Basic syntax

Edit page Last modified: 08 February 2023

This is a collection of basic syntax elements with examples. At the end of every section, you'll find a link to a detailed description of the related topic.

You can also learn all the Kotlin essentials with the free Kotlin Basics track ↗ on JetBrains Academy.

Package definition and imports

Package specification should be at the top of the source file.

```
package my.demo

import kotlin.text.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

See Packages.

Program entry point

An entry point of a Kotlin application is the `main` function.

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

Open in Playground → Target: JVM Running on v.1.8.10

Another form of `main` accepts a variable number of `String` arguments.

```kotlin
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Print to the standard output

`print` prints its argument to the standard output.

```kotlin
print("Hello ")
print("world!")
```

Open in Playground → Target: JVM Running on v.1.8.10

`println` prints its arguments and adds a line break, so that the next thing you print appears on the next line.

```kotlin
println("Hello world!")
println(42)
```

Open in Playground → Target: JVM Running on v.1.8.10

Functions

A function with two `Int` parameters and `Int` return type.

```kotlin
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Open in Playground → Target: JVM Running on v.1.8.10

A function body can be an expression. Its return type is inferred.

```kotlin
fun sum(a: Int, b: Int) = a + b
```

Open in Playground → Target: JVM Running on v.1.8.10

A function that returns no meaningful value.

```kotlin
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

`Unit` return type can be omitted.

```kotlin
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

See Functions.

Variables

Read-only local variables are defined using the keyword `val`. They can be assigned a value only once.

```kotlin
val a: Int = 1  // immediate assignment
val b = 2   // `Int` type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

Open in Playground → Target: JVM Running on v.1.8.10

Variables that can be reassigned use the `var` keyword.

```kotlin
var x = 5 // `Int` type is inferred
x += 1
```

Open in Playground → Target: JVM Running on v.1.8.10

You can declare variables at the top level.

```kotlin
val PI = 3.14
var x = 0
```

Open in Playground → Target: JVM Running on v.1.8.10

```
fun incrementX() {
    x += 1
}
```

See also Properties.

Creating classes and instances

To define a class, use the `class` keyword.

```
class Shape
```

Properties of a class can be listed in its declaration or body.

```
class Rectangle(var height: Double, var length: Double) {
    var perimeter = (height + length) * 2
}
```

The default constructor with parameters listed in the class declaration is available automatically.

```
val rectangle = Rectangle(5.0, 2.0)
println("The perimeter is ${rectangle.perimeter}")
```

Open in Playground → Target: JVM Running on v.1.8.10

Inheritance between classes is declared by a colon (`:`). Classes are final by default;

to make a class inheritable, mark it as `open`.

```kotlin
open class Shape

class Rectangle(var height: Double, var length: Double): Shape() {
    var perimeter = (height + length) * 2
}
```

See classes and objects and instances.

Comments

Just like most modern languages, Kotlin supports single-line (or *end-of-line*) and multi-line (*block*) comments.

```kotlin
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

Block comments in Kotlin can be nested.

```kotlin
/* The comment starts here
/* contains a nested comment */
and ends here. */
```

See Documenting Kotlin Code for information on the documentation comment syntax.

String templates

```kotlin
var a = 1
// simple name in template:
val s1 = "a is $a"

a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

Open in Playground → Target: JVM Running on v.1.8.10

See String templates for details.

Conditional expressions

```kotlin
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

Open in Playground → Target: JVM Running on v.1.8.10

In Kotlin, `if` can also be used as an expression.

```kotlin
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Open in Playground → Target: JVM Running on v.1.8.10

See `if`-expressions.

for loop

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

Open in Playground → Target: JVM Running on v.1.8.10

or

```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

See for loop.

while loop

```kotlin
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

Open in Playground → Target: JVM Running on v.1.8.10

See while loop.

when expression

```kotlin
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

Open in Playground → Target: JVM Running on v.1.8.10

See when expression.

Ranges

Check if a number is within a range using `in` operator.

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Check if a number is out of range.

```
val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range, too")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Iterate over a range.

```
for (x in 1..5) {
    print(x)
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Or over a progression.

```
for (x in 1..10 step 2) {
    print(x)
}
```

n in Playground → Target: JVM Running on v.1.8.10

```
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

See Ranges and progressions.

Collections

Iterate over a collection.

```
for (item in items) {
    println(item)
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Check if a collection contains an object using `in` operator.

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Open in Playground → Target: JVM Running on v.1.8.10

Using lambda expressions to filter and map collections:

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
```

```
fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.uppercase() }
    .forEach { println(it) }
```

See Collections overview.

Nullable values and null checks

A reference must be explicitly marked as nullable when `null` value is possible. Nullable type names have `?` at the end.

Return `null` if `str` does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

n in Playground → Target: JVM Running on v.1.8.10

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after nul
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

or

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

Open in Playground → Target: JVM Running on v.1.8.10

See Null-safety.

Type checks and automatic casts

The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked bra
    return null
}
```

or

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
```

Open in Playground → Target: JVM Running on v.1.8.10

or even

```
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand si
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
```

Open in Playground → Target: JVM Running on v.1.8.10

See Classes and Type casts.