Table of Contents

Basic types

"Primitives"

Kotlin provides Java's eight primitive types as built-in classes:

- `kotlin.Byte` — 8-bit signed integer
- `kotlin.Short` — 16-bit signed integer
- `kotlin.Int` — 32-bit signed integer
- `kotlin.Long` — 64-bit signed integer
- `kotlin.Float` — 32-bit IEEE 754 floating-point
- `kotlin.Double` — 64-bit IEEE 754 floating-point
- `kotlin.Boolean` — true/false
- `kotlin.Char` — 16-bit Unicode character (**not** a numeric type!)

In addition to these types, Kotlin also provides four *unsigned integer* types:

- `kotlin.UByte` — 8-bit unsigned integer
- `kotlin.UShort` — 16-bit unsigned integer
- `kotlin.UInt` — 32-bit unsigned integer
- `kotlin.ULong` — 64-bit unsigned integer

The format of numeric literals in Kotlin is almost identical to that of Java; octal literals have been removed.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
val creditCardNumber = 1234_5678_9012_3456L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
val bool = true
val char = 'A'
val uint = 20U // UInt
val ulong = 9082348590UL // ULong
val pi = 3.14 // Double
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float
```

[1]

> **TIP** Everything in Kotlin is a class! There are no primitive types; the compiler will automatically deal with boxing/unboxing for you, making working with generics a breeze.

Values of the built-in types cannot be converted between each other implicitly, since in other languages this is a common cause of error or loss of precision. To do so one must use the built-in `toXXX` functions:

```
val char = 'A'
val double: Double = char * 2 // ERROR
val double: Double = char.toDouble() * 2 // ok
```

> **TIP** Often, this is not necessary because the standard arithmetic operators have overloads for most appropriate conversions [2]. For example, adding a `Long` and an `Int` will compile without any conversions, and will result in a `Long` value.

The conversion functions are compiler intrinsics and thus do not introduce any function call overhead.

Arrays

Arrays in Kotlin are represented by the generic `kotlin.Array<T>` class. For performance reasons, primitive arrays have their own specialized types. These map

directly to the respective primitive array type (i.e. `IntArray` = `int[]`).

Kotlin does not have array literals (they exist, but are only allowed in annotations); to create an array you can use the built-in `arrayOf` function:

```
val strings: Array<String> = arrayOf("Hello", "world")
val ints: IntArray = intArrayOf(1, 2, 3)
```

Strings

Strings are defined using double quotes.

String templates (also known as string interpolation) are the recommended way to concatenate values with a string:

```
val foo = 3
println("The value of foo is $foo") // "The value of foo is 3"
println("Half of foo is ${foo / 2}") // "Half of foo is 1"
```

[3]

Multi-line (raw) strings, which unfortunately didn't make it into JDK 13, can be defined using triple quotes:

```
val text = """
foo
bar
"""
```

NOTE	Multi-line strings do not support escape characters. If you need to escape a `$` in a raw string, you can replace it with `${'$'}`.

To achieve nicer indentation, you can add a margin and trim it:

```
val text = """
    foo
    bar
""".trimIndent()
```

NOTE	Don't worry about using `trimIndent` — this function is actually a compiler intrinsic.

Collections

The base class of all collections in Kotlin is `Collection<T>`. Unlike Java, Kotlin's collections provide no mutating operations:

```
val list: List<Int> = ArrayList<Int>()
list.add(3) // ERROR: List has no add method
```

Only `Collection`'s subtype, `MutableCollection`, allows the object to be mutated. The same applies to all of the built-in collection types.

Similarly to arrays, you can create collections using the builder functions `xxxOf`:

```
val unmodifiableList = listOf(1, 2, 3)
val modifiableList = mutableListOf(1, 2, 3)
val arrayList = arrayListOf(1, 2, 3)
val hashMap = hashMapOf("key" to "value", "hello" to "world") // Takes
a list of Pair<,>s
val set = setOf(1, 2, 3, 2) // {1, 2, 3}
```

TIP	The syntax `"key" to "value"` makes use of the infix function `to`, which takes two values and constructs a `Pair` from them. You could also have written `hashMapOf(Pair("key", "value"), Pair("hello", "world"))`, but it wouldn't have been as readable.

[1] https://kotlinlang.org/docs/reference/basic-types.html

[2] https://kotlinlang.org/docs/reference/basic-types.html#explicit-conversions

[3] This is somehow the context for my second-most popular StackOverflow answer: https://stackoverflow.com/a/48800990