



Types

Types 



Summary of the basic types
in the Kotlin programming
language

Numbers

Kotlin handles numbers in a way close to Java, but not exactly the same. For example, there are no implicit widening conversions for numbers, and literals are slightly different in some cases.

Kotlin provides the following built-in types representing numbers (this is close to Java):

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Note that characters are not numbers in Kotlin.

Literal Constants

There are the following kinds of literal constants for integral values:

- Decimals: `123`
 - Longs are tagged by a capital `L`: `123L`
- Hexadecimals: `0x0F`
- Binaries: `0b00001011`

NOTE: Octal literals are not supported.

Kotlin also supports a conventional notation for floating-point numbers:

- Doubles by default: `123.5`, `123.5e10`
- Floats are tagged by `f` or `F`: `123.5f`

Underscores in numeric literals (since 1.1)

You can use underscores to make number constants more readable:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Representation

On the Java platform, numbers are physically stored as JVM primitive types, unless we need a nullable number reference (e.g. `Int?`) or generics are involved. In the latter cases numbers are boxed.

Note that boxing of numbers does not necessarily preserve identity:



```
val a: Int = 10000
println(a === a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```



Target platform: JVM Running on kotlin v. 1.2.71

On the other hand, it preserves equality:



```
val a: Int = 10000
println(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA == anotherBoxedA) // Prints 'true'
```



Explicit Conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:  
val a: Int? = 1 // A boxed Int (java.lang.Integer)  
val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)  
print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the other i
```

So equality would have been lost silently all over the place, not to mention identity.

As a consequence, smaller types are NOT implicitly converted to bigger types. This means that we cannot assign a value of type `Byte` to an `Int` variable without an explicit conversion

+

```
val b: Byte = 1 // OK, literals are checked statically  
val i: Int = b // ERROR
```



Target platform: JVM Running on kotlin v. 1.2.71

We can use explicit conversions to widen numbers

+

```
val i: Int = b.toInt() // OK: explicitly widened  
print(i)
```



Every number type supports the following conversions:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

Absence of implicit conversions is rarely noticeable because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example

```
val l = 1L + 3 // Long + Int => Long
```


Operations

Kotlin supports the standard set of arithmetical operations over numbers, which are declared as members of appropriate classes (but the compiler optimizes the calls down to the corresponding instructions). See [Operator overloading](#).

As of bitwise operations, there're no special characters for them, but just named functions that can be called in infix form, for example:

```
val x = (1 shl 2) and 0x000FF000
```

Here is the complete list of bitwise operations (available for `Int` and `Long` only):

- `shl(bits)` – signed shift left (Java's `<<`)
- `shr(bits)` – signed shift right (Java's `>>`)
- `ushr(bits)` – unsigned shift right (Java's `>>>`)
- `and(bits)` – bitwise and
- `or(bits)` – bitwise or
- `xor(bits)` – bitwise xor
- `inv()` – bitwise inversion

Floating Point Numbers Comparison

The operations on floating point numbers discussed in this section are:

- Equality checks: `a == b` and `a != b`
- Comparison operators: `a < b`, `a > b`, `a <= b`, `a >= b`
- Range instantiation and range checks: `a..b`, `x in a..b`, `x !in a..b`

When the operands `a` and `b` are statically known to be `Float` or `Double` or their nullable counterparts (the type is declared or inferred or is a result of a [smart cast](#)), the operations on the numbers and the range that they form follow the IEEE 754 Standard for Floating-Point Arithmetic.

However, to support generic use cases and provide total ordering, when the operands are **not** statically typed as floating point numbers (e.g. `Any`, `Comparable<...>`, a type parameter), the operations use the `equals` and `compareTo` implementations for `Float` and `Double`, which disagree with the standard, so that:

- `NaN` is considered equal to itself
- `NaN` is considered greater than any other element including `POSITIVE_INFINITY`
- `-0.0` is considered less than `0.0`

Characters

Characters are represented by the type `Char`. They can not be treated directly as numbers

```
fun check(c: Char) {
    if (c == 1) { // ERROR: incompatible types
        // ...
    }
}
```

Character literals go in single quotes: `'1'`. Special characters can be escaped using 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'`.

We can explicitly convert a character to an `Int` number:

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // Explicit conversions to numbers
}
```

Like numbers, characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.

Booleans

The type `Boolean` represents booleans, and has two values: `true` and `false`.

Booleans are boxed if a nullable reference is needed.

Built-in operations on booleans include

- `||` - lazy disjunction
- `&&` - lazy conjunction
- `!` - negation

Arrays

Arrays in Kotlin are represented by the `Array` class, that has `get` and `set` functions (that turn into `[]` by operator overloading conventions), and `size` property, along with a few other useful member functions:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

To create an array, we can use a library function `arrayOf()` and pass the item values to it, so that `arrayOf(1, 2, 3)` creates an array `[1, 2, 3]`. Alternatively, the `arrayOfNulls()` library function can be used to create an array of a given size filled with null elements.

Another option is to use the `Array` constructor that takes the array size and the function that can return the initial value of each array element given its index:

```
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
asc.forEach { println(it) }
```


As we said above, the `[]` operation stands for calls to member functions `get()` and `set()`.


Note: unlike Java, arrays in Kotlin are invariant. This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure (but you can use `Array<out Any>`, see [Type Projections](#)).

Kotlin also has specialized classes to represent arrays of primitive types without boxing overhead: `ByteArray`, `ShortArray`, `IntArray` and so on. These classes have no inheritance relation to the `Array` class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:


```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

Strings

Strings are represented by the type `String`. Strings are immutable. Elements of a string are characters that can be accessed by the indexing operation: `s[i]`. A string can be iterated over with a `for`-loop:




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




Target platform: JVM Running on kotlin v. 1.2.71

You can concatenate strings using the `+` operator. This also works for concatenating strings with values of other types, as long as the first element in the expression is a string:



```
val s = "abc" + 1  
println(s + "def")
```



Target platform: JVM Running on kotlin v. 1.2.71

Note that in most cases using [string templates](#) or raw strings is preferable to string concatenation.

String Literals

Kotlin has two types of string literals: escaped strings that may have escaped characters in them and raw strings that can contain newlines and arbitrary text. An escaped string is very much like a Java string:

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

Escaping is done in the conventional way, with a backslash. See [Characters](#) above for the list of supported escape sequences.

A raw string is delimited by a triple quote (""""), contains no escaping and can contain newlines and any other characters:

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

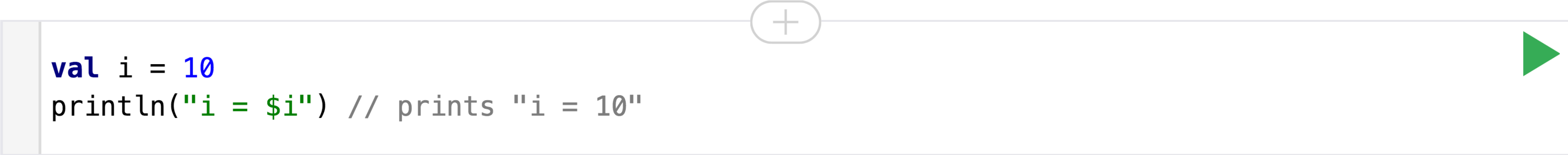

You can remove leading whitespace with [trimMargin\(\)](#) function:

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

By default `|` is used as margin prefix, but you can choose another character and pass it as a parameter, like `trimMargin(">")`.

String Templates

Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign (\$) and consists of either a simple name:



```
val i = 10
println("i = $i") // prints "i = 10"
```

Target platform: JVM Running on kotlin v. 1.2.71

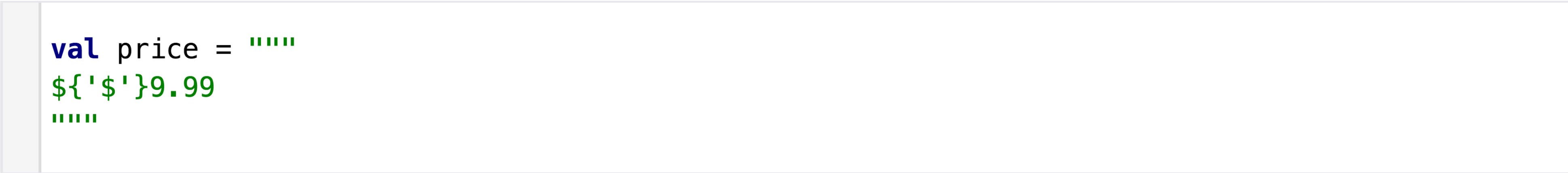
or an arbitrary expression in curly braces:



```
val s = "abc"
println("$s.length is ${s.length}") // prints "abc.length is 3"
```

Target platform: JVM Running on kotlin v. 1.2.71

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal \$ character in a raw string (which doesn't support backslash escaping), you can use the following syntax:



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