Kotlin: A Brief Analysis

Ryan Wittmers

Structures of Programming Languages

Dr. John Walker Orr, PhD.

2021-12-01

Contents

1 Introduction

Kotlin is a statically typed, expressive, open-source language that supports both functional and object-oriented language paradigms. Its syntax draws similarities from a variety of other languages including Java, Scala, and C#, with a large emphasis being on its readability, writability, generalisability, and portability. Released in 2011 by JetBrains—the company behind many popular IDEs such as IntelliJ, CLion, and DataGrip—Kotlin was their solution to many of the issues that developers had been facing with existing popular programming languages. Kotlin introduced itself as a more modern, concise language with built-in null safety and interoperability with any library on the Java Virtual Machine (JVM). Additionally, it allowed developers to choose their target to be the JVM, Android, Native, or JavaScript depending on if they were creating a backend, cross-platform, server-side, front-end, Android, or multi-platform application, with a large focus being on multi-platform programming.

1.1 Origin

Although the project started in 2010, Kotlin's first stable release was not until 2016. Its name originates from a small Russian island off the west coast of Saint Petersburg (where most of the JetBrains employees live). Kotlin's lead designer Andrey Breslav chose the name because Java was named based on an island in Indonesia, and Kotlin was developed to be an alternative solution to Java. Kotlin was developed to be a more concise, safe, interoperable, pragmatic, toolchain-compatible, collections-focused language with the motivation to do so coming from what the designers and engineers determined to be Java's limitations. The designers realized that although the Scala language had already been developed, its slow compile time became a large obstacle for large-scale development. However, despite being developed as an alternative to Java, its actual compilation and execution process is the same as Java's when the JVM is being used as the compilation target as seen in **Figure 1**.

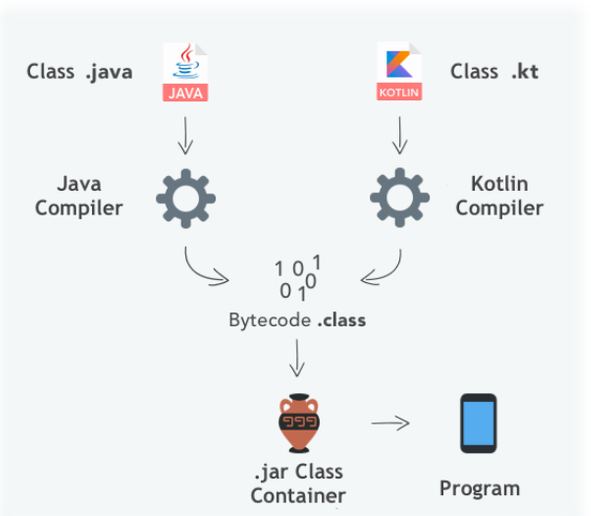

Figure 1: *Kotlin v. Java Compilation to the JVM [1]*

1.2 Iterations & Usage

Kotlin quickly rocketed into popularity when Google announced that Kotlin would be officially supported as a development language for Android in 2017. Then, in 2019, Google announced that Kotlin is their preferred language for Android development. Consequently,

Kotlin became a language that is the backbone for many of the applications used today. Companies such as Cash App, VMware, Netflix, Philips, Amazon, and Uber have utilized Kotlin because of its advantages and ease of implementation over other popular development languages [2]. The reason behind its development originates from the need to more safely manage and combat arguably the most egregious aspect of Java and what is often referred to as "The Billion Dollar Mistake": the `NullPointerException`. Kotlin has utilized what has been learned over the years about the advantages and disadvantages of popular languages such as Java, Scala, and C# and created a language with a focus on succinct, safe, multi-platform code that can be used to write native applications for Windows, Unix, web, and Android environments. However, the designers wanted the learning curve for Kotlin to be as minimal as possible, which is why it has 100% interoperability with Java and the JVM, resulting in full Object-Oriented support. Further, Kotlin has full support for functional programming features, including first-class functions, immutability, and side-effect elimination.

2 Syntax & Semantic Structures

A large motivation behind the initial development of Kotlin was to take the robust features of Java and create a more modern and safe language. With this motivation in mind, a summation of the developers' intent results in an overall focus on the readability and writability of their language. This more concise code starts on the variable declaration level but can be seen throughout function and class definitions as well. For example, the "Hello, World!" program is one of the most famous ways to show a few language elements in a short snippet. The examples of such programs are demonstrated in Java and Kotlin below.

```java
public static void main(String[] args)
{
    System.out.println("Hello, World!");
}
```

Figure 2: *Java Hello World*

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

Figure 3: *Kotlin Hello World*

2.1 Variables & Types

Kotlin allows the use of type declaration and the use of type inference. Using Pascal notation (i.e. `<name>: <Type>`) [3], variables use the keywords `val` or `var` upon each variable declaration to indicate whether the variable can be modified after its first assignment, with `val` being read-only and `var` allowing both read and write access after assignment. There are a few ways to declare a variable in Kotlin:

```kotlin
// Type declaration
val readOnly: Short = 10

// Type inference
var readOrWrite = 10

// Special case where variable does not need to be initialized
// However, it must be initialized before it can be used
// in any expression or operation
val read: Int
```

2.2 Basic Data Types

Due to Kotlin being able to compile to the JVM and be interpreted as JavaScript code it must support the types included in each. By default, each type is non-nullable, and must be forced to be nullable if the programmer desires by adding the `?` to the end of the type (e.g. `String?`). Additionally, each type and class inherits from the `Any?` class, which is the superclass of all types and the root of the Kotlin type hierarchy.

2.2.1 Numbers

Kotlin has a variety of ways to express a numeric value. It does not have primitive types and instead uses objects to represent each numeric type, of which are each of the familiar types including `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`. Since Kotlin supports mixed-mode arithmetic, any of the aforementioned numeric types can be combined in an expression, with the result being stored as a `real` number if one of the two operands represents a Non-Integer. Additionally, Kotlin supports the use of *literal constants* when declaring or assigning numeric values. For example, if one needed to create a variable holding a `Long` value, the only declaration needed would be:

```
val bigNumber = 1L // <-- 'L' = literal constant to represent a Long

// Instead of:
// Note: Underscores can be used to make nums more readable
val bigNumber = 3_000_000_000
```

On the JVM, numbers are only stored as primitive types unless there is a case where a nullable reference to the number object is created by adding a "?" character to the end of the declaration (i.e. `Int?`). However, despite Kotlin's portability between numeric values, it does not consider smaller numeric types to be subtypes of larger types, as shown in the following equality check:

```
val a: Int? = 42 // A nullable Int
val b: Long? = a // A nullable Long

print(b == a)
```

In the above scenario, if the code were to compile—which it does not because the compiler knows that a Long cannot be equal to an Int—the `print(b == a)` statement would evaluate to false, because of the Long's `.equals` function that would check if both `a` and `b` are of the `Long` type [4].

2.2.2 Characters

The `Char` type in Kotlin represents a standard 16-bit Unicode character for JavaScript and the JVM, with non-nullable values represented as a primitive `char` value and literals being stored within single quotes (e.g. 'a'). Its common functions include `compareTo()` and `equals()`, and supports standard Unicode escape characters preceded by a backslash such as `\t`, `\n`, `\b`, `\r`, `\'`, `\"`, `\\`, `\$`, and all standard Unicode characters (e.g. '`\u0026`'). Further, it supports the standard arithmetic operators `+`, `-`, `*`, `/`, and `%`.

2.2.3 Strings

Strings are represented by the `String` type, which is an iterable sequence of characters stored in double-quotes. The string class in Kotlin is of the form:

```
class String : Comparable<String>, CharSequence
```

and has functions such as `length()`, `replace()`, and `substring()`, to name a few. Strings are accessible by index, are immutable, and can be manipulated with arithmetic addition (i.e. `string1 + string2`) for concatenation. A string and a numeric value can be added together to create a new string, which is stored as a new `String` object since they are immutable in Kotlin. Additionally, because strings are `Comparable` character sequences, they can be compared with the `==` or `!=` operators instead of the `equals()` function.

2.2.3.1 Raw Strings

In addition to string arithmetic concatenation, Kotlin's `String` supports *Raw Strings*, which is a string that can contain expressions and escape characters that are not interpreted by the compiler. Raw strings are interpreted as they are written, and thus can be used to create multi-line templates that can be stored as formatted strings. Raw strings are enclosed within triple-quotes (i.e. `""" <STRING> """`) and can contain any character that the user may want to verbosely include in the string. This means that any escape characters will be printed inside the string instead of being interpreted as an escape (e.g. `\n` will print instead of a newline being added).

```
val rawString = """
This is an example of a for loop in Python
    for i in foo:
        print(i)
"""

println(rawString)

/* Output:
    This is an example of a for loop in Python
    for i in foo:
        print(i)
*/
```

Notice that the indentation of the raw string is preserved to reflect the indentation in the original raw string, which is why raw strings are often used when the programmer wants to print out a multi-line string that has specific indentation and formatting.

2.2.3.2 String Interpolation / Templatization

String interpolation in Kotlin allows a programmer to insert a variable directly into a string, without having to manually concatenate the variable with the string or having to use the `String.format()` function. For example, the following code will print out the value of the variable `foo`:

```
val foo = "bar"
println("The value of foo is $foo")

// Output: The value of foo is bar
```

This is extremely useful especially when a variable is used multiple times within a string or when the variable holds the result of a calculation, function, or expression. It arguably makes the string output more readable and writable because it does not use conversion specifiers like `%d` or `%s`, for example, to format the string. Further, if one wanted to include an expression directly inside a string, they can do so by enclosing the expression in curly braces (e.g. `${foo.length - 1}`). This is useful when the expression is not a constant, but rather a calculation that is dependent on the variable. One common use of string interpolation is in a class's `toString()` function because it makes the printing of the object's state incredibly easy and readable.

2.2.4 Arrays

Arrays are represented by the `class Array<T>`, which has a notion of its size and the elements contained within. The array's elements can be accessible by index, and its `.indices` can be iterated over using the `for` loop. One can create an array of any type by using the `arrayOf()` function, which takes a list of elements as an argument. For example, the following code creates an array of integers:

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

// OR, just to initialize an array of 5 integers.
val array = IntArray(5) { it * 1 } // [0, 1, 2, 3, 4]
```

Arrays are not dynamically resizable which means that they must be initialized with a fixed size. An abstract structure such as a `List` or `ArrayList` must be utilized to create a dynamically sized array.

2.3 Null Safety

As mentioned in the introduction, Kotlin has a built-in null safety feature that is made to combat the `NullPointerException` and other errors that can occur when using Java due to its lack of null safety.

2.3.1 ? and let

The null-safety feature is implemented by the use of the keyword `?` in the declaration of a variable. For example, the following code is valid:

```
val nonNullableString: String = "Hello, World!"
nonNullableString = null // Throws compilation error
...
val nullableString: String? = "Hello, World!"
nullableString = null // No error
```

The only way for a variable to be null is if it is explicitly set to null and uses the `?` to indicate that assignment. Additionally, the language has a few other features to not break one's program when null is encountered. For example, the `let` keyword when called on a nullable variable will return the value of `this` with null omitted. For example, if there was a list that had null values mixed with String values, calling `list?.let { println(it) }` would print each non-null value in the list, where `it` is the current value of the iteration.

2.3.2 The Elvis Operator

The Elvis Operator (?:) is used to return a default value if the variable is null, with its unique name coming from it looking like Elvis's iconic smirk when turned on its side to the left. Its usage is similar to that of a ternary expression where one variable may be null. For example, one could express "if foo isn't null, store it, otherwise store some non-null value". This could be expressed in Kotlin as: `val bar = foo ?: "non-null"` This functionality can be used to avoid the need for nullable variables in many cases.

2.4 Control Flow Statements

Kotlin provides a few ways to manage the flow of a program. Namely, those types are the standard `Boolean`, the `when` clause, `for` and `while` loops, and the `break` / `continue` statements. However, it is important to note that Kotlin does not support the use of ternary operations (i.e. `condition ? then : else`) since the operator `?` is reserved to specify that

a data type can be `null`. To compensate for this, Kotlin considers Booleans types to be expressions, with the `if` statement returning a value.

2.4.1 Booleans

`kotlin.Boolean` is the Boolean logic type of Kotlin and can represent either a `true` or `false` value. Additionally, it can be assigned to a variable as an expression with the evaluated value being stored as the result or passed as a parameter to a function. As mentioned above, the `if` statement returns a value since it is considered an expression.

```
val a = if (i == 3) bar() else foo()
```

In the above code snippet, the return value `bar()` will be assigned to `a` if `i = 3`, and the return value of `foo()` if `i != 3`. This statement allows for ternary-like conciseness with the readability of a regular `if` statement, which is especially advantageous for single-line logic when more complicated expressions are involved.

2.4.2 When expression

In an effort to be more concise, Kotlin reimagined the concept of a standard C-like `switch` statement and instead created a more readable and functional structure that has a `case -> result` format. In the following snippets, one can see the readability, functionality, writability gained, and the lines saved with the `with` statement.

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  default:
    System.out.println("Not a valid day");
}

// Output: Thursday
```

Figure 4: *Java Switch Statement*

```
val day = 4
when (day) {
    1 -> println("Monday")
    2 -> println("Tuesday")
    3 -> println("Wednesday")
    4 -> println("Thursday")
    else -> println("Not a valid day")
}

// Output: Thursday
```

Figure 5: *Kotlin "when" statement*

The `when` statement is not only a better alternative to the bulky `switch` statement, but also offers a variety of other uses and benefits. The `when` statement can be represented and accessed as an expression, can be used without an argument (i.e. to simplify an `if/else` block), and can have arbitrary conditional expressions [5]. It simplifies potentially complex `case` statements, removes the need for a `break` after every case, and can group two choices that should store the same result. Consider the trivial snippet below that checks if a character is A and assigns the result to a `isValidCharacter`.

```
val character = 'B'
val isValidCharacter = when (character) {
        'A' -> "Character is A!"
```

```
        'B', 'C', 'D', 'E' -> "Character $character is not A"
        else -> "Not a valid character"
    }

// Ouput: Character B is not A
```

As shown, the `when` statement is a large improvement over the traditional switch statement in terms of both readability and writability especially when it involves multiple cases that include the same outcome without necessarily being the default case.

2.4.3 Loops

Loops reflect the C-like `for`, `while`, and `do-while` statements. The `for` statement is used to iterate over a collection or range of values. The `while` statement is used to iterate over a block of code while a condition is true. The `for` statement is the most commonly used loop in Kotlin and is similar to the `for` loop in C with a few differences. The `for` loop is a more readable and functional structure that is most commonly used to iterate over a collection of values. The following snippets show a variety of uses for the `for` loop.

```
for (i in 1..10) { print(i) } // Output: 12345678910
for (i in 1..10 step 2) { print(i) } // Output: 13579
for (i in foo) { print(i) }
for (i in foo.indices) { print(foo[i]) }
```

The `for` loop is versatile and dynamic in nature; however, if a programmer wanted to simply iterate while a condition is true, the `while` loop is the preferred choice. The syntax is identical to a Java or C-like while loop.

2.4.4 Break / Continue

The `break` and `continue` statements can be used to break the flow of a loop or to skip a case in a `when` statement. The `break` statement can be used to break out of a loop or a switch statement. The `continue` statement can be used to skip the current iteration of a loop or to skip a case in a `when` statement. Additionally, the `break` statement can jump to an expression marked with a label, which is identified with the "@" symbol. A label paired with a `break` can be used to break out of a loop or switch statement, which can be used to move out of a current scope to another one in the same function. The following snippet shows a simple example of using `break`s paired with labels.

```
outer@ for (i in 1..2) {
    for (j in 1..3) {
        if (i == 3 && j == 3) {
            break@outer
        }
        print("$i, $j")
    }
} // Output: 1, 11, 21, 32, 12, 22, 3
```

Although this is not a commonly utilized functionality due to effects on readability, it is incredibly useful when wanting to more granularly control the flow of any of the control flow statements in an unnatural manner [6].

3 Abstract Data Types

In addition to the aforementioned primitive data types, Kotlin also has several abstract types that are used to represent collections of values. The snippet exemplifies a non-comprehensive list of abstract data types in Kotlin:

```kotlin
val list = listOf(1, 2, 3) // List<Int>
val mutableList = mutableListOf(1.2, 2.3, 3.4) // MutableList<Double>
val set = setOf(1, 2, 3) // Set<Int>
val map = mapOf(1 to "one", 2 to "two") // Map<Int, String>
val array = arrayOf(1, 2, 3) // IntArray
val hashSet = hashSetOf(1, 2, 3) // HashSet<Int>
val hashMap = hashMapOf(1 to "one", 2 to "two") // HashMap<Int, String>
```

Although Kotlin has its own abstract data types, it is still possible to use Java's abstract types because of the interoperability of the two languages. This interoperability allows for a programmer to write more functional and safe code without sacrificing the power of the JVM and its existing data structures.

4 Program & Subprogram Structure

Kotlin is an object-oriented and functional programming language because of the mix of lambda expressions, operator overloading, higher-order functions, and lazy evaluation, meaning it supports the use of objects, classes, data classes, and interfaces. The following snippet shows a few of the options at one's disposal to structure a Kotlin program:

```kotlin
fun main(args: Array<String>) {
    println("Hello, world!")
}

class MyClass {
    fun myMethod() {
        println("Hello, world!")
    }
}

object MyObject {
    fun myMethod() {
        println("Hello, world!")
    }
}
```

As mentioned and shown in the snippet above, Kotlin provides several ways to structure a program.

4.1 Classes

Classes are the most common way to structure a program in Kotlin. Classes are declared using the `class` keyword and are public unless otherwise specified. A class can be declared as a data class, which is a class that has a primary constructor and a companion object—where an object is a static instance of a class there is only one of—or a normal class that has a primary constructor and, optionally, a secondary constructor.

4.1.1 Primary Constructors

The primary constructor is the constructor called when an object of the class is created. The primary constructor is the only constructor that is called when an object of the class is created. A simple constructor may look like this:

```kotlin
class MyClass(val myValue: Int)
{
    class MyClass constructor (val myValue: Int)
}
```

The primary constructor is the only constructor that is required to be present. As seen in the above snippet, the parameter passed to the `constructor` is automatically initialized upon object creation, which means no code is required within the constructor to initialize the class arguments and is why Kotlin prohibits code within the constructor. Additionally, the `constructor` keyword can be omitted if there are no visibility modifiers or method annotations. However, one could use the `init` function to initialize the class arguments if extra initialization is preferred. The following snippet shows a simple class and constructor:

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

During the instance initialization, the `init` blocks are called in the order that they appear since multiple can be used. The `.also` syntax calls the `println` on `this` upon object creation, which is incredibly useful when specifying state that may exist, but will not attempt to print if the object is null, for example [7].

4.1.2 Secondary Constructors

The secondary constructor is used to initialize the class arguments in a different scope and a different order. A functional example of secondary constructor usage is shown in the below definitions of a `Person` and a `Car` and making each `Car` have an owner upon creation.

```
class Person(val vehicles: MutableList<Car> = mutableListOf())

class Car{
    constructor(owner: Person) {
        // Add this car to the owner's list of cars
        owner.vehicles.add(this)
    }
}
```

Additionally, if classes need to be extended or interfaces implemented, it can be easily done using this short snippet of code:

```
public class Student extends Person
    implements Comparable<Student>
```

Figure 6: *Extending/Implementing in Java*

```
class Student : Person, Comparable<Student>
```

Figure 7: *Kotlin: Extends/Implements*

The above figures highlight the more concise usage of the `extends` and `implements` keywords in Kotlin, which makes writability much more clear.

4.2 Other Classes & Instantiation

Many other types of classes can be used to structure a program. As mentioned above, data classes are classes that have a primary constructor and a companion object, where an object

is a static instance of a class that there is only one of. There is full support for abstract classes and interfaces, which can be specified using the `abstract` or `interface` keywords. A class can be declared abstract if one would like to use the class as a base or "template" for other classes. Furthermore, all Kotlin classes have a common superclass of `Any`, and thus inherit from it. By default, all classes are final, meaning they cannot be extended or inherited; however, if one wishes to extend or inherit froxm a class, one must explicitly declare the class as `open`.

Once any of the mentioned classes have been defined, one can create a new instance of the class using a couple of methods:

```
val myClass = MyClass()
val myObject = MyObject()
```

As a final note on Kotlin class creation, the language does not use the `new` keyword to create an instance of a class.

4.3 Functions

Functions are defined using the `fun` keyword and are called using the standard `()` approach, using dot notation to invoke member functions, and can have explicit or implicit `return` statements. Function parameters can be default values or named arguments. For example, a short function definition could be:

```
fun listLength(length: Int = list.size): Int
```

If a function does not return a value, the `Unit` type is used but can be omitted from the function definition. Because functions are first-class, they can be passed as arguments to other functions and can be returned from other functions. This functionality also means that lambda expressions can be defined as well, such as the simple example of a sum function used as a lambda expression:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
// vs.
val sumLambda = { a: Int, b: Int -> a + b }
println(sumLambda(1, 2)) // 3
println(sum(1, 2)) // 3
```

It is also very common to see a lambda expression with a single parameter, which means the verbose parameter can be omitted and the `it` keyword can be used in place of the `param: Type -> <value>` syntax. The following example shows a trivial example of the `it` keyword, where the parameter is simply added to 1 and returned:

```
intList.filter { it > 0 }
// vs.
intList.filter { value -> value > 0 }
```

Although it is much more writable to use the `it` keyword, it may add to one's confusion when reading the code when they do not already know what type the `it` is implicitly referring to.

4.4 Exceptions

All exceptions in Kotlin inherit from the `Throwable` class and can be caught in a `try/catch` block, or explicitly thrown using the `throw` keyword. Due to full Java interoperability, if there is a Java exception that is thrown, it can be caught in Kotlin using the `catch` block. All

exceptions require a message, which makes the debugging process much more readable and manageable. Something that many programmers view as an oversight in Kotlin is that there is no *checked exceptions* like there are in Java. Kotlin provides a convincing explanation as to why there are no checked exceptions in Kotlin, and that in an effort to be more safe,

> *Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality* [8].

The Kotlin designers believe that exceptions are necessary to keep a project safe, but don't believe specifying that a function will throw an exception is a good design choice. They explain it almost like that functionality is setting the function up for failure, and should be crafted in a way that makes it so the code doesn't throw an exception.

5 Development Environment

One of Kotlin's key strengths is its ability to be used for multi-platform programming. It greatly reduces the amount of time, code, and developers used to manage the same application for use on multiple platforms. The Kotlin Multiplatform, or KMM, is the bridge between what has been previously separated into multiple platform: Native, JVM, and Web-based code. Similar to the JVM for Java, the KMM is a collection of libraries that can be used to write Kotlin code on multiple platforms as shown in Figure 8.

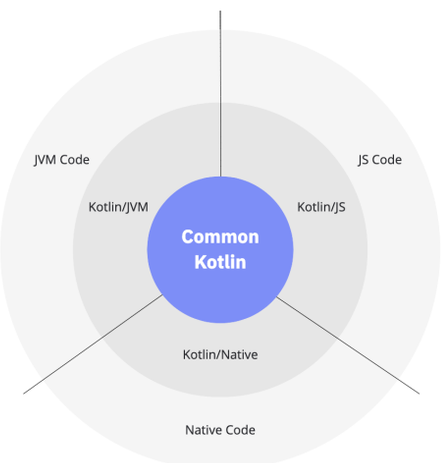

Figure 8: *Kotlin Multiplatform Development Environment*

A variety of tools are available to help with multi-platform development as shown in **Figure 6**. *Common Kotlin* includes all core libraries and basic tools and can be used to create a Kotlin project for any platform. If a programmer wants interoperability between different platforms, they can use platform-native versions of Kotlin, such as Kotlin/JS, Kotlin/Native, and Kotlin/JVM. This figure shows how a Kotlin project can be compiled to each of the platforms:

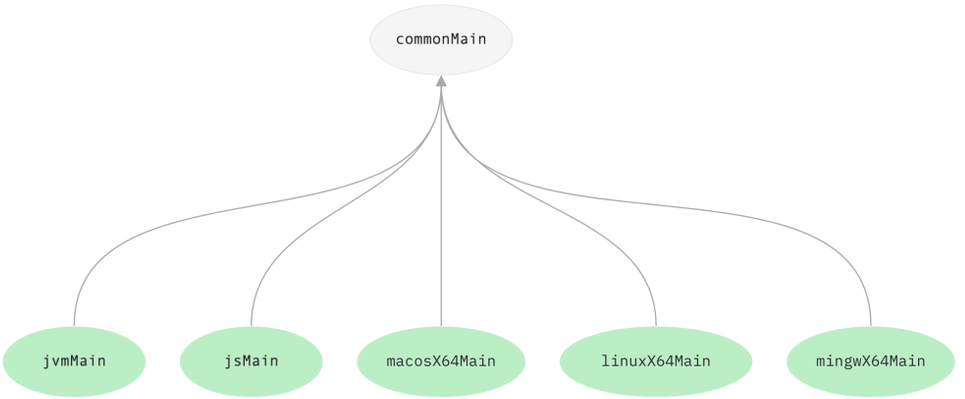

Figure 9: *Sharing Code Between Platforms*

A single development platform can be used to create a project that compiles to an executable on the JVM, macOS, Linux, and Windows. This provides an incredibly robust way to easily manage product development, which is why it is the Android development language of choice and why the KMM is so popular. Aside from application development, Kotlin can be utilized to create server-side applications, which can be used to create RESTful APIs, or to create a server-side language for use in a database using the Ktor framework: a JetBrains-engineered framework for client/server applications[9].

5.1 IDEs and Toolchain

Depending on the type of application being developed, different IDE's and toolchains can be used to create and manage the project. The most commonly used toolchains for each Kotlin project are IntelliJ IDEA, Android Studio, and NetBeans for OS-level development.

6 Hands-On Experience

I have thoroughly enjoyed working in Kotlin (albeit for only a few projects) and can see why it is such a popular language amongst both freelancers or individuals building commercial solutions. It is familiar because of its similarities to previous languages I've used, such as Java and Scala, while improving upon each of those languages because of its null-safety features and concise code. Its interoperability with Java and added safety features make it an excellent general-purpose language for use in a variety of different areas. A criticism of Kotlin is that it was initially created to be a quicker, more concise language. However, even in my own, fairly small and relatively trivial projects, it felt that compilation time was much longer than it would've been in even Java. However, this makes sense for a couple of reasons. First, Kotlin's compiler is often described as a "smart compiler," which means that the compiler does a variety of optimizations such as static code analysis and smart type casting by performing null-checks and explicit type casts.

Despite these smart features, it is not a catch-all for each problem that can arise in a project. For example, a project that uses a lot of null-safety features can be difficult to debug, as the compiler will often not be able to detect the problem and will instead just throw an exception. This is because the compiler is not able to perform static analysis, and so it cannot know if the code is correct.

An additional reason for longer type casts is that to run a project built on the JVM, the programmer must first compile the Kotlin source files (`.kt`) to Java `jar` files then run those `jar` files using the java interpreter. For someone who had never worked with a Kotlin project before and was new to the language, creating a project running it was as simple as creating

a new Kotlin project in IntelliJ IDEA and running the project. However, passing command-line arguments to the project required a fair amount of research to get the project to run. For example, to get the project to compile and run with a grammar file as an argument, the following arguments must be provided:

```
kotlinc Grammar.kt -include-runtime -d Grammar.jar && java -jar Grammar.jar grammar.g
```

Although this is a composite command and can be split up, it still results in a fairly long overall run time for the project because it goes through both the Kotlin and Java compilers before finally executing the `jar` file.

7 Final Thoughts

Minor grievances aside, I would assert that Kotlin does exactly what it was designed to do, which is to be a functional, multi-platform, and easily maintainable language that can be utilized as a toolbelt for the development of almost any application. Furthermore, it was designed as an alternative and eventual replacement to Java—which I believe is an objective it executes very well for being on Version 1.0—and it achieves its goal to provide functional, concise, and safe code without sacrificing readability. It was made by developers, for developers for use in personal, industrial, and commercial applications. For programmers who are not active in the industry such as students or freelancers, adopting Kotlin is made easy because of its simplicity, familiarity, and interoperability with Java and its libraries. Overall, I can see and appreciate the value of Kotlin as a multi-platform, multi-paradigm language that does not cut corners regarding robustness, readability, writability, and ease of use. Despite these advantages, I recognize that Kotlin is continuing to grow, develop, and improve as a general-use language that is a quicker and easier alternative to the language giants that dominate the industry.

References

[1] B. Philippe and B. Wixted. "Grasp the Kotlin origin story". In: *OpenClassrooms* (2019).

[2] J. Skeen and D. Greenhalgh. *Kotlin Programming: The Big Nerd Ranch Guide*. Pearson Technology Group, 2018.

[3] *Basic syntax: Kotlin*. Sept. 2021. URL: https://kotlinlang.org/docs/basic-syntax.html#variables.

[4] *Basic types: Kotlin*. 2021. URL: https://kotlinlang.org/docs/basic-types.html#numbers.

[5] G. Tomassetti. *Kotlin when: A switch with superpowers*. June 2019. URL: https://superkotlin.com/kotlin-when-statement/.

[6] *Returns and jumps: Kotlin*. Oct. 2021. URL: https://kotlinlang.org/docs/returns.html#break-and-continue-labels.

[7] *Classes: Kotlin*. Nov. 2021. URL: https://kotlinlang.org/docs/classes.html#constructors.

[8] B. Eckel. "Exceptions: Kotlin". In: *Kotlin Help* (Sept. 2021). URL: https://kotlinlang.org/docs/exceptions.html#checked-exceptions.

[9] *Welcome: KTOR*. Dec. 2020. URL: https://ktor.io/docs/welcome.html.

8 Programs

8.1 Factorial

```kotlin
/**
 * Return the factorial value of a given [number]
 *
 * @author rwittmers19@georgefox.edu
 */
fun recursiveFactorial(number: Long): Long
{
    // Base case, stored as a Long because factorial can get very big!
    var factorial: Long = 1

    // If not one (because 1! == 1)
    if (number >= 1)
    {
        // Recursively calculate factorial of number and store in "factorial"
        factorial = recursiveFactorial(number - 1) * number
    }

    return factorial
}

/**
 * Taken in a command line argument [args[0]] for the number to
 * calculate factorial value of.
 */
fun main(args: Array<String>)
{
    // Stores usage string to print if there are errors
    // Compiles this file to a .jar file, then runs it using the JVM compiler
    val usage = "Usage: kotlinc Factorial.kt -include-runtime -d Factorial.jar
        && java -jar Factorial.jar <N_VALUE>"

    // Get and validate input for argument n from command line
    try
    {
        // Basic validation; program only takes one input
        if (args.size == 1)
        {
            val factorialResult: Long = recursiveFactorial((args[0]).toLong())

            // Print the result
            println("\n${args[0]}! equals $factorialResult\n")
        }
        else
        {
            println(usage)
        }
    }
    catch (parse_error: TypeCastException)
    {
        println(usage)
    }
}
```

8.2 Structured Information Sorter

```
import java.io.BufferedReader
import java.io.FileNotFoundException
import java.io.FileReader
import java.lang.IndexOutOfBoundsException
import java.lang.NumberFormatException
import kotlin.math.round

/**
 * A Person object is a list of Person's sorted by name with an attached age.
 *
 * NOTE: By default, each function in Kotlin is public, which means the "public"
 * specification is redundant.
 *
 * @param _name the name of the Person
 * @param _age the age of the Person
 *
 * @author rwittmers19@georgefox.edu
 */
class Person(var _name: String, var _age: Int)
{
    /**
     * Construct a new Person object
     */
    constructor(name: String, age: String) : this(name, age.toInt())

    /**
     * Get name of the Person
     */
    fun getPersonName(): String
    {
        return _name
    }

    /**
     * Get age of the Person
     */
    fun getPersonAge(): Int
    {
        return _age
    }

    /**
     * Set the name of the Person
     */
    fun setPersonName(name: String)
    {
        _name = name
    }

    /**
     * Set the age of the Person
     */
    fun setPersonAge(age: Int)
    {
        _age = age
    }
}

/**
 * Class PersonDB reads input from a file and creates a collection of Persons
 * It then sorts the collection by name and outputs the name of each Person
```

```
 * and the average age of all the Persons.
 */
class StructuredInformationSorter
{
    /**
     * Reads and parses a file in the same directory and returns a list of Persons
     *
     * @param fileName the name of the file to read
     * @return return a list of Persons sorted by name
     */
    fun readFile(file: BufferedReader): List<Person>
    {
        val people = mutableListOf<Person>()

        try
        {
            /**
             * The "?" is a usage of null safety.
             * If the file is null, return an empty string,
             * which makes a try / catch a redundant and unneeded step.
             * However, parse errors should still be caught.
             **/
            var line: String? = file.readLine()

            // Read each line of the file and create a Person object
            while (line?.isNotEmpty() == true)
            {
                // Split name and age by commas
                val personInformation = line.split(",")

                // Initialize a new Person object
                val person = Person("", 0)

                // Create Person using name ( person[0] ) and age ( person[1] )
                person.setPersonName(personInformation.first())
                person.setPersonAge(personInformation.last().toInt())

                // Add Person to the list
                people.add(person)

                // Move to the next line
                line = file.readLine()
            }
        }
        catch (ioError: IndexOutOfBoundsException)
        {
            error("An out of bounds index was being accessed. "
                + "Verify data is formatted properly.")
        }
        catch (formatError: NumberFormatException)
        {
            error("A person's age is improperly formatted,"
                + " Please verify all ages are an Integer type. ")
        }
        finally
        {
            // Cleanup open file
            file.close()
        }

        // Return a list of Person's sorted by name
        // "it" is an implicit name of a single parameter. In this case, "name".
        // "it" is similar to persons[0] - with better readability.
```

```
        return people.sortedBy { it.getPersonName() }
    }

    /**
     * Calculate the average age of all Persons
     *
     * @param people the list of Persons created by the datafile
     * @return return the average age of all Person's age
     */
    fun averageAge(people: List<Person>): Double
    {
        // Calculate the average age of all Persons
        // "it" is an implicit name of a single parameter. In this case, "age"
        return (people.sumOf { it.getPersonAge().toDouble() } / people.size)
    }
}

/**
 * Driver function for program. Reads a txt file in the same directory
 * and outputs the name of each Person and the average age of all the Persons.
 */
fun main()
{
    // Create a variable to store the list of people
    val persons: List<Person>

    // Create a variable to store the average age of all people
    val averageAge: Double

    try
    {
        // Read the datafile.
        val datafile = BufferedReader(FileReader("datafile.txt"))

        // Pass in a datafile containing Person information.
        persons = StructuredInformationSorter().readFile(datafile)

        // Assign the average age of all people
        averageAge = StructuredInformationSorter().averageAge(persons)

        /*
            Print the name of each Person and the average age of all Persons.

            "it" is an implicit name of a single parameter.
            In this case, "name" and "age"
        */
        persons.forEach { println(it.getPersonName()) }
        print("\nAverage age: $averageAge")
    }
    catch (e: FileNotFoundException)
    {
        error("File not found")
    }
}
```

8.3 Random Text Generator

```
import java.io.*
import java.util.*
import kotlin.random.Random
import kotlin.system.exitProcess

/**
 * Class Grammar generates a grammar dictionary from a file.
 *
 * @author rwittmers19@georgefox.edu
 */
class Grammar
{
    private val grammar: MutableMap<String, MutableList<String>> =
        mutableMapOf()
    private val startSymbol: String = "<start>"
    private final val START_SYMBOL = '{'
    private final val END_SYMBOL = '}'
    private final val PRODUCTION_END_SYMBOL = ';'
    private final val NON_TERMINAL_PATTERN = "(?<=<).+?(?=>)".toRegex()

    /**
     * Gets the current state of the grammar dictionary
     *
     * @return return the current state of the grammar dictionary
     */
    private fun getGrammar(): MutableMap<String, MutableList<String>>
    {
        return grammar
    }

    /**
     * Reads a file and parses it to a grammar dictionary.
     *
     * @param fileName The name of the file to be read.
     * @return The grammar dictionary.
     */
    private fun makeGrammar(fileName: String):
        MutableMap<String, MutableList<String>>
    {
        // Opens a grammar file input by the user via command line and returns
        // the file contents as a string.
        try
        {
            // Try to open the file
            val file = File(fileName)
            val fileReader = FileReader(file)
            val bufferedReader = BufferedReader(fileReader)

            // Remove each '}' from the file and store it as the file
            val grammarFile: String =
                bufferedReader.readText().replace(END_SYMBOL.toString(), "")

            // For each line in the construct split on '{'
            for (symbol in grammarFile.split(START_SYMBOL.toString()))
            {
                // Initialize the non-terminal as blank since one hasn't been
                // found yet
                var nonTerminal = ""

                // If a non-terminal is found, store the nonTerminal
                if (NON_TERMINAL_PATTERN.containsMatchIn(symbol.trim()))
```

```
            {
                // Find all "Not-None" (!!) non-terminals
                nonTerminal =
                    NON_TERMINAL_PATTERN.find(symbol.trim())!!.value
            }

            val productions = mutableListOf<String>()

            // Store each production as the associated values to each
            // non-terminal key
            if (nonTerminal != "")
            {
                // For each production that's split on the ';' symbol
                for (production in symbol.split(PRODUCTION_END_SYMBOL))
                {
                    val nonTerminalPattern = "<$nonTerminal>"
                    // Add the production to the grammar, removing the first
                    // production symbol because it's the non-terminal
                    productions.add((
                        production.split("\n ").first().replace(
                            nonTerminalPattern, "")
                    ).trim())
                }

                // Assign each non-terminal to its productions if non-empty
                if (nonTerminal != "" && productions.isNotEmpty())
                {
                    // Removes dangling blank line from production list
                    productions.removeLast()

                    // Assign each non-terminal its production
                    grammar[nonTerminal] = productions
                }
            }
        }
    }
    catch (e: FileNotFoundException)
    {
        print(
            "\nFile '$fileName' not found, please ensure your grammar file is"
                + " in the same directory as your JAR file.\n\n"
        )
        exitProcess(1)
    }
    catch (io: IOException)
    {
        print("\nThere was a problem reading '$fileName', try again.\n\n")
        exitProcess(1)
    }

    // Return the current state of the grammar
    return getGrammar()
}

/**
 * Gets a random production from a given non-terminal
 *
 * @param nonTerminal The non-terminal to get a production from
 * @return A random production from the given non-terminal
 */
private fun getRandomProduction(nonTerminal: String): String
{
    // Get a random production using the given non-terminal
```

```
    var randomProduction = grammar[nonTerminal]!!.random()

    // However, if the production contains non-terminals, keep replacing
    // them with productions until there aren't anymore
    if (randomProduction.contains(NON_TERMINAL_PATTERN))
    {
        // Get the non-terminal from the production
        val subNonTerminal =
            NON_TERMINAL_PATTERN.findAll(randomProduction).first().value

        // Add a random production
        val nonTerminalPattern = "<$subNonTerminal>"
        randomProduction += getRandomProduction(subNonTerminal)

        // Replace the non-terminal with the random production
        randomProduction = randomProduction.replace(nonTerminalPattern, "")
    }

    return randomProduction
}

/**
 * Accepts grammar map and returns a random sentence using the grammar rules
 *
 * @param fileName The name of the grammar file
 * @return A random sentence generated from the grammar rules
 */
fun generateSentence(fileName: String): String
{
    val generatedGrammar = makeGrammar(fileName)
    var randomSentence = ""

    for (production in generatedGrammar["start"]!!)
    {
        // For each word in the production, check if it's a non-terminal,
        // if it is, replace it with a random production of that non-terminal
        for (word in production.split(" "))
        {
            if (NON_TERMINAL_PATTERN.find(word) != null)
            {
                // Gets the non-terminal word from the word
                // (e.g. <foo> -> "foo")
                val nonTerminal =
                    NON_TERMINAL_PATTERN.find(word)!!.value

                val nonTerminalPattern = "<$nonTerminal>"

                // Replace each non-terminal with a randomly chosen production
                randomSentence +=
                    word.replace(
                        nonTerminalPattern,
                        getRandomProduction(nonTerminal)) + " "
            }
            else
            {
                // If the word is not a non-terminal, append it to the string
                randomSentence += "$word "
            }
        }
    }

    // Finally, replace any verbose newline characters with a newline
    return randomSentence.replace("\\n", "\n")
```

```
    }

}

/**
 * The drive function for the program
 *
 * @param args The command line argument by which a user passes in a grammar file
 */
fun main(args: Array<String>)
{
    // Read command line argument as the grammar file name.
    val fileName = args[0]
    val grammar = Grammar()

    // Print out the random sentence
    print(grammar.generateSentence(fileName))
}
```