Jonathon Meney

Professor Cezar Campeanu

Comparative Programming Languages

30 November 2022

Kotlin, a Descendant of Java and Other Java Alternatives

Introduction

Kotlin is a relatively new language but is nonetheless a fast-growing one. Released originally in 2011 it has been around for 11 years. During those 11 years, the language has seen a plethora of updates and had a few major milestones. This report will aim to go further in depth into the language's history, accompanied by an overview of the language itself. A description of Kotlin's features, and an evaluation of Kotlin will also follow. At the end of this report also, will be examples of Kotlin programs.

Historical Development

Kotlin has been around for just over 11 years, releasing in early July 2011. Since then, it has seen numerous bug fixes and features implemented, and has had a few major lifetime milestones. Kotlin was originally created by JetBrains, a company known for its easy to use, language specific, integrated development environments (IDEs for short). Their original desire to build a new language began when the CEO asked his employee's: "Guys, what do you think JetBrains can do in terms of big things that would benefit the community and be noticeable" (Naik).

Since developers at JetBrains created multiple IDEs in many languages, they had lots of exposure to different languages. As a result, they understood the advantages and disadvantages of each language to some degree. This is what began the Kotlin project. Their first goal was to create a language which would combine the things people enjoyed in different languages and leave the things they did not enjoy behind (Naik). The developers at JetBrains were also looking for Java alternatives so they could leave behind the limitations that Java had underlying in its design. Since many of their projects were written in Java already, they needed to find a language that would still work alongside their existing Java code (Bogode). One of the languages that worked quite well for this problem was Scala, but it still had some limitations they did not like. As a result, they set out to improve what Scala was doing on the Java Virtual Machine (JVM for short) at the time and to improve what Java already was (Naik). This is also why Kotlin is written so much like Java code.

The biggest milestone for Kotlin was the announcement that Google "will officially support Kotlin on Android as a 'first-class' language" (Miller). This came because JetBrains was also developing and maintaining Android Studio, the major IDE for Android development, which is used by Google. Prior to this announcement android applications were written in only Java with a few slight variations. Android uses a variation of the JVM, but is still almost the same, and since it is so close, Kotlin can operate side-by-side with the existing Java code that applications are already written in. This makes it easy for any Java developer to easily make the switch to Kotlin.

Today Kotlin is still gaining popularity as many Android Java developers are making the switch. The language still receives regular updates and is open source for the community to help build. Additionally, it was reported in 2021 that: “Kotlin has been used by 4,800,000+ developers for server-side, mobile multi-platform, Android, and front-end development. In addition to this, there are about 194 Kotlin user groups worldwide, and 45 of the top-200 universities are teaching Kotlin” (Naik). It is also estimated that about 80% of android apps are already using Kotlin in their source code (Naik). In short, Kotlin is only just getting started in its journey to be a better Java.

Language Overview

In general, “Kotlin is an open-source, statically-typed programming language that supports both object-oriented and functional programming” (Android Developers). Kotlin’s primary purpose was to improve languages that existed on the JVM, and it did just that. Now, as mentioned, Kotlin is the first language used by all new Android applications (Miller). Kotlin’s biggest advantage is also its ability to be 100% compatible with all existing Java code. This means that Android applications that already exist in Java can start using Kotlin at any time, one of its main design goals when creating Kotlin. Since this is the case, users of Kotlin can utilize the vast libraries in Java to their benefit (Garg).

Another major design goal for the developers of Kotlin was to improve the existing implementation of Java and other Java alternatives. As a result of this improvement, Kotlin code is much simpler than its Java counterpart. It was also estimated that Kotlin programs would be at least 20% less code than if the program were written in Java (Garg). Due to this reduction of coding complexity, it allows programmers who use Kotlin to build applications quicker and with much less boilerplate code.

Another important part of Kotlin to mention is that it has null safety. What this means is that Kotlin’s standard types cannot hold null values. In the event they do they will be caught at compile time. Kotlin does, however, allow standard types to hold null values if they are initialized with the safety operator, denoted by a question mark (see Appendix A, Figure 13). Kotlin provides further null safety by using safe calls when accessing object properties or methods that may return null. Functions can also be marked with the safety operator if the function may return null (see Appendix A, Figure 14). Variables, calls, or functions that hold or return null must all be handled with the safety operator, otherwise, the “Kotlin system refuses to compile code that tries” (Garg).

Lastly, due to its vast similarity to Java, Kotlin is a quite easy to learn language, especially for those already using Java. This was also another design goal for the language. It was intended that anybody could spend a few hours reading the documentation and be able to read and write their own programs (Garg). As with any language, understanding its more in-depth features will of course take more time.

Language Features

General Syntax

The syntax of Kotlin is much like Java. It uses curly braces for blocks of code, but many keywords are different, and semicolons are optional at the end of statements. A general program also follows a basic ordering of its content. The top of the file must begin with any package definitions or imports. Package definitions and imports use the keywords, package, and import, respectively (See Appendix A, Figure 2) (Kotlin Foundation).

After package definitions and imports, is class and function definitions. If no classes or functions are defined, then next will be the main function (See Appendix A, Figure 9). The main function serves as the entry point for a Kotlin program. The main function can also optionally have a parameter which can accept a list of command line arguments. For the typical “Hello World” program that is written in the main function, one can use the *print* or *println* function (see Appendix A, Figure 9) (Kotlin Foundation).

The typical function definition begins with the *fun* keyword. It is then followed by a name and a set of parentheses. These parentheses contain the parameter list; each parameter must also be followed by a type annotation. After the parameter list is the return type of the function. Following the return type, is a code block surrounded by braces, or an equals sign followed by what it returns (see Appendix A, Figure 11). Depending on the form of the function definition, the return type may be left out (Kotlin Foundation).

As mentioned, Kotlin is also object-oriented supporting class definitions. Classes are marked with the *class* keyword (see Appendix A, Figure 12). If the class is inheritable, it also needs to be preceded by the *open* keyword (see Appendix A, Figure 12). A class can inherit another class by writing the class definition followed by a colon and the name of the class being inherited. If the class is defined with parameters in its definition, they become the parameters for the default constructor. Their values can be accessed by their parameter name anywhere in the class's scope. Kotlin classes also support additional secondary constructors (Kotlin Foundation).

One of the most basic things in any programming language is variables. Variables are quite simple in Kotlin. They come in only two forms, constant and non-constant. Constants are defined with the *val* keyword, and non-constants are defined with the *var* keyword. Variables can also be declared and initialized, or only declared for assignment later (see Appendix A, Figure 3). If a variable is declared but not initialized, it must be declared with a type annotation. Variables that are declared and initialized can optionally have a type annotation, as the type can be inferred (Kotlin Foundation).

The Kotlin compiler will not allow the null type to be held by a variable or be returned by a function under regular circumstances. To have a variable hold the null type, or for a function to return the null type, the variable type or return type must be marked with question mark at the end of the type (see Appendix A, Figure 13) (Kotlin Foundation).

Lastly, comments come in two forms. Single-line and multi-line, or block, comments. A single-line comment begins with two forward slashes followed by the comment. Multi-line comments begin with a forward slash and then an asterisk and are closed by an asterisk and then a forward slash. Kotlin also has its own version of Javadoc called KDoc for documenting methods (see Appendix A, Figure 1) (Kotlin Foundation).

Data Types

Kotlin supports a wide range of primitive types that can be used. For numerical types, *Byte*, *Short*, *Int*, *Long*, *Float*, and *Double* are the available types. Unsigned integers can also be defined using different variations of the previous types. *Boolean* is also a primitive type in Kotlin for holding logical values. Kotlin also has the primitive types of *Char* (character) and *String*. Kotlin also defines arrays as a type. Arrays in Kotlin can hold any type, but arrays can also be of a specific type. These include, *ByteArray, ShortArray, IntArray, LongArray, FloatArray, DoubleArray,* and *CharArray*. Integer arrays also have an unsigned version. Types can be checked against with the *is* keyword and can be cast with the *as* keyword (see Appendix A, Figure 16) (Kotlin Foundation). For a full list of the types available in Kotlin see Appendix B, Table 1.

Primitive Operations

Kotlin provides by default thirty-three primary operators. This includes all the mathematical operators, augmented assignment operators, and increment operators. Logical operators, such as *and, or,* and *not,* are all included, and relational operators are available as well. Null-based operators are also included. The method reference operator from Java is also available. Kotlin also has a special operator for defining ranges as well (Kotlin Foundation). There are also more operators than listed above but they are too vast and complex to explain here. An example of each operator listed above and those not listed can be found in Appendix B, Table 2.

Sequence Control

The control flow mechanisms that can be found in Kotlin are much like others found across many different languages. The standard *if* statement is available, but it may take the form of an expression instead of a block of code to execute (see Appendix A, Figure 4). Kotlin also has a *when* statement (see Appendix A, Figure 5). The *when* statement is like the C family's *switch* statement. It considers some value and does something if it matches a *case* in the when code block. The *when* statement must also contain a mandatory *else* clause to default to if all cases fail (Kotlin Foundation).

Kotlin also supports *for* and *while* loops. *For* loops work as *foreach* loops in other languages, using an iterator to loop over code (see Appendix, Figure 6). *While* loops come in two different forms a standard *while*, and a *do-while*. A *do-while*, compared to a regular *while*, will execute its looping code block once and then check its looping condition. A regular *while* loop will check its condition first before ever running its looping code block (see Appendix, Figure 7).

Lastly, Kotlin offers three simple control statements that can be used in conjunction with if statements and loops. These three control statements are denoted by the keywords, *break, continue,* and *return. Break* and *continue* can be used with loops, to break out of loop early, or begin the next loop iteration, respectively. Kotlin also supports labeled loops which give more control over where a *break* or *continue* statement can break out of or continue from (see Appendix A, Figure 8). The *return* keyword is also like implementations in other languages allowing a function call to end early and return a value at the point of a *return* statement (see Appendix A, Figure 9) (Kotlin Foundation).

Programming Environment

Furthermore, the available environments for programming in Kotlin are IntelliJ and Android Studio, both of which are also made by JetBrains, the creators of Kotlin. Each of these is quite commonly used, though more so Android Studio, as it is also used for making Android applications. IntelliJ is still used for Kotlin development, however, but large-scale non-Android Kotlin projects are rarely implemented (Kotlin Foundation).

Evaluation

In an evaluation of Kotlin, it can be said by many that it is a super easy to learn and use language. "Kotlin is a great fit for server-side development" (Zubchenko), but as mentioned the larger of the two languages used on Android. Kotlin's support for both object-oriented programming and functional programming also improves its overall usability. Kotlin's object-oriented programming design also makes it possible to cut down many of the lines that would need to be written for the same class in Java (Zubchenko).

Kotlin's simplicity also makes it an easy to read and write language. Kotlin's simple syntax makes it easy to read and understand blocks of code. The language also has a small set of reserved and restricted words which reduces the complexity and learning curve of the language. Since Kotlin is also statically typed it makes understanding what type a variable will be much easier. There is also a wide variety of types that increase the readability further. Kotlin also offers many shorthand notations which improves the writability of the language.

Kotlin is also an exceptionally reliable language, its null safety, and its type checks make this so. "Compared to Java, variables in Kotlin cannot contain null if the compiler does not know about it. In fact, while declaring a variable you have to specifically indicate whether it can be nullable or not" (Zubchenko). This null safety allows the programmer to not have to deal with null pointer exceptions, and in turn fewer cases need to be managed appropriately. Kotlin does, however, have methods of dealing with errors and exceptions, so if null pointer exceptions are possible then they can be handled accordingly (see Appendix A, Figure 15). The use of type checks also improves the reliability of the language greatly as it gives protection making sure data of a given type is passed around properly.

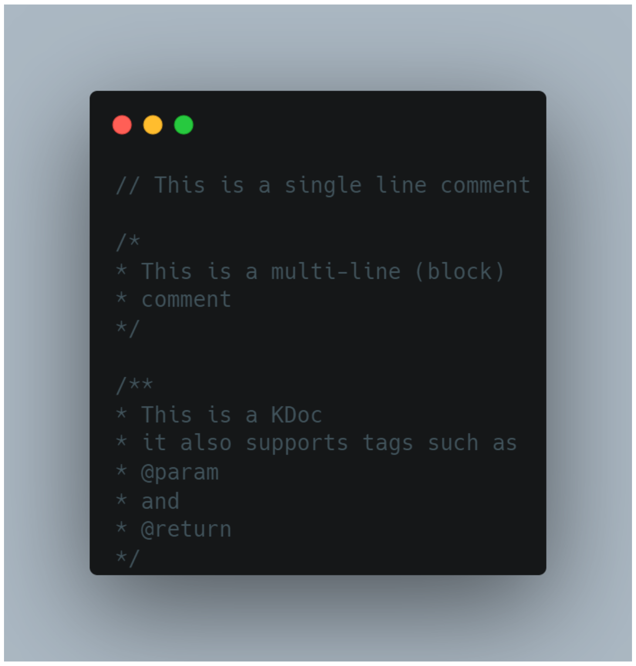
Overall, the Kotlin language is well designed. Its light syntax makes it easily readable as well as easily writable. Its support for multiple programming paradigms also makes it a well-designed language, as it can serve a range of purposes. Lastly, Kotlin's null safe features and type checking make it reliable and allow the developer to quickly find bugs and fix them. The exception handling that exists also makes it even more reliable, allowing the program to handle errors appropriately and continue running.

Conclusion

To conclude, Kotlin may be a newer language in the grand scheme of the programming world, but it has already solidified itself in the mainstream. Google's official support has been the driving force behind its growing use, and Android has been its catalyst. Kotlin's design goals, and its features, make it an easy to read and write language, with an easy learning curve. Finally, an evaluation of Kotlin shows that it is a well-designed language through its readability, writability, and reliability, and its ability to be a general-purpose language.

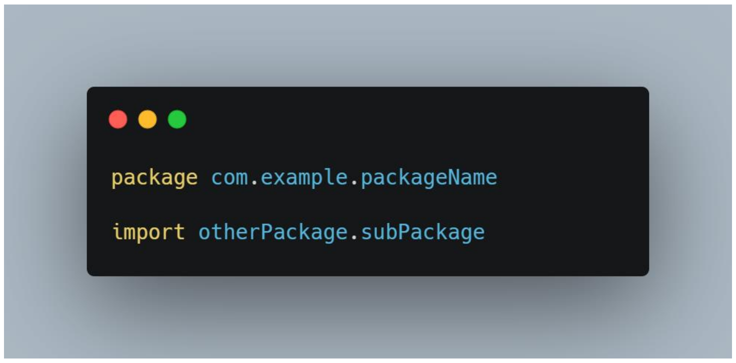
Appendix A:

Example Programs



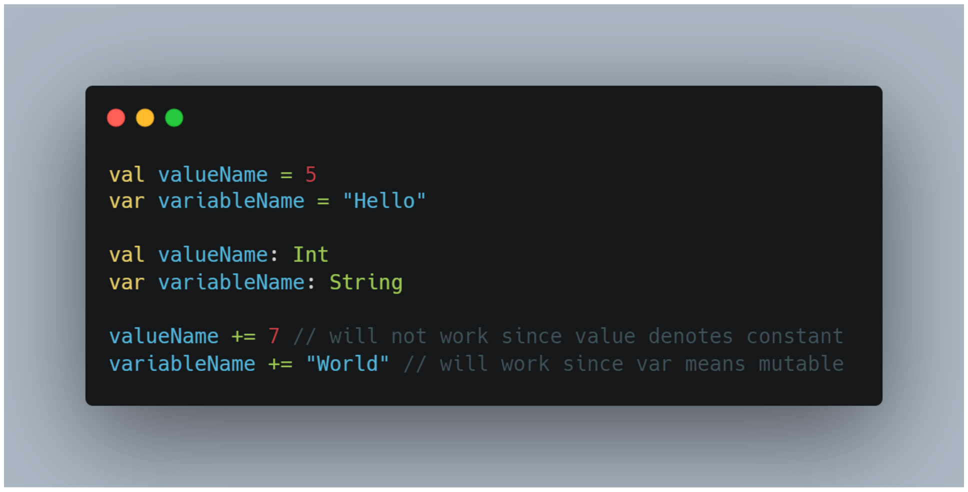
```kotlin
// This is a single line comment

/*
* This is a multi-line (block)
* comment
*/

/**
* This is a KDoc
* it also supports tags such as
* @param
* and
* @return
*/
```

Figure 1. An example of all types of comments.



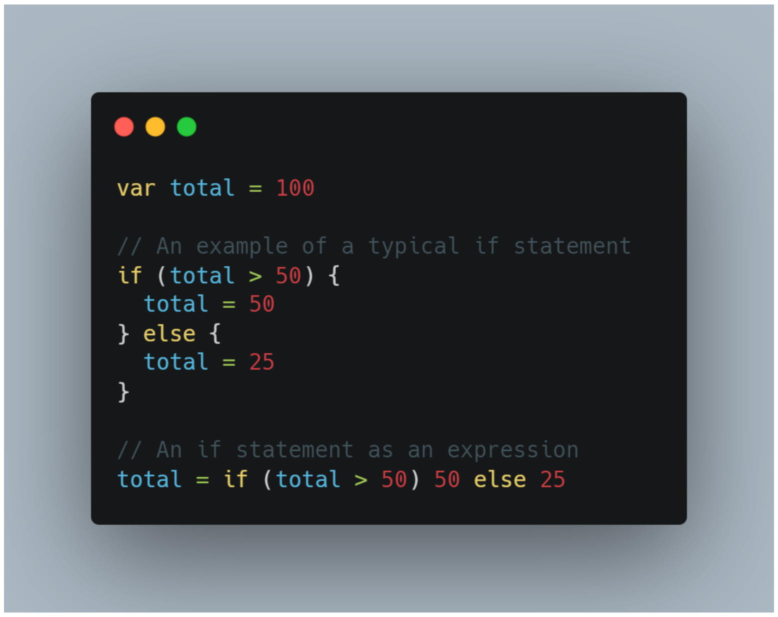
```kotlin
package com.example.packageName

import otherPackage.subPackage
```

Figure 2. An example of declaring the package a file is in, and importing another package



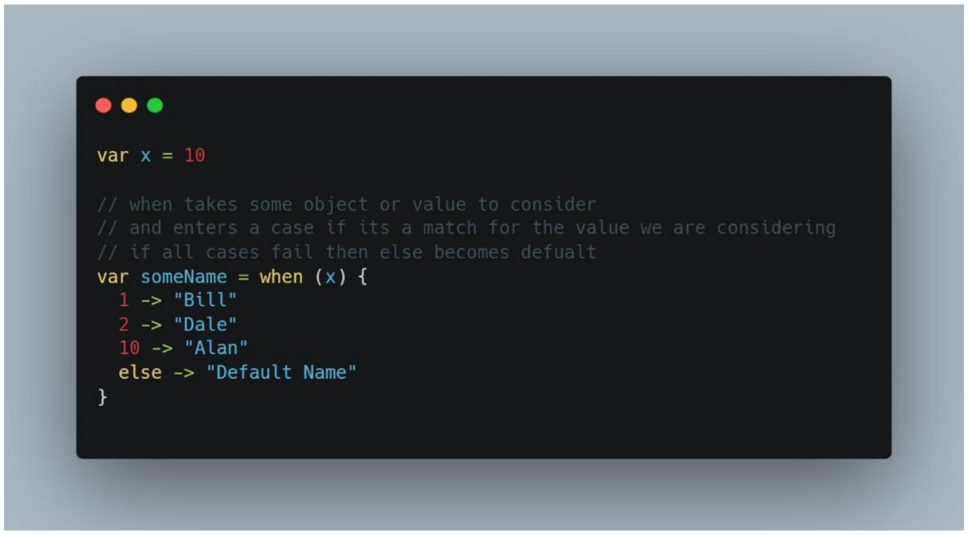
```kotlin
val valueName = 5
var variableName = "Hello"

val valueName: Int
var variableName: String

valueName += 7 // will not work since value denotes constant
variableName += "World" // will work since var means mutable
```

Figure 3. An example of *val* and *var* for variable declaration.



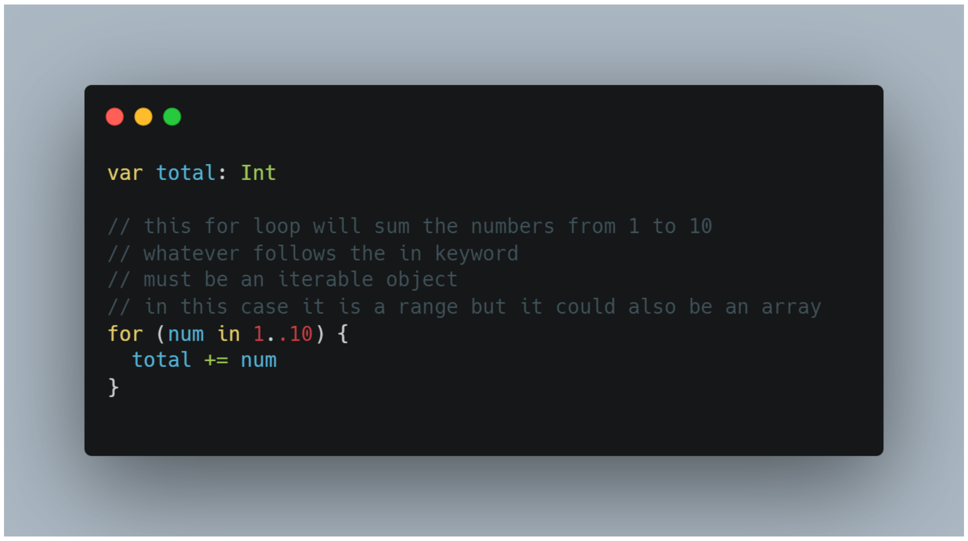
```
var total = 100

// An example of a typical if statement
if (total > 50) {
  total = 50
} else {
  total = 25
}

// An if statement as an expression
total = if (total > 50) 50 else 25
```

Figure 4. An example of both forms of the *if* statement.



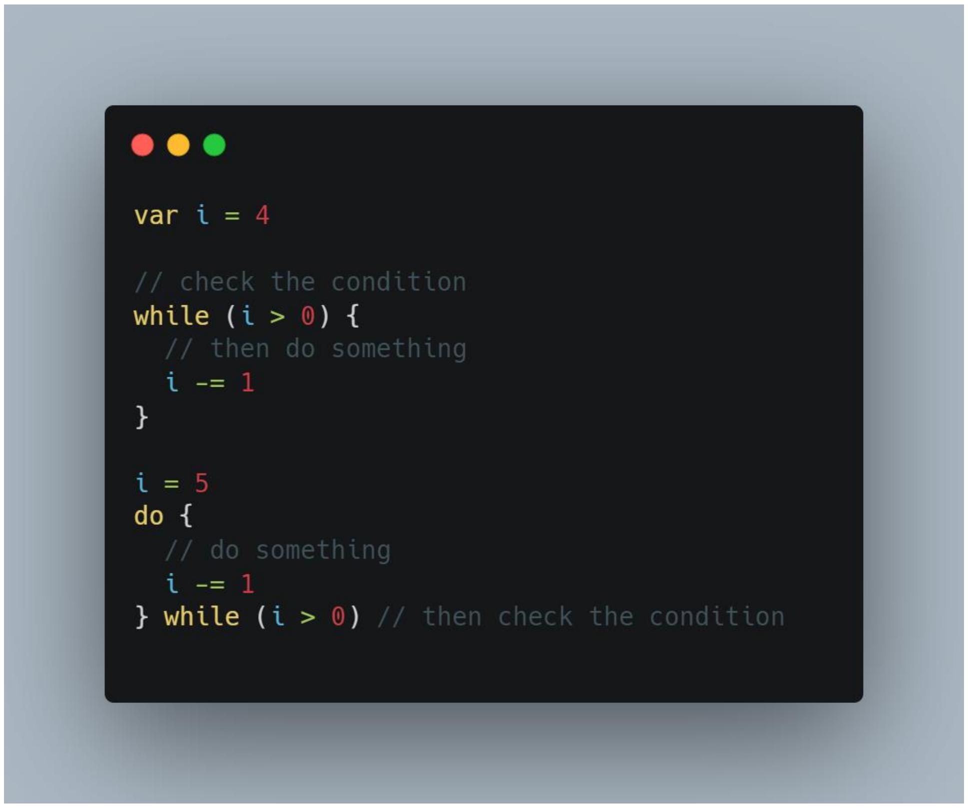
```
var x = 10

// when takes some object or value to consider
// and enters a case if its a match for the value we are considering
// if all cases fail then else becomes defualt
var someName = when (x) {
  1 -> "Bill"
  2 -> "Dale"
  10 -> "Alan"
  else -> "Default Name"
}
```

Figure 5. An example of the *when* statement.



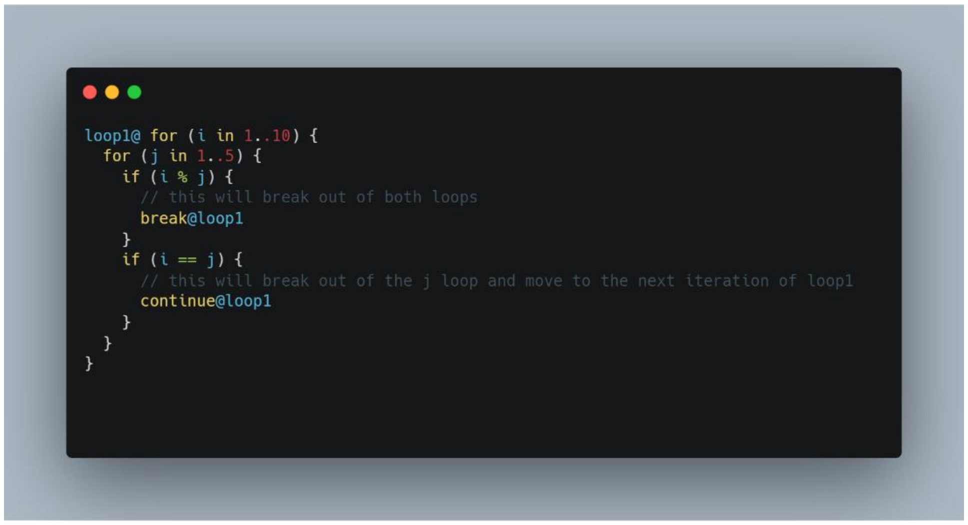
```
var total: Int

// this for loop will sum the numbers from 1 to 10
// whatever follows the in keyword
// must be an iterable object
// in this case it is a range but it could also be an array
for (num in 1..10) {
  total += num
}
```

Figure 6. An example of a *for* loop.



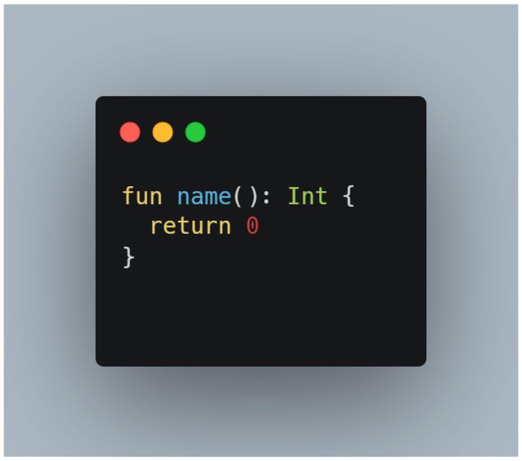
```
var i = 4

// check the condition
while (i > 0) {
  // then do something
  i -= 1
}

i = 5
do {
  // do something
  i -= 1
} while (i > 0) // then check the condition
```

Figure 7. An example of a *while* loop, and a *do-while* loop.



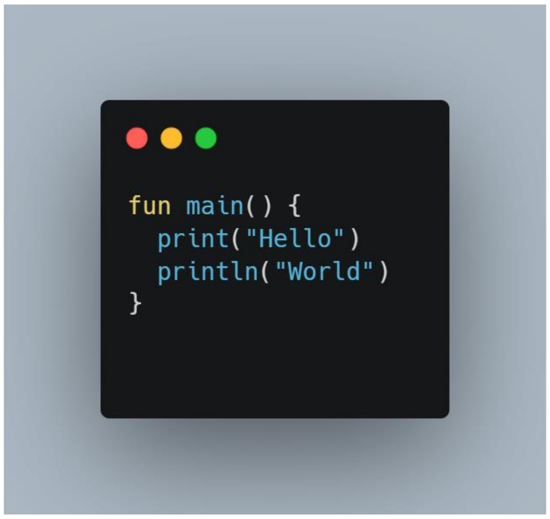
```kotlin
loop1@ for (i in 1..10) {
  for (j in 1..5) {
    if (i % j) {
      // this will break out of both loops
      break@loop1
    }
    if (i == j) {
      // this will break out of the j loop and move to the next iteration of loop1
      continue@loop1
    }
  }
}
```

Figure 8. An example of the *break* and *continue* statements. It also shows labeled loops.



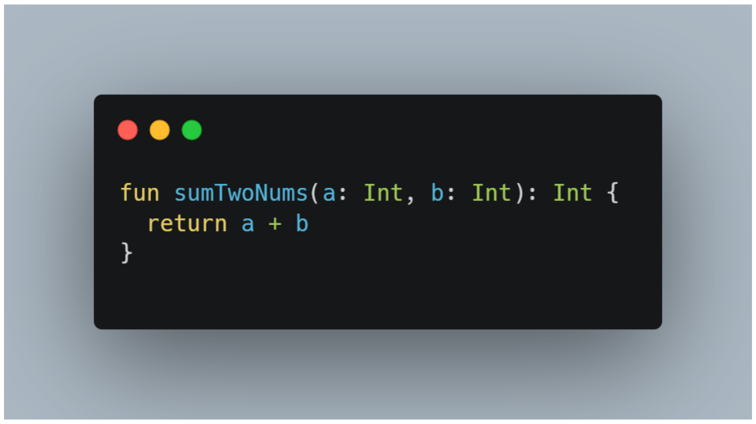
```kotlin
fun name(): Int {
  return 0
}
```

Figure 9. A simple *return* statement.



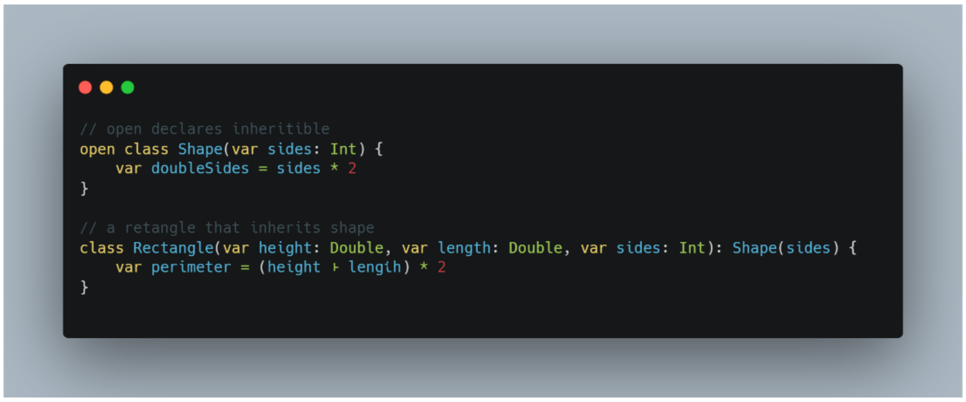
```kotlin
fun main() {
  print("Hello")
  println("World")
}
```

Figure 10. The syntax of the main function. It also shows the print statement syntax.



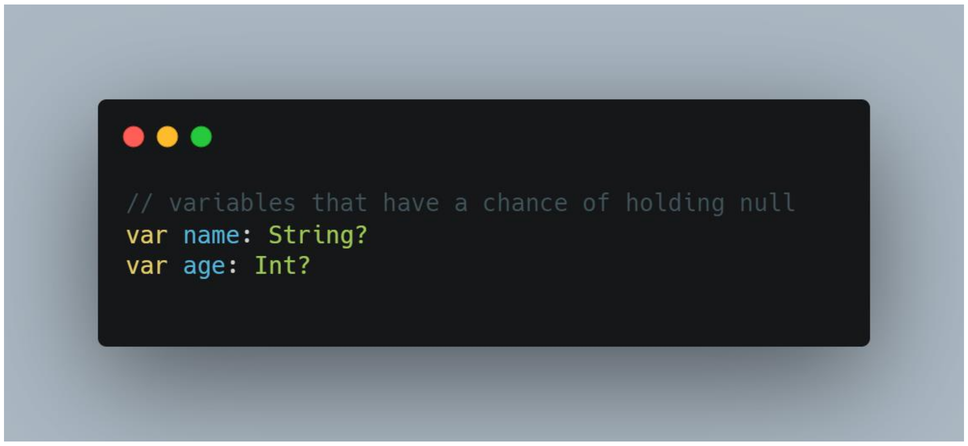
```
fun sumTwoNums(a: Int, b: Int): Int {
  return a + b
}
```

Figure 11. A simple summation function.



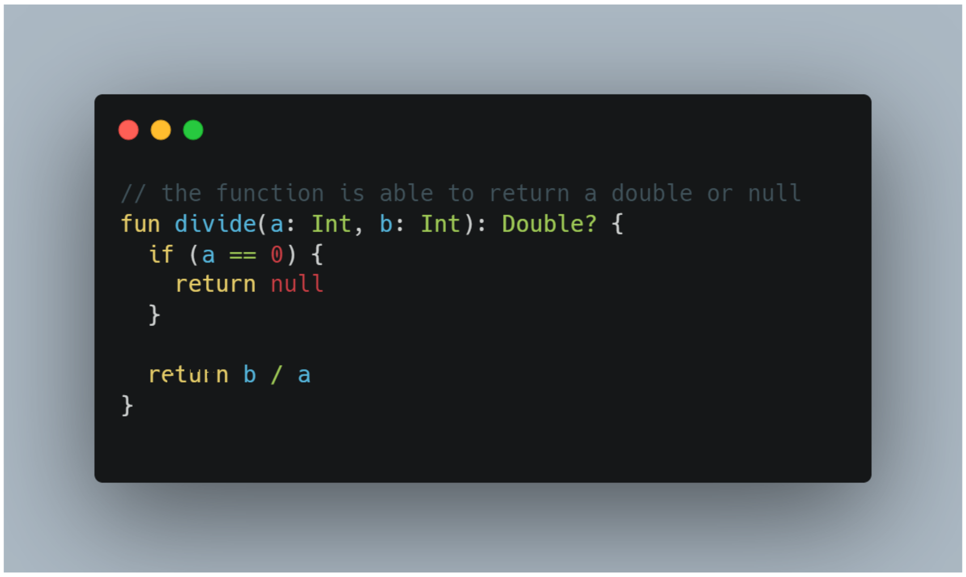
```
// open declares inheritible
open class Shape(var sides: Int) {
    var doubleSides = sides * 2
}

// a retangle that inherits shape
class Rectangle(var height: Double, var length: Double, var sides: Int): Shape(sides) {
    var perimeter = (height + length) * 2
}
```

Figure 12. An example of classes and inheritance.



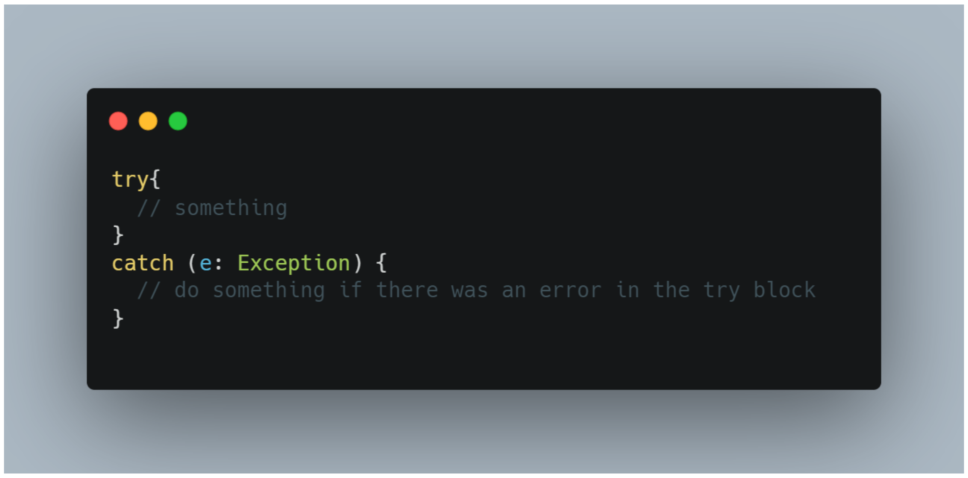
```
// variables that have a chance of holding null
var name: String?
var age: Int?
```

Figure 13. An example of variables that might hold null.



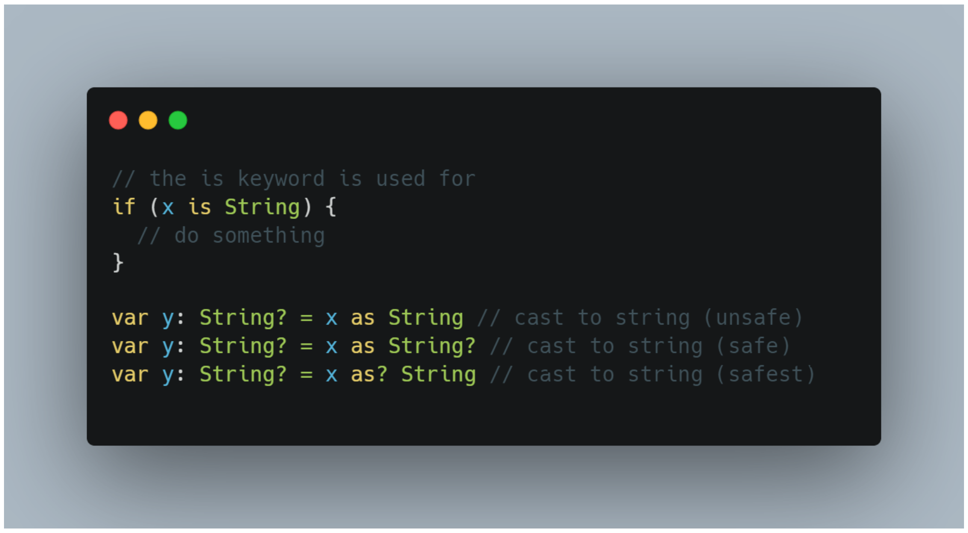
```kotlin
// the function is able to return a double or null
fun divide(a: Int, b: Int): Double? {
  if (a == 0) {
    return null
  }

  return b / a
}
```

Figure 14. An example of a function that can return a double or null.



```kotlin
try{
  // something
}
catch (e: Exception) {
  // do something if there was an error in the try block
}
```

Figure 15. An example of exception handling.



```
// the is keyword is used for
if (x is String) {
  // do something
}

var y: String? = x as String // cast to string (unsafe)
var y: String? = x as String? // cast to string (safe)
var y: String? = x as? String // cast to string (safest)
```

Figure 16. An example of type checking, and typecasting.

Appendix B:

Types & Operators

Numerical	Logical	Characters / Strings	Arrays	Unsigned
Byte	Boolean	Char	ByteArray	UByte
Short		String	ShortArray	UShort
Int			IntArray	UInt
Long			LongArray	ULong
Float			FloatArray	
Double			DoubleArray	UByteArray
			CharArray	UShortArray
				UIntArray
				ULongArray

Table 1. All available types.

Operator	Purpose
+, -, *, /, %	Mathematical operators
=	Assignment operator
+=, -=, *=, /=, %=	Augmented assignment operators
++, --	Increment and decrement operators
&&, ||, !	Logical 'and', 'or', 'not' operators
==, !=	Equality operators
===, !==	Referential equality operators
<, >, <=, >=	Comparison operators
!!	Assert that an expression is not null
?.	Performs a safe call
?:	Takes right-hand if the left-hand value is null (Elvis operator)
::	Method reference operator
..	Range operator
:	Separates variable name and type
?	Marks a type as possibly null
;	Separates multiple statements on the same line
$	References a variable in a string template

Table 2. All primary operators available.

Works Cited

Android Developers. “Kotlin Overview.” *Android Developers*, Google Developers, 27 Dec. 2019, https://developer.android.com/kotlin/overview.

Bogode, Stanley, et al. “Discover the History of Kotlin.” *OpenClassrooms*, OpenClassrooms, 28 June 2022, https://openclassrooms.com/en/courses/5774406-learn-kotlin/5930526-discover-the-history-of-kotlin.

Garg, Priyanka. “What Is Kotlin? 12 Interesting Facts about Kotlin.” *OpenXcell*, OpenXcell, 20 Dec. 2021, https://www.openxcell.com/blog/12-things-must-know-kotlin/.

Kotlin Foundation. “Kotlin Programming Language.” *Kotlin*, JetBrains, https://kotlinlang.org.

Miller, Paul. “Google Is Adding Kotlin as an Official Programming Language for Android Development.” *The Verge*, Vox Media, 17 May 2017, https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017.

Naik, Amit Raja. “Ten Years of Kotlin Programming Language.” *Analytics India Magazine*, AIM, 13 Oct. 2021, https://analyticsindiamag.com/ten-years-of-kotlin-programming-language/.

Zubchenko, Alexander. “The Complete Kotlin Programming Language Review - Software Development.” *Waverley*, Waverley Software Inc, 7 Oct. 2022, https://waverleysoftware.com/blog/kotlin-review/.

Confirmation Information

```
lock okprocessing
searching ... found it at line 60
Jonathon==Jonathon/15|A0007B|==|A0007B|set language366Kotlin
Kotlin :348074 Jonathon 2022-10-17-09-48-42 348550 1
Set language Kotlin (366) for 348074, confirmation code=348550
All done
```