Fall 2020

Benchmarking Java and Kotlin in Android Runtime environment

Arnoldo Montoya-Gamez

Benchmarking Java and Kotlin in Android Runtime environment

by

Arnoldo Montoya-Gamez

A Creative Component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph A. Zambreno, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ABSTRACT

In 2017, Google announced that the Kotlin Programming Language would become an official Android Development language. In the meantime, it has become one of the fastest-growing programming languages. In this study, we studied the trade-offs between using Kotlin and what was officially the Android programming language, Java. This study focuses on seeing if there are differences between Java and Kotlin in execution speeds, Application size, and lines of code.

To analyze the trade-offs, we wrote two Android Apps, one using only Java, and the other using only Kotlin. To measure runtime, we wrote 5 commonly used algorithms using the same datasets. Each algorithm was executed 100 times; then, the time was averaged. To measure Application size, we analyzed the final app result. Finally, to measure lines of code, we looked at the final app using an Android Studio plugin called "Statistic."

After project completion, we found that both languages have pros and cons. Specifically, we found that, on average, Java is faster than Kotlin at executing code by about 20%. We also found the Java Android App to be smaller than Kotlin by about 39%. However, Kotlin does have one good thing going for it, which is lines of code. On average, writing code with the same purpose was about 19% shorter in Kotlin code than Java code.

In conclusion, this study found that if Android application developers want a slightly faster app or want their app to take up a little less space, Java may be the language to consider. But, if application developers want to develop features faster, Kotlin is the way to go due to it performing the same tasks with less lines of code.

CHAPTER 1. INTRODUCTION

Android is the most widely used mobile operating system with a current market share of about 75% worldwide, with IOS at about 25%. The Android OS has been the leading mobile OS since August 2012, which explains why there are about twice as many Android Developers as there are IOS developers.

From the start, developing apps for Android required knowledge in Java. Java is a language that was developed in the 90s and is one of the most widely used programming languages at the moment. Java was the original Android Development language; however, engineers have been switching to develop more recently using a newer language called Kotlin.

Kotlin is a simpler and safer language developed by Jetbrains and Google. The language's premise is it is more fun to write, safer to write, and people can start to write code much faster than using java. In 2017, Google announced that Kotlin was now the preferred and official Android Development language. from that day forward, Kotlin has become of the fastest-growing programming languages.

This study benchmarks performance differences between Kotlin and Java to give Android Developers more data when deciding whether to use one language over the other. This research aims to answer the following question:
What are the performance differences between Java and Kotlin?

CHAPTER 2. BACKGROUND

2.1 Android Operating System

In this study, Kotlin and Java executed on the Android Operating System will be benchmarked. The non-trivial background readers will need for this study is that the Android OS is a mobile operating system that executes on the Linux OS. Like Linux, Android's processes share resources between them, such as memory or CPU. Because of that, benchmarking the runtime of different algorithms on an Android device isn't trivial. There is a risk that other processes will use I/O and CPU while the algorithms are being executed. This is something that was taken into account during this study.

2.2 Android application package (APK)

When writing Android Software, developers write the code using C++, Java, or Kotlin. After the code is written, when developers want to see the outcome of their changes, code must be compiled, packaged for it to be installable and executable on an Android Device. This can be done in multiple ways, but the most common way is by using the Android Studio IDE. This IDE compiles and packages the code into an APK, a file that ends in the ".apk" extension. This study analyzes how much space this APK takes depending on the programming language used.

2.3 Language Differences

Android apps can be written using Kotlin, Java, and C++ languages. In this study, however, we only focus on seeing the performance differences between Kotlin and Java. This section will cover some of the main differences between both languages in terms of syntax.

2.3.1 Type declaration

(a) Java (b) Kotlin

Figure 2.1: Differences in type declaration (Part 1)

Like other modern languages, Kotlin is a language that does not require users to declare variables types. In figure 2.1, both sides are equivalent, but as seen in the images, Kotlin needs Var/Val to declare variables, and it does not require a semicolon at the end. The difference between val and var is, developers declare a var if the variable may change or is mutable, and developers declare val when it shouldn't be re-assigned, like adding final on a Java variable.

(a) Java (b) Kotlin

Figure 2.2: Differences in type declaration (Part 2)

As mentioned, Kotlin type declaration is optional, but if developers wanted to declare types to keep variables consistent, they can. Figure 2.2 shows the code equivalent to figure 2.1, but with type declaration.

2.3.2 Nullability

One of the things that make Kotlin very powerful in terms of error handling is nullability. By default, all variables (vars and vals) are non-null, meaning, Null can't be assigned to them by default. This is a powerful feature because this feature alone makes it difficult for developers to get null-pointer exceptions at runtime due to the compiler not allowing for null to be assigned to

```java
MyObject x = new MyObject();
x = null; // This is ok to do
MyObject y = null; // This is also ok to do
```

(a) Java

```kotlin
var x = MyObject()
x = null; //Error!
var y: MyObject? = null //This is ok
```

(b) Kotlin

Figure 2.3: Nullability example

any variable.

Figure 2.3 shows an example of this. In Java, developers can assign null to any variable by default. However, the Kotlin language does not allow for null to be assigned unless we specifically say it is nullable by using the "?" after the type declaration.

2.3.3 String interpolation

```java
String name = "Michael";
String language = "Spanish";
String message = name + " speaks " + language;
//if message is printed, it will say
//"Michael speaks Spanish"
```

(a) Java

```kotlin
val name = "Michael"
val language = "Spanish"
val message = "$name speaks $language."
//if message is printed, it will say
//"Michael speaks Spanish."
```

(b) Kotlin

Figure 2.4: String Interpolation example

Like other modern languages, Kotlin allows for developers to more concisely create Strings through String interpolation. In figure 2.4, both languages are creating the same String. However, due to String Interpolation, Kotlin does not require anything other than the "$¡variable_name¿" for the compiler to know that it just needs to replace that value with the ToString() method of the variable. On the other hand, in Java, we need to concatenate strings together using the plus sign, which makes it slightly more difficult to get right the first time.

```java
String concatMessage(String a, String b){
   return a + " " + b;
}
```

(a) Java

```kotlin
fun concatMessage(a: String, b: String): String {
   return "$a $b"
}
//Both Function Declarations are equivalent
fun concatMessage(a: String, b:String): String = "$a $b"
```

(b) Kotlin

Figure 2.5: Function Expression example

2.3.4 Function Expression Body

Function Declarations is another thing that is different between the two languages. As seen in figure 2.5, when writing a function in Java, we start by writing the return type of that function, followed by the function name, followed by the parameters. Kotlin, on the other hand, starts function declarations with the keyword "fun" (function), followed by the method name, followed by the parameters, which is finally followed by the return type.

One interesting thing to notice in figure 2.5 is, Kotlin allows us to omit the function brackets and the return keyword. Instead, we can assign a function using "="; then, the following expression is the return value.

2.3.5 Function Named Parameters

```java
String concatMessage(String a, String b){
   return a + " " + b;
}

String msg = concatMessage("a", "dog");
System.out.println(msg);
```

(a) Java

```kotlin
fun concatMessage(a: String, b:String): String = "$a $b"

val msg = concatMessage(a="a", b="dog")
println(msg)
```

(b) Kotlin

Figure 2.6: Function named parameters example

Another interesting difference between Java and Kotlin is named parameters. Like Java, Kotlin allows developers to name their function parameters, but unlike Java, Kotlin allows developers to name their input parameters when calling a function. This is useful when developers want to

know what parameters are given for what purpose when calling functions.

As seen in figure 2.6, we specify what we want parameters a and b to be assigned to on the Kotlin side. On the other hand, the Java side does not support named parameters, so we have to call it with the parameters in the order it was defined.

2.3.6 Function Default Parameters

```java
public String foo(String name, int number, boolean toUpperCase) {
    return (toUpperCase ? name.toUpperCase() : name) + number;
}
public String foo(String name, int number) {
    return foo(name, number, false);
}
public String foo(String name, boolean toUpperCase) {
    return foo(name, 42, toUpperCase);
}
public String foo(String name) {
    return foo(name, 42);
}
```

(a) Java

```kotlin
fun foo(name: String, number: Int = 42, toUpperCase: Boolean = false) {
    return (if (toUpperCase) name.toUpperCase() else name) + number
}
```

(b) Kotlin

Figure 2.7: Function default parameters example

As many Java developers are aware, Java allows developers to overload functions, meaning developers can write several functions with the same name as long as the signature (parameters) is different. This is very useful when developers want the same-named function to take different parameters. However useful it may seem, there are some flaws, such as writing the same function definition multiple times.

Kotlin addresses this issue through default parameters. What default parameters allow developers to do is that developers can create one function with all of the possible parameters it may take and what it should be if omitted by the function caller. A developer can call the one function with no parameter, some parameters, or all parameters without rewriting the same function with this in place.

As seen in figure 2.7, both sides are again equivalent, but thanks to default parameters, Kotlin only needs one function definition while Java requires four.

CHAPTER 3. EXPERIMENTS

To determine differences between Java's and Kotlin's Performance, multiple experiments were run to see the differences. These experiments attempted to measure runtime, memory usage, and lines of code needed to accomplish the same thing in both languages.

3.1 Runtime

Runtime is difficult to measure, especially inside the Android OS, because there was no way to tell the compiler only to execute what was being benchmarked. To ensure any CPU noise by the system was minimized, all algorithms were executed about 100 times; then runtimes were averaged to make sure noise was reduced.

3.1.1 Algorithms used to measure runtime

Runtime was measured using slow runtime algorithms. The difficult part was determining which algorithms should be used, how many to use, and how many times they should be executed to ensure no CPU noise is present.

The original plan was to run algorithms on different data structures, such as lists, graphs, and trees. However, after much experimentation, it was determined that measuring slow algorithms on lists would suffice.

The problem with measuring slow algorithms on other data structures is, the most common algorithms for graphs and tree algorithms are recursive for the most part. Thus, running these algorithms on huge graphs and trees causes issues due to the call stack, specifically, stack overflow

exception due to going too deep in these data structures. The algorithms chosen for this benchmark were the following:

- List Permutation Generator - $O(n!)$ runtime
- Recursive Fibonacci - $O(2^n)$ runtime
- Bubble Sort - $O(n^2)$ runtime
- Selection Sort - $O(n^2)$ runtime
- Merge Sort - $O(nlog(n))$ runtime

3.1.2 Datasets used to measure runtime

Finding datasets was another difficult task. This was difficult because if the datasets were too small, differentiating between the two app implementation results would be difficult. If, on the other hand, the dataset was too large, then the mobile device would probably not have enough memory to process it and crash.

Also, as previously mentioned, there is no guarantee that the device will focus all resources (CPU and memory) on one process, so the risk of having noise in the results is considered. For that reason, each experiment needed to run multiple times, and times needed to be averaged.

To determine the best dataset size, many set sizes were run to keep the processing time between 15-30 Seconds on the Kotlin implementation. Once the correct datasets were found, these datasets were saved as a .json file and used for all experiments. Each sorting algorithm had its own dataset, with datasets shared between the Java and the Kotlin implementation.

In the end, our datasets ended up being the following sizes:

- Bubble Sort: 33,000 element Integer array

- Selection Sort: 66,000 element Integer array
- Merge Sort: 3,500,000 element Integer array

Since sorting algorithms were not the only algorithms used for benchmarking purposes, values for both Fibonacci and Permutations had to be determined. In the case of those two, the following was executed:

- Fibonacci: Fibonacci(45) was executed
- Permutation: 10 element Integer array

To clarify, the permutation dataset's order did not matter since the dataset content does not affect the final runtime of permutation, unlike some sorting algorithms. So, the permutation was executed with a randomly created Integer array of size 10 for each app's execution.

3.2 Memory

To understand how memory was measured, we may need to understand an Android Application Package (APK). An APK is just the package file format used by the Android Operating System. In other words, once you write the code needed for the program, you compile it, and an APK is generated, which can then be recognized by the Android Mobile Device and installed onto it.

Although there are other ways to measure memory, only the APK size was measured in this study. This is an important benchmark because having two identical applications, each with identical algorithms, however, written in different languages, the expectation is that the APK size shouldn't be different, but the study will prove this hypothesis.

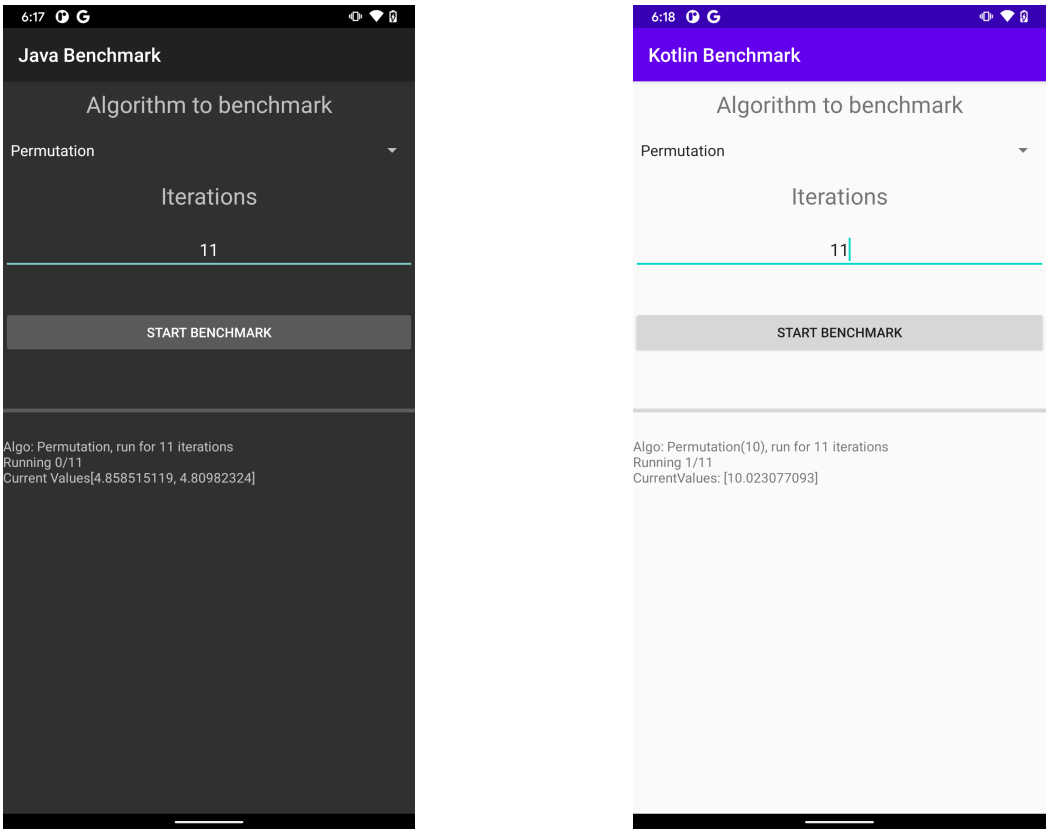
This benchmark can help developers decide when it comes to what language to consider because if either language makes too big of a difference, and devices that have limited memory are being used. Memory footprint may be a factor for users of an Android App.

3.3 Lines Of Code

Lines of code (LOC) was important to measure because the premise behind using Kotlin is that it takes roughly 25% less LOC than Java Juravle (2020). To measure LOC, this study identified identical tasks within the two languages, then measured the number of lines it takes to write these tasks. To be as subjective as possible, this study looked at the most significant differences where

- Kotlin $>$ Java
- Java $>$ Kotlin
- Java $\approx$ Kotlin
- Overall lines of code comparison within the apps

Doing that is important because it can show the different tasks where there are major differences between the two languages and the overall size. Given that Kotlin is supposed to avoid a lot of the boilerplate, this study wanted to prove that.



(a) Java UI (b) Kotlin UI

Figure 3.1: Android App Benchmarking User Interface

3.4 User Interface

For the benchmarking User Interface (UI), this study used the same UI for Kotlin as Java. The UI consists of an algorithm picker, which has the 5 algorithms previously mentioned. The UI also has an iteration input, which decides how many times to run each algorithm.

There is a progress bar that shows the user how many algorithms the app has executed and some information a developer would log to see the progress. This information is the current progress, current algorithm. Once the benchmarking app finishes, it will give the user some useful information, such as average, max, and min times.

CHAPTER 4. EXPERIMENT RESULTS

4.1 Runtime

Algorithm	Input	Java runtime (S)	Kotlin runtime (S)	Comparison
Bubble Sort	33,000	21.25	15.77	-34.76%
Selection Sort	66,000	16.99	15.92	-6.74%
Merge Sort	3,500,000	3.59	15.35	76.64%
Fibonacci	45	17.83	21.84	18.34%
Permutation	10	4.84	10.63	54.45%

Figure 4.1: Runtime Results Table

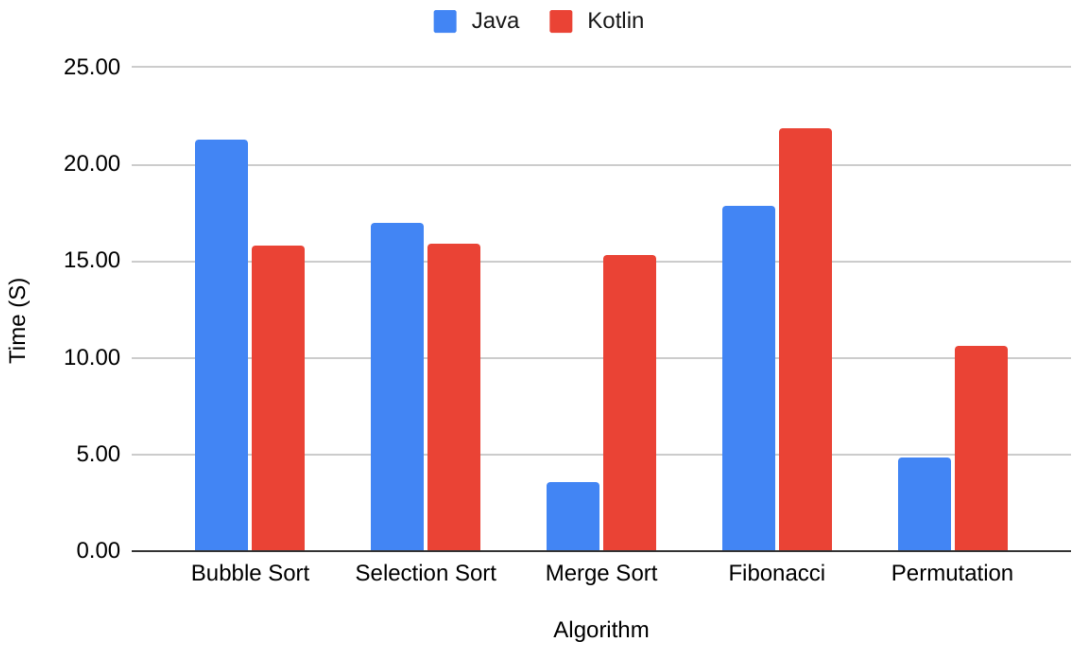


Figure 4.2: Runtime Results Chart

As previously mentioned, to get rid of CPU noise in the benchmark for each algorithm, each one was executed 100 times, then it was averaged. Also, to clarify, the time was only taken during

algorithm execution and not while generating or loading the data sets. The average was then taken, and the results are shown in figure 4.2.

As shown in figure 4.1, Java seems to outperform Kotlin in 3 out of 5 benchmarks. More noticeably in merge sort, which I believe may have to do with small implementation differences, meaning Kotlin uses similar but different data structures. Another factor that affects this may be that merge sort for Java works with the same list, while Merge Sort for Kotlin generates new lists in each recursive call.

Overall, Java outperforms Kotlin by about 20%. This is unsurprising given that Java has been around longer, which means it has had more time than Kotlin to evolve.

4.2 Memory

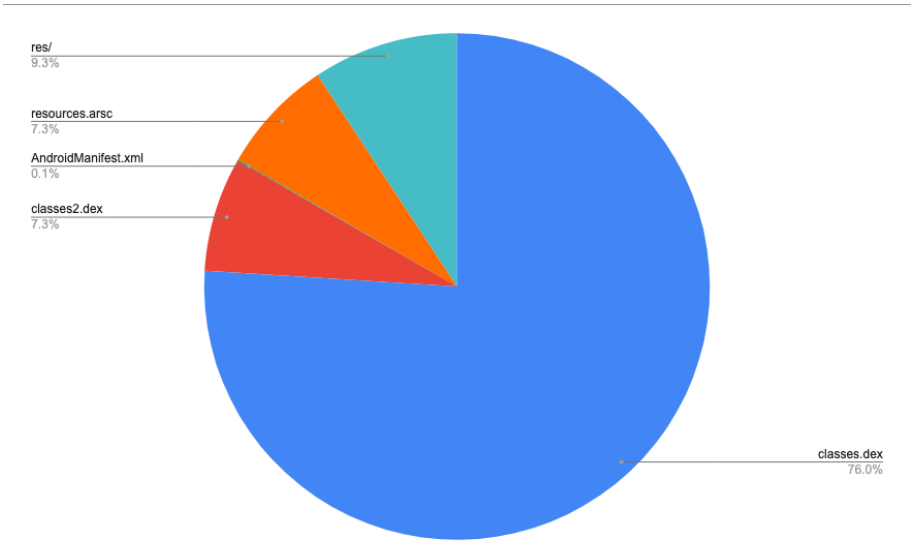


Figure 4.3: APK Composition

For this Study, memory footprint analysis was only done on the compiled Android Package (APK). A comparison between the Kotlin APK and the Java APK was made using the "APK Analyzer". Figure 4.3 shows what an APK is composed of. As seen in the pie chart, classes.dex is

File	Java Size (MB)	Kotlin Size (MB)	Comparison
classes.dex	2.60	6.10	57.4%
kotlin/	0.00	0.02	100.0%
classes2.dex	0.25	0.26	5.1%
META-INF/	0.00	0.01	98.8%
DebugProbesKt.bin	0.00	0.00	100.0%
AndroidManifest.xml	0.00	0.00	0.0%
resources.arsc	0.25	0.25	0.0%
res/	0.32	0.32	0.0%
Total	**3.42**	**6.97**	**50.91%**

Figure 4.4: APK file breakdown table

what makes up most of the APK and is the element that changes if we change languages. This is why classes.dex is the only part compared in the two APKs.

As seen in figure 4.4, the Java app's classes.dex file took up about 2.6MB, while the Kotlin app's classes.dex file took up about 6.1MB. This means that in a small application like the one used in this study, the size would decrease by about 60%. This indicates that if Android developers believe memory footprint could be a problem, they may want to consider Java over Kotlin.

4.3 Lines of code

Source File	All Lines Count	Source Code Lines Count
Kotlin	293	248
Java	351	305
% Smaller	16.52%	18.69%

Figure 4.5: Lines Of Code Table

As previously stated, Kotlin decreases the lines of code by 25% in comparison to Java Juravle (2020). As seen in figure 4.5, this study found the lines of code to be reduced by about 19%, which is aligned with the expectations.

This study indicates that if we assume Android Application Developers type both languages at about the same rate, using Kotlin would save about 20% of their time, which would allow for more work to be done in the order of months or years.

CHAPTER 5. REVIEW OF LITERATURE

Because this study focuses on benchmarking and comparing different languages, looking at other people's work in the benchmarking area was useful. To come up with my own experiments, I had to try and solve a few problems. The problems were benchmarking CPU, memory, and language comparison. These problems have been researched in the past, so my approaches were similar to others in this area.

5.1 Previous Work

Because Android has been a popular OS for about 10 years, benchmarking of Android has been done by researchers. But, because Android is an OS that can run web apps and mobile android apps, benchmarking can compare multiple combinations. For example, previous research has been done comparing web apps vs. java android apps Rösler et al. (2014), Kotlin Android Apps vs. Java Android Apps Gorelovs (2019); Schwermer (2018), and Native Android code vs. others Lin et al. (2011); Kim et al. (2012).

However, because technology tends to move forward, and tooling for Java and Kotlin should have improved since the last time they were benchmarked, it was important to run this study to determine if the languages were coming closer to one another in terms of performance.

Researching others' work on the Android OS was useful, but ideas for this study also came from other places. Other researchers benchmarked the differences between popular programming languages, such as c vs. c++, and Java vs. Python. Reading about the Android benchmarking

and the general benchmarking was very useful because most follow a very similar pattern: to run the same benchmark multiple times, then averaging out the results. (Gherardi et al. (2012); Prechelt (2000); Gorelovs (2019); Lin et al. (2011))

Lastly, Bull et al. (2000) created a benchmark suite for large scale or Grande applications. Their study was useful in considering other approaches to solving the benchmarking problem. Specifically, how to present the results, things to consider during the benchmarking, and the effect hardware has on benchmarking.

5.2 Lines of code

The two languages are quite different. Because of that, previous work on language analysis had to be researched and analyzed. Again, because Kotlin is about twenty years younger than Java, The expectation is that Kotlin is better, and researchers highlighted the benefits Kotlin has over Java.

The pattern found in these research papers was, Kotlin is safer due to nullability (highlighted in section 2.3), writing the same task in Kotlin takes about 25% less code, and Kotlin developers take advantage of over 50% of new features Kotlin has available, but Java doesn't. Flauzino et al. (2018); Banerjee et al. (2018); Mateus and Martinez (2020); Emomaliyev and Popov (2019)

5.3 Kotlin adoption in Android Development

Another topic that was slightly researched was the developer adoption of Kotlin. The main reason this part was done was to determine what developers saw in terms of performance once Kotlin was adopted and if there were any obvious differences between productivity due to the less LOC premise of Kotlin.

There are multiple takeaways from these researchers. For example, Oliveira et al. (2020) found that in general, developers believe Kotlin improves code quality, readability, and productivity.

They also found that the most common questions developers asked on stack overflow were related to new language features or how to set up their IDE.

Lastly, there was some other research done on the adoption of Kotlin by open source projects. Still, these studies found that in 2019, about 10% of Android Apps originally written in Java were transitioning to Kotlin. These studies also found that productivity seemed not to be affected negatively or positively when transitioning. Mateus and Martinez (2019); Coppola et al. (2019)

CHAPTER 6. CONCLUSION

Although this study does not cover every scenario or every algorithm, it does highlight a few pros and cons of both languages. This study indicated that Android developers might want to consider Java if they want their app to have a smaller memory footprint or a faster app. If, on the other hand, memory or execution time is not a problem, but they want to get more features done in the same period of time, then Kotlin is the language to consider for their app.

6.1 Future Work

As previously stated, these experiments were run on an app that was created specifically for this experiment. Although this process generally did work, if anyone decides to take this work further, they may want to consider using the Android SDK benchmarking tools highlighted in the Android Documentation. These tools were not used in this study because they were found after project completion.

BIBLIOGRAPHY

Banerjee, M., Bose, S., Kundu, A., and Mukherjee, M. (2018). A comparative study: Java vs kotlin programming in android application development. *International Journal of Advanced Research in Computer Science*, 9(3):41.

Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. (2000). A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):375–388.

Coppola, R., Ardito, L., and Torchiano, M. (2019). Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, pages 8–14.

Emomaliyev, M. and Popov, A. (2019). Kotlin and android development. *Modern Science*, (1):58–61.

Flauzino, M., Veríssimo, J., Terra, R., Cirilo, E., Durelli, V. H., and Durelli, R. S. (2018). Are you still smelling it? a comparative study between java and kotlin language. In *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, pages 23–32.

Gherardi, L., Brugali, D., and Comotti, D. (2012). A java vs. c++ performance evaluation: a 3d modeling benchmark. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. Springer.

Gorelovs, M. (2019). Comparison of android application development using java and kotlin programming languages.

Juravle, C. (2020). Busting android performance myths.

Kim, Y.-J., Cho, S.-J., Kim, K.-J., Hwang, E.-H., Yoon, S.-H., and Jeon, J.-W. (2012). Benchmarking java application using jni and native c application on android. In *2012 12th International Conference on Control, Automation and Systems*, pages 284–288. IEEE.

Lin, C.-M., Lin, J.-H., Dow, C.-R., and Wen, C.-M. (2011). Benchmark dalvik and native code for android system. In *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pages 320–323. IEEE.

Mateus, B. G. and Martinez, M. (2019). An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, 24(6):3356–3393.

Mateus, B. G. and Martinez, M. (2020). On the adoption, usage and evolution of kotlin features in android development. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12.

Oliveira, V., Teixeira, L., and Ebert, F. (2020). On the adoption of kotlin on android development: A triangulation study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216. IEEE.

Prechelt, L. (2000). An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29.

Rösler, F., Nitze, A., and Schmietendorf, A. (2014). Towards a mobile application performance benchmark. In *International Conference on Internet and Web Applications and Services*, volume 9, pages 55–59.

Schwermer, P. (2018). Performance evaluation of kotlin and java on android runtime.