

# ANDROID BOOK

1. Introduction .....	3
1.1. Mobile Communication Technologies .....	3
1.2. Android .....	4
1.3. iOS .....	9
2. Types of mobile applications .....	11
2.1. Native applications .....	11
2.2. Responsive web applications (WebApps) .....	11
2.3. Hybrid applications .....	11
2.4. Progressive web applications .....	12
2.5. Compiled applications .....	13
3. Kotlin Language .....	14
3.1. Introduction .....	14
3.1.1. Kotlin installation .....	14
3.1.2. Variables and data types .....	15
String Templates .....	16
Nullable Types and Elvis operator .....	16
3.1.3. Operators .....	17
Assignment operator .....	17
Arithmetic operators .....	17
Relational operators .....	17
Logical operators .....	18
Ternary conditional operator .....	18
3.1.4. Control structure .....	18
If-Else .....	18
When .....	18
3.1.5. Exceptions .....	20
3.1.6. Functions .....	21
Definition and invocation .....	21

Lambda Expressions basics .....	21
Anonymous functions .....	23
3.1.7. OOP .....	24
Classes and Objects.....	24
Inheritance .....	24
Encapsulation.....	26
Use of Generics .....	27
Enum Classes.....	27
Singleton Objects .....	28
3.1.8. Complex data typeS .....	29
Classes and Data Classes.....	29
List / MutableList .....	30
Set / MutableSet .....	31
Map / MutableMap.....	31
Arrays .....	32
High-order functionS with collections .....	32
forEach().....	32
map() .....	33
filter() .....	34
groupBy().....	34
fold().....	35
sortedBy().....	36

## 1. INTRODUCTION

In this introductory section, we will take a quick look at the primary technologies at our disposal for portable app development.

Now, find yourself wanting to create a mobile app:

- What's the first thing that crosses your mind?
- Which programming language should be used for development?
- In which development environment?
- And for which devices? Mobile phones? Tablets? Smartwatches? Televisions?

It's a question somehow comparable in complexity to when we contemplate developing an application for a personal computer.

### 1.1. Mobile Communication Technologies

The mobile communication network has evolved over time with different generations bringing changes to the previous one. The different generations and their main characteristics are:

- **Generation 0:** This generation can be considered the first communications between mobile devices using radio waves.
- **First generation (1G technology):** the first automated mobile communication network was launched in 1979 in Japan and had problems with voice communications and security limitations since the voice calls were replicated on the radio towers.
- **Second generation (2G technology):** 1990s. It was the first to provide digital voice and data, as well as international roaming. From the 2000s on it introduced packets into the network to provide high-speed data transfer and high-speed Internet.
- **Third generation (3G technology):** Its main objective was to offer an increase in data rates, facilitating growth and offering greater voice and data capacity, as well as lowering the cost of transmissions.
- **Fourth generation (4G technology):** The main objective was to provide high speed, high quality, high capacity and all this at low prices for both voice and data, as well as increasing the security of communications.
- **Fifth generation (5G technology):** This technology, which began its commercial launch in 2019, is 10 times faster than 4G, reaching 1Gbps transmission speed.

- **Sixth generation (6G technology):** Its main objective will be to reduce the latency of the connections and increase the transmission speed. It is estimated that commercialization will be in 2030 and that the first real use cases could arrive as from 2026.

## 1.2. ANDROID

Android is an OS developed by Google and based on the Linux Kernel, made specifically for touch-screen portable devices: mobile phones, tablets, smart watches, TVs or even some cars.


The Android mobile operating system's journey commenced with the public debut of its inaugural beta on November 5, 2007. Subsequently, the initial commercial iteration Android 1.0 saw the light on September 23, 2008. Google has consistently overseen the development of this operating system on an annual basis since 2011. Major new releases are unveiled during Google I/O, simultaneously accompanied by the release of the first public beta for supported Google Pixel devices. The stable version is subsequently rolled out later in the year.



NAME	VERSION	LAUNCHING DATE	MAIN IMPROVEMENTS
Android 1.0	1.0 - 1.1	september 2008	First stable version
Android Cupcake	1.5	april 2009	Refined design Virtual keyboard Widgets for apps <b>Copy and paste in browser</b> Animated transitions <b>Automatic screen rotation</b>
Android Donut	1.6	september 2009	Quick search Revamped Android Market Adapted to more screen formats Voice synthesiser Camera and gallery improvements CDMA and VPN support

<b>Android Eclair</b>	2.0 - 2.1	october 2009	<b>Routes in Maps</b> Support for multiple accounts Live Wallpapers <b>Flash and zoom support</b> Improvements to pre-installed apps like Maps, browser or calendar
<b>Android Froyo</b>	2.2 - 2.2.3	may 2010	Voice commands <b>Wi-Fi hotspots</b> Improved browser performance Flash support C2DM push notifications Move apps to SD
<b>Android Gingerbread</b>	2.3 - 2.3.7	december 2010	API for games <b>NFC</b> First easter egg Icon design changes Support for WXGA resolution and higher Select before copy Support for multiple cameras Gyroscope and barometer support Video calling in Hangouts
<b>Android Honeycomb</b>	3.0 - 3.2.6	february 2011	<b>Adapted for tablets</b> System Bar Quick settings Browser tabs Hardware acceleration USB OTG support
<b>Android Ice Cream Sandwich</b>	4.0 - 4.0.4	october 2011	Holo interface Navigation bar Folders Roboto Typography <b>Screenshots</b> <b>Face unlock</b> Dismiss notifications one by one
<b>Android Jelly Bean</b>	4.1 - 4.3.1	july 2012	Google Now Smoother movement Quick settings Better accessibility Widgets on the lock screen Native emoji support
<b>Android KitKat</b>	4.4 - 4.4.4	october 2013	Design changes <b>Immersive mode</b> ART Revamped Clock, Phone and Downloads app
<b>Android Lollipop</b>	5.0 - 5.1.1	november 2014	Material Design New lock screen (without widgets) Performance improvements Recent improvements Settings finder

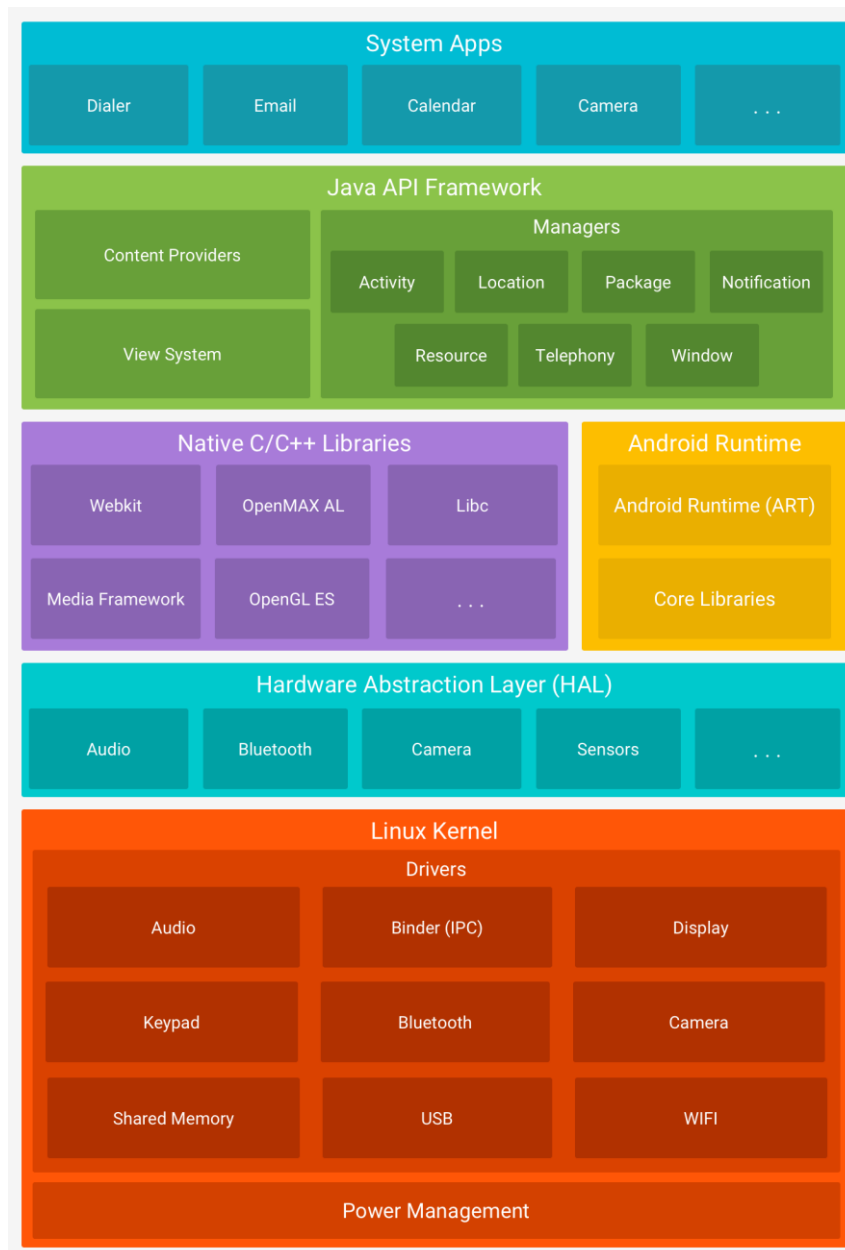
<b>Android Marshmallow</b>	6.0 - 6.0.1	october 2015	Runtime permissions Doze mode USB-C and 4K support <b>Fingerprint reader support</b> Experimental multi-window Direct Share Now on Tap
<b>Android Nougat</b>	7.0 - 7.1.2	august 2016	Doze improvements JIT compiler improvements Daydream VR <b>Multi-window mode</b> PIP on Android TV Vulkan 3D Quick app settings Launcher shortcuts
<b>Android Oreo</b>	8.0 - 8.1	august 2017	Project Treble Mobile PIP mode Adaptive icons Notification changes Autocomplete API Performance optimisations
<b>Android Pie</b>	9.0	august 2018	Privacy enhancements Brightness and smart battery App actions App slices <b>Digital well-being</b> Gesture navigation
<b>Android 10 (Quince Tart)</b>	10.0	september 2019	Dark mode Real-time subtitles Intelligent responses New gesture navigation <b>Foldable optimisations</b> Privacy improvements Google Play system updates
<b>Android 11 (Red Velvet Cake)</b>	11.0	september 2020	Changes to notifications <b>Chat bubbles</b> <b>Native screen recorder</b> Domotics in the shutdown menu One-time permission Android Auto wireless for all
<b>Android 12 (Snow Cone)</b>	12.0	october 2021	Material You Privacy Enhancements Approximate location permission <b>Microphone, camera and location usage flags</b> Domotics disappears from shutdown menu Performance improvements Scrolling screenshots
<b>Android 12L (Snow Cone V2)</b>	12.1	march 2022	Optimised for tablets, foldables and computers Lower taskbar Multitasking and multi-window improvements Two-column layout

<b>Android 13 (Tiramisu)</b>	13.0	august 2022	More Material You customisation Permission changes New notification permissions <b>Choose language for each app</b> QR reader New photo selector Active apps New clipboard menu
<b>Android 14 (Upside Down Cake)</b>	14.0	october 2023 on Pixel devices	
<b>Android 15 (Vanilla Ice Cream)</b>	15.0	expected 2024	

Ref: 1 - <https://www.xatakandroid.com/sistema-operativo/todas-versiones-android-historia> and Wikipedia

The Android software stack is composed of applications that run in a Java framework of object-oriented applications on top of the Java core libraries. The Java virtual machine on which these applications run was Dalvik until version 5.0, to change in later versions to the Android Runtime (ART) environment. The main difference between these virtual machines was that Dalvik performed the compilation at runtime, while ART compiles the Java bytecode during the installation of the application.

The used libraries are written in C/C++ languages and include (among others): a graphical interface manager (surface manager), an OpenCore framework, a SQLite relational database, an OpenGL ES 2.0 3D graphical programming interface, a WebKit rendering engine, an SGL graphics engine, SSL and a standard C library.



Ref: 2 - <https://developer.android.com/guide/platform>

The language for Android development has traditionally been Java. However, Google has adopted Kotlin as the official Android programming language, which is a more powerful language that generates executable code directly in the JVM.



Kotlin	Java
<pre> data class Person(var name: String, var age: Int)  /* var: read and write    val: read-only i.e. no setters */ </pre>	<pre> public class Person {     private String name;     private String age;      public Person(String name, String age) {         this.name = name;         this.age = age;     }      public String getName() {         return name;     }      public void setName(String name) {         this.name = name;     }      public String getAge() {         return age;     }      public void setAge(String age) {         this.age = age;     }      @Override     public boolean equals(Object o) {         /*         Code for equals function         */     }      @Override     public int hashCode() {         /*         Code for hashCode function         */     } } </pre>

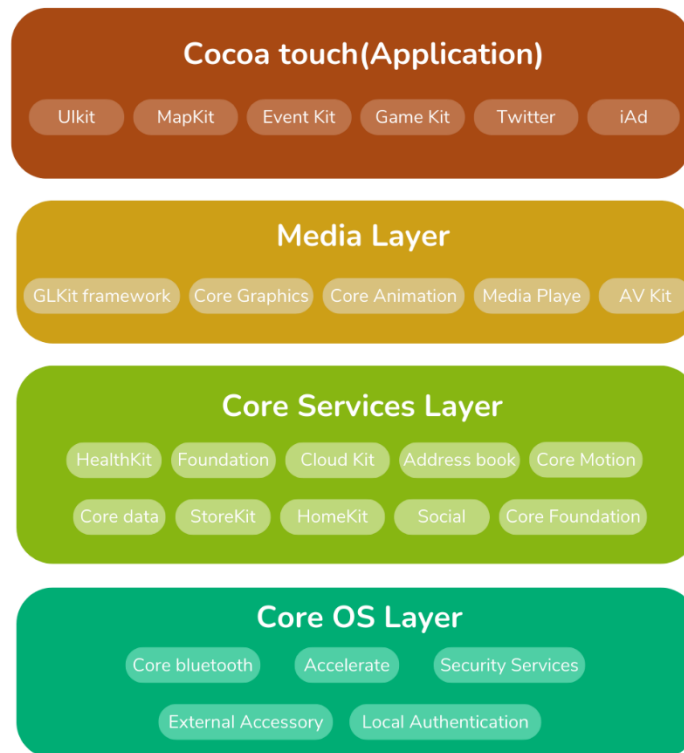
### 1.3. iOS

Android is the best-selling operating system and after that it comes iOS, owned by Apple®. It was created for the iPhone, and later adopted in the iPod touch and the iPad. iOS does not allow installation on hardware from other companies.

iOS provides various control elements, such as sliders, toggles and buttons, giving an immediate response to user commands and a fluid interface. The user can interact with the operating system through gestures, swipes, touches, pinches...

iOS is derived from macOS, and macOS is derived from Darwin BSD, which is a Unix-like operating system.

The iOS architecture has four abstraction layers: the Core OS layer, the Core Services layer, the Media layer and the Cocoa Touch layer. Native application development for iOS involves languages such as Objective-C and Swift.



Ref: 3 - <https://redfoxsec.com/blog/ios-architecture/>

## 2. TYPES OF MOBILE APPLICATIONS

### 2.1. NATIVE APPLICATIONS

These are the applications that are developed specifically for the operating system in which they will be executed, and therefore, will make a better use of all their resources. In addition, they allow access to all the functionalities of the platforms and users will have the new features available right from the beginning.

For native development on Android, Java and Kotlin are used, while Android also supports Objective-C and Swift (iOS), and no additional steps are needed to bring the code to native code (bytecode).

All this results in fluid applications that offer the best user experience. As a negative point, we will have an increase in the cost of production and maintenance.

### 2.2. RESPONSIVE WEB APPLICATIONS (WebApps)

These are applications based on web technology: HTML + CSS + JavaScript. To run they only need a web browser. Being responsive means that its interface adapts to any device.

For this type of application, it is not necessary to develop in native code. They are fully cross-platform since they run on the web browser of the operating system. Same app can run on Android, iOS, Windows, Linux or Mac.

So, one has a single code to run on all platforms, with the disadvantage that it will not offer as good experience to the user as native apps, especially if it is a complex application.

### 2.3. HYBRID APPLICATIONS

Hybrid applications refer to those that use web technology to build a website (HTML + CSS + JS) and load it in a *WebView*. A *WebView* is nothing but a web browser without the navigation bar and other options, so it looks like a native app on the device.

In addition, hybrid applications can access to some features of the device, such as location, accelerometers, etc.

The frameworks most used for this type of application are **Ionic** and **PhoneGap**. These are two frameworks widely used by web application developers to create applications for mobile devices. The idea behind these frameworks is simple: The web application is packaged in HTML + CSS + JS code together with *WebView*. The *WebView* will be in native code for each system, but the web application is the same. The disadvantage of these technologies is that

they react slower since they must communicate with the website to obtain much of the content.

These frameworks are comparable to Electron or NW.js (previously known as *node-webkit*) in desktop environments.

## 2.4. PROGRESSIVE WEB APPLICATIONS

A little closer to native applications are progressive web applications (PWA), which are revolutionizing the current landscape. These applications increase the functionalities according to the mobile device on which they run, to extract more potential from it (they adapt progressively to the device).

These applications provide the user with many of the advantages of native applications, but with development based on web technologies.

Unlike hybrid applications, they allow operation with a poor connection to the server. Using technologies like Service Workers, Cache API, and Web Storage API, PWAs can store recently accessed app information. That way, when navigating back to a previous page, user can see what was already loaded instead of getting the dreaded “currently offline” page. Once connected again, PWAs can seamlessly update content and manage the network requests.

There are many frameworks for the development of PWAs, among which are the main web development frameworks: **React** PWA Library, **Angular** PWA Framework, **Vue** PWA Framework, Ionic PWA Framework, Svelte, PWA Builder or Polymer.

	NATIVE	WEB	HYBRID	PROGRESSIVE
DESC	Built for a specific platform	Rely on a web browser and a working Internet connection to run	Combine the functionalities of native and web apps	Web apps designed to be faster, more lightweight, and borrow native app features
PROS	Faster speed Processing efficiency Smoother UI Hardware compatibility Access to the device's functionality	Cheaper to make No device memory or storage issues Easy maintenance Accessibility	Faster development Cross-platform compatibility Cost-effective Offline capability	No installation needed Data efficiency Versatility Automatic updates
CONS	Programming is not easy Takes time Multiple codebases for the same app	Browser dependent Useless without Internet Limited functionalities	Slower speed Limited hardware access Less smooth UI	Hardware integration issues Limited hardware access Browser UI issues

Ref: 4 - <https://www.bitstudios.com/blog/types-of-mobile-applications/>

## 2.5. COMPILED APPLICATIONS

These are technologies to develop native applications more than a type of mobile app. They aim to use only one programming language to generate native applications. The general idea is to work with a single technology and programming language and compile the code to be native on the different platforms.

Some of the technologies most used in this type of application are:

- **React Native and Native Script:** React is a framework created by Facebook, which uses the JavaScript language and the React library and allows to create interfaces based on its components. In these applications, the JavaScript code is executed in a separate execution thread, while the user interface elements are compiled to machine language.  
On the other hand, Native Script allows to create native applications using pure JavaScript or using other libraries such as Angular or Vue. It also comes with several pre-built components for user interfaces. Like React Native, they don't work with HTML.
- **Flutter:** It is a framework developed and maintained by Google, which uses the Dart programming language. The principle is that all code is written in Dart and compiled to native code that runs entirely on the device. So, Flutter compiles on ARM and generates C/C++ libraries, being closer to native code and therefore faster. The way Flutter works is to design user interfaces called widgets. Flutter already has several default widgets, such as buttons, navigation bars, etc.

## 3. KOTLIN LANGUAGE

### 3.1. INTRODUCTION

Kotlin is a programming language that was developed by JetBrains, a software company based in Russia. It was first announced in 2011 and has become a popular programming language in the world of software development, especially in Android mobile app development.

Kotlin is characterized as a modern and concise programming language that is designed to be interoperable with Java, which means that you can use it in projects that are already written in Java and vice versa. Some of the key features of Kotlin include:

- **Security:** Kotlin incorporates features that help prevent common errors in the code, such as those related to null safety. This reduces the likelihood of runtime errors related to null references.
- **Conciseness:** Kotlin allows you to write code more concisely compared to Java. This means you can do more with fewer lines of code, making the software easier to read.
- **Interoperability:** Kotlin can seamlessly interact with existing Java code and libraries. This means that you can gradually migrate a Java project to Kotlin without needing to rewrite all the code from scratch.
- **Functional programming:** Kotlin supports functional programming, which means you can write more declarative code and take advantage of concepts like higher-order functions, lambdas, and extension functions.
- **Object-oriented:** Kotlin is an object-oriented language, which means that you can use the concepts of object-oriented programming naturally.
- **Cross-platform support:** Kotlin can also be used for cross-platform development, meaning you can write code that runs on different platforms, such as Android, iOS, web ...

In short, Kotlin is a modern and versatile programming language that has gained popularity in the software development community due to its features and advantages over other languages. Its primary use is in Android application development.

#### 3.1.1. KOTLIN INSTALLATION

To install the Kotlin language, we will follow the following steps:

- We have to install the desired version of java, in this case openjdk-11-jdk

```
sudo apt install openjdk-11-jdk
```

- The next step is to install Kotlin

```
sudo snap install --classic kotlin
```

- Create a text editor to create a file called hello.kt

```
fun main() {  
  
    println("Hello World!")  
  
}
```

- Compile the file

```
kotlinc hello.kt -include-runtime -d hello.jar
```

- Run it

```
java -jar hello.jar
```

One can also start some Kotlin coding and testing using a Kotlin playground:

- <https://play.kotlinlang.org>
- <https://developer.android.com/training/kotlinplayground>

### 3.1.2. VARIABLES AND DATA TYPES

In Kotlin data types are classes, so we can access their properties and member functions. Some of these types, such as numbers, characters or logical values, may be represented internally as primitive values at run time, but all of this is transparent to the user. Remember that all primitive types in Java are also available in Kotlin.

To define values in Kotlin we use the reserved words *var* or *val*:

- We will use *var* to define mutable variables
- We will use *val* to define immutable variables, or what in Java would be constant values

In general, to have greater security and performance when working in several threads of execution, it is recommended to use constant values if we know that they won't be modified.

The following assignments would be correct:

```
val pi = 3.14 // Constant
val subject = "PMDM"
var x = 1
x = x + 1
```

If we look closely, we see that we have not defined the type of variables. Kotlin can infer the type of variables from the values with which we initialize them, so it is not necessary to indicate them explicitly. It will only be necessary to indicate the type of a variable if we do not give a value to the declaration. To indicate the type we will do:

```
var nameVariable : Type
```

---

## STRING TEMPLATES

String Templates or string literals can contain template expressions: fragments of code that will be evaluated and their result concatenated into the string. With this we can include values, variables or even expressions in a string.

Template expressions begin with the dollar sign \$ and consist of a variable name or an expression between keys { }.

```
val temp = 27

println("${temp}° ${ if (temp > 24) "Hot" else "Cold" })"
```

---

## NULLABLE TYPES AND ELVIS OPERATOR

Kotlin is a safe language, and among other things, it prevents us from programming errors such as *NullPointerException* since it does not allow variable values to be *null* by default.

If we want to specify that a variable can contain a *null* value, it is necessary to explicitly define it as nullable. To do this, when we define it, we add a question mark "?" to its type:

```
val name : String? = null
```

In addition, Kotlin also provides us with the "?:" operator, known as the Elvis operator, to specify an alternative value when the variable is *null*.

```
val name : String? = null
println(name?.length ?: -1)
```



### 3.1.3. OPERATORS

Now we will see the different operators that we can use in Kotlin, and their meaning.

---

#### ASSIGNMENT OPERATOR

It is used to give value to a variable.

```
n = 4
```

---

#### ARITHMETIC OPERATORS

They are used to perform arithmetic operations with variables. The result will be a numerical value. They may be:

- + (sum operator)
- - (subtraction operator)
- % (modulo operator)
- \* (multiplication operator)
- / (division operator)

```
5 + 3
```

```
5++    // increment
```

---

#### RELATIONAL OPERATORS

They are used to make comparisons between variables and return a logical value.

- == just like
- != different than
- < less than

- > greater than
- <= less than or equal
- >= greater than or equal

Additionally, with Kotlin, we can use:

- === It is the same object
- !== Not the same object

---

## LOGICAL OPERATORS

They are used to perform operations between logical type variables. Its result is also logical type.

- ! Negation
- || OR
- && AND

---

## TERNARY CONDITIONAL OPERATOR

value = condition ? expression\_1 : expression\_2

In Kotlin, a conditional statement is not a statement, but an expression, so it can be assigned directly to a variable.

---

### 3.1.4. CONTROL STRUCTURE

---

#### IF-ELSE

This structure is the same as the Java versions

---

#### WHEN

Switch does not exist in Kotlin, the most similar structure is the “When”. It can be expressed as a statement or as an expression, which we will see in the following example:

```
when (eValue) {  
    value1 -> if_value1
```

```

value2 -> if_value2

...

valueN -> if_valueN

else -> _default
}

var x = when (exprValue) {

    value1 -> value_for_1

    value2 -> value_for_2

    ...

    valueN -> value_for_n

    else -> default_value

}

```

We can also use the *when* operator without arguments (as if, then, else) and with the “is” and “in” operators

```

when {

    t < 15 -> println("COLD")

    t in 15..24 -> println("OK")

    t > 25 -> println("HOT")

}

```

```

when(month) {

    in 1..3 -> println("winter")

    in 4..6 -> println("spring")

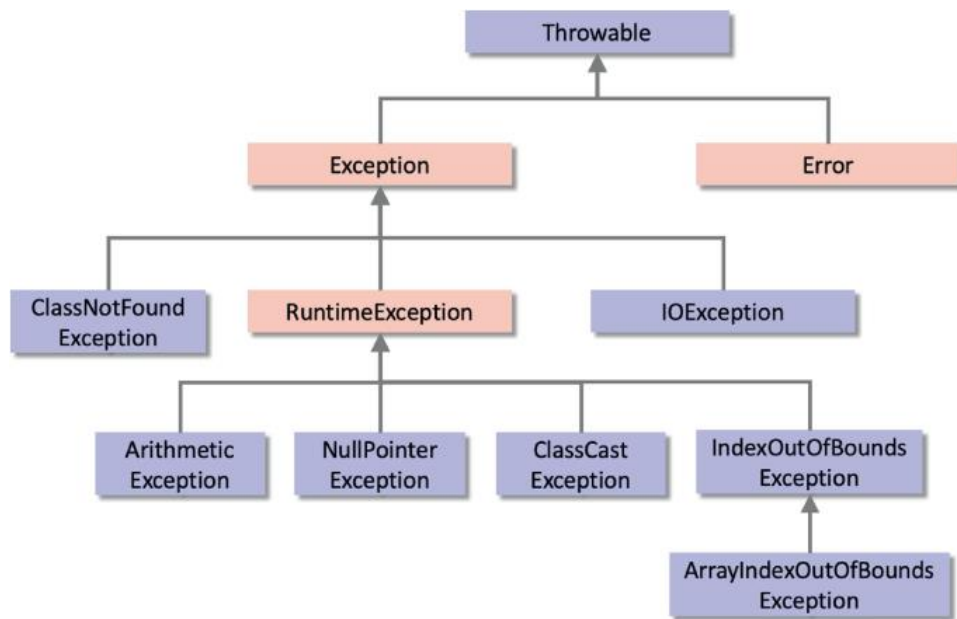
    in 7..9 -> println("summer")

    in 10..12 -> println("autumn")

}

```

### 3.1.5. EXCEPTIONS



Exceptions in Kotlin are only of the not-reviewed type:

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `NullPointerException` ...

On a practical level, this means that *throws* should not be included in function declarations where they may occur.

This does not mean that we cannot handle exceptions or *throw* exceptions in our code.

```
try {  
    // some code  
} catch (e: SomeException) {  
    // handler  
} finally {  
    // optional finally block  
}
```

```

1 fun foo() {
2     try {
3         throw Exception("Exception message")
4     } catch (e: Exception) {
5         println("Exception handled")
6     } finally {
7         println("inside finally block")
8     }
9 }

```

Ref: 5 - <https://code.tutsplus.com/es/kotlin-from-scratch-exception-handling--cms-29820t>

### 3.1.6. FUNCTIONS

#### DEFINITION AND INVOCATION

To declare a function in Kotlin we do:

```

fun funcName(param1 : Type1, param2 : Type2...) : ReturnType {

    // function body

    return

}

```

We look at some features of function declarations:

- They are declared using the keyword *fun*
- Names start with lower case and are expressed in camelCase
- Function parameters are specified after the name, in parentheses, and in the form *parameter : Type*. These types must necessarily be specified
- The return type of the function may be specified after the parenthesis with the argument list, followed by “:” .
- When the function does not return any significant value, its default return type is *Unit*, which would be the equivalent of *void* in Java or C.

#### LAMBDA EXPRESSIONS BASICS

A lambda expression represents the block of a function and simplifies the code.

Characteristics of lambda expressions:

- It is expressed in { }
- It does not have the keyword *fun*

- It has no access modifiers (*private*, *public* or *protected*) as it does not belong to any class
- It is an anonymous function, which does not have a name
- It does not specify the return type, as it is inferred by the compiler
- The last expression in a lambda is considered the return value
- Parameters are not enclosed in parentheses
- In addition, we can assign a lambda expression to a variable and execute it

In this section, we will look at some ways to create lambda expressions:

- Lambda expression without parameters and assigned to a variable

```
val msg = { println("Hi! I'm a lambda function") }

msg()
```

- Lambda expression with parameters

```
val msg = { text : String -> println(text) }

msg("Hi Kotlin!")

msg("Good morning!")

val writeSum = { s1: Int, s2: Int ->

    println("Let's add $s1 y $s2")

    val result = s1 + s2

    println("The Sum is: $result")

}

writeSum(3,2)
```

- Lambda expression omitting parameters: use of *it* when only one param

```
val coins : (Int) -> String = { quantity ->
```

```

        "$quantity quarters"
    }

    println(coins(3))    // 3 quarters

    val coins : (Int) -> String = { "$it quarters" }

    println(coins(3))    // 3 quarters

```

---

## ANONYMOUS FUNCTIONS

In Kotlin, anonymous functions can be created using the *fun* keyword in a similar way to how regular functions are defined, but without giving them a name. These functions can be assigned to variables or passed as arguments to other functions. They are often used in Kotlin to implement functional interfaces, such as *Runnable* or *OnClickListener*. Here are some examples of simple anonymous functions with arguments in Kotlin:

```

val sum = fun(x : Int, y : Int) : Int {

    return x + y

}

println(sum(5, 3))    // Prints "8"


fun calculate(a : Int, b : Int, operation : (Int, Int) -> Int ) : Int {

    return operation(a, b)

}

val sum = calculate(10, 5, fun(x : Int, y : Int) : Int { return x + y} )

val diff = calculate(10, 5, fun(x : Int, y : Int): Int { return x - y} )

println("SUM: $sum")    // Prints "SUM: 15"

println("DIFF: $diff ")    // Prints "DIFF: 5"

```

### 3.1.7. OOP

Object-Oriented Programming (OOP) in Kotlin is one of the language's key features, as it revolves around objects and classes for organizing and structuring code. Find below the fundamental concepts of OOP in Kotlin.

---

#### CLASSES AND OBJECTS

```
class Person(val name : String, val age : Int) {  
  
    fun greet() {  
  
        println("Hello, I'm $name and I'm $age years old.")  
  
    }  
  
}  
  
fun main() {  
  
    val person1 = Person("John", 30)  
  
    val person2 = Person("Mary", 25)  
  
    person1.greet()  
  
    person2.greet()  
  
}
```

In this example, we've defined a *Person* class with two properties (name and age) and a *greet* method. Then, we create two objects of the *Person* class and call the *greet* method on each of them.

---

#### INHERITANCE

Kotlin supports inheritance, which means you can create a new class based on an existing class. Kotlin classes and their functions are final by default (in Java are open by default). To allow a class to be extended it must be marked open. To allow class functions and fields to be overridden, they must also be marked open.

Here's an example of inheritance:



```
open class Animal(val name : String) {  
    open fun makeSound() {  
        println("$name makes a sound.")  
    }  
}  
  
class Dog(name : String) : Animal(name) {  
    override fun makeSound() {  
        println("$name barks.")  
    }  
}  
  
class Cat(name : String) : Animal(name) {  
    override fun makeSound() {  
        println("$name meows.")  
    }  
}  
  
fun main() {  
    val dog = Dog("Max")  
    val cat = Cat("Whiskers")  
    dog.makeSound()  
    cat.makeSound()  
}
```

In this example, we've defined a base class `Animal` with a *makeSound* method, and then we've created two derived classes (`Dog` and `Cat`) that inherit from `Animal`. Each derived class provides its own implementation of the *makeSound* method.

---

## ENCAPSULATION

Kotlin allows you to control access to a class's properties and methods using access modifiers like *private*, *protected*, *internal* (module) and *public* (default). This makes it easy to implement the encapsulation principle (<https://kotlinlang.org/docs/visibility-modifiers.html>)

```
class BankAccount(private var balance : Double) {
    fun deposit(amount : Double) {
        if (amount > 0) {
            balance += amount
        }
    }

    fun withdraw(amount : Double) {
        if (amount > 0 && balance >= amount) {
            balance -= amount
        }
    }

    fun getBalance() : Double {
        return balance
    }
}

fun main() {
    val account = BankAccount(1000.0)
    account.deposit(500.0)
    account.withdraw(200.0)
    println("Current balance: ${account.getBalance()}")
}
```

In this example, the `balance` property is declared as *private*, meaning it can only be accessed from within the *BankAccount* class. The *deposit*, *withdraw* and *getBalance* methods are used to interact with the `balance` property.

These are the basic concepts of Object-Oriented Programming in Kotlin. You can create classes, objects, apply inheritance and control access to properties and methods to build modular and maintainable code.

---

## USE OF GENERICS

Kotlin allows the use of type parameters in classes definition.

Definition:

```
class class name < generic data type > (  
    val property name : generic data type  
)  
  
class Question<T>(  
    val questionText: String,  
    val answer: T,  
    val difficulty: String  
)
```

Use:

```
val instance name = class name < generic data type > ( parameters )  
  
fun main() {  
    val q1 = Question<String>("Capital of China is __", "Beijing", "medium")  
    val q2 = Question<Boolean>("The sky is green. True or false", false, "easy")  
    val q3 = Question<Int>("How many days are in July?", 31, "easy")  
}
```

Ref: 6 - <https://kotlinlang.org/docs/generics.html>

---

## ENUM CLASSES

The *enum* classes are used to prevent programmers and users wrong typing. Using *enum* classes force a given set of values to be the only accepted ones (type-safe).

Definition:

```
enum class enum name {  
    Case 1 , Case 2 , Case 3  
}
```

Use:

```
enum name . case name
```

```

class Question <T> (
    val questionText : String,
    val answer : T,
    val difficulty: Difficulty
)
enum class Difficulty{
    EASY, MEDIUM, HARD
}

fun main() {
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)
}

```

Ref: 7 - <https://kotlinlang.org/docs/enum-classes.html>

---

## SINGLETON OBJECTS

There are many scenarios where you want a class to only have one instance. For example:

- Player stats in a mobile game for the current user
- Interacting with a single hardware device, like sending audio through a speaker
- Authentication, where only one user should be logged in at a time

In the above scenarios, you'd probably need to use a class. However, you'll only ever need to instantiate one instance of that class --> Singleton Object

A singleton object can't have a constructor as you can't create instances directly. Instead, all the properties are defined within the curly braces and are given an initial value.

```

object object name {
    

class body 1


}

```

```

data class Question<T>{
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
}
enum class Difficulty {
    EASY, MEDIUM, HARD
}
object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}

fun main() {
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)
    StudentProgress.answered = StudentProgress.answered + 1
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
}

```

Ref: 8 - <https://kotlinlang.org/docs/object-declarations.html#object-declarations-overview>

### 3.1.8. COMPLEX DATA TYPES

In Kotlin, complex data types are often implemented using classes, data classes, or collections (such as *lists*, *sets*, and *maps*) to represent structured data. Here are some examples of complex data types in Kotlin:

---

#### CLASSES AND DATA CLASSES

Classes are used to create custom complex data types with properties and methods. Data classes, on the other hand, are a special type of class designed primarily for holding data. They automatically provide useful functions like `toString()`, `equals()`, and `hashCode()` based on their properties. A data class needs to have at least one parameter in its constructor.

```

data class Person(val name : String, val age : Int)

val person = Person("Alice", 30)

println(person)      // Output: Person(name=Alice, age=30)

```

---

## LIST / MUTABLELIST

Lists are ordered collections that can store elements of the same or different types. You can create lists of complex data types. *List* is an interface that defines properties and methods related to a read-only ordered collection of items. *MutableList* extends the *List* interface by defining methods to modify a list, such as adding and removing elements.

```
val fruits = listOf("Apple", "Banana", "Cherry")

val people = listOf(Person("Alice", 30), Person("Bob", 25))

val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars")

println(solarSystem.size)

println(solarSystem[2])

println(solarSystem.get(2))

println(solarSystem.indexOf("Earth"))


val solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars")

solarSystem.add("Pluto")

solarSystem.add(3, "Theia")

solarSystem[3] = "Future Moon"

solarSystem.removeAt(9)

solarSystem.remove("Future Moon")

println(solarSystem.contains("Pluto"))

println("Future Moon" in solarSystem)
```

Lists are collections easy to iterate using a *for* loop.

```
for ( element name in collection name ) {  
    body  
}
```

```
for (planet in solarSystem) {  
    println(planet)  
}
```

<https://kotlinlang.org/docs/collections-overview.html>

---

## SET / MUTABLESET

A set is a collection that does not have a specific order and does not allow duplicate values. The secret is a *hash code*. A hash code is an *Int* produced by the *hashCode()* method of any Kotlin class. A small change to the object, such as adding one character to a String, results in a vastly different hash value.

Searching for a specific element in a set is fast—compared with lists—especially for large collections but Sets tend to use more memory than lists for the same amount of data.

```
val uniquePeople = setOf(Person("Alice", 30), Person("Bob", 25), Person("Alice", 30))  
  
println(uniquePeople)           // [Person(name=Alice, age=30), Person(name=Bob, age=25)]  
  
val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "Mars")  
  
println(solarSystem.size)  
  
solarSystem.add("Pluto")  
  
println(solarSystem.contains("Pluto"))    // "Pluto" in solarSystem is equivalent  
  
solarSystem.remove("Pluto")
```

---

## MAP / MUTABLEMAP

Maps associate keys with values, allowing you to create complex data structures to represent relationships or configurations. It's called a map because unique keys are mapped to other values.

```
val map name = mapOf(  
    key to value ,  
    key to value ,  
    key to value ,  
)  
mutableMapOf<key type , value type > ()
```

```
val ages = mutableMapOf("Alice" to 30, "Bob" to 25)  
  
println(ages) // {Alice=30, Bob=25}  
  
println(ages.get("Bob"))  
  
ages.remove("Alice")  
  
val peopleMap = mapOf(1 to Person("Alice", 30), 2 to Person("Bob", 25))  
  
println(peopleMap) // {1=Person(name=Alice, age=30), 2=Person(name=Bob, age=25)}
```

---

## ARRAYS

Arrays are fixed-size collections that can store elements of the same type. The data type is optional as it can be inferred.

```
val variable name = arrayOf<data type> ( element1 , element2 , ... )
```

```
val numbers = arrayOf(1, 2, 3, 4, 5)
```

---

## HIGH-ORDER FUNCTIONS WITH COLLECTIONS

A higher-order function is a function that takes functions as parameters or returns a function.

### FOREACH()

The `forEach()` function can be combined with string templates and lambdas to iterate along a collection and perform actions on each element of the collection.



```

class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Chocolate Chip",
        softBaked = false,
        hasFilling = false,
        price = 1.69
    ),
    Cookie(
        name = "Banana Walnut",
        softBaked = true,
        hasFilling = false,
        price = 1.49
    ), ...
)

```

```

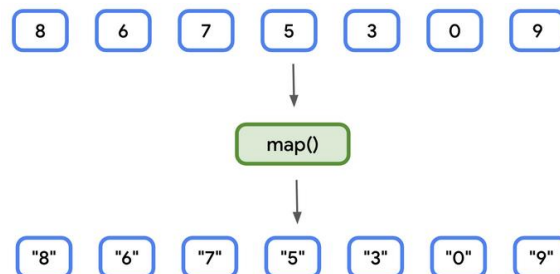
fun main() {
    cookies.forEach {
        println("Menu item: ${it.name}")
    }
}

```

Ref: 9 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/for-each.html>

## MAP()

The `map()` function lets you transform a collection into a new collection with the same number of elements while adding some transformation.



Ref: 10 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html>

```

fun main() {
    val fullMenu = cookies.map {
        "${it.name} - ${it.price}"
    }
    println("Full menu:")
    fullMenu.forEach {
        println(it)
    }
}

```

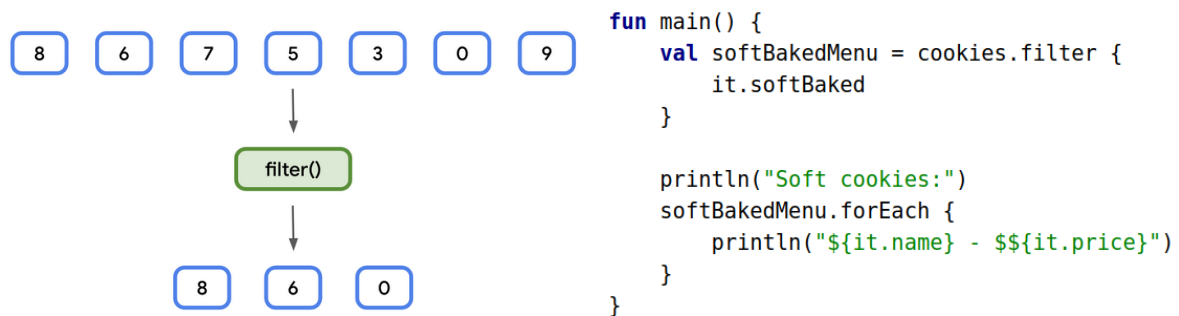
```

Full menu:
Chocolate Chip - $1.69
Banana Walnut - $1.49
Vanilla Creme - $1.59

```

## FILTER()

The `filter()` function lets you create a subset of a collection. The lambda has a single parameter representing each item in the collection and returns a Boolean value.



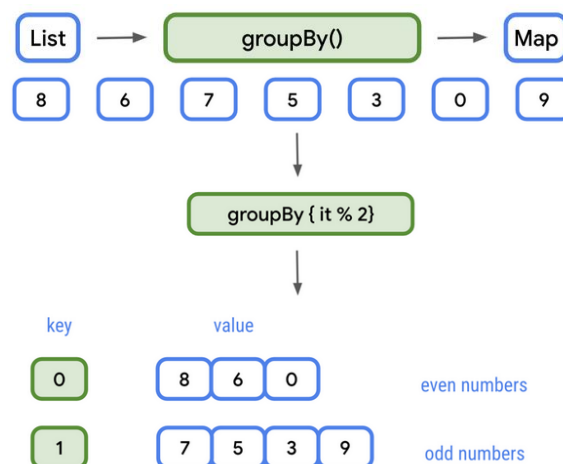
Ref: 11 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html>

For each item in the collection:

- If the result of the lambda expression is true, then the item is included in the new collection.
- If the result is false, the item is not included in the new collection

## GROUPBY()

The `groupBy()` function can be used to turn a *list* into a *map*, based on a function. Each unique return value of the function becomes a *key* in the resulting map. The values for each *key* are all the items in the collection that produced that unique return value.



Ref: 12 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html>

```

fun main() {
    val groupedMenu = cookies.groupBy {it.softBaked}
    val softBakedMenu = groupedMenu[true] ?: emptyList()
    val crunchyMenu = groupedMenu[false] ?: emptyList()

    println("Soft cookies:")
    softBakedMenu.forEach {
        println("${it.name} - ${it.price}")
    }
    println("Crunchy cookies:")
    crunchyMenu.forEach {
        println("${it.name} - ${it.price}")
    }
}

```

---

## FOLD()

Used to generate a single value from a collection.

The fold() function takes two parameters:

- An initial value
- A lambda expression that returns a value with the same type as the initial value

The lambda expression additionally has two parameters:

- Accumulator: Each time the lambda expression is called, the accumulator is equal to the return value from the previous time the lambda was called
- The second is the same type as each element in the collection

Initial value of accumulator

```

val totalPrice = cookies.fold(0.0) {total, cookie ->
    total + cookie.price
}
println("Total price: ${totalPrice}")

```

Accumulator

total = total + cookie.price  
 return total

Inferred

Ref: 13 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections.fold.html>

## SORTEDBY()

Lets you specify a lambda that returns the property you'd like to sort by.

As far as the data type of the value has a natural sort order—strings are sorted alphabetically, numeric values are sorted in ascending order—it will be sorted just like a collection of that type.

```
val alphabeticalMenu = cookies.sortedBy {  
    it.name  
}  
println("Alphabetical menu:")  
alphabeticalMenu.forEach {  
    println(it.name)  
}
```

Ref: 14 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by.html>

These examples demonstrate various ways you can work with complex data types in Kotlin. Depending on your application's needs, you can use classes, data classes, lists, sets, maps, or arrays to represent and manipulate structured data effectively.