Unit 2.1. The Kotlin language

Ricardo Sánchez

Contents

1 The Kotlin language

Kotlin is a cross-platform, statically typed, general-purpose high-level programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library.

Kotlin mainly targets the JVM, but also compiles to JavaScript or native code.Language development costs are borne by JetBrains, while the Kotlin Foundation protects the Kotlin trademark.

The Android Kotlin compiler emits Java 8 bytecode by default (which runs in any later JVM), but allows targeting Java 9 up to 20, for optimizing.

On 7 May 2019, Google announced that the Kotlin programming language had become its preferred language for Android app developers. Since the release of Android Studio 3.0 in October 2017, Kotlin has been included as an alternative to the standard Java compiler.

References:

- Kotlin docs
- Kotlin for Android
- Kotlin playground

1.1 Hello world

Open the Kotlin Playground and write and execute this code:

```
fun main() {
    println("Hello, world!")
}
```

`fun main()` is the entry point of the program. All Kotlin programs are required to have a main function, which is the specific place in your code where the program starts running.

`println` is a function that takes an argument as a String and outputs its content to the console.

!!! tip You can also run Kotlin code in Android Studio creating a new file on an exisiting project and running that file.

1.2 Variables

To create a variable, use `var` or `val`, and assign a value to it with the equal sign (`=`):

```
var name = "Mary"
val birthyear = 1974
```

The difference between `var` and `val` is that variables declared with the `var` keyword can be modified, while `val` variables cannot. `val` variables are **immutable**.

Kotlin uses type inference, but you can specify the type when create a variable:

```
var name: String = "Mary"
val birthyear: Int = 1974
```

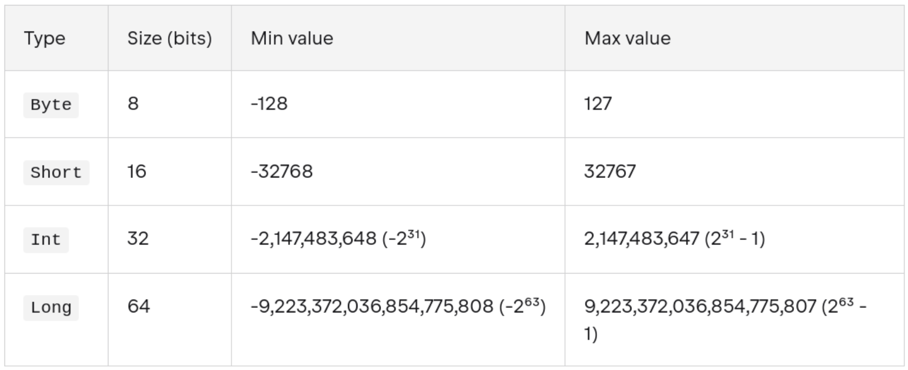
The general rule for naming Kotlin variables are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names should start with a letter, $ and _
- Names are case sensitive (“myVar” and “myvar” are different variables)
- Names should start with a lowercase letter and it cannot contain whitespace
- Reserved words (like Kotlin keywords, such as var or String) cannot be used as names

1.2.1 Types

In Kotlin, everything is an object in the sense that you can call member functions and properties on any variable.

For **integer numbers**, there are four types with different sizes and value ranges:



Type	Size (bits)	Min value	Max value
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-2^{31})	2,147,483,647 (2^{31} - 1)
`Long`	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 (2^{63} - 1)

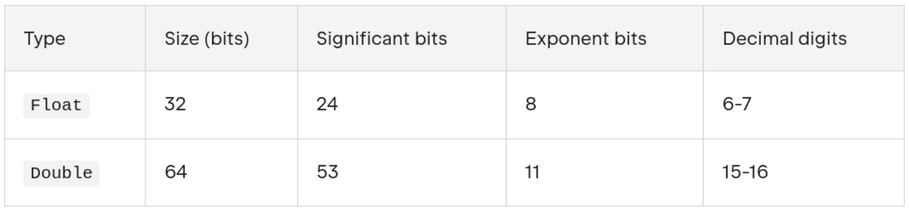
Figure 1: Int types

When you initialize a variable with no explicit type specification, the compiler automatically infers the type with the smallest range enough to represent the value starting from Int. If it doesn’t exceed the range of Int, the type is Int. If it does exceed that range, the type is Long. To specify the Long value explicitly, append the suffix L to the value. To use the Byte or Short type, specify it explicitly in the declaration.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

!!! info In addition to signed integer types, Kotlin also provides unsigned integer types

For real numbers, Kotlin provides floating-point types **Float** and **Double** that adhere to the IEEE 754 standard. Float reflects the IEEE 754 single precision, while Double reflects double precision.



Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

Figure 2: Float types

For variables initialized with fractional numbers, the compiler infers the Double type.

The **String** data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

```
val myText: String = "Hello World"
```

String literals may contain **template** expressions (pieces of code that are evaluated and whose results are concatenated into a string). When a template expression is processed, Kotlin automatically calls the **`.toString()`** function on the expression's result to convert it into a string. A template expression starts with a dollar sign ($) and consists of a variable name:

```
var name: String = "Mary"
val age: Int = 34
println("Hello $name!")
println("Yout age is $age")
```

Template expressions can also hold an expression in curly braces:

```
val num1 = 3
val num2 = 4
println("$num1 + $num2 is ${num1 + num2}")
```

The **Boolean** data type can only take the values **`true`** or **`false`**:

```
val isTrue: Boolean = true
val isFalse: Boolean = false
```

The **Char** data type is used to store a single character. A `char` value must be surrounded by single quotes:

```
val letter = 'A'
```

1.2.2 Type Conversion

In Kotlin, numeric type conversion is different from Java. For example, it is not possible to convert an Int type to a Long type with the following code:

```
val x: Int = 5
val y: Long = x
println(y) // Error: Type mismatch
```

To convert a numeric data type to another type, you must use one of the following functions: `toByte()`, `toShort()`, `toInt()`, `toLong()`, `toFloat()`, `toDouble()`, `toString()` or `toChar()`:

```
val x: Int = 5
val y: Long = x.toLong()
println(y)
```

1.2.3 Nullable types and Elvis operator

Kotlin is a safe language, and among other things, it prevents us from programming errors such as *NullPointerException* since it does not allow variable values to be *null* by default.

If we want to specify that a variable can contain a *null* value, it is necessary to explicitly define it as **nullable**. To do this, when we define it, we add a question mark `?` to its type:

```
var name: String? = null //Nullable type
var age: Int = 50   //Non-nullable type
age = null //Error
```

In addition, Kotlin also provides us with the `?:` operator, known as the **Elvis operator**, to specify an alternative value when the variable is *null*.

```
var name : String? = null
println(name.length) // Error
println(name?.length ?: -1) //prints -1
name = Mary
println(name?.length ?: -1) //prints 4
```

In this example we've used the `?.` **safe call operator**. It prevents to cause an exception when the variable is null and Kotlin can't call the member function (length in this case).

1.3 Constants

We can declare constants in Kotlin using the `const` keyword. Constants must be initialized with a value at the time of declaration, and their value cannot be changed later. Constants can only be of primitive types and String.

```
const val PI = 3.14159
const val APP_NAME = "MyKotlinApp"
```

!!! tip Use **snake_case** for naming constants: all uppercase letters with words separated by underscores.

The difference between `val` and `const val` is that `val` can be assigned a value at runtime, while `const val` must be assigned a value at compile time. Additionally, `const val` can only be used for top-level or object-level properties, while `val` can be used in any scope.

1.4 Operators

1.4.1 Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value from another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value by 1	++x
–	Decrement	Decreases the value by 1	–x

1.4.2 Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

1.4.3 Comparison Operators

Comparison operators are used to compare two values, and returns a **Boolean** value: either `true` or `false`.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

The `===` operator (and its negated counterpart `!==`) checks for **referencial equality**. `a === b` evaluates to true if and only if `a` and `b` point to the same object:

```kotlin
fun main() {
    var a = "Hello"
    var b = a
    var c = "world"
    var d = "world"

    println(a === b)
    // true
    println(a === c)
    // false
    println(c === d)
    // true
}
```

For values represented by primitive types at runtime (for example, `Int`), the `===` equality check is equivalent to the `==` check.

1.4.4 Logical Operators

Operator	Name	Description	Example
&&	Logical **and**	Returns true if both statements are true	`x < 5 && x < 10`
||	Logical **or**	Returns true if one of the statements is true	`x < 5 || x < 4`
!	Logical **not**	Reverse the result, returns false if the result is true	`!(x < 5)`

1.5 Comments

Kotlin has single-line comments and multi-line comments:

```kotlin
// This is a single-line comment

/* This is
  a multi-line
  comment */
```

1.6 Control structures

1.6.1 if - else

The structure is similar to other languages:

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

In Kotlin, `if-else` can return a value for each case that can be assigned to a variable:

```
val message = if (age < 18) {
  "You are under-age."
} else {
  "You are adult."
}
```

!!! warning When using `if` as an expression, you must also include `else` (required).

That can be shortened to:

```
val msg = if (age < 18) "You are under-age." else "You are adult."
```

This is equivalent to the *ternary operator* of other languages.

1.6.2 when

The `when` statement is similar to `switch-case` of C/Java:

```
when (trafficLightColor) {
    "Red" -> println("Stop")
    "Yellow" -> println("Slow")
    "Green" -> println("Go")
    else -> println("Invalid traffic-light color")
}
```

In the same way that if, when can return the result and can be assigned to a variable:

```
val msg = when (trafficLightColor) {
    "Red" -> "Stop"
    "Yellow" -> "Slow"
    "Green" -> "Go"
    else -> "Invalid traffic-light color"
}

println(msg)
```

1.6.3 while and do-while

`while` and `do-while` loops are similar to C/Java:

```
while (condition) {
  // code block to be executed
}
```

```
do {
  // code block to be executed
}
while (condition);
```

In the same way, we have `break` and `continue` statements.

1.6.4 for

Unlike Java and other programming languages, there is no traditional `for` loop in Kotlin.

In Kotlin, the `for` loop is used to loop through arrays, ranges, and other things that contains a countable number of values.

To loop through array elements, use the `for` loop together with the `in` operator:

```
val numbers = arrayOf(10, 14, 2, 15, 20)
for (x in numbers) {
  println(x)
}
```

With the `for` loop, we can also iterate ranges:

```
for (x in 0..10) {
  println(x)
}
```

1.7 Functions

1.7.1 Definition and invocation

To declare a function in Kotlin we do:

```
fun funcName(param 1 : Type1, param2 : Type2...) : ReturnType
{
    // function body
    return
}
```

Some examples:

```
fun simplefunction()
{
    println("Simple function")
}

fun functionWithParams(name: String): Unit
{
    println("Hello $name")
}

fun sum(x: Int, y: Int): Int {
  return (x + y)
}

//Function call
println(sum(4, 5)) //prints 9
```

We look at some features of function declarations:

- They are declared using the keyword *fun*
- Names start with lower case and are expressed in camelCase
- Function parameters are specified after the name, in parentheses, and in the form *parameter : Type*. These types must necessarily be specified
- The return type of the function may be specified after the parenthesis with the argument list, followed by `:`.
- When the function does not return any significant value, its default return type is `Unit` , which would be the equivalent of `void` in Java or C.

!!! warning Unlike in some languages, such as Java, where a function can change the value passed into a parameter, parameters in Kotlin are immutable. You cannot reassign the value of a parameter from within the function body.

1.7.2 Named parameters

You can use named parameters when call a function:

```
fun hello(name: String, age: Int)
{
    println("Hello $name, you are $age years old")
}

hello(age = 16, name = "Sean") // Hello Sean, you are 16 years old
```

In this case, you can write the arguments in any order.

1.7.3 Default arguments

Function parameters can also specify default arguments.

```
fun hello(name: String = "Nonamed", age: Int)
{
    println("Hello $name, you are $age years old")
}

hello(age = 16, name = "Sean") // Hello Sean, you are 16 years old
hello(age = 16) // Hello Nonamed, you are 16 years old
```

1.7.4 Single-expression functions

When the function body consists of a single expression, the curly braces can be omitted and the body specified after an = 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
```

1.7.5 Lambda expressions

Lambda expressions provide a concise syntax to define a function without the `fun` keyword. You can store a lambda expression directly in a variable without a function reference on another function.

Before the assignment operator (`=`), you add the `val` or `var` keyword followed by the name of the variable, which is what you use when you call the function. After the assignment operator (`=`) is the lambda expression, which consists of a pair of curly braces that form the function body:

```
fun main() {
    hello() // prints "Hello world!"
}

val hello = {
    println("Hello world!")
}
```

You can assign the lambda function to a variable and use it as a function too:

```
fun main() {
    val myHello = hello
    myHello() // prints "Hello world!"
}

val hello = {
    println("Hello world!")
}
```

The full syntactic form of lambda expressions is as follows:

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

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the `->`.
- If the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

```
val sum = { x: Int, y: Int -> x + y }
```

Another example of a lambda expression without arguments and return type:

```
val hello: () -> Unit = { println("Hello world!") }
```

1.7.5.1 Trailing lambdas

If the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val product = items.fold(1) { acc, e -> acc * e }
```

Such syntax is also known as **trailing lambda**.

If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

1.7.5.2 `it` for single parameter functions

If the lambda has a single argument, we can use the keyword `it`, which represents that argument passed to the lambda function.

The expression:

```
array.forEach { item -> println(item * 4) }
```

can be shortened to:

```
array.forEach { println(it * 4) }
```