Basic Types

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

numbers, characters and booleans can be represented as primitive values at runtime.

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

Note that characters are not numbers in Kotlin.

NOTE: Octal literals are not supported in Kotlin.

### **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(a == b) // Surprise! This prints "false" as Long's equals() check for other part to be Long as well

So not only identity, but even equality would have been lost silently all over the place.

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

We can use explicit conversions to widen numbers

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

Every number type supports the following conversions:

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

**Array Concepts:**

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.

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 Arrayclass, but they have the same set of methods and properties.

### **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

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

Unlike Java, Kotlin does not have a separate ["import static"](https://docs.oracle.com/javase/8/docs/technotes/guides/language/static-import.html) syntax; all of these declarations are imported using the regular import keyword.

If

In Kotlin If you're using if as an expression rather than a statement,it returns a value.

The expression is required to have an else branch.

// As expression **val** max = **if** (a > b) a **else** b

## **When Expression**

when replaces the switch operator of C-like languages. In the simplest form it looks like this

**when** (x) {

1 -> print("x == 1")

2 -> print("x == 2")

**else** -> { // Note the block

print("x is neither 1 nor 2")

}

}

when matches its argument against all branches sequentially until some branch condition is satisfied. whencan be used either as an expression or as a statement. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored.

If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions.

If many cases should be handled in the same way, the branch conditions may be combined with a comma:

**when** (x) {

0, 1 -> print("x == 0 or x == 1")

**else** -> print("otherwise")

}

We can use arbitrary expressions (not only constants) as branch conditions

**when** (x) {

parseInt(s) -> print("s encodes x")

**else** -> print("s does not encode x")

}

We can also check a value for being in or !in a [range](https://kotlinlang.org/docs/reference/ranges.html) or a collection:

**when** (x) {

**in** 1..10 -> print("x is in the range")

**in** validNumbers -> print("x is valid")

!**in** 10..20 -> print("x is outside the range")

**else** -> print("none of the above")

}

**fun** hasPrefix(x: Any) = **when**(x) {

**is** String -> x.startsWith("prefix")

**else** -> **false**

}

when can also be used as a replacement for an if-else if chain.

## **For Loops**

for loop iterates through anything that provides an iterator.

**for** (item **in** collection) print(item)

The body can be a block.

**for** (item: Int **in** ints) {

// ...

}

for iterates through anything that provides an iterator.

A for loop over an array is compiled to an index-based loop that does not create an iterator object.

If you want to iterate through an array or a list with an index, you can do it this way:

**for** (i **in** array.indices) {

print(array[i])

}

Note that this "iteration through a range" is compiled down to optimal implementation with no extra objects created.

Alternatively, you can use the withIndex library function:

**for** ((index, value) **in** array.withIndex()) {

println("the element at $index is $value")

}

## **Break and continue in loops**

Kotlin supports traditional break and continue operators in loops.