1 Kotlin & Kotlin Standard Library Written by Márton Braun

Kotlin is a modern, multi-paradigm programming language developed by JetBrains. It first appeared in 2011 and slowly evolved into one of the most popular languages available today.

One of the reasons developers love Kotlin so much is because it makes app development easier by providing a significant amount of out-of-the-box classes and utilities. Kotlin's classes and utilities are wrapped inside the **Kotlin Standard Library**, which contains the core components of the Kotlin language. Inside this library, you'll find a variety of tools and types to help build your apps.

Before you start building your own custom data structures, it's essential to know about the primary data structures that the Kotlin Standard Library already provides.

In this chapter, you'll start by learning a few things about Kotlin like variables, types, nullability, conditionals, loops and functions. You'll then focus on two specific data structures, `List` and `Map`, both of which are included with the Kotlin Standard Library. You'll end this chapter with a discussion about mutability in the context of these two data structures.

Introduction to Kotlin

To understand data structures and algorithms in Kotlin, you first need to understand the main features of the language. But don't worry: There's nothing overly complicated about Kotlin, especially if you have experience with other modern programming languages. However, regardless of your experience, there are a few things you need to know before diving deep into the details of data structures:

- How to declare variables and functions.
- How to create custom types.

- How to manipulate data in loops and decision-making structures.

Once you get comfortable with the basics, you can move on to something a little more complicated: `Generics`. You can find most of the code of this chapter in the provided projects.

Ready to get started?

Variables and types

A variable is a way to store information. Typically, a variable has a name and a type. Variables can also have modifiers that add extra options or restrictions to it.

In Kotlin, there are two types of variables, `val` and `var`:

```
val name = "Bill Clinton"
var country = "Romania"
```

The difference between `val` and `var` is that variables declared with `val` cannot be reassigned:

```
name = "Matei Suica" // compile error
country = "Pakistan" // Ok
```

Since `name` was defined with `val` its value cannot be changed from "Bill Clinton", but since `country` was defined with `var`, its value can be updated.

> **Note:** You can think of `var` as a **variable**, while `val` is a **value**.

With regard to types: The Kotlin compiler can sometimes determine the type of the variable. This is referred to as **type inference**, which is a feature of many modern programming languages.

The variables in the previous example are of type `String`. This is clear to the compiler because they were initialized when they were declared. Since nothing was ambiguous about them, you did not have to declare the type. However, this may not always be the case. For example, you can declare a variable that references a number but initializes it later:

```
var score: Int
```

In this example, since no initial value is set, its type cannot be inferred and so you must explicitly set the type for `score`.

There are several types in Kotlin that are already defined for you. The Kotlin Standard Library includes more than are covered here, but the basic types are:

- **Numbers**: `Double`, `Float`, `Long`, `Int`, `Short`, `Byte`.
- **Characters**: `Char`, `String`.
- **Other**: `Boolean`, `Array`.

As you work through this book, you'll encounter most of these types. However, at this point, you don't need to study their specifics, only acknowledge their existence. Later, you'll create complex structures to store these kinds of types.

Null-safety

Many programming languages, including Kotlin, have the concept of a `null` value. You can assign `null` to a variable whenever you want to signal that it doesn't have a value.

For example, if you have a variable that can hold a `Car` but you've not yet created a `Car` object, the variable can hold a `null`:

```
var car: Car? = null
```

Upon object creation, you could easily reassign the variable:

```
car = Car("Mercedes-Benz")
```

The problem with the presence of `null` is that you might try to use it. Assuming that `Car` defines a `drive()` method, you might decide to try something like this:

```
car.drive()
```

When you have a value assigned, like `car = Car("Mercedes-Benz")`, you won't have an issue; however, if you try to do this with a `null` value, the program will crash. This is where the infamous **NPE**, Null-Pointer Exception, was born.

To prevent an NPE, Kotlin has a neat system baked into the language. Noticed the **?** after the `Car` type in the first declaration? That question mark changes the variable type to a **nullable** type. This type tells the compiler that your variable *could* either contain a `Car` object or a `null` value. This small detail triggers a chain reaction in the code.

For example, with a nullable type, you cannot use:

```
car.drive()
```

Instead, you need to use the safe-call operator `?.`:

```
car?.drive()
```

Using a safe-call operator means that this function will only execute if the object is not `null`.

To fall back to another value in case a variables holds a `null` value, you can use `?:`, which is also known as the Elvis operator:

```
val realCar: Car = car ?: Car("Porsche")
```

This code does a lot in a single line:

1. Creates a variable `realCar` that cannot be reassigned.
2. The `realCar` variable has a non-nullable `Car` type. You can be sure that there's a real car that you can `drive()` in that variable.
3. `realCar` can be either the same as the `car` or a Porsche if `car` happens to be `null` (not contain a real car).

These language features are nice, but there are cases where you don't want to play by the rules.

You know that your variable is not a `null` — even though it's nullable — and you need to tell the compiler to use the value that it holds. For this, Kotlin has the not-null assertion operator `!!`. You can use it instead of the safe-call operator:

```
car!!.drive()
```

This calls `drive()` on the non-null value that `car` holds; if, however, it holds a `null`, it'll throw an NPE. Therefore, you should think twice before using it. That could be why the JetBrains team made this operator a double-bang: Think! Twice!

Conditional statements

Programs in Kotlin execute linearly; in other words, one line at a time. While this is easy to follow and clean, it's not very useful. There are a lot of situations where making a decision or repeating a step can come in handy. Kotlin has structures that resolve both of these problems in a concise fashion.

For decision making, Kotlin has two constructs, `if-else` and `when`. Unlike other languages, there's no ternary operator in Kotlin; you have to use `if-else` to get the same result. Here's an example:

```
val a = 5
val b = 12
var max = -1

if (a > b) {
  max = a
} else {
  max = b
}

println(max) // prints 12
```

The code above makes a decision based on the condition inside the brackets.

If `a > b` is `true`, the first block of code is executed, and `max` takes the value of `a`. If the condition is `false`, then `max` takes the value of `b`.

In the structure, the `else` part is optional. You might want only to do something if the condition is `true` but not when it's `false`. Instead of leaving the `else` block empty, you can omit it.

`when` is much like a series of `if-else` that can handle many cases:

```
val groupSize = 3

when (groupSize) {
  1 -> println("Single")
  2 -> println("Pair")
  3 -> { // Note the block
    println("Trio")
  }
  else -> println("This is either nobody or a big crowd")
}
```

In this example, the `when` structure makes a decision based on the value of `groupSize`. It has some particular cases like 1, 2 or 3, and then an `else` clause that handles everything that isn't specified above.

For the `when` structure, the `else` can be optional if the compiler determines that you already handled all of the possible values.

Loops

There are two types of loops in Kotlin, `for` and `while`. Although you can do just fine only using `while`, in some situations, using `for` is easier and more elegant.

Let's start with the elegant one:

```
for (i in 1..3) {
  println(i)
}
```

`for` can iterate over any iterable collection of data. In this example, `1..3` creates an `IntRange` that represents the numbers from 1 to 3. The `i` variable takes each value, one at a time, and goes into the code block with it. In the block, `println()` is executed and the value of `i` goes into the standard output.

Here's a more generic example:

```
for (item in collection) println(item)
```

This prints all of the items in the `collection`. Note how braces are not mandatory in a `for` loop, though they usually make the code more readable.

The second type of loop is the `while` loop, which executes the same block of code as long as its condition remains `true`:

```
var x = 10
while (x > 0) {
  x--
}
```

This code starts with `x` having the value of `10` and since `10` is greater than `0`, it executes the code inside the block. There, `x` decreases by `1`, becoming `9`. Then, the loop goes back to the condition: "Is 9 greater than 0?". This continues until eventually `x` gets to `0` and the condition becomes `false`.

There is a variation of `while` known as `do-while`. The `do-while` loop first executes the code and then checks for the condition to continue. This ensures that the block of code is executed at least once:

```
var x = 10
do {
  x--
} while (x > 0)
```

One thing to notice with `while` loops is that you can easily create an infinite loop. If you do, your program will get stuck and eventually die in a pitiful StackOverflowException or something similar:

```
var x = 10
while (x > 0) {
  x++
}
println("The light at the end of the tunnel!")
```

This time, `x` gets incremented instead of decremented: `10`, `11`, `12`, `13`, and so on. It never gets less than or equal to `0`, so the `while` loop has no reason to stop. In other words, you'll never get to see the light at the end of the tunnel.

Functions

Functions are an important part of any programming language, especially in Kotlin as it's a multi-paradigm programming language. Kotlin has a lot of functional programming features, so it treats functions with the respect they deserve!

In general, programming is based on small units of code that can be abstracted and reused. Functions are the smallest units of code that you can easily reuse. Here's an example of a function:

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

This is a simple function that compares two numbers and determines which is higher.

Functions are declared using the `fun` keyword, followed by the name of the function. By naming functions, you can then call them using their name, as you'll see momentarily.

Functions might also have a list of **parameters**. Each parameter has a name that you can use to refer to them, as well as a type. This function has two parameters, `a` and `b`; however, functions can also have more parameters or no parameters at all.

After a colon `:`, there's another type; this is the function's return type. In other words, the result of this function will have that type. A function cannot have more than one return type; however, it's possible for it to have no return type specified at all.

Lastly, there's the code block describing the steps the function will execute: its **body**. Because this function has a return type, it also needs to `return` a value of that type. In this case, it returns either `a` or `b`, and since both are `Int`s, the return type is also `Int`.

Here's an example of a function that has no return type specified:

```
fun printMax(c: Int, d: Int) {
  val maxValue = max(c, d)
  println(maxValue)
}
```

Again, because this function does not declare a return type, there's no need for a `return` keyword. So what's the point of this function if it contains no return value?

Well, if you look closely, you'll see that this function calls `max` by its name, and passes in two parameters, `c` and `d` (which `max` renames to `a` and `b`). From there, `printMax` takes the result of `max`, stores it in a variable named `maxValue` and prints it the console.

> **Note**: This chapter does not cover higher-order functions and lambdas as these concepts are more complex. You will, however, touch on them in later chapters.

Generics

Generics are a great way to abstract your code whenever you can manipulate multiple types in the same way.

Consider a class that emulates a box. A class is simply a collection of data and functions that are logically grouped to perform a set of specific tasks. When creating a class, think about how you might use it. In this case, with a box, you:

- Put something in it.
- Grab something out of it.

- Check if the box is empty.

Here's some code that can perform these tasks:

```
class Box {
  var content: Any? = null

  fun put(content: Any?) {
    this.content = content
  }

  fun retrieve(): Any? {
    return content
  }

  fun isEmpty(): Boolean {
    return content == null
  }
}
```

This is a simple class that can store a value via `put()`, can retrieve a value via `retrieve()`, and can check if the box is empty via the `isEmpty()` method.

Since you want the box to store different kinds of objects, the type is set to `Any` since the `Any` class is the superclass of all objects in Kotlin.

This could work, but there's one drawback: Once you put something into the box, you lose the knowledge of the object's type since you had to use the `Any` type to store any kind of object.

To get a more specialized box, you could replace `Any` with the type you need; for example, a `Cat` or a `Radio`. But you'd need to create a different type of Box for every type of object you'd want to store, i.e. you'd have to create `CatBox` and `RadioBox` separately.

Generics are an excellent way to keep the code abstract and let the objects specialize once instantiated. To abstract `Box`, you can write it like this:

```
class Box<T> {
  var content: T? = null

  fun put(content: T?) {
    this.content = content
  }

  fun retrieve(): T? {
    return content
  }

  fun isEmpty(): Boolean {
    return content == null
  }
}
```

Now, to benefit from a specialized box for this generic, you need to instantiate it:

```
val box = Box<Int>()
box.put(4)

val boolBox = Box<Boolean>()
boolBox.put(true)
boolBox.isEmpty()
```

Your box can handle any type you want, and you'll be sure that whatever you put in it, has the same type when you remove it from the box.

You can also apply generics at a function level, and there can be restrictions applied to the kind of types the generic will accept. In Kotlin, there's a way to say "I want all functions to return this generic type" or "I want only the input parameters to be this generic type".

There's a lot to learn about Generics, and you'll need to research it as you progress with your data structures and algorithms. But for now, you'll start with two of the most common generic data structures that are already provided by the Kotlin Standard Library.

The Kotlin Standard Library

With the Kotlin Standard Library you can get away with not using any third-party libraries for most things. It contains useful classes and functions for text manipulation, math, streams, multithreading, annotations, collections and more.

There are many things to mention, but this book can't cover everything now, so keep your focus on the parts of the library that will help you with the algorithms.

Here are a few things to consider:

Package kotlin

This package contains many helpful higher-order functions. It also contains the definition of the most basic classes, exceptions and annotations. In this package, you'll find `Any`, `Array`, `Int`, `ClassCastException` and `Deprecated` to name a few. The most interesting things are the scoping functions defined in this package.

`let`

The `let` function helps you with null-checks and creates a new local scope to safely perform operations. Here's an example:

```
fun printCar(car: Car?) {
  val isCoupe = car?.let {
    (it.doors <= 2)
  }

  if (isCoupe == true) {
    println("Coupes are awesome")
  }
}
```

Inside `let`, `it` holds the value of `car`, but its type is `Car` instead of `Car?`. Since you're using the safe-call operator `?.`, the code block won't run if `car` is `null`. That's how the compiler can give `it` the non-nullable type that's easier to work with. As you might notice, `let` can return anything. In this case, it returns a `Boolean` telling you if the printed car was a coupé.

`let` gives you the instance of the class you called it on as `it` inside the block. This is helpful in a lot of situations. There are other functions that have a different approach.

`run`

`run` is similar to `let`, but it's more focused on the target object — the one you're using to call the function. Inside the block, `run` provides the target object as `this` and isolates the block from the outer scope.

```
fun printCar2(car: Car?) {
  val isCoupe = car?.run {
    (this.doors <= 2)
  }

  if (isCoupe == true) {
    println("Coupes are awesome")
  }
}
```

This is the same example, but now you're isolated inside `run`. The return value can still be anything.

These two functions are "transformational" functions. They're called "transformational" because the object they return can be different from the object you call the function on. This is not the case with the following "mutating" functions.

`also`

If you try to replace `run` with `also`, you'll get compile errors. Unlike with `let` or `run` which return a transformation, the `also` function returns the original object.

Now, don't get tricked into thinking that original means that it's unmodified. It's just the same object. `also` uses `it` to refer to the object inside of the block.

```
fun printCar3(car: Car?) {
  car?.also {
    it.doors = 4
  }.let {
    if (it?.doors != null && it.doors <= 2) {
      println("Coupes are awesome")
    }
  }
}
```

Since `also` returns the same `car` object, you can use it to mutate the object and then chain other calls to it. In this example, the check to see if the car is a coupe is within a `let` block, but since it was modified to have 4 doors within `also`, it won't print "Coupes are awesome".

`apply`

By now, you might be able to guess how `apply` works. It's an `also` that is isolated like a `run`. It returns the same object as the target, and it uses `this` inside the block:

```
fun printCar4(car: Car?) {
  car?.apply {
    doors = 4
  }.let {
    if (it?.doors != null && it.doors <= 2) {
      println("Coupes are awesome")
    }
  }
}
```

Again, the car has been updated to have 4 doors so it also won't print "Coupes are awesome"

These functions will come in handy from time to time, especially if you want to write clean and concise code.

There's one more function defined in **Standard.kt** that you'll see a lot. It's not the most useful one, but it's very common.

`TODO`

The JetBrains team decided to define `TODO` inside the Standard Kotlin Library to prevent one of the decades-old habits of software developers: forgetting about **TODOs**.

Have a look at the definition of `TODO`:

```
public inline fun TODO(): Nothing = throw NotImplementedError()
```

`TODO()` throws an error when the code reaches one of these TODOs. This is a clever trick to prevent forgetting that you still have to write something. You'll see this every time IntelliJ generates a piece of code for you to implement. Just don't forget about it!

List

The second important package in the Kotlin Standard Library is **kotlin.collections**. You'll use it a lot in the following chapters and even more in real-life programming. For this introduction, you'll focus only on two basic collections, `List` and `Map`.

A `List` is a general-purpose, generic container for storing an ordered collection of elements; it's used commonly in many types of Kotlin programs.

You can create a `List` by using a helper function from the Kotlin Standard Library named `listOf()`. For example:

```
val places = listOf("Paris", "London", "Bucharest")
```

> **Note**: Kotlin defines lists using interfaces. Each of these interface layers more capabilities on the list. For example, a `List` is an `Iterable`, which means that you can iterate through it at least once.
>
> It's also a `Collection`, which means it can be traversed multiple times, non-destructively, and it can be accessed using a subscript operator `[]`.
>
> For `List`, the positional access function `get()` guarantees access efficiency and it's the same as using the subscript operator.

Because Kotlin differentiates between mutable and read-only data structures, you'll want to create a `MutableList` to talk about all the operations lists have.

You'll learn more about mutability in Kotlin later in this chapter, but for now, just add another layer on top of your list, and create it like this:

```
val mutablePlaces = mutableListOf("Paris", "London", "Bucharest")
```

The Kotlin `List` is known as a **generic collection** because it can work with any type. In fact, most of the Kotlin standard library is built with generic code.

As with any data structure, there are certain notable traits of which you should be aware. The first of these is the notion of **order**.

Order

Elements in a list are explicitly **ordered**. Using the above `places` list as an example, `Paris` appears before `London`.

All of the elements in a list have a corresponding **zero-based**, integer index. For example, `places` from the above example has three indices, one corresponding to each element, starting with `0`.

You can retrieve the value of an element in the list by writing the following:

```
places[0] // Paris
places[1] // London
places[2] // Bucharest
```

The order should not be taken for granted. Some data structures, such as `Map`, have a weaker concept of the order or no order at all. You can end up with a different order when you try to access elements out of different collections.

Random-access

Random-access is a trait that data structures can claim if they can handle element retrieval in a constant amount of time.

For example, getting `"London"` from `places` takes constant time. This means that there's no performance difference in accessing the first element, the 3rd element or any other element of the list. Again, this performance should not be taken for granted. Other data structures such as `Linked Lists` and `Trees` do not have constant time access.

For linked lists, the further the element is, the longer it takes to access it. You'll learn more about the complexity of the operations in the next chapter.

List performance

Aside from being a random-access collection, there are other areas of performance that are of interest on how well or poorly does the data structure fare when the amount of data it contains needs to grow. For lists, this varies on two factors.

Insertion location

The first factor is one in which you choose to insert the new element inside the list. The most efficient scenario for adding an element to a list is to append it at the end of the list:

```
mutablePlaces.add("Budapest")
println(mutablePlaces) // prints [Paris, London, Bucharest, Budapest]
```

Inserting `"Budapest"` using `add()` places the string at the end of the list. This is a **constant-time** operation, meaning the time it takes to perform this operation stays the same no matter how large the list becomes.

However, there may come a time that you need to insert an element in a particular location, such as in the middle of the list. To help illustrate, consider the following analogy. You're standing in line for the movies. Someone new comes along to join the lineup. If they just go to the end of the line, nobody will even notice the newcomer. But, if the newcomer tried to insert themselves into the middle of the line, they would have to convince half the lineup to shuffle back to make room. And if they were *terribly* rude, they may try to insert themselves at the head of the line. This is the worst-case scenario because every single person in the lineup would need to shuffle back to make room for this new person in front!

This is exactly how lists work. Inserting new elements from anywhere aside from the end will force elements to shift back to make room for the new element:

```
mutablePlaces.add(0, "Kiev")
// [Kiev, Paris, London, Bucharest, Budapest]
```

To be precise, every element must shift back by one index. If we consider the number of items in the list to be *n*, this would take *n* steps. The time for this operation grows as the number of elements in the list grows. If the number of elements in the list doubles, the time required for this `add` operation will also double.

If inserting elements in front of a collection is a common operation for your program, you may want to consider a different data structure to hold your data.

Capacity

The second factor that determines the speed of insertion is the list's **capacity**.

Underneath the hood, Kotlin lists are allocated with a predetermined amount of space for its elements. If you try to add new elements to a list that is already at maximum capacity, the `List` must restructure itself to make more room for more elements.

This is done by copying all the current elements of the list in a new and bigger container in memory. However, this comes at a cost. Each element of the list has to be accessed and copied. This means that any insertion, even at the end, could take *n* steps to complete if a copy is made.

> **Note**: The Standard Library employs a strategy that minimizes the times this copying needs to occur. Each time it runs out of storage and needs to copy, it doubles the capacity.

Map

A `Map` is another generic collection that holds **key-value** pairs. For example, here's a map containing a user's name and a score:

```
val scores = mutableMapOf("Eric" to 9, "Mark" to 12, "Wayne" to 1)
```

There's no restriction on what type of object the `Key` is, but you should know that a `Map` uses the `hashCode()` function to store the data. Usually, the `Key` is one of the Standard Library types which have the `hashCode()` function implemented. But if you want to use your own type, you need to implement the function yourself.

It's not difficult to override `hashCode()`, you just have to investigate a little bit what are the most common strategies to get the best result out of the `Map`.

You can add a new entry to the map with the following syntax:

```
scores["Andrew"] = 0
```

This creates a new key-value pair in the map:

```
{Eric=9, Mark=12, Wayne=1, Andrew=0}
```

Maps are unordered, so you can't guarantee where new entries will be put. This is because maps put data into different *buckets*, depending on the result that the `hashCode()` function returns. The data in each bucket is ordered, but the general order of the data in the map is unpredictable.

It is possible to traverse through the key-values of a map multiple times as the `Collection` protocol affords. This order, while not defined, will be the same until the collection is changed.

The lack of explicit ordering disadvantage comes with some redeeming traits.

Unlike the list, maps don't need to worry about elements shifting around. Inserting into a map always takes a constant amount of time.

> **Note**: Lookup operations also take a constant amount of time, which is significantly faster than finding a particular element in a list which requires a walk from the beginning of the list to the insertion point.

Mutable vs. read-only

As you've seen throughout the chapter, there's a distinction between mutable and read-only data structures in Kotlin.

When referring to the concept of a `List`, it's usually referring to the Kotlin's `MutableList`. Unlike `List`, `MutableList` also has functions for adding and removing elements. Kotlin doesn't allow a `List` to be changed in any way.

To change a data structure, you must express this intent by using the `Mutable` version of that data structure. These data structures have functions for adding and removing elements.

So why would you ever use the read-only version? **For safety**.

Whenever you need to pass your data structure as a parameter, and you want to be sure that the function doesn't produce a side effect, you should use an read-only collection as the parameter.

Consider this code:

```
fun noSideEffectList(names: List<String>) {
  println(names)
}

fun sideEffectList(names: MutableList<String>) {
  names.add("Joker")
}

fun mutableVsReadOnly() {
  val people = mutableListOf("Brian", "Stanley", "Ringo")
  noSideEffectList(people) // [Brian, Stanley, Ringo]
  sideEffectList(people)   // Adds a Joker to the list
  noSideEffectList(people) // [Brian, Stanley, Ringo, Joker]
}
```

The `sideEffectList` function adds a Joker to it. These kind of side-effects are usually the ones generating bugs. Avoiding them by using a `List` instead of a `MutableList` is preferred.

Key points

- Every data structure has advantages and disadvantages. Knowing them is key to writing performant software.
- Functions such as `add(Int, Any)` for `List` have performance characteristics that can cripple performance when used haphazardly. If you find yourself needing to use `add(Int, Any)` frequently with indices near the beginning of the list, you may want to consider using a different data structure such as the `Linked List`.
- `Map` trades the ability to maintain the order of its elements for fast insertion and searching.
- Kotlin encourages you to use read-only variants of collections whenever possible.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.