 20.2: Use as a singleton

Kotlin objects are actually just singletons. Its primary advantage is that you don't have to use `SomeSingleton.INSTANCE` to get the instance of the singleton.

In java your singleton looks like this:

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

In kotlin, the equivalent code is

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

It's obviously less verbose to use.

第21章：协程

Kotlin实验性（尚未稳定）协程实现的示例

第21.1节：简单协程，延迟1秒但不阻塞

（来自官方文档）

```
fun main(args: Array<String>) {
launch(CommonPool) { // 在公共线程池中创建新协程
        delay(1000L) // 非阻塞延迟1秒（默认时间单位为毫秒）
        println("World!") // 延迟后打印
    }
println("Hello,") // 主函数在协程延迟时继续执行
    Thread.sleep(2000L) // 阻塞主线程2秒以保持JVM存活
}
```

结果

```
Hello,
World!
```


Chapter 21: coroutines

Examples of Kotlin's experimental(yet) implementation of coroutines

Section 21.1: Simple coroutine which delay's 1 second but not blocks

(from official doc)

```
fun main(args: Array<String>) {
    launch(CommonPool) { // create new coroutine in common thread pool
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main function continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

result

```
Hello,
World!
```

第22章：注解

第22.1节：元注解

声明注解时，可以使用以下元注解包含元信息：

- @Target：指定可以用该注解标注的元素类型（类、函数、属性、表达式等）
- @Retention指定注解是否存储在编译后的类文件中，以及是否在运行时通过反射可见（默认两者均为true）。
- @Repeatable允许在单个元素上多次使用相同的注解。
- @MustBeDocumented指定该注解是公共API的一部分，应包含在生成的API文档中显示的类或方法签名中。

示例：

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

第22.2节：声明注解

注解是将元数据附加到代码上的一种方式。要声明一个注解，请将注解修饰符放在类的前面：

```
注解类 Strippable
```

注解可以有元注解：

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION, AnnotationTarget.VALUE_PARAMETER,
AnnotationTarget.EXPRESSION)
注解类 Strippable
```

注解，像其他类一样，可以有构造函数：

```
注解类 Strippable(val importanceValue: Int)
```

但与其他类不同，注解仅限于以下类型：

- 对应于Java原始类型的类型（Int、Long等）；
- 字符串
- 类（ Foo:: class）
- 枚举
- 其他注解
- 上述类型的数组

Chapter 22: Annotations

Section 22.1: Meta-annotations

When declaring an annotation, meta-info can be included using the following meta-annotations:

- `@Target`: specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.)
- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true.)
- `@Repeatable` allows using the same annotation on a single element multiple times.
- `@MustBeDocumented` specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

Example:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Section 22.2: Declaring an annotation

Annotations are means of attaching metadata to code. To declare an annotation, put the annotation modifier in front of a class:

```
annotation class Strippable
```

Annotations can have meta-annotations:

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION, AnnotationTarget.VALUE_PARAMETER,
AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

Annotations, like other classes, can have constructors:

```
annotation class Strippable(val importanceValue: Int)
```

But unlike other classes, is limited to the following types:

- types that correspond to Java primitive types (Int, Long etc.);
- strings
- classes (Foo:: class)
- enums
- other annotations
- arrays of the types listed above

第23章：类型别名

通过类型别名，我们可以为其他类型指定别名。它非常适合为函数类型命名，例如(String) -> Boolean或泛型类型如Pair<Person, Person>。

类型别名支持泛型。别名可以替代带泛型的类型，别名本身也可以是泛型的。

第23.1节：函数类型

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

第23.2节：泛型类型

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```


Chapter 23: Type aliases

With type aliases, we can give an alias to other type. It's ideal for giving a name to function types like `(String) -> Boolean` or generic type like `Pair<Person, Person>`.

Type aliases support generics. An alias can replace a type with generics and an alias can be generics.

Section 23.1: Function type

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Section 23.2: Generic type

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

第24章：类型安全构建器

第24.1节：类型安全的树结构构建器

构建器可以定义为一组扩展函数，这些函数以带接收者的 lambda 表达式作为参数。在这个例子中，正在构建一个JFrame的菜单：

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
menu.init()
    add(menu)
}

fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
menuItem.init()
    add(menuItem)
}
```

这些函数随后可以用来轻松构建对象的树形结构：

```
class MyFrame : JFrame() {
    init {
menuBar {
菜单("Menu1") {
                菜单项("Item1") {
                    // 初始化菜单项并绑定某个操作
                }
菜单项("Item2") {}
            }
菜单("Menu2") {
                菜单项("Item3") {}
                菜单项("Item4") {}
            }
        }
    }
}
```


Chapter 24: Type-Safe Builders

Section 24.1: Type-safe tree structure builder

Builders can be defined as a set of extension functions taking lambda expressions with receivers as arguments. In this example, a menu of a `JFrame` is being built:

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}

fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
}
```

These functions can then be used to build a tree structure of objects in an easy way:

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

第25章：委托属性

Kotlin 可以将属性的实现委托给处理对象。一些标准处理器已包含在内，
例如延迟初始化或可观察属性。也可以创建自定义处理器。

第25.1节：可观察属性

```
var foo : Int 由 Delegates.observable("1") 委托 { property, oldValue, newValue ->
    println("${property.name} 的值从 $oldValue 变为 $newValue")
}
foo = 2
```

该示例输出foo 的值从 1 变为 2

第25.2节：自定义委托

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

该示例输出Delegated value

第25.3节：延迟初始化

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

该示例输出2。

第25.4节：基于映射的属性

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

该示例输出1

第25.5节：Delegate 可以用作减少样板代码的层

考虑 Kotlin 的空类型系统和WeakReference<T>。

假设我们需要保存某种引用并且想要避免内存泄漏，这时就用到了
WeakReference。

举个例子：

```
class MyMemoryExpensiveClass {
    companion object {
```


Chapter 25: Delegated properties

Kotlin can delegate the implementation of a property to a handler object. Some standard handlers are included, such as lazy initialization or observable properties. Custom handlers can also be created.

Section 25.1: Observable properties

```
var foo : Int by Delegates.observable("1") { property, oldValue, newValue ->
    println("${property.name} was changed from $oldValue to $newValue")
}
foo = 2
```

The example prints `foo was changed from 1 to 2`

Section 25.2: Custom delegation

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

The example prints `Delegated value`

Section 25.3: Lazy initialization

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

The example prints 2.

Section 25.4: Map-backed properties

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

The example prints `1`

Section 25.5: Delegate Can be used as a layer to reduce boilerplate

Consider Kotlin's Null Type system and `WeakReference<T>`.

So let's say we have to save some sort of reference and we wanted to avoid memory leaks, here is where `WeakReference` comes in.

take for example this:

```
class MyMemoryExpensiveClass {
    companion object {
```

```
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
it.get()?.let(block)
            }
        }
    }

    init {
reference = WeakReference(this)
    }
}
```

现在这只是使用一个弱引用。为了减少这些样板代码，我们可以使用自定义属性委托来帮助我们，像这样：

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
reference = WeakReference(value)
    }
}
```

所以现在我们可以像使用普通的可空变量一样使用被WeakReference包装的变量！

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }

    init {
reference = this
    }
}
```



```
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
                it.get()?.let(block)
            }
        }
    }

    init {
        reference = WeakReference(this)
    }
}
```

Now this is just with one WeakReference. To Reduce this boilerplate, we can use a custom property delegate to help us like so:

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        reference = WeakReference(value)
    }
}
```

So Now we can use variables that are wrapped with `WeakReference` just like normal nullable variables !

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }

    init {
        reference = this
    }
}
```

第26章：反射

反射是语言在运行时而非编译时检查代码的能力。

第26.1节：引用类

要获取表示某个类的KClass对象的引用，请使用双冒号：

```
val c1 = String::class
val c2 = MyClass::class
```

第26.2节：与Java反射的互操作

要从Kotlin的KClass获取Java的Class对象，请使用.java扩展属性：

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java

val c2: Class<MyClass> = MyClass::class.java
```

后一个例子将被编译器优化，不会分配中间的KClass实例。

第26.3节：引用函数

函数在 Kotlin 中是一等公民。你可以使用双冒号获取它的引用，然后将其传递给另一个函数：

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

第26.4节：获取类的所有属性值

给定继承自BaseExample类且带有一些属性的Example类：

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
BaseExample(baseField) {

    val field3: String
        get() = "没有后备字段的属性"

    val field4 by lazy { "委托值" }

    private val privateField: String = "私有值"
}
```

可以获取一个类的所有属性：

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
println("${member.name} -> ${member.get(example)}")
```


Chapter 26: Reflection

Reflection is a language's ability to inspect code at runtime instead of compile time.

Section 26.1: Referencing a class

To obtain a reference to a `KClass` object representing some class use double colons:

```
val c1 = String::class
val c2 = MyClass::class
```

Section 26.2: Inter-operating with Java reflection

To obtain a Java's `Class` object from Kotlin's `KClass` use the `.java` extension property:

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java

val c2: Class<MyClass> = MyClass::class.java
```

The latter example will be optimized by the compiler to not allocate an intermediate `KClass` instance.

Section 26.3: Referencing a function

Functions are first-class citizens in Kotlin. You can obtain a reference on it using double colons and then pass it to another function:

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

Section 26.4: Getting values of all properties of a class

Given `Example` class extending `BaseExample` class with some properties:

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
    BaseExample(baseField) {

    val field3: String
        get() = "Property without backing field"

    val field4 by lazy { "Delegated value" }

    private val privateField: String = "Private value"
}
```

One can get hold of all properties of a class:

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
```

```
}
```

运行此代码将导致抛出异常。属性private val privateField被声明为私有，调用member.get(example)将无法成功。处理此问题的一种方法是过滤掉私有属性。为此，我们必须检查属性的Java getter的可见性修饰符。对于private val，getter不存在，因此我们可以假设是私有访问权限。

辅助函数及其用法可能如下所示：

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

另一种方法是使用反射使私有属性可访问：

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
println("${member.name} -> ${member.get(example)}")
}
```

第26.5节：设置类的所有属性的值

举例来说，我们想设置一个示例类的所有字符串属性

```
class TestClass {
    val readOnlyProperty: String
        get() = "只读！"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
```



```
}
```

Running this code will cause an exception to be thrown. Property `private val privateField` is declared private and calling `member.get(example)` on it will not succeed. One way to handle this it to filter out private properties. To do that we have to check the visibility modifier of a property's Java getter. In case of `private val` the getter does not exist so we can assume private access.

The helper function and it's usage might look like this:

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Another approach is to make private properties accessible using reflection:

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
    println("${member.name} -> ${member.get(example)}")
}
```

Section 26.5: Setting values of all properties of a class

As an example we want to set all string properties of a sample class

```
class TestClass {
    val readOnlyProperty: String
        get() = "Read only!"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
```

```
    }
}
```

获取可变属性是基于获取所有属性，然后按类型过滤可变属性。我们还需要检查可见性，因为读取私有属性会导致运行时异常。

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
.forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)}")
        }
```

要将所有String属性设置为"Our Value"，我们可以额外按返回类型过滤。由于Kotlin基于Java虚拟机，存在类型擦除，因此返回泛型类型如List<String>的属性将与List<Any>相同。遗憾的是反射并非万能，没有合理的方法避免这一点，所以你需要在使用场景中注意。

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // 我们只想要字符串类型
.filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
.forEach { prop ->
            // 我们不是打印属性，而是将其设置为某个值
prop.setter.call(instance, "我们的值")
        }
```



```
    }
}
```

Getting mutable properties builds on getting all properties, filtering mutable properties by type. We also need to check visibility, as reading private properties results in run time exception.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)}")
        }
```

To set all `String` properties to `"Our Value"` we can additionally filter by the return type. Since Kotlin is based on Java VM, Type Erasure is in effect, and thus Properties returning generic types such as `List<String>` will be the same as `List<Any>`. Sadly reflection is not a golden bullet and there is no sensible way to avoid this, so you need to watch out in your use-cases.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // We only want strings
        .filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            // Instead of printing the property we set it to some value
            prop.setter.call(instance, "Our Value")
        }
```

第27章：扩展方法

第27.1节：潜在陷阱：扩展方法是静态解析的

要调用的扩展方法是在编译时根据被访问变量的引用类型确定的。无论变量在运行时的实际类型是什么，始终会调用相同的扩展方法。

```
开放类 Super

类 Sub : Super()

函数 Super.myExtension() = "为Super定义"

函数 Sub.myExtension() = "为Sub定义"

函数 callMyExtension(myVar: Super) {
println(myVar.myExtension())
}

callMyExtension(Sub())
```

上述示例将打印"为Super定义"，因为变量myVar的声明类型是Super。

第27.2节：顶层扩展

顶级扩展方法不包含在类中。

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size - 1) {
        dest[i] += this[i]
    }
}
```

上面为类型IntArray定义了一个扩展方法。注意，扩展方法所定义的对象（称为接收者）是通过关键字this访问的。

该扩展方法可以这样调用：

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

第27.3节：延迟扩展属性的解决方法

假设你想创建一个计算开销较大的扩展属性。因此你希望通过使用lazy属性委托并引用当前实例（this）来缓存计算结果，但正如Kotlin问题KT-9686和KT-13053中所解释的那样，你无法这样做。不过，这里提供了一个官方的解决方法。

在示例中，扩展属性是color。它使用了一个显式的colorCache，可以与this一起使用，因为不需要lazy：

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()
```


Chapter 27: Extension Methods

Section 27.1: Potential Pitfall: Extensions are Resolved Statically

The extension method to be called is determined at compile-time based on the reference-type of the variable being accessed. It doesn't matter what the variable's type is at runtime, the same extension method will always be called.

```
open class Super

class Sub : Super()

fun Super.myExtension() = "Defined for Super"

fun Sub.myExtension() = "Defined for Sub"

fun callMyExtension(myVar: Super) {
    println(myVar.myExtension())
}

callMyExtension(Sub())
```

The above example will print `"Defined for Super"`, because the declared type of the variable `myVar` is `Super`.

Section 27.2: Top-Level Extensions

Top-level extension methods are not contained within a class.

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size - 1) {
        dest[i] += this[i]
    }
}
```

Above an extension method is defined for the type `IntArray`. Note that the object for which the extension method is defined (called the **receiver**) is accessed using the keyword **`this`**.

This extension can be called like so:

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

Section 27.3: Lazy extension property workaround

Assume you want to create an extension property that is expensive to compute. Thus you would like to cache the computation, by using the lazy property delegate and refer to current instance (**`this`**), but you cannot do it, as explained in the Kotlin issues KT-9686 and KT-13053. However, there is an official workaround provided here.

In the example, the extension property is `color`. It uses an explicit `colorCache` which can be used with **`this`** as no `lazy` is necessary:

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()
```

```
val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

第27.4节：扩展Java 7+ Path类的示例

扩展方法的一个常见用例是改进现有API。以下是向Java 7+的Path类添加存在（exist）、不存在（notExists）和递归删除（deleteRecursively）方法的示例：

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

现在可以在以下示例中调用这些方法：

```
val dir = Paths.get(dirName)
if (dir.exists()) dir.deleteRecursively()
```

第27.5节：扩展long以渲染人类可读字符串的示例

给定任何类型为Int或Long的值，将其渲染为人类可读的字符串：

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + " " +
units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

然后可以轻松使用：

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

第27.6节：示例——扩展Java 8 Temporal类以渲染ISO格式字符串

通过以下声明：

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

现在你可以简单地：

```
val dateAsString = someInstant.toIsoString()
```

第27.7节：使用扩展函数提升可读性

在 Kotlin 中，你可以这样编写代码：


```
val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

Section 27.4: Sample extending Java 7+ Path class

A common use case for extension methods is to improve an existing API. Here are examples of adding `exist`, `notExists` and `deleteRecursively` to the Java 7+ `Path` class:

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

Which can now be invoked in this example:

```
val dir = Paths.get(dirName)
if (dir.exists()) dir.deleteRecursively()
```

Section 27.5: Sample extending long to render a human readable string

Given any value of type **`Int`** or **`Long`** to render a human readable string:

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + " " +
units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

Then easily used as:

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

Section 27.6: Sample extending Java 8 Temporal classes to render an ISO formatted string

With this declaration:

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

You can now simply:

```
val dateAsString = someInstant.toIsoString()
```

Section 27.7: Using extension functions to improve readability

In Kotlin you could write code like:

```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("重要文件不存在")
}
```

但是使用apply并不能清楚表达你的意图。有时创建一个类似的扩展函数来重命名该操作，使其更直观，会更清晰。这不应被滥用，但对于非常常见的操作，比如验证：

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

你现在可以这样写代码：

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("重要文件不存在")
}
```

这让人们能够清楚地知道lambda参数中会发生什么。

注意，verifiedBy的类型参数T与T: Any?相同，意味着即使是可空类型也能使用该版本的扩展函数。虽然verifiedWith要求非空类型。

第27.8节：对伴生对象的扩展函数（静态函数的表现形式）

如果你想像静态函数一样扩展一个类，例如为类Something添加一个看起来像静态的函数fromString，这只有在该类有一个伴生对象且扩展函数是在伴生对象上声明时才有效：

```
class Something {
    伴生对象 {}
}

class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...

fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
Something.fromString("") //有效，因为扩展函数声明在
                         //伴生对象上

SomethingElse().fromString("") //有效，函数在实例上调用，非
                               //静态调用

SomethingElse.fromString("") //无效
}
```



```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not exist")
}
```

But the use of `apply` is not that clear as to your intent. Sometimes it is clearer to create a similar extension function to in effect rename the action and make it more self-evident. This should not be allowed to get out of hand, but for very common actions such as verification:

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

You could now write the code as:

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not exist")
}
```

Which now let's people know what to expect within the lambda parameter.

Note that the type parameter `T` for `verifiedBy` is same as `T: Any?` meaning that even nullable types will be able to use that version of the extension. Although `verifiedWith` requires non-nullable.

Section 27.8: Extension functions to Companion Objects (appearance of Static functions)

If you want to extend a class as-if you are a static function, for example for class `Something` add static looking function `fromString`, this can only work if the class has a companion object and that the extension function has been declared upon the companion object:

```
class Something {
    companion object {}
}

class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...

fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
    Something.fromString("") //valid as extension function declared upon the
                             //companion object

    SomethingElse().fromString("") //valid, function invoked on instance not
                                   //statically

    SomethingElse.fromString("") //invalid
}
```

第27.9节：便于代码中引用视图的扩展

您可以使用扩展来引用视图，创建视图后无需再写大量样板代码。

原创想法来自Anko库

扩展

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T = itemView?.findViewById(id)
as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as? T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
itemView?.findViewById(id) as? T
```

用法

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```


Section 27.9: Extensions for easier reference View from code

You can use extensions for reference View, no more boilerplate after you created the views.

Original Idea is by Anko Library

Extensions

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T = itemView?.findViewById(id)
as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as? T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
itemView?.findViewById(id) as? T
```

Usage

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```

第28章：DSL构建

专注于语法细节，在Kotlin中设计内部DSLs。

第28.1节：构建DSL的中缀方法

如果你有：

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

你可以在测试中编写如下类似DSL的代码：

```
@Test
fun test() {
   100.plusOne() 应该是 101
}
```

第28.2节：使用带有lambda的操作符

如果你有：

```
val r = Random(233)
中缀内联操作符函数Int.rem(block: () -> Unit) {
  如果(r.nextInt(100) < this) block()
}
```

你可以编写如下类似DSL的代码：

```
20 % { println("你看到这条消息的概率是20%") }
```

第28.3节：重写invoke方法以构建DSL

如果你有：

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

你可以在生产代码中编写如下类似DSL的代码：

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger是在`ex`上下文中定义的
        // 你只能在此上下文中调用此方法
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

第28.4节：在lambda中使用扩展

如果你有：

Chapter 28: DSL Building

Focus on the syntax details to design internal DSLs in Kotlin.

Section 28.1: Infix approach to build DSL

If you have:

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

you can write the following DSL-like code in your tests:

```
@Test
fun test() {
  100.plusOne() shouldBe 101
}
```

Section 28.2: Using operators with lambdas

If you have:

```
val r = Random(233)
infix inline operator fun Int.rem(block: () -> Unit) {
  if (r.nextInt(100) < this) block()
}
```

You can write the following DSL-like code:

```
20 % { println("The possibility you see this message is 20%") }
```

Section 28.3: Overriding invoke method to build DSL

If you have:

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

you can write the following DSL-like code in your production code:

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger is defined in the context of `ex`
        // you can only call this method inside this context
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

Section 28.4: Using extensions with lambdas

If you have:

```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this${e.message}") }}
```

你可以编写如下类似DSL的代码：

```
“它应该返回2” {
parse("1 + 1").buildAST().evaluate() 应该是 2
}
```

如果你对上面的 shouldBe 感到困惑，请参见示例 中缀方法构建DSL。


```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this\n${e.message}") }
}
```

You can write the following DSL-like code:

```
"it should return 2" {
   parse("1 + 1").buildAST().evaluate() shouldBe 2
}
```

If you feel confused with `shouldBe` above, see the example `Infix approach to build DSL`.

第29章：惯用语

第29.1节：Kotlin中的Serializable和serialVersionUID

在Kotlin中为类创建 `serialVersionUID` 有几种选择，均涉及向类的伴生对象添加成员。

最简洁的字节码来自于一个**`private const val`**，它将成为包含类中的私有静态变量，在本例中是`MySpecialCase`：

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

你也可以使用这些形式，每种形式都会带有 getter/setter 方法的副作用但这些方法对于序列化来说并非必要...

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

这会创建静态字段，同时也会在伴生对象上创建一个 getter 方法 getSerialVersionUID 这是不必要的。

```
class MySpecialCase : Serializable {
    companion object {
@JvmStatic private val serialVersionUID: Long = 123
    }
}
```

这会创建静态字段，同时也会在包含类 MySpecialCase 上创建一个静态 getter 方法 getSerialVersionUID 这是不必要的。

但所有这些方法都可以作为向 Serializable 类添加 serialVersionUID 的方式。

第29.2节：委托给一个类，但不在公共构造函数中提供它

假设你想要委托给一个类，但你不想在构造函数参数中提供被委托的类。相反，你想私下构造它，使构造函数的调用者对此一无所知。起初这似乎不可能，因为类委托只允许委托给构造函数参数。然而，有一种方法可以做到，如这个答案所示：

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table {

    constructor() : this(TreeBasedTable.create()) // 或者如果需要，可以使用不同类型的表

}
```

这样，你就可以像这样调用MyTable的构造函数：MyTable()。该Table<Int, Int, Int>是

Chapter 29: Idioms

Section 29.1: Serializable and serialVersionUid in Kotlin

To create the `serialVersionUID` for a class in Kotlin you have a few options all involving adding a member to the companion object of the class.

The most concise bytecode comes from a **`private const val`** which will become a private static variable on the containing class, in this case `MySpecialCase`:

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

You can also use these forms, **each with a side effect of having getter/setter methods** which are not necessary for serialization...

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

This creates the static field but also creates a getter as well `getSerialVersionUID` on the companion object which is unnecessary.

```
class MySpecialCase : Serializable {
    companion object {
        @JvmStatic private val serialVersionUID: Long = 123
    }
}
```

This creates the static field but also creates a static getter as well `getSerialVersionUID` on the containing class `MySpecialCase` which is unnecessary.

But all work as a method of adding the `serialVersionUID` to a `Serializable` class.

Section 29.2: Delegate to a class without providing it in the public constructor

Assume you want to delegate to a class but you do not want to provide the delegated-to class in the constructor parameter. Instead, you want to construct it privately, making the constructor caller unaware of it. At first this might seem impossible because class delegation allows to delegate only to constructor parameters. However, there is a way to do it, as given in this answer:

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table {

    constructor() : this(TreeBasedTable.create()) // or a different type of table if desired

}
```

With this, you can just call the constructor of `MyTable` like that: `MyTable()`. The `Table<Int, Int, Int>` to which

MyTable 委托将被私有创建。构造函数调用者对此一无所知。

此示例基于这个SO问题。

第29.3节：使用let或also简化可空对象的操作

let 在 Kotlin 中从调用它的对象创建一个局部绑定。示例：

```
val str = "foo"
str.let {
    println(it) // it
}
```

这将打印"foo"并返回Unit。

let 和 also 的区别在于你可以从 let 块中返回任何值。而 also 则总是返回 Unit。

你可能会问这有什么用？因为如果你调用一个可能返回 null 的方法，并且你只想在返回值不为 null 时执行某些代码，你可以像这样使用 let 或 also：

```
val str: String? = someFun()
str?.let {
println(it)
}
```

这段代码只有在 str 不为 null 时才会执行 let 块。注意 null 安全操作符（?）。

第29.4节：使用 apply 初始化对象或实现方法链

apply 的文档说明如下：

> 调用指定的函数块，以 this 作为接收者，并返回 this 值。

虽然 kdoc 说明不是很详细，但 apply 确实是一个有用的函数。通俗地说，apply 建立了一个作用域，其中 this 绑定到你调用 apply 的对象。这使你在需要对一个对象调用多个方法并最终返回该对象时，可以节省一些代码。例如：

```
File(dir).apply { mkdirs() }
```

这与写成以下内容是一样的：

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

第29.5节：Kotlin中的流畅方法

Kotlin中的流畅方法可以与Java相同：

`MyTable` delegates will be created privately. Constructor caller knows nothing about it.

This example is based on this SO question.

Section 29.3: Use let or also to simplify working with nullable objects

`let` in Kotlin creates a local binding from the object it was called upon. Example:

```
val str = "foo"
str.let {
    println(it) // it
}
```

This will print `"foo"` and will return **`Unit`**.

*The difference between `let` and `also` is that you can return any value from a `let` block. `also` in the other hand will always return **`Unit`**.*

Now why this is useful, you ask? Because if you call a method which can return **`null`** and you want to run some code only when that return value is not **`null`** you can use `let` or `also` like this:

```
val str: String? = someFun()
str?.let {
    println(it)
}
```

This piece of code will only run the `let` block when `str` is not **`null`**. Note the **`null`** safety operator (?).

Section 29.4: Use apply to initialize objects or to achieve method chaining

The documentation of `apply` says the following:

> calls the specified function block with **`this`** value as its receiver and returns **`this`** value.

While the kdoc is not so helpful `apply` is indeed an useful function. In layman's terms `apply` establishes a scope in which **`this`** is bound to the object you called `apply` on. This enables you to spare some code when you need to call multiple methods on an object which you will then return later. Example:

```
File(dir).apply { mkdirs() }
```

This is the same as writing this:

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

Section 29.5: Fluent methods in Kotlin

Fluent methods in Kotlin can be the same as Java:

```
fun doSomething() {
   someOtherAction()
   return this
}
```

但你也可以通过创建扩展函数使它们更具函数式，例如：

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

这就允许更明显的流畅函数：

```
fun doSomething() {
   return 流畅地 { someOtherAction() }
}
```

第29.6节：过滤列表

```
val list = listOf(1,2,3,4,5,6)

//过滤出偶数

val even = list.filter { it % 2 == 0 }

println(even) //返回 [2,4]
```

第29.7节：创建DTO（POJOs/POCOs）

Kotlin中的数据类是专门用来存储数据的类。这类类用data标记：

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

上面的代码创建了一个User类，自动生成了以下内容：

- 所有属性的Getter和Setter（对于val属性仅生成Getter）
- equals()
- hashcode()
- toString()
- copy()
- componentN()（其中N是按声明顺序对应的属性）

就像函数一样，也可以指定默认值：

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int = 20)
```

更多详情请参见这里Data Classes。


```
fun doSomething() {
   someOtherAction()
   return this
}
```

But you can also make them more functional by creating an extension function such as:

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

Which then allows more obviously fluent functions:

```
fun doSomething() {
   return fluently { someOtherAction() }
}
```

Section 29.6: Filtering a list

```
val list = listOf(1,2,3,4,5,6)

//filter out even numbers

val even = list.filter { it % 2 == 0 }

println(even) //returns [2,4]
```

Section 29.7: Creating DTOs (POJOs/POCOs)

Data classes in kotlin are classes created to do nothing but hold data. Such classes are marked as **data**:

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

The code above creates a `User` class with the following automatically generated:

- Getters and Setters for all properties (getters only for **val**s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (where `N` is the corresponding property in order of declaration)

Just as with a function, default values can also be specified:

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int = 20)
```

More details can be found here Data Classes.

第30章：Kotlin中的RecyclerView

我只是想分享一些关于使用Kotlin实现RecyclerView的小知识和代码。

第30.1节：主类和适配器

我假设你已经了解了一些Kotlin的语法和用法，只需在
activity_main.xml文件中添加RecyclerView，并设置适配器类即可。

```
class MainActivity : AppCompatActivity(){

        lateinit var mRecyclerView : RecyclerView
        val mAdapter : RecyclerAdapter = RecyclerAdapter()

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
            val toolbar = findViewById(R.id.toolbar) as Toolbar
            setSupportActionBar(toolbar)

            mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

            mRecyclerView.setHasFixedSize(true)
mRecyclerView.layoutManager = LinearLayoutManager(this)
            mAdapter.RecyclerAdapter(getList(), this)
mRecyclerView.adapter = mAdapter
        }

        private fun getList(): ArrayList<String> {
            var list : ArrayList<String> = ArrayList()
            for  (i in 1..10) { // 相当于 1 <= i && i <= 10
              println(i)
list.add("$i")
            }
            return list
        }
    }
```

这是你的 RecyclerView adapter 类，并创建你想要的 main_item.xml 文件

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
        this.mItems = item
        this.mClick = mClick;
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }
```


Chapter 30: RecyclerView in Kotlin

I just want to share my little bit knowledge and code of RecyclerView using Kotlin.

Section 30.1: Main class and Adapter

I am assuming that you have aware about the some syntax of **Kotlin** and how to use, just add **RecyclerView** in **activity_main.xml** file and set with adapter class.

```
class MainActivity : AppCompatActivity(){

        lateinit var mRecyclerView : RecyclerView
        val mAdapter : RecyclerAdapter = RecyclerAdapter()

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
            val toolbar = findViewById(R.id.toolbar) as Toolbar
            setSupportActionBar(toolbar)

            mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

            mRecyclerView.setHasFixedSize(true)
            mRecyclerView.layoutManager = LinearLayoutManager(this)
            mAdapter.RecyclerAdapter(getList(), this)
            mRecyclerView.adapter = mAdapter
        }

        private fun getList(): ArrayList<String> {
            var list : ArrayList<String> = ArrayList()
            for  (i in 1..10) { // equivalent of 1 <= i && i <= 10
                println(i)
                list.add("$i")
            }
            return list
        }
    }
```

this one is your recycler view **adapter** class and create **main_item.xml** file what you want

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
        this.mItems = item
        this.mClick = mClick;
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
        holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }
```

```
    override fun getItemCount(): Int {
        return mItems.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```



```
    override fun getItemCount(): Int {
        return mItems.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
            card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```

第31章：Kotlin中的日志记录

第31.1节：kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "发生错误" }
  }
}
```

使用 kotlin.logging 框架

Chapter 31: logging in kotlin

Section 31.1: kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
    logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "Error occurred" }
  }
}
```

Using kotlin.logging framework

第32章：异常

第32.1节：使用try-catch-finally捕获异常

在Kotlin中捕获异常看起来与Java非常相似

```
try {
doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

你也可以捕获多个异常

```
try {
doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
最终 {
    清理()
}
```

try 也是一个表达式，可能会返回值

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Kotlin 没有受检异常，所以你不必捕获任何异常。

```
fun fileToString(file: File) : String {
    //readAllBytes 会抛出 IOException，但我们可以省略捕获它
fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```


Chapter 32: Exceptions

Section 32.1: Catching exception with try-catch-finally

Catching exceptions in Kotlin looks very similar to Java

```
try {
    doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

You can also catch multiple exceptions

```
try {
    doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
finally {
    cleanup()
}
```

`try` is also an expression and may return value

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Kotlin doesn't have checked exceptions, so you don't have to catch any exceptions.

```
fun fileToString(file: File) : String {
    //readAllBytes throws IOException, but we can omit catching it
    fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```

第33章：JUnit

第33.1节：规则

向测试夹具添加 JUnit rule：

```
@Rule @JvmField val myRule = TemporaryFolder()
```

必须使用`@JvmField`注解以公开与`myRule`
属性具有相同可见性（public）的支持字段（参见answer）。JUnit规则要求被注解的规则字段为public。

Chapter 33: JUnit

Section 33.1: Rules

To add a JUnit rule to a test fixture:

```
@Rule @JvmField val myRule = TemporaryFolder()
```

The `@JvmField` annotation is necessary to expose the backing field with the same visibility (public) as the `myRule` property (see answer). JUnit rules require the annotated rule field to be public.

第34章：Kotlin Android扩展

Kotlin内置了Android视图注入，允许跳过手动绑定或使用ButterKnife等框架。其优点包括更优雅的语法、更好的静态类型检查，从而减少错误。

第34.1节：使用视图

假设我们有一个名为activity_main.xml的示例布局的活动：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

    <Button
       android:id="@+id/my_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="My button"/>
```

我们可以使用 Kotlin 扩展来调用按钮，而无需任何额外的绑定，示例如下：

```
import kotlinx.android.synthetic.main.activity_main.my_button

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
setContentView(R.layout.activity_main)
        // my_button 已经被正确地转换为“Button”类型
        // 而不是“View”类型
my_button.setText("Kotlin rocks!")
    }
}
```

你也可以使用 * 符号导入布局中出现的所有 id

```
// my_button 可以像之前一样使用
import kotlinx.android.synthetic.main.activity_main.*
```

合成视图不能在未加载该布局的 Activities/Fragments/Views 之外使用：

```
import kotlinx.android.synthetic.main.activity_main.my_button

class NotAView {
    init {
        // 这个示例无法编译！
my_button.setText("Kotlin rocks!")
    }
}
```

第34.2节：配置

从一个正确配置的gradle项目开始。

在你的project-local（非顶层）build.gradle文件中，在Kotlin插件声明下面添加extensions插件声明，

Chapter 34: Kotlin Android Extensions

Kotlin has a built-in view injection for Android, allowing to skip manual binding or need for frameworks such as ButterKnife. Some of the advantages are a nicer syntax, better static typing and thus being less error-prone.

Section 34.1: Using Views

Assuming we have an activity with an example layout called `activity_main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

    <Button
       android:id="@+id/my_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="My button"/>
</LinearLayout>
```

We can use Kotlin extensions to call the button without any additional binding like so:

```
import kotlinx.android.synthetic.main.activity_main.my_button

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
        setContentView(R.layout.activity_main)
        // my_button is already casted to a proper type of "Button"
        // instead of being a "View"
        my_button.setText("Kotlin rocks!")
    }
}
```

You can also import all ids appearing in layout with a `*` notation

```
// my_button can be used the same way as before
import kotlinx.android.synthetic.main.activity_main.*
```

Synthetic views can't be used outside of Activities/Fragments/Views with that layout inflated:

```
import kotlinx.android.synthetic.main.activity_main.my_button

class NotAView {
    init {
        // This sample won't compile!
        my_button.setText("Kotlin rocks!")
    }
}
```

Section 34.2: Configuration

Start with a properly configured gradle project.

In your **project-local** (not top-level) `build.gradle` append extensions plugin declaration below your Kotlin plugin,

位于顶层缩进级别。

```
buildscript {
    ...
}

apply plugin: "com.android.application"
...
apply plugin: "kotlin-android"
apply plugin: "kotlin-android-extensions"
...
```

第34.3节：使用Kotlin扩展，现在获取视图完全绘制完成通知的痛苦监听器变得如此简单且强大

```
mView.afterMeasured {
  // 在此代码块内视图已完全绘制
  // 你可以获取视图的高度/宽度，it.height / it.width
}
```

底层原理

```
内联函数 View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    重写 fun onGlobalLayout() {
        如果 (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

第34.4节：产品风味

Android扩展同样适用于多个Android产品风味。例如，如果我们在
build.gradle 中有如下配置：

```
android {
productFlavors {
        paid {
            ...
        }
free {
            ...
        }
    }
}
```

例如，只有免费版本有购买按钮：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
```


on top-level indentation level.

```
buildscript {
    ...
}

apply plugin: "com.android.application"
...
apply plugin: "kotlin-android"
apply plugin: "kotlin-android-extensions"
...
```

Section 34.3: Painful listener for getting notice, when the view is completely drawn now is so simple and awesome with Kotlin's extension

```
mView.afterMeasured {
  // inside this block the view is completely drawn
  // you can get view's height/width, it.height / it.width
}
```

Under the hood

```
inline fun View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    override fun onGlobalLayout() {
        if (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

Section 34.4: Product flavors

Android extensions also work with multiple Android Product Flavors. For example if we have flavors in `build.gradle` like so:

```
android {
    productFlavors {
        paid {
            ...
        }
        free {
            ...
        }
    }
}
```

And for example, only the free flavor has a buy button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
```

```
    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="购买完整版"/>
```

我们可以专门绑定到该版本：

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```



```
    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy full version"/>
</LinearLayout>
```

We can bind to the flavor specifically:

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```

第35章：面向Java开发者的Kotlin

大多数学习Kotlin的人都有Java编程背景。

本主题收集了Java与Kotlin的对比例子，突出最重要的差异以及Kotlin相较于Java提供的宝贵特性。

第35.1节：声明变量

在 Kotlin 中，变量声明看起来与 Java 有些不同：

```
val i : Int = 42
```

- 它们以val或var开头，声明为final（“value”）或variable。
- 类型写在名称后面，用:分隔
- 得益于Kotlin的类型推断，如果有赋值且编译器能够明确检测类型，则可以省略显式类型声明

Java	Kotlin
`int i = 42;`	`var i = 42`（或`var i : Int = 42`）
`final int i = 42;`	`val i = 42`

第35.2节：快速事实

- Kotlin 不需要 ; 来结束语句
- Kotlin 是 **空安全的**
- **Kotlin 与 Java 100% 互操作**
- Kotlin 没有 **原始类型**（但如果可能，会优化其在 JVM 上的对象对应类型）
- Kotlin 类有 **属性，而非字段**
- Kotlin 提供了 **数据类**，自动生成 `equals/hashCode` 方法和字段访问器
- Kotlin 只有运行时异常，**没有受检异常**
- Kotlin 没有 new 关键字。创建对象只需像调用其他方法一样调用构造函数。
- Kotlin 支持（有限的）运算符重载。例如，访问映射的值可以写成：
  `val a = someMap["key"]`
- Kotlin 不仅可以编译成 JVM 的字节码，还可以编译成 **JavaScript**，使你能够用 Kotlin 编写后端和前端代码
- Kotlin 与 **Java 6 完全兼容**，这对于支持（不那么）旧的 Android 设备尤其有意义
- Kotlin 是 官方支持的 Android 开发语言
- Kotlin 的集合内置区分了 **可变集合和不可变集合**。
- Kotlin 支持协程（实验性）

第35.3节：相等性与身份

Kotlin 使用 `==` 来判断相等（即内部调用 equals），使用 `===` 来判断引用身份。

Java	Kotlin
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

Chapter 35: Kotlin for Java Developers

Most people coming to Kotlin do have a programming background in Java.

This topic collects examples comparing Java to Kotlin, highlighting the most important differences and those gems Kotlin offers over Java.

Section 35.1: Declaring Variables

In Kotlin, variable declarations look a bit different than Java's:

```
val i : Int = 42
```

- They start with either **`val`** or **`var`**, making the declaration **`final`** ("**val**ue") or **var**iable.
- The type is noted after the name, separated by a `:`
- Thanks to Kotlin's *type inference* the explicit type declaration can be omitted if there is an assignment with a type the compiler is able to unambiguously detect

Java	Kotlin
`int i = 42;`	`var i = 42` (*or* `var i : Int = 42`)
`final int i = 42;`	`val i = 42`

Section 35.2: Quick Facts

- Kotlin does not need **`;`** to end statements
- Kotlin is **null-safe**
- Kotlin is **100% Java interoperable**
- Kotlin has **no primitives** (but optimizes their object counterparts for the JVM, if possible)
- Kotlin classes have **properties, not fields**
- Kotlin offers **data classes** with auto-generated `equals/hashCode` methods and field accessors
- Kotlin only has runtime Exceptions, **no checked Exceptions**
- Kotlin has **no `new` keyword**. Creating objects is done just by calling the constructor like any other method.
- Kotlin supports (limited) **operator overloading**. For example, accessing a value of a map can be written like:
  `val` `a = someMap["key"]`
- Kotlin can not only be compiled to byte code for the JVM, but also into **Java Script**, enabling you to write both backend and frontend code in Kotlin
- Kotlin is **fully compatible with Java 6**, which is especially interesting in regards for support of (not so) old Android devices
- Kotlin is an **officially supported** language **for Android development**
- Kotlin's collections have built-in distinction between **mutable and immutable collections**.
- Kotlin supports **Coroutines** (experimental)

Section 35.3: Equality & Identity

Kotlin uses `==` for equality (that is, calls `equals` internally) and `===` for referential identity.

Java	Kotlin
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

参见：https://kotlinlang.org/docs/reference/equality.html

第35.4节：IF、TRY等是表达式，而非语句

在 Kotlin 中，if、try 及其他是表达式（因此它们会返回一个值），而非（void）语句。

例如，Kotlin 没有 Java 的三元 Elvis Operator，但你可以写类似如下的代码：

```
val i = if (someBoolean) 33 else 42
```

更陌生但同样表达力强的是try 表达式：

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```


See: https://kotlinlang.org/docs/reference/equality.html

Section 35.4: IF, TRY and others are expressions, not statements

In Kotlin, `if`, **`try`** and others are expressions (so they do return a value) rather than (void) statements.

So, for example, Kotlin does not have Java's ternary *Elvis Operator*, but you can write something like this:

```
val i = if (someBoolean) 33 else 42
```

Even more unfamiliar, but equally expressive, is the **`try`** *expression*:

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```

第36章：Java 8 流等价物

Kotlin为集合和可迭代对象提供了许多扩展方法，用于应用函数式风格的操作。
专门的`Sequence`类型允许对多个此类操作进行惰性组合。

第36.1节：将名称累积到列表中

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // 不需要 toList()
```

第36.2节：收集示例#5 - 查找成年人员，输出格式化字符串

```
// Java:
String phrase = persons
        .stream()
.filter(p -> p.age >= 18)
        .map(p -> p.name)
.collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// 在德国 Max 和 Peter 和 Pamela 都是成年人。
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
.joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// 在德国，马克斯、彼得和帕梅拉都已成年。
```

顺便提一下，在Kotlin中我们可以创建简单的数据类，并按如下方式实例化测试数据：

```
// Kotlin:
// 数据类自动拥有equals、hashcode、toString和copy方法
data class Person(val name: String, val age: Int)

val persons = listOf(Person("马克斯", 18), Person("大卫", 12),
                     Person("彼得", 23), Person("帕梅拉", 23))
```

第36.3节：收集示例#6 - 按年龄分组人，打印年龄和姓名

```
// Java:
Map<Integer, String> map = persons
        .stream()
.collect(Collectors.toMap(
                p -> p.age,
p -> p.name,
                (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
```


Chapter 36: Java 8 Stream Equivalents

Kotlin provides many extension methods on collections and iterables for applying functional-style operations. A dedicated `Sequence` type allows for lazy composition of several such operations.

Section 36.1: Accumulate names in a List

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // toList() not needed
```

Section 36.2: Collect example #5 - find people of legal age, output formatted string

```
// Java:
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
        .joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// In Germany Max and Peter and Pamela are of legal age.
```

And as a side note, in Kotlin we can create simple data classes and instantiate the test data as follows:

```
// Kotlin:
// data class has equals, hashcode, toString, and copy methods automagically
data class Person(val name: String, val age: Int)

val persons = listOf(Person("Max", 18), Person("David", 12),
                     Person("Peter", 23), Person("Pamela", 23))
```

Section 36.3: Collect example #6 - group people by age, print age and names together

```
// Java:
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
                p -> p.age,
                p -> p.name,
                (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
```

```
// {18=马克斯, 23=彼得;帕梅拉, 12=大卫}
```

好的，这里有一个更有趣的Kotlin案例。首先是错误答案，用来探索从集合/序列创建Map的各种变体：

```
// Kotlin:
val map1 = persons.map { it.age 转换为 it.name }.toMap()
println(map1)
// 输出: {18=Max, 23=Pamela, 12=David}
// 结果: 重复项被覆盖，没有类似于Java 8的异常

val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// 输出: {18=Max, 23=Pamela, 12=David}
// 结果: 与上面相同，更冗长，重复项被覆盖

val map3 = persons.toMapBy { it.age }
println(map3)
// 输出: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// 结果: 重复项再次被覆盖

val map4 = persons.groupBy { it.age }
println(map4)
// 输出: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
age=23)], 12=[Person(name=David, age=12)]}
// 结果: 更接近，但现在是 Map<Int, List<Person>> 而不是 Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// 输出: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// 结果: 更接近了，但现在是 Map<Int, List<String>> 而不是 Map<Int, String>
```

现在给出正确答案：

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }

println(map6)
// 输出: {18=Max, 23=Peter;Pamela, 12=David}
// 结果: 太棒了!!
```

我们只需要将匹配的值连接起来，合并列表，并为joinToString提供一个转换器，将Person实例转换为Person.name。

第36.4节：不同类型的流 #7 - 惰性迭代 Double，映射为 Int，映射为 String，打印每个元素

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
```



```
// {18=Max, 23=Peter;Pamela, 12=David}
```

Ok, a more interest case here for Kotlin. First the wrong answers to explore variations of creating a `Map` from a collection/sequence:

```
// Kotlin:
val map1 = persons.map { it.age to it.name }.toMap()
println(map1)
// output: {18=Max, 23=Pamela, 12=David}
// Result: duplicates overridden, no exception similar to Java 8

val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// output: {18=Max, 23=Pamela, 12=David}
// Result: same as above, more verbose, duplicates overridden

val map3 = persons.toMapBy { it.age }
println(map3)
// output: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// Result: duplicates overridden again

val map4 = persons.groupBy { it.age }
println(map4)
// output: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
age=23)], 12=[Person(name=David, age=12)]}
// Result: closer, but now have a Map<Int, List<Person>> instead of Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// output: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// Result: closer, but now have a Map<Int, List<String>> instead of Map<Int, String>
```

And now for the correct answer:

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }

println(map6)
// output: {18=Max, 23=Peter;Pamela, 12=David}
// Result: YAY!!
```

We just needed to join the matching values to collapse the lists and provide a transformer to `joinToString` to move from `Person` instance to the `Person.name`.

Section 36.4: Different Kinds of Streams #7 - lazily iterate Doubles, map to Int, map to String, print each

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
```

```
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

第36.5节：应用过滤器后计算列表中的项目数

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
// 但最好不要过滤，而是用谓词计数
val count = items.count { it.startsWith('t') }
```

第36.6节：将元素转换为字符串并用逗号分隔连接它们

```
// Java:
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // 默认使用", "作为分隔符
```

第36.7节：计算员工工资总和

```
// Java:
int total = employees.stream()
.collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

第36.8节：按部门分组员工

```
// Java:
Map<Department, List<Employee>> byDept
     = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

第36.9节：计算各部门工资总和

```
// Java:
Map<Department, Integer> totalByDept
     = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment,
                    Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

第36.10节：将学生分为及格和不及格

```
// Java:
Map<Boolean, List<Student>> passingFailing =
```



```
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

Section 36.5: Counting items in a list after filter is applied

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
// but better to not filter, but count with a predicate
val count = items.count { it.startsWith('t') }
```

Section 36.6: Convert elements to strings and concatenate them, separated by commas

```
// Java:
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // ", " is used as separator, by default
```

Section 36.7: Compute sum of salaries of employee

```
// Java:
int total = employees.stream()
                     .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

Section 36.8: Group employees by department

```
// Java:
Map<Department, List<Employee>> byDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

Section 36.9: Compute sum of salaries by department

```
// Java:
Map<Department, Integer> totalByDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                     Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

Section 36.10: Partition students into passing and failing

```
// Java:
Map<Boolean, List<Student>> passingFailing =
```

```
students.stream()
.collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

第36.11节：男性成员的姓名

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
.filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
.collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

第36.12节：按性别分组成员名单

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
      roster.stream().collect(
Collectors.groupingBy(
            Person::getGender,
            Collectors.mapping(
Person::getName,
                Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

第36.13节：将列表过滤到另一个列表

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

第36.14节：查找列表中最短的字符串

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

第36.15节：不同类型的流 #2 - 如果存在则惰性使用第一个元素

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
```



```
    students.stream()
            .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

Section 36.11: Names of male members

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

Section 36.12: Group names of members in roster by gender

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
      roster.stream().collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.mapping(
                Person::getName,
                Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

Section 36.13: Filter a list to another list

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

Section 36.14: Finding shortest string a list

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

Section 36.15: Different Kinds of Streams #2 - lazily using first item if exists

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
```

```
.ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

第36.16节：不同类型的流 #3 - 迭代一个整数范围

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin：（包含范围）
(1..3).forEach(::println)
```

第36.17节：不同类型的流 #4 - 迭代数组，映射值，计算平均值

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
.average()
.ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

第36.18节：不同类型的流 #5 - 惰性迭代字符串列表，映射值，转换为整数，查找最大值

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
.ifPresent(System.out::println);  // 3
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

第36.19节：不同类型的流 #6 - 惰性迭代整数流，映射值，打印结果

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin：（包含范围）
(1..3).map { "a$it" }.forEach(::println)
```



```
    .ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

Section 36.16: Different Kinds of Streams #3 - iterate a range of Integers

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin:  (inclusive range)
(1..3).forEach(::println)
```

Section 36.17: Different Kinds of Streams #4 - iterate an array, map the values, calculate the average

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

Section 36.18: Different Kinds of Streams #5 - lazily iterate a list of strings, map the values, convert to Int, find max

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

Section 36.19: Different Kinds of Streams #6 - lazily iterate a stream of Ints, map the values, print results

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:  (inclusive range)
(1..3).map { "a$it" }.forEach(::println)
```

第36.20节：流的工作原理——过滤、转换为大写，然后排序列表

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
.filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
.sorted()
.forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

第36.21节：不同类型的流 #1 - 使用第一个元素的急切操作（如果存在）

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
.findFirst()
.ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

或者，在 String 上创建一个名为 ifPresent 的扩展函数：

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// 现在使用新的扩展函数：
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

另见：apply()函数

另见：扩展函数

另见：?.安全调用操作符，以及一般的可空性：
http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563#34498563

第36.22节：收集示例#7a - 映射名称，使用分隔符连接

```
// Java（详细版）：
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // 供应者
        (j, p) -> j.add(p.name.toUpperCase()),  // 累加器
        (j1, j2) -> j1.merge(j2),               // 合并器
        StringJoiner::toString);                // 完成器
```


Section 36.20: How streams work - filter, upper case, then sort a list

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

Section 36.21: Different Kinds of Streams #1 - eager using first item if it exists

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

or, create an extension function on String called ifPresent:

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// now use the new extension function:
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

See also: apply() function

See also: Extension Functions

See also: ?. Safe Call operator, and in general nullability:
http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563#34498563

Section 36.22: Collect example #7a - Map names, join together with delimiter

```
// Java (verbose):
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher
```

```
String names = persons
.stream()
.collect(personNameCollector);

System.out.println(names);   // MAX | PETER | PAMELA | DAVID

// Java (简洁版)
          String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" | "));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

第36.23节：收集示例 #7b - 使用 SummarizingInt 收集

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
.collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:

// 用于保存统计数据的东西...
数据类 SummaryStatisticsInt(变量 count: Int = 0,
                            变量 sum: Int = 0,
                            变量 min: Int = Int.MAX_VALUE,
                            变量 max: Int = Int.MIN_VALUE,
                            变量 avg: Double = 0.0) {
    函数 accumulate(newInt: Int): SummaryStatisticsInt {
        count++
sum += newInt
min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
         返回 this
    }
}

// 现在手动执行 fold，因为 Stream.collect 实际上就是 fold
变量 stats = persons.fold(SummaryStatisticsInt()) { stats, person -> stats.accumulate(person.age) }

打印(stats)
// 输出: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

但最好创建一个扩展函数，实际上是为了匹配Kotlin标准库中的样式：

```
// Kotlin:
inline fun 集合<整数>.总结整数(): 整数统计摘要
        = this.折叠(整数统计摘要()) { 统计, 数字 -> 统计.累积(数字) }

inline fun <T: 任意> 集合<T>.总结整数(转换: (T)->整数): 整数统计摘要 =
        this.折叠(整数统计摘要()) { 统计, 项目 -> 统计.累积(转换(项目)) }
```

现在你有两种方式来使用新的总结整数函数：

```
val 统计2 = 人员.映射 { 它.年龄 }.总结整数()
```



```
String names = persons
        .stream()
        .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID

// Java (concise)
String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" | "));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

Section 36.23: Collect example #7b - Collect with SummarizingInt

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
           .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:

// something to hold the stats...
data class SummaryStatisticsInt(var count: Int = 0,
                                var sum: Int = 0,
                                var min: Int = Int.MAX_VALUE,
                                var max: Int = Int.MIN_VALUE,
                                var avg: Double = 0.0) {
    fun accumulate(newInt: Int): SummaryStatisticsInt {
        count++
        sum += newInt
        min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
        return this
    }
}

// Now manually doing a fold, since Stream.collect is really just a fold
val stats = persons.fold(SummaryStatisticsInt()) { stats, person -> stats.accumulate(person.age) }

println(stats)
// output: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

But it is better to create an extension function, 2 actually to match styles in Kotlin stdlib:

```
// Kotlin:
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt
        = this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }

inline fun <T: Any> Collection<T>.summarizingInt(transform: (T)->Int): SummaryStatisticsInt =
        this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
```

Now you have two ways to use the new `summarizingInt` functions:

```
val stats2 = persons.map { it.age }.summarizingInt()
```

```
// 或者

val 统计3 = 人员.总结整数 { 它.年龄 }
```

所有这些都会产生相同的结果。我们也可以创建这个扩展以适用于序列和适当的原始类型。


```
// or

val stats3 = persons.summarizingInt { it.age }
```

And all of these produce the same results. We can also create this extension to work on `Sequence` and for appropriate primitive types.

第37章：Kotlin 注意事项

第37.1节：对可空类型调用 toString()

在使用 Kotlin 中的 toString 方法时需要注意的一点是如何处理与 `String?` 结合的 null。

例如，你想从 Android 中的 EditText 获取文本。

你可能会有如下代码：

```
// 错误示范：
val text = view.textField?.text.toString() ?: ""
```

你可能期望如果该字段不存在，值应为空字符串，但在这种情况下它是 "null"。

```
// 正确示范：
val text = view.textField?.text?.toString() ?: ""
```


Chapter 37: Kotlin Caveats

Section 37.1: Calling a toString() on a nullable type

A thing to look out for when using the `toString` method in Kotlin is the handling of null in combination with the `String?`.

For example you want to get text from an `EditText` in Android.

You would have a piece of code like:

```
// Incorrect:
val text = view.textField?.text.toString() ?: ""
```

You would expect that if the field did not exists the value would be empty string but in this case it is `"null"`.

```
// Correct:
val text = view.textField?.text?.toString() ?: ""
```

附录 A：配置 Kotlin 构建

第A.1节：Gradle配置

`kotlin-gradle-plugin` 用于使用 Gradle 编译 Kotlin 代码。基本上，它的版本应对应你想使用的 Kotlin 版本。例如，如果你想使用 Kotlin `1.0.3`，那么你也需要应用 `kotlin-gradle-plugin` 版本 `1.0.3`。

将此版本外部化到 gradle.properties 或 ExtraPropertiesExtension 中是个好主意：

```
buildscript {
ext.kotlin_version = '1.0.3'

    repositories {
mavenCentral()
    }

dependencies {
classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

然后你需要将此插件应用到你的项目中。针对不同平台，应用方式有所不同：

针对 JVM

```
apply plugin: 'kotlin'
```

针对 Android

```
apply plugin: 'kotlin-android'
```

针对 JS

```
apply plugin: 'kotlin2js'
```

以下是默认路径：

- kotlin 源代码：`src/main/kotlin`
- java 源代码：`src/main/java`
- kotlin 测试：`src/test/kotlin`
- java 测试：`src/test/java`
- 运行时资源：`src/main/resources`
- 测试资源：`src/test/resources`

如果你使用自定义项目布局，可能需要配置SourceSets。

最后，你需要将 Kotlin 标准库依赖添加到你的项目中：

```
dependencies {
compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

如果您想使用 Kotlin 反射，您还需要添加 compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"

Appendix A: Configuring Kotlin build

Section A.1: Gradle configuration

`kotlin-gradle-plugin` is used to compile Kotlin code with Gradle. Basically, its version should correspond to the Kotlin version you want to use. E.g. if you want to use Kotlin `1.0.3`, then you need to apply `kotlin-gradle-plugin` version `1.0.3` too.

It's a good idea to externalize this version in `gradle.properties` or in `ExtraPropertiesExtension`:

```
buildscript {
   ext.kotlin_version = '1.0.3'

   repositories {
     mavenCentral()
   }

   dependencies {
     classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
   }
}
```

Then you need to apply this plugin to your project. The way you do this differs when targeting different platforms:

Targeting JVM

```
apply plugin: 'kotlin'
```

Targeting Android

```
apply plugin: 'kotlin-android'
```

Targeting JS

```
apply plugin: 'kotlin2js'
```

These are the default paths:

- kotlin sources: `src/main/kotlin`
- java sources: `src/main/java`
- kotlin tests: `src/test/kotlin`
- java tests: `src/test/java`
- runtime resources: `src/main/resources`
- test resources: `src/test/resources`

You may need to configure `SourceSets` if you're using custom project layout.

Finally, you'll need to add Kotlin standard library dependency to your project:

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

If you want to use Kotlin Reflection you'll also need to add `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

A.2节：使用Android Studio

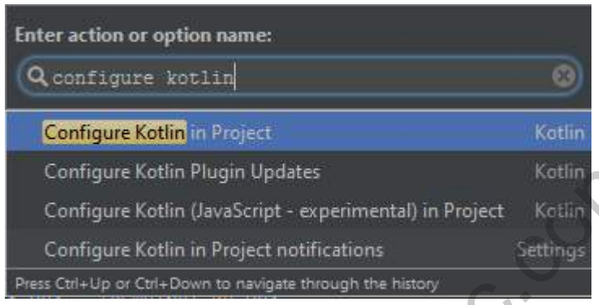
Android Studio可以在Android项目中自动配置Kotlin。

安装插件

要安装Kotlin插件，请依次进入文件 > 设置 > 编辑器 > 插件 > 安装JetBrains插件... > Kotlin > 安装，然后在提示时重启Android Studio。

配置项目

像往常一样创建一个Android Studio项目，然后按下 Ctrl + Shift + A 在搜索框中，输入“在项目中配置 Kotlin”并按回车键。



Android Studio 会修改你的 Gradle 文件以添加所有必要的依赖项。

转换 Java

要将你的 Java 文件转换为 Kotlin 文件，请按 Ctrl + Shift + A 并找到“将 Java 文件转换为 Kotlin 文件”。这将把当前文件的扩展名更改为.kt，并将代码转换为 Kotlin。

Section A.2: Using Android Studio

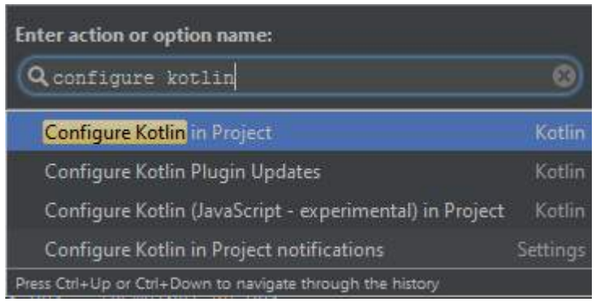
Android Studio can configure Kotlin automatically in an Android project.

Install the plugin

To install the Kotlin plugin, go to File > Settings > Editor > Plugins > Install JetBrains Plugin... > Kotlin > Install, then restart Android Studio when prompted.

Configure a project

Create an Android Studio project as normal, then press Ctrl + Shift + A. In the search box, type "Configure Kotlin in Project" and press Enter.



Android Studio will alter your Gradle files to add all the necessary dependencies.

Converting Java

To convert your Java files to Kotlin files, press Ctrl + Shift + A and find "Convert Java File to Kotlin File". This will change the current file's extension to `.kt` and convert the code to Kotlin.

```java
package com.orangeflash81.myapplication;

public class Foo {
    private String name = "Joe Bloggs";

    String getName() { return name; }

    void setName(String value) { name = value; }
}
```


A.3 节：从使用 Groovy 脚本的 Gradle 迁移到 Kotlin 脚本

步骤：

- 克隆gradle-script-kotlin项目
- 从克隆的项目复制/粘贴到你的项目中：
    - `build.gradle.kts`
    - `gradlew`
    - `gradlew.bat`
    - `settings.gradle`
- 根据您的需求更新build.gradle.kts的内容，您可以参考刚克隆的项目或其示例中的脚本作为灵感
- 现在打开Intellij并打开您的项目，在资源管理器窗口中，它应该被识别为Gradle项目，
  如果没有，先展开它。
- 打开后，让Intellij运行，打开build.gradle.kts并检查是否有错误。如果高亮不起作用和/或全部标红，则关闭并重新打开Intellij
- 打开Gradle窗口并刷新它

如果您使用的是Windows，可能会遇到此bug，下载完整的Gradle 3.3发行版并使用它替代提供的版本。相关内容。


```java
package com.orangeflash81.myapplication;

public class Foo {
    private String name = "Joe Bloggs";

    String getName() { return name; }

    void setName(String value) { name = value; }
}
```


Section A.3: Migrating from Gradle using Groovy script to Kotlin script

Steps:

- clone the gradle-script-kotlin project
- copy/paste from the cloned project to your project:
    - `build.gradle.kts`
    - `gradlew`
    - `gradlew.bat`
    - `settings.gradle`
- update the content of the `build.gradle.kts` based on your needs, you can use as inspiration the scripts in the project just cloned or in one of its samples
- now open Intellij and open your project, in the explorer window, it should be recognized as a Gradle project, if not, expand it first.
- after opening, let Intellij works, open `build.gradle.kts` and check if there are any error. If the highlighting is not working and/or is everything marked red, then close and reopen Intellij
- open the Gradle window and refresh it

If you are on Windows, you may encounter this bug, download the full Gradle 3.3 distribution and use that instead the one provided. Related.

OSX和Ubuntu开箱即用。

小提示，如果您想避免发布到Maven等的所有麻烦，可以使用Jitpack，添加的行与Groovy几乎相同。您可以参考我的这个项目。

OSX and Ubuntu work out of the box.

Small bonus, if you want to avoid all the hassle of publishing on Maven and similar, use Jitpack, the lines to add are almost identical compared to Groovy. You can take inspiration from this project of mine.

致谢

非常感谢所有来自Stack Overflow Documentation的人员，他们帮助提供了这些内容，
更多更改可以发送至web@petercv.com以发布或更新新内容

亚伦·克里斯蒂安森	第7章、第29章和第38章
阿卜杜拉	第9章
亚当·阿罗尔德	第29章
亚历克斯·法乔鲁索	第9章
升天	第5章
atok	第26章
阿维吉特·卡尔马卡尔	第17章
baha	第7章
布拉德	第36章
byxor	第8章和第16章
凯鲁姆	第22章
cyberscientist	第1章
达维德·蒂马尔	第27章
大卫·索罗科	第6章
迪维娅	第19章和第20章
egor.zhdan	第4章
选择	第38章
埃斯彭	第12章
杰森	第36章
glee8e	第7章和第20章
埃克托尔	第22章和第29章
快捷键	第18章
ice1000	第28章
扬·弗拉基米尔·莫斯特特	第19章
雅努森	第3章
杰森·米纳德	第1、7、18、19、27、29和36章
杰莫·姆格布里什维利	第21和34章
jenglert	第33章
jereksel	第32章
KeksArmee	第7、14、16和18章
凯文·罗巴特尔	第23章
基里尔·拉赫曼	第6、9和26章
康拉德·亚姆罗兹克	第27、29和31章
madhead	第7、26和38章
mayojava	第10和29章
memoizr	第13章
莫希特·苏塔尔	第30章
Mood	第22章
尼哈尔·萨克塞纳	第11章
olivierlemasle	第31章
oshai	第31章
帕克·霍耶斯	第1章和第27章
razzledazzle	第10章、第14章、第27章和第29章
里奇·库兹马	第13章
Ritave	第19章、第26章和第34章
罗宾	第9章、第10章、第14章和第19章
Ruckus T	第1章

Credits

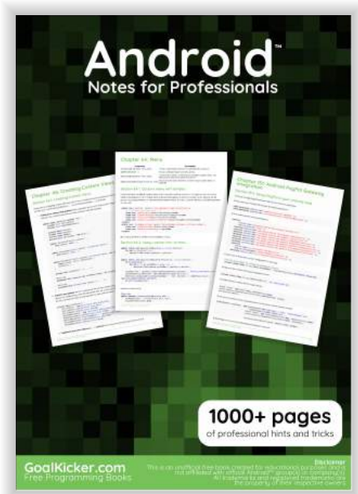
Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

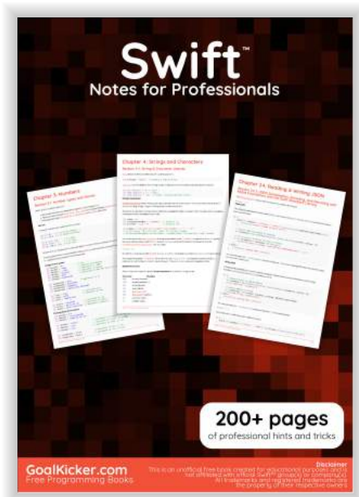
Aaron Christiansen	Chapters 7, 29 and 38
Abdullah	Chapter 9
Adam Arold	Chapter 29
Alex Facciorusso	Chapter 9
Ascension	Chapter 5
atok	Chapter 26
Avijit Karmakar	Chapter 17
baha	Chapter 7
Brad	Chapter 36
byxor	Chapters 8 and 16
Caelum	Chapter 22
cyberscientist	Chapter 1
Dávid Tímár	Chapter 27
David Soroko	Chapter 6
Divya	Chapters 19 and 20
egor.zhdan	Chapter 4
elect	Chapter 38
Espen	Chapter 12
Gerson	Chapter 36
glee8e	Chapters 7 and 20
Héctor	Chapters 22 and 29
hotkey	Chapter 18
ice1000	Chapter 28
Jan Vladimir Mostert	Chapter 19
Januson	Chapter 3
Jayson Minard	Chapters 1, 7, 18, 19, 27, 29 and 36
Jemo Mgebrishvili	Chapters 21 and 34
jenglert	Chapter 33
jereksel	Chapter 32
KeksArmee	Chapters 7, 14, 16 and 18
Kevin Robatel	Chapter 23
Kirill Rakhman	Chapters 6, 9 and 26
Konrad Jamrozik	Chapters 27, 29 and 31
madhead	Chapters 7, 26 and 38
mayojava	Chapters 10 and 29
memoizr	Chapter 13
Mohit Suthar	Chapter 30
Mood	Chapter 22
Nihal Saxena	Chapter 11
olivierlemasle	Chapter 31
oshai	Chapter 31
Parker Hoyes	Chapters 1 and 27
razzledazzle	Chapters 10, 14, 27 and 29
Rich Kuzsma	Chapter 13
Ritave	Chapters 19, 26 and 34
Robin	Chapters 9, 10, 14 and 19
Ruckus T	Chapter 1

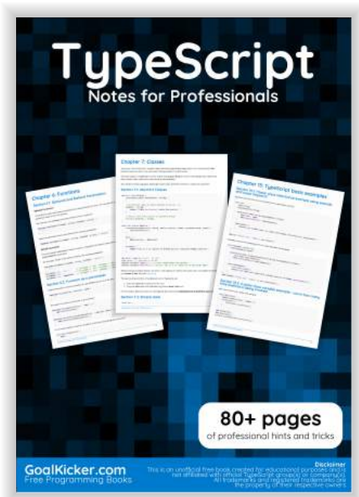
萨赫	第1章
萨姆	第3章、第4章、第15章和第25章
肖恩·赖利	第1章
Seaskyways	第25章
SerCe	第6章和第14章
Shinoo Goyal	第2章
Slav	第16章和第24章
Spidfire	第7章、第9章、第14章和第37章
technerd	第14章
托尔斯滕·施莱因策	第14章和第35章
特拉维斯	第12章
未知	第1章和第4章

Sach	Chapter 1
Sam	Chapters 3, 4, 15 and 25
Sean Reilly	Chapter 1
Seaskyways	Chapter 25
SerCe	Chapters 6 and 14
Shinoo Goyal	Chapter 2
Slav	Chapters 16 and 24
Spidfire	Chapters 7, 9, 14 and 37
technerd	Chapter 14
Thorsten Schleinzer	Chapters 14 and 35
Travis	Chapter 12
UnKnown	Chapters 1 and 4

你可能也喜欢







You may also like