**University of Brighton**

2015/2016

**Scala on Android: the-current-state-of-the-art**

**Project Report**

**Chris Howell**

Contents

[Abstract 2](#_Toc447636216)

[Measuring Success 2](#_Toc447636217)

[Introduction 3](#_Toc447636218)

[Why Scala? 3](#_Toc447636219)

[Java on Android 4](#_Toc447636220)

[Development Process 4](#_Toc447636221)

[Setting up the environment 5](#_Toc447636222)

[Using conscript and giter8 5](#_Toc447636223)

[SBT vs Gradle 6](#_Toc447636224)

[SBT Project Structure 6](#_Toc447636225)

[Memory overhead and APK size 6](#_Toc447636226)

[Reflection 7](#_Toc447636227)

[Scala on Android 8](#_Toc447636228)

[Immutability and Mutability 8](#_Toc447636229)

[Objects and Classes 9](#_Toc447636230)

[Case Classes 9](#_Toc447636231)

[Pattern Matching 10](#_Toc447636232)

[Options 11](#_Toc447636233)

[Futures and Promises 11](#_Toc447636234)

[Implicit conversion 12](#_Toc447636235)

[Traits 13](#_Toc447636237)

[The Akka Actor Model 14](#_Toc447636238)

[Akka Remote 14](#_Toc447636239)

[Java Interoperability 15](#_Toc447636240)

[Scala Type System 15](#_Toc447636241)

[Java and Scala Type Systems Objects and Primitives 16](#_Toc447636242)

[Generics 17](#_Toc447636243)

[Variance Annotation 17](#_Toc447636244)

[Bounded Types 18](#_Toc447636245)

[Implicit Parameters 18](#_Toc447636246)

[Partial Application 18](#_Toc447636247)

[Structural Typing 19](#_Toc447636248)

[Type Inference 19](#_Toc447636249)

[Anonymous Functions 20](#_Toc447636250)

[First Class Functions 20](#_Toc447636251)

[Higher Order Functions 20](#_Toc447636252)

[Type Functions 20](#_Toc447636253)

[Higher Kinded Types 21](#_Toc447636254)

[Operator Overloading 21](#_Toc447636255)

[Kotlin 21](#_Toc447636256)

[Comparison with Scala 22](#_Toc447636257)

[Type System 22](#_Toc447636258)

[Null Values 22](#_Toc447636259)

[Java Interoperability 22](#_Toc447636260)

[Developer Support 22](#_Toc447636261)

[Project Reflection 23](#_Toc447636262)

# Abstract

This document features a review of the viability of an alternative programming language as the basis for an Android application. I will present a test case, a review of the development process, an in-depth study of an alternative programming language in comparison to Java and a reflective review detailing my experience and opinion on the subject matter.

# Measuring Success

* A working test case.
* A review of the development process with a discussion of any issues that arose from this development period.
* A detailed discussion of the pros and cons of using Scala on Android with code examples and comparison to Java.
* A review/comparison of the Scala and Java type systems.
* A discussion of the programming language Kotlin
* A summary and conclusion stating my opinion on the future of Android application development.

# Introduction

This report describes my research into an alternative programming language for use on Android. I hope that my research will highlight the viability of using alternative languages when creating applications and the additional benefits the alternative language can offer in contrast to Java. I wanted to spend the next few paragraphs talking about the reason why I have decided on this project and what it means to me.

I picked this project because it is an area of personal interest to me. With Java being the first language I learnt in detail, and generally the first language I turn to when building new applications. As my experience in programming has grown, I have started to branch out into learning other languages, with these including PHP, JavaScript, Haskell, C++ and finally Scala.

With Android application programs on the increase I felt like my skills and experience have reached the point where I am now competent enough to create Android applications. Understandably I was frustrated to know that I would be forced to continue using Java when there are many other languages available as a computer programmer.

Fortunately the Java developers design choice of write once, run anywhere enables alternative languages that compile down into Java byte code to also run on Google’s version of the JVM called Dalvik. Any programming language that executes on the JVM, can also execute on Dalvik. In recent years the JVM has seen a growth in the number of languages that are capabal of

# Why Scala?

While Java is indisputably the current go-to language when developing android applications, I hope that my test case highlights the possibility that android developers have when creating applications. The android applications byte code executing on the VM increases the prospect of an alternative language that will one day dethrone Java. It would require no significant change in hardware/software as the VM is already well established.

The test code I have prototyped for this project implements Scala on Android to create a picture hiding application that uploads the picture to a server and removes it from the device. As I have stated in my discussion of the development process, while the initial set-up can require some initial configuring, once this stage has been complete you can take advantage of a less verbose language than Java.

While Scala is sometimes referred to a Java without the semicolon, this is a statement that could not be further from the truth. Scala is, in my opinion, currently one of the bests options for developers to blend object oriented principles with functional programming when writing code. Because it is neither a complete imperative language nor a purely functional language, it is essentially a new programming paradigm that redefines the rules when coding. It is an expressive language that provides a much richer type system than Java currently supports. It is a truer representation of the object oriented principles because everything is an object.

A strong type system like Scala's is an effective method of ensuring the code stays bug free during runtime, with most of the errors being caught during compilation.

# Java on Android

The most recent version of Java that has been released is Java 8 and with it provides some of the more common aspects of a functional programming language. This is a positive step for the language as a whole even if it is only provides features that other languages, such as Scala, have feature for multiple iterations.

What should be noted at the start of this study is that the currently supported Java version for Android is Java 7 which is now over half a decade old. In comparison to this the version of Scala available for use on Android is version 2.11.7, with this being the most current version of Scala. Immediately one of the advantages of Scala on Android is clear in the fact that if a developer chooses Scala over Java they are using the most up to date version of that programming language.

It should also be noted that there is currently a library for Android called RetroLambda. This library allows for the use of lambda functions when writing Android applications in Java code and then converts this Java 8 feature back to Java 7. This approach is more of a workaround to using Java 8 on Android. Google recently announced that Android N will support some features of Java 8, but at present this is not the currently supported Android version and is reported to cause issues with other libraries on Android. Android N is currently in Alpha stage, so will not see public release and widespread use for several months.

When discussing the advantages of Scala on Android in later sections it should be noted that while Java 8 is the most current version of Java, the actual comparison for fairness is against Java 7 as that is currently the version of Java supported on Android by Google.

The review of the advantages of Scala on Android will make references to new features implemented by Java 8 even though they are currently unavailable for use on Android, but again it must be stressed the best currently supported version of Java on Android is version 7, with many applications having been built in Java 6.

# Development Process

During my research stage of the project I came across one recurring theme in many of the experiences of other developers keen for an alternative to Java on Android. The recurring theme is the difficulty in setting up the development environment. From my own personal experience I can agree with this still being the case.

I am of the opinion that until the setup process has been streamlined to enable a developer to spend more time coding their application than configuring the environment, there will be no significant leap forwards in a new language taking prominence over Java. I believe this situation will remain the same until Google decide to throw their support behind a successor/alternative language to Java.

What is interesting and I will cover in greater detail in a later section is that JetBrains are currently working on seamlessly incorporating their Kotlin programming language into IntelliJ IDEA. This is an extremely interesting development as it’s a programming language written by programmers for programmers. It shares a lot of similarities to Scala in that it is a hybrid functional/imperative language.

It should be noted that I have only tested one alternative language on Android, in this case Scala, so I cannot state for a fact if it is easier or harder setting up the environment when using a language like Groovy. From reviewing some experiences by other developers of setting up the development environment to utilize Groovy the general experience seems to be slightly easier than Scala. Arguably because Groovy is an interpreted language means it has worse performance than Scala, but this falls outside the subject area for this report.

In recent years a large number of programming languages have appeared that run on the JVM, I believe this is in part to developers wanting to branch out from Java. The opportunity that is provided is that the JVM is already well established, so any machine capable of running the JVM or Java code is also capable of running any of these alternative languages. With the fact that the JVM is already a well-supported method of launching applications that are machine independent it is the logical approach for developers to use this as a base for building new programming languages.

The next few sections of this report will cover what is in my opinion the easiest method for setting up an android application that uses SBT to compile and launch the application.

## Setting up the environment

Ideally when setting up the environment it is advisable to first download sbt to use as a command line build tool for compiling and debugging your projects. Sbt is a flexible, powerful and easy to use build tool written in Scala for Scala projects.

One of the major advantages of using SBT is that it allows a developer to state the dependencies for the project and will then manage these by downloading all the required libraries. This approach saves you having to add the individual JAR files to the class path when importing Akka actors etc. Another reason why it is an advantage to use SBT is because it features incremental compilation, with this meaning that after the initial project compilation only classes that have been altered in some way need to be compiled again. This saves vast amounts of time and almost completely removes the issue of slow compilation for android apps written in Scala.

Along with SBT the Scala programming language needs to be downloaded and install onto the development machine. The latest iteration of Scala is version 2.11.7. After installing Scala the development machine also need to the Android SDK install along with the appropriate API level. You will need to correctly set the path the Android SDK or this will cause issues on both Linux and Windows operating systems.

## Using conscript and giter8

The easiest method for quickly setting up and compiling SBT-based android applications is via the command line build tool conscript and a giter8 template. It is possible to use the command line to create the project structure, this approach is more time consuming than downloading a template.

Conscript is a build tool that enables a developer to quickly download and build templates created by giter8 and other groups. This approach is the least confusing when building Scala-based Android applications. The main reason for this is that they quickly provide the developer with a basic working template without having to configure the dependencies etc.

Once conscript has been downloaded and install from the GitHub account it allows for the quick creation of new sbt-based android template via the command line. This project contains no code, but does help to quickly get a new project started without having to manually implement the basic project structure.

This approach is excellent for reducing the setup stage and for quickly getting a new app featuring Scala and SBT running with minimal effort.

## SBT vs Gradle

The standard android application is often built using the Gradle build system to manage the project dependencies. In theory these build tools are similar in that they allow the developer to explicitly state what dependencies the project requires and to download them during the compilation state.

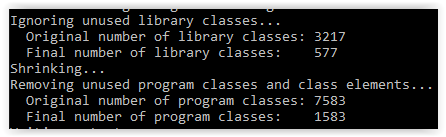
Of the two build tools, when using Scala on Android then SBT is the preferred choice because it features incremental compilation, which is an essential tool when compiling Scala Android applications. Incremental compilation allows for only edited classes to be compiled

## SBT Project Structure

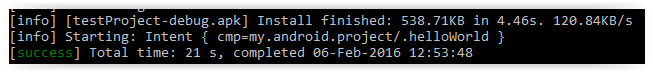
The basic structure of an SBT android project structures differs slightly from a more standard gradle build with the difference being that dependencies for the project are all placed in the SBT build file.

## Memory overhead and APK size

A topic of much discussion is that the android apk file requires all its dependencies to be included in a single apk file. Unfortunately, this requires the whole Scala language to be included into the apk file when it is created. While in the past this did cause the base apk file size to be exceedingly large, often reaching 8MB+ for the base project, this fortunately is no longer a real issue when using Scala.



One of the features of Proguard is that it can remove classes that aren’t needed when compiling the apk file. One interesting feature of SBT is that we also define the Proguard configurations inside our build.sbt file. By using Proguard I was able to reduce the APK by a significant amount to a much more reasonable size. As the above image shows the number of classes remove was in excess of 6000. This is a significant number of redundant classes that aren’t included in the final APK and helps reduce the memory overhead significantly.

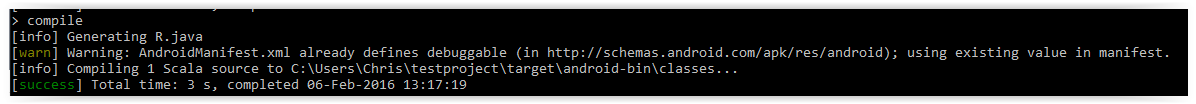


As you can see it was possible to have the base APK file for this project, including the Scala Library and Akka actors compiling to a relatively tiny 538.71KB. This is a more than reasonable base APK size and I believe it is approximately the same base size when creating a Java project. We can also see that the total time it took to compile the project on a clean build was 21 seconds. As Proguard caches some compilation information when you recompile the program the resulting APK is slightly larger unless you clean the cache before building the project again.

## Reflection

The main lesson learnt from the development process is that it is entirely possible to write fast and responsive android applications using Scala as opposed to Java. While the memory overhead of using Scala was a problem when developers first started utilizing Scala on android, this is no longer a major issue in the development phase as Proguard removes the redundant classes from the Scala library when the APK is being generated.

Another well documented issue is the compilation time required to compile the project, with this often being in excess of two minutes. Fortunately, this is another non-issue at this stage of creating Scala Android applications. Because SBT feature incremental compilation the only classes that are compiled again are those that have been changed in some way. On average I waited no more than a few seconds for a class to compile after I had made some changes.



Using Conscript and Giter8 to quickly create the template project structure was an effective way of quickly overcoming the lack of official Android Scala support from IDE creators like JetBrains and the Android O/S owners Google. It takes a matter of second to quickly get a Java-based android project ready for coding when using an IDE like IntelliJ IDEA, and using Conscript/Giter8 approach you can reproduce this base template creation in a quick and easy fashion.

I would say overall while it was initially frustrating to get a project up and running using Scala, once I had created my first project the subsequent projects were much simpler to create as the environmental variables had already been correctly set and the languages/ tools had been installed and configured.

It should be noted that I wrote all of the code for this project using Notepad++ with the Scala Plugin. When I did try and open the program using IntelliJ IDEA it complained that the android packages weren’t recognized. For me this wasn’t a problem as I didn’t find it an issue to write my code in Notepad++. To build and test my code I used the command line and SBT. None of the reported issues by the IDEA IDE were reported or caused the application not to run when I build it from the command line with SBT.

Again I must iterate for me this was not a problem as using a text editor and command line tool to build/run the application is a similar experience to coding Haskell applications.

# Scala on Android

Some examples are given below to highlight some of the options available using Scala.

* Immutability
* Objects and Classes
* Case Classes
* Pattern Matching
* Option type
* Implicit conversion
* Traits
* Futures and Promises
* Akka

This list provides some of the features available to Scala developers when creating android applications. This may change in time as Java borrows more and more from Scala. At its core Java will always be an imperative and verbose programming language as this is deeply ingrained in the makeup of the language.

It should also be noted that while this list tries to capture the essence of the power that programmers can harness with Scala, this list should not be considered exhaustive as the language is large and extensive. All the examples shown are runnable on my Samsung A3 phone and are targeted for the Android API Level 16.

## Immutability and Mutability

One of the real cornerstones of functional programming is the approach to global state. The imperative approach is to have large numbers of global variables that help define the changing state of the programming. This approach can often lead to unexpected results when data is effect in some unforeseen way and this can often lead to runtime errors.

For example, Java requires a developer to write lots of codes usually using if/else statements to check that the data matches the expected result. If the developer doesn’t include these checks at the appropriate points in a program, then the result will often be runtime errors. One method of ensuring shared data isn’t corrupted is by utilizing synchronized methods, but again this can lead to verbose and hard to understand code.

A functional programming languages approach is to reduce the amount of global variables to the absolute minimum amount required, with this often being none at all. This approach to changing state results in less runtime error as each method created can be consider a pure function in that it only has control over local variables.

Scala provides the choice of allowing global variables to effect changing state in a traditional imperative manner. This is not the preferred choice when coding in Scala and favouring immutability is often encouraged by the community unless absolutely required. When immutability is applied to code then generally it is less buggy and unlikely to crash during runtime due to unexpected data.

The use of mainly immutable data is a community driven effort and often the encouraged method of creating applications in Scala in a more functional style. Martin Oderskey himself has stated that mutable data is a good idea, but only when used in the appropriate situation. Mutable data can be global or local to a function in Scala and when used locally it can be beneficial to the developer.

This is another area where the hybrid nature of Scala shines as it is easily available for the developer to choose between using mutable or immutable data when needed. The same could be said of Java that it does feature immutable data via the use of the final keyword, but this is often not the encourage method by the community and is better suited to special circumstance, such as setting the frame rate of a game developed in Java.

## Objects and Classes

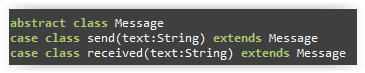
Classes work in a similar fashion to Java in that they are essentially the blueprint that contains all the methods accessible to that class. These blueprints can then be reused multiple times to create object instances from. The main difference between Scala classes and Java classes would be that Java favours global variables and mutable state, while Scala classes often feature no global variables and favours immutability. This means that Scala classes are likely to be less buggy during runtime as functions only effect local variables.

While Scala classes are similar to Java classes where these two languages do differ is that Scala allows you to create a singleton object. The closest comparison to this you can find in Java in declaring every method and variable inside a class static to essentially create a static object. Scala objects allow you to create a single instance of the object by using the object keyword instead of the class keyword.

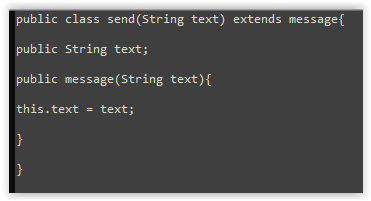
## Case Classes

Case classes are classes with attributes that are created in a clear and concise syntax. They offer a powerful and flexible way of quickly defining case classes that are easily combinable with pattern matching.

In the image below and abstract class called Message is defined which features two case classes that both contain String values. At a later date it will be a trivial matter to revisit and revise these attributes as required. While the value of this approach may not be immediately apparently the Akka actor model supports pattern matching on the receive function being called. With this approach I will be able to apply pattern matching to the message case class to determine the appropriate response.



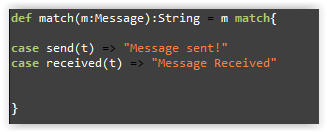
The next image shows the more verbose approach employed by Java, with this implementation still missing the code for the abstract Message superclass and the Received class. With the code included for all three required classes the verbose nature of Java becomes apparent as significantly more code is required to achieve the same result as Scala case classes.



## Pattern Matching

In addition to feature imperative if and else statements Scala also features pattern matching, with this being the more traditional approach to selection in functional programming languages. Pattern matching is often combined with recursion in functional programming; this approach is an alternative to iteration.

One powerful option that Scala allows is the ability to pattern match against classes, with the image below showing an example of this. The function takes an instance of the Message class that was defined earlier in the code and then uses pattern matching to determine which variation of the class was received as the input.



Using pattern matching to decode case classes is common practice and provides a flexible way to write functions that react according to the concrete instance of the abstract class.

While I have mainly focused on discussing pattern matching on types you can also pattern match on input data such as numbers or characters.

Java does not currently support pattern matching, with the closest available option to a developer being via the use of a switch statement, but this is extremely limited as it only allows for the switch choice to be based on an integer input, with this being extremely simple implementation in comparison to Scala’s ability to pattern match on case classes and abstract data types.

## Options

One useful feature of the Scala programming language is the use of Option types. This feature works in a similar way to Haskell’s Maybe Monad resulting in the removal of the null pointer exception from occurring.

This is a common exception that often occurs in Java program development when an unexpected null is returned when data is required. As returning null values are heavily incorporated into Java programs, this approach has carried over into Android application development and has created the same issue for Android developers.

Scala excludes null values and instead implements the Option type. The Option type allows for the possibility of some data to be return or none. This approach generally results in less error prone code being written that has null pointer exceptions appearing unexpectedly. It also removes the need for try/catch blocks, with this resulting in cleaner code.

As of Java 8 option types are available in the Java language as well. They generally work in a similar way to Scala Options, but one of their failings is that they don’t force the developer to ensure there is a procedure in place in the event of no data being return, with this result in an exception being thrown.

## Futures and Promises

One interesting piece of functionality provided by the Scala language is the use of futures. A Scala future is a non-blocking piece of data that has yet to be determined. By stating that a value is a future, the program can continue to operate until the value is actually needed. By using futures the code is non-blocking as it isn’t required to wait for the value to be resolved until it is absolutely needed.

Futures and promises in Scala allow for the developer to create non-blocking code. While Java does also feature futures, again they feature a more verbose approach than Scala and when retrieving the data can often result in the program blocking while waiting for the data to be returned.

Scala on the other hand allows the developer to firstly define the code to be executed by the future and to then explicit state the successful execution result and the failed execution result.

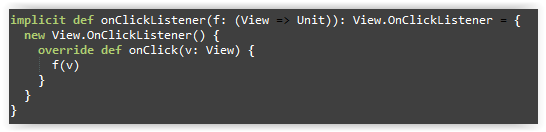
Futures and promises in Scala can be combined in an interesting fashion to create non-blocking code. The network connection for the test case, which if written in Java would commonly require a class to implement asyntask, although this is not a requirement it is often encouraged by the community as the ideal method for network related tasks to ensure the UI thread isn’t blocked while network related tasks are executed. The use of futures and promises allows the developer to avoid having to create and define classes that implement asynctask.

Examples of this can be seen in the test case where network related tasks were required. Rather than creating an inner class that implements callable to return a value and then using the get method to retrieve this value which may result in blocking while the task is completed, by using futures and promises the tasks was executed and if successful the required result returns only when it has been successfully executed, resulting in no blocking taking place.

## Implicit conversion

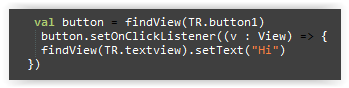
## 

This image shows one way of creating a new button in an activity and setting the on click response for when the button has been pressed. Unfortunately, this approach is still verbose and so similar to Java it is essentially Java without the semi colons. Fortunately Scala does provide an interesting method of reducing the verboseness of creating button a button on click event.

[15]

Scala implements implicit functions that take implicit parameters as inputs. In this case the function takes an input value f, which is itself a function that takes a view and returns a unit. This approach allows for the creation of an implicit definition for the button on click event. In this example the button is passed a lambda function for the button onClickListener. This means that only one instance of the button listener needs to be defined and then whenever the developer wants to create a new button click event they can use this.

This approach reduces the amount of code required by a significant amount as it is no longer required to define the actions of a new listener for every button. Ideally, implicit click event listener can then be placed into a separate trait; this will allow it to then be utilized across the whole program when and where the functionality is required.



As you can now see all we need to do now when we want to define the buttons behaviour is write the code instead of the boilerplate required to create a new on click listener for each button whenever we want to define some behaviour. The implicit function is available to any button within scope and is ideally placed inside the trait so that it is reachable wherever needed in the project.

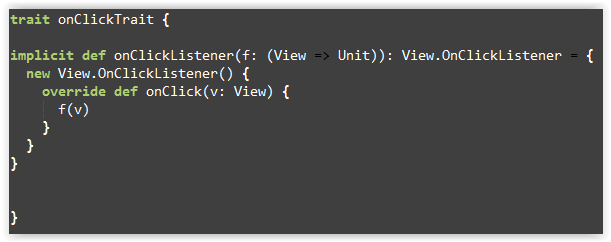
## Traits

A Scala trait works in a similar manner to a Java interface, but differs in the level of abstractness that is offered. For example, in Java an interface will only contain the method names, the method inputs and the return type. This is probably as abstract as is possible and is Java’s answer to how to implement multiple inheritance.

By only allow to directly inherit from one superclass at a time, the only way it is possible to simulate multiple inheritances is by incorporating interfaces and having to write the concrete methods in each class including that interface. This highlights the verbose nature of Java as a concrete implementation of the interface needs to be coded whenever a class implements them. Scala traits allow for a concrete method implementation to be defined and utilized wherever needed.

Scala traits are similar to interfaces in that they contain the method stubs, but differ in that they also allow the code for these methods to be given a concrete implementation. Then using the “with” keyword a Scala Class can implement as many traits as required. By using this approach to inheritance Scala Classes can include multiple traits in a single class and reuse their provided functionality whenever needed. If method name clashes are detected between the traits that a class is implementing, then they are given priority in order of right to left, so this is something a developer should always consider when including multiple traits in a class.

It should be stated that in Java 8, which is not currently supported on Android, that interfaces do now allow for concrete method implementations for the methods contained inside that interface but they still do not allow for global variables, unlike Scala traits which allow for trait wide global variables to be assigned to a trait. This is a positive step forwards for Java in reducing the amount of code required when using an interfaced, but it is not currently common practice when developing Android applications in Java.



As you can see I have now removed the implicit button click function and placed it inside its own trait. This means that even though I no longer have this method inside the class displayed below, I can still attach this to the buttons contained inside this class because I have included the onClickTrait. This approach would be impossible to implement in Java because you could only extend one class and would be forced to write an interface without the concrete implementation of the functionality, with this forcing you to then write that functionality inside any class that uses the interface.



## The Akka Actor Model

While languages like Java feature thread-based concurrency Scala omits this approach in favour of the actor-based concurrency system. One of the major drawbacks of using mutable state and thead-based concurrency is that the developer has to spend large amounts of time reasoning about how to reduce issue such as deadlock or thread starvation from occurring.

By using the actor model approach to concurrency Scala allows the developer to write concurrent, non-blocking programs. The actor model features actors that carry out some predefined functionality. The actors each have an inbox of received messages and process these messages sequentially.

An actor has one main function that needs to be overridden. This function is the receive function which handles how the actor response to different messages. Combined with case classes this is a powerful system for handling concurrency.

Unfortunately, using Akka on android can be a painful experience for the developer as the tree shake required to allow Scala to work on android often removes classes required by Akka, with this resulting in hard to trace errors. During the development of the test case the original approach was to use Akka for concurrency, but this approach proved hard to implement and eventually was removed as using futures and promises was the easier to implement approach to running concurrent operations without the need to configure Proguard to support Akka.

If Scala was to become a more dominant language for use in creating android applications it would be better if a more lightweight or better integrated version of Akka were created. In a later section Kotlin’s approach to concurrency will be discussed.

## Akka Remote

The Akka framework allows actors to be remotely distributed across multiple physical locations via the use of remote actors in a peer-to-peer fashion. This is one of the more complicated aspects to setup on Android as it requires merging multiple Akka JAR file reference configurations into a single configuration file.

While attempting to implement Akka remote in the test case it was an unsuccessful venture as it is simply not a good fit in its current state for android development. In the future this may change as the benefits of the actor model are apparent, but the amount of effort required by the developer to correctly configure this approach could be consider a waste of time as it requires a fair amount of effort to configure when arguably it is not the most suitable means of enabling a connection between two devices.

## Java Interoperability

As both Scala and Java execute on the virtual machine and Scala has often focused on providing interoperability to Java. Scala and Java classes, while differing in the amount of syntax required, when compiled to bytecode are essentially the same. This approach allows a developer to freely mix Java classes with their Scala classes.

As Java is such a well establish language with a huge number of libraries that provide a huge variety of functionality, it is a huge advantage to Scala that the developer can at any point simply import a Java library and use that functionality in their program without having to find a suitable Scala alternative or write their own implementation of the Java library.

When programming in Scala on Android this is a useful feature as it means that if the developer runs into a problem they are having trouble solving in Scala, they can simply import the required Java library and use that. The same is true of Android specific libraries as they compile down to the same byte code they also combine well with Scala.

This interoperability works both ways. A Scala class can extend a Java class and override any of the methods it wishes with more elegant method implementations. This Scala class can then be extended by a Java class without issue and provides the functionality of that class but written in a more functional Scala style.

As previously stated Scala features traits in place of interfaces and are more flexible in allowing the developer to define methods and values inside these traits. These traits can be combined with Java interfaces and then implemented by a Java class. This approach provides a good workout to Java lacks of multiple inheritance.

# Scala Type System

The true advantage of Scala comes in its rich and extensive type system. One of the keys aspects for why Scala could be a much more effective language when programming Android applications is the extensive type system it provides. In this respect Java simply cannot compete as its type system isn’t as expressive in comparison to Scala's type system. While Scala allows for clear and elegant use of its type system, Java has a verbose imperative approach to its type system, with this often resulting in less elegant type declarations and code.

While Java 8 has certainly made a move towards implementing some of the most common examples of the functional paradigm, but at best these are simply the most common features of any language that has functional programming features. One example would be that Java has only just introduced the lambda function as an input for a method, while even C++ has featured lambdas since version 11.

The java type system is built around everything being either a class or a primitive type and even in Java 8 no fundamental change has taken place to this idea. Take for example the lambda function, while on the surface it looks like you are just defining a function that is executed, in reality this code is converted into a class before being executed.

Scala has a much bigger and richer type system than Java. Scala provides a truly flexible type system that has much greater depth than Java and allows a skilled developer to create code of a much higher quality than Java. While Java does implement many of the same type system features as Scala they are implemented in a verbose manner and have been added on top of the language longer after its initial design period, with this meaning that while it does support these type related features they are often Some of the features of the Scala type system are listed below.

* Generics
* Variance Annotation
* View Bound Type
* Implicit Parameters
* Partial Application
* Structural Types
* Type Inference
* Type Functions
* Lambda Functions
* Higher Order Functions
* Higher Kinded Types
* Operator Overloading

# Java and Scala Type Systems Objects and Primitives

Both Java and Scala are statically type programming languages that compile before execution, with most errors being caught during the compilation process. At this most high level view they share many similarities as programming languages, but when reviewing the class hierarchy and implementation of the various types, the differences between the languages become clearer.

Java as a programming language features two primary types, with these being objects/classes and primitive values like integers or characters. All classes in Java extend from the Object class which sits at the top of the Java class hierarchy. While primitives are treated separately, they are not objects, but simply values. These values are available either local to a method or globally to a class. One point of interested is that Java primitives because they are not objects, they cannot be treated as generic type parameters unlike Scala.

Scala is a truer representation of a pure object oriented programming language in that everything is an object, with this including classes and what would be considered primitives in Java. Located at the top of the Scala object hierarchy is the object Any. Every object in Scala extends from the Any object and it enables them access to a universal set of methods and operators.

Located directly below the Any object in the hierarchy is the AnyVal object. The AnyVal object enables what would be consider a primitive in Java to extend from the AnyVal object. This approach means that values which are considered primitives in Java and not actual classes are treated in Scala as objects with these objects useable as type parameters. By extending AnyVal a developer can create their own value types.

Next to the AnyVal object in the Scala hierarchy is the AnyRef object. This object allows the interoperability that Scala provides with Java. It is at this point in the Scala object hierarchy that the Java class hierarchy is combined with the Scala class hierarchy. It is essentially Scala’s top level object superclass that all classes extend from similarly from how all Java classes extend from the object superclass.

Of special note is what class sits at the bottom of the Scala hierarchy, with this being the Nothing class. The Nothing class extends all other classes and enables the developer to, for example, define a list that contains the type Nothing. This list would then allow you to at runtime place any of the values classes that Nothing extends from into the list.

# Generics

Scala supports the creation of generic type parameters, which are particularly useful for creating generic collections classes. Java also features generic types, but the syntax is slightly more verbose and less elegant than Scala because of the better type inference provided by the language.

Generics are useful when writing types or classes in which flexibility of the type is needed. For example, a singly linked list in Scala will feature a generic type signature, with this enabling you to create an instance of this class, but modify the input when and where needed. For example, the list could be created and passed an integer as the concrete type in place of the generic type. All the concrete methods of the linked list will then be available, with these ranging from removing elements to adding them.

The flexibility of using generics when creating classes and collection classes is that the input can change, but the class will act in a similar fashion regardless of the input type. A good example of this using the previous generically defined linked list is that the input type could be a string or a char and still continue to operate in the same manner. The list could even contain collections of custom created types.

As previously stated Java has in fact since Java 5 supported the creation of generic types. It is of note that in comparison to Scala the Java generic types are consider first –order, while Scala generics are higher kinded, this will be covered in more detail in a later section.

These generic types work in a similar fashion to Scala, but as stated require a slightly more verbose and less elegant approach to creating generic parameters. Scala generics parameters are more flexible in that they allow a simpler and easier method of defining the variance of the generic type via the use of the use of a simpler variance annotation when creating the generic parameters.

## Variance Annotation

Scala features declaration-site variance, with this meaning that the variance of a type is stated in the type parameter. For example, we have an input of type [T], in the type signature for this type we want to state that any input of type [T] is a subclass of type [C]. Using Scala declaration-site variance we can then explicitly state that type [T] is a subset of type [C] by adding a plus symbol. So now when stating the input of this function is [T] the developer would write [T+]. This means that any input of type [T], is a subclass of another type.

A better example of this would be if we created an abstract class called car. We then created two classes called Toyota and Honda, with both of these classes extending the abstract car class. Now when creating a method with the generic type T with the + symbol attached to the T symbol we can pass this function an instance of Toyota or Honda because both of these classes extend the abstract Car class.

By using declaration-site variance the developer is provided with an elegant method of defining if a generic type is covariant or contravariant.

## Bounded Types

In Scala using bounded types a type can be specified as having some relation to another type. For example we have type A and type B. Using bounded types the developer can specify to the compiler that there is a relationship between type A and type B. This allows functions to have a certain type as an input, but then treat that input as if it is a different type to the input type.

This relationship between two types can be expressed using the symbol <: for upper type bounds and >: for lower type bounds. While this might seem like a simplified method of expressing relationships between two types, it is in fact a flexible and powerful tool for declaring relationships between types in the function parameters. So for example, if the developer needs to state the relationship is that the function input type A is related in some way to type B, then the developer can simply write in the function parameters as A <: B. This means that when providing the function with its input it can receive an input of type A, but treat it as if it’s an input of type B.

Bounded types in this manner are how Scala provides it’s interoperability with Java. Using bounded types the Scala compiler can treat Java classes and code as if it is Scala code. When applying the example given to Java and Scala classes how works becomes clearer. By stating that the input type is type A, in this case a Java class, and then stating the relationship is <: to B with B being a Scala class, the Java classes is treated as if it was in fact a class written in Scala.

## Implicit Parameters

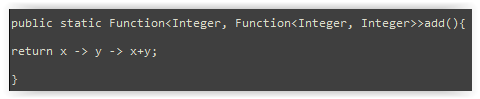


## Partial Application

Another common feature found in functional programming languages is partial application of functions and Scala also incorporates this functionality into the language.

A partially applied function is a function





## Structural Typing

Scala implements a type safe functionality called structural typing. In declarative language this is known as duck typing. The common description for duck typing is “if it walks like a duck and quacks like a duck, then it’s a duck”. Duck typing in a dynamic language can often result in runtime errors as the object being assumed to be a duck isn’t checked until the required method is needed.

The advantage Scala provides when using structural typing is that the compiler can ensure that at compile time that the structural typing functionality implemented is type safe.

Structural typing allows the developer to define a function that has an object as a type parameter that contains a certain method or methods. You can then pass this function an object and as long as that object contains that method then the developer isn’t required to explicitly state that the object has some relationship with that function. If you pass an object to that function that doesn’t contain the required method, then the compiler will flag this during the compilation phase.

Structural typing allows for a flexible approach to programming in that it allows for the object methods to determine if it is valid for use with the defined function rather than if it has the correct inheritance to support use with that function. Because Scala is a type safe language it provides a better interface for using structural typing than dynamic languages as the compiler will flag issues, while in a dynamic language the error will happen at runtime if the object doesn’t contain the appropriate methods.

## Type Inference

Scala features an advanced compile time type inference system. Advanced type inference like Scala provides lets the compiler calculate the type signature of variables. One of the benefits of type inference as good as Scala’s, is that nearly all bugs in the code are caught when compiling and not during runtime. A strong type inference system reduces the amount of required code as obviously type values don’t need to be stated by the developer.

Val x = “String”

String str = “String”;

Java in comparison requires the developer to explicitly state the type of the variable as the compiler isn’t as strong at type inference as the Scala compiler.

While the example given is a simple example, when considering the use of generics and variance annotation, the Scala compiler is better equipped to handle type inference. Similarly to when initializing new

It should be noted that even though Scala does offer good type inference it is still possible for the developer to explicitly state via the use of annotation the type of a val.

Unfortunately the Scala compiler cannot infer the types in recursive functions.

## Anonymous Functions

An anonymous function, also called a lambda function, is a function that can be defined in source and then passed to a function that requires a higher order function as one of its inputs. They are extremely useful as they allow you to create functions for one time use and that are defined where and when you need them.

Filter for example takes a function as an input and then returns the resulting values. Scala allows the developer to define a type function, which could be passed to the filter function as its input, but you could pass a lambda function, with this being the preferred method when the function is for one time use only.

## First Class Functions

One key aspect of a functional programming language is how functions are treated. In a pure functional programming language a function is considered first-class, with this meaning the function is treated as just another data type. These functions can then be defined as a value and passed to functions and returned from functions just as if they were normal values. First-class functions are at their most powerful when combined with higher order functions.

Both first-class functions and higher order functions are supported by Scala. Java on the other hand doesn’t have native support for first-class functions. It does allow for the simulation of first-class functions, but this approach is a verbose work around to achieve the same effect that is easily available for developers when using Scala.

## Higher Order Functions

A higher order function is a function that takes a function as an input and returns a value or a function that returns a function as the output instead of a value. There are plenty of examples of higher order functions in the Scala programming language as it provides a host of built-in functions such as filter or map which take functions as arguments and return values.

As already stated first-class functions are available for use in the Scala language. These functions can be assigned to values and then passed as arguments to higher order functions or even returned as a function from the higher order function.

Scala the programming language contains a host of predefined higher order functions such as filter, map and fold etc. One of the key advantages to using a language that comes with built-in higher order functions is that the developer is only required to supply the logical aspect of how the data is supposed to be manipulated, often via the use of a lambda or first-class function, without worry about the lower level details of how these actual operation is implemented.

## Type Functions

When defining a type signature in Scala, we often see functions being passed as type parameters. One useful feature of the Scala type system is that the developer can actually define the function as a type and then when declaring a type parameter that is actually a function, the predefined function can be passed as the input. This is useful when multiple functions take the same function as a type parameter, as this function only needs to be defined once and can then be passed around to whatever input requires it. Type functions are an example of Scala’s use of first-class functions.

Java similarly supports type functions, but again in a less elegant manner than Scala.

## Higher Kinded Types

## Operator Overloading

One interesting feature that Scala’s type system implements is the use of operator overloading. Operator overloading allows a developer to define a symbol as the name of a function. While the advantage of this may not be immediate apparent, it is a flexible system as it allows developers to still use a symbol-based approach when writing their functions.

For example, the developer creates a new object type called T. Objects of type T are in some way comparable to each other, although this has not currently been coded by the developer. Using implicit conversion the developer can then write a new function that takes two objects of type T and in some way compares them and returns the result to the developer. Using operator overload the developer can name the implicit function +.

Now when the developer writes the functionality that requires two objects of T being compared to each other the only requirement is using the + symbol. The compiler will search for and conclude that the default + symbol function does not know how to compare two objects of type T, but an implicit function has been written for using the + symbol to compare objects of type T.

It should be noted that while operator overloading is a flexible system for defining symbol-based functions, it can often lead to some frustration. In some cases it can make the code harder to understand to other developers as in some instances using keywords is easier to reason about and understand than using symbols, this is especially apparent in larger programs which feature multiple operator overloads.

# Kotlin

While Scala is evidently an extremely powerful and flexible language at present it has only found a small corner of the android development market to occupy. While the test case highlights the strengths that the language can bring to Android application development, without real support from Google it will be extremely difficult to find widespread use. Currently Scala on Android requires tweaking by the developer and this can have an impact on productivity, with hard to find errors when proguard’s tree shake removes a required class.

Kotlin is a relatively new language, with version 1.0 only recently being released, that is created by Jet Brains the creators of Intellij IDEA IDE. It is another language that runs on the Java Virtual Machine. This development is extremely important as not only do JetBrains offer easy integrations with their IDE and Kotlin, but also because Kotlin implements many features found in Scala, the language syntax offers a similar level of expressiveness.

While still being a relatively new language, having only just released version 1.0, it is an extremely interesting development. Kotlin is a language with similarities to Scala and could be considered Androids answer to Apple’s Swift language. Both language share a syntactic similarity with Scala and implement a large number of the features found in Scala.

One of the more important aspects of this development is that Android Studio is based on Intellij IDEA IDE. The reason this is important is that Jet Brains are providing as easy integration as possible with their IDE and Kotlin. One of the biggest barriers to using Scala on Android is the lack of support from Google, with this requiring the developer to spend time tweaking the project structure to support the use of Scala on Android.

## Comparison with Scala

As stated Kotlin and Swift both share more than a passing resemblance to Scala, both syntactically and in their type systems. Both Scala and Kotlin feature great support for writing applications that utilize functional properties over more imperative languages.

While Kotlin does have an expressive type system it features some features that Scala doesn’t support, while supporting other features that are absent from Scala.

## Type System

Its type system, while being expressive, is not as capable as Scala’s type system. This may change in subsequent versions as new features are implemented into the language and others are deprecated. As Kotlin’s type system matures it may reach a similar level of expressiveness as Scala.

## Null Values

Kotlin has a different approach to dealing with null values than language likes Scala and Java. While Scala features option types for handling values that could be null, this type is simply a wrapper around a value and can still return null values.

Kotlin by default doesn’t allow null values to be returned.

## Java Interoperability

One of the main aims of Kotlin is to provide seamless integration with Java and all its libraries. One interesting way that Jet Brains have implemented this into the Kotlin language is by having their IDE automatically convert Java source code into Kotlin code. This is an excellent approach as it allows developers to quickly convert old Java code into Kotlin syntax without having to refactor all the code from one language into another. This ease of use when switching from Java code to Kotlin is a

## Developer Support

As previously stated the developers of Kotlin are the same group that created the IDEA IDE, which is the base for Android Studio. Google dropped support for Android programming on the Eclipse IDE and are supporting Android Studio as the primary IDE for Android application development. Because of this and the ease of converting Java class files into Kotlin files, there is a strong possibility as the language matures, more developers will choose Kotlin over Java when creating their applications.

Because Google have chosen to support Android Studio as their primary IDE of choice when developer android applications they could potentially choose to also support Kotlin as an alternative to Java and in doing this reduce the setup burden required by a developer who wishes to use an alternative JVM language for app creation.

# Project Reflection

Generally the outcome of this project and the progress made during the study of Scala on Android was an enjoyable and educational experience. With this involving the creation of a working test case to a satisfactory standard and a report providing an in-depth discussion of the Scala language features, of its type system and of the potential Kotlin has at being the successor language to Java on Android.

It should be stated that there was a requirement to alter the application from a messenger app that made use of remote actors in favour of a picture hiding application that utilized PHP and a more traditional client/server relationship for moving data over the network. The main reason for this was the difficulty in setting up remote actors and akka actors on Android for establishing a communication between devices.

Each Akka jar contains a reference configuration file that contains the various configurations for each of the Akka jars. Unfortunately, when Proguard performs the tree shake to remove unneeded classes and methods, it also removes the reference configuration file for the Akka remote jar. One workaround that was attempted was combining both configuration files into one large file and pointing SBT towards this configuration file. After this the program did compile and attempt to send remote message, but unfortunately no communication could be established between the nodes. At this point it became clear that Akka Remote in its current form is not a good fit for Android development.

Using Proguard is vital when developing Android applications written in Scala. Its tree shake to reduce the number of classes and keep the APK size to a minimum is vital to reducing the memory overhead. Unfortunately, this also leads too hard to locate issues when attempting to utilize libraries like Akka remote actors, as vital classes that are required by these libraries can unfortunately is remove in the tree shake.

While I do believe that given more time I could have implemented remote actors, some developers have had success, I felt that I would be better spending my time on creating a more complete application, with good backend functionality and a good UI experience for the user.

I was able to implement the actor model inside my application to handle the internal concurrency. This highlights that even though the Akka framework is generally more suited to server applications, the actor model approach to concurrency is still a good fit for Android app development. As Android applications require good use of concurrency, the actor model provides an excellent method for creating concurrent non-blocking

Ideally in the future a more lightweight version of the actor model will be made available for use in Android development. With good support from Google using the actor model for concurrency could become the new standard for writing concurrent Android applications. In my opinion this is the likely future of concurrency on Android, but only if Google chooses to support it.

While I do still stand firm in my belief that Scala is a much more expressive and powerful language than Java, if I was to state my opinion on the future of Android application development I would place Kotlin as the most likely candidate to provide a true alternative to Java for Android development.

Kotlin for example has a relatively small size when included in Android applications, with this being the clear winner of the memory overhead provided by Scala on Android. Because of the smaller base library size the developer doesn’t require Proguard or its tree shake to remove classes and methods. By not require Proguard to remove classes, this removes the trouble of classes relying on other classes that have been removed, with this allowing the developer to spend more time coding and less time fixing Proguard related issues.

While this report provides a strong argument that Scala is a viable language for developing android applications, it also highlights some of the pitfalls of this approach, such as proguard’s tree shake causing hard to identify issues.

Another issue that is about to occur is that Scala 2.12 release is focused on improved interoperability with Java 8. As previously stated Java 8 is currently unavailable on Android without some configuration required by the developer before starting to create their app. With the release of Scala 2.12, this means that setting up Scala on Android will require even more configuration by the developers before creating their app and result in less time spent on the app and more time spent on setting up the environment.

It should be noted that Martin Oderskey has confirmed that following version 2.12 of Scala with 2.13, they will be focusing on overhauling the Scala language at a fundamental level. One approach they are considering is by dividing the core Scala libraries and the additional libraries to make the language a more modular language. The average development cycle for a new version of Scala is approximately 18 months. With a more modular approach to the language it may then be easier for developer to use the core Scala libraries in an android application without having to spend time configuring Proguard to not remove classes that are needed by the application, but again this is purely speculation as it will be some time before Scala version 2.13 is released.

In closing, it should be state that the test case highlights that it is a perfectly viable option to use Scala to create android application, but does require some developer setup to achieve this. Arguably without some setup by the developer the default Java version available for development on android is Java Version 6, while the developer is capable of using Scala version 2.11.7.With the release of Kotlin version 1.0 and the supporting the language is currently receiving from the language creators and the IDE support they are providing, that within the next few years Kotlin could become the dominate language for creating less verbose android applications in a more functional than imperative style than is the current norm when using Java as the primary language.

Some of the major issues encountered during the development of the application were the result of Scala’s implementation of lists. Lists, in Scala, are by default immutable, with this mean whenever a new element is added to a list, a new list containing all previous elements along with the new element. While this might not be an issue on a more powerful machine, on a handheld device with limited resources, this can result in stack overflow errors occurring.