2015

The Future of iOS Development: Evaluating the Swift Programming Language

Garrett Wells
Claremont McKenna College

Claremont McKenna College

The Future of iOS Development: Evaluating the Swift Programming Language

submitted to
Professor Arthur Lee, advisor
and
Dean Nicholas Warner

by
Garrett Wells

for
Senior Thesis
Spring 2015
April 28, 2015

Abstract

Swift is a new programming language developed by Apple for creating iOS and Mac OS X applications. Intended to eventually replace Objective-C as Apple's language of choice, Swift needs to convince developers to switch over to the new language. Apple has promised that Swift will be faster than Objective-C, as well as offer more modern language features, be very safe, and be easy to learn and use. In this thesis I test these claims by creating an iOS application entirely in Swift as well as benchmarking two different algorithms. I find that while Swift is faster than Objective-C, it does not see the speedup projected by Apple. I also conclude that Swift offers many advantages over Objective-C, and is easy for developers to learn and use. However there are some weak areas of Swift involving interactions with Objective-C and the strictness of the compiler that can make the language difficult to work with. Despite these difficulties Swift is overall a successful project for Apple and should attract new developers to their platform.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the summer of 2008 Apple launched the App Store for the iPhone and iPod touch. Originally containing only 522 apps, as of 2014 the App store houses over 1 million apps and has seen over 75 billion app downloads [Fri13]. This platform has attracted thousands of developers to create applications for iOS devices, and has launched thousands of careers and companies. One of the constraints that Apple put on developers wishing to create iOS applications was that every app had to be written in the language Objective-C. Designed by Brad Cox and Tom Love in 1983 and based on the SmallTalk language, Objective-C extends the C programming language by adding object-oriented features and message passing [Koc99]. While the language was initially used in many different fields and applications, it soon became known as the primary language used by Apple. Apple uses mainly Objective-C for the OS X and iOS operating systems, as well as their respective API frameworks Cocoa and Cocoa Touch [Koc99].

As Objective-C aged it became harder for new developers, unfamiliar with C and SmallTalk, to learn and understand. Languages such as Java, Python, and Javascript became widely used and began to set the standard for modern programming languages. Developers began to complain that Objective-C was difficult to learn and uncomfortable to use. This difficulty made fewer developers interested in making iOS and OS X applications and several began to switch over to Android operating system, which allowed them to write in Java. However Apple could not switch which language it supported without rewriting the Cocoa and Cocoa Touch frameworks, as well as abandoning any legacy Objective-C code. Apple needed a way to use a different language while simultaneously continuing to support all of its Objective-C code.

Eventually Apple settled on a solution to its language problem. In June of 2014 during their annual Worldwide Developers Conference, Apple unveiled a new programming language for developing iOS and Mac OS applications. This new language, christened Swift, was developed by Chris Lattner, an Apple employee, beginning in June of 2014. Initially a personal side project for two years the language eventually attracted the attention of others at Apple and the project became a major focus of the Apple Developer Tools group in July 2013 [Lat14]. This new language would look completely different than Objective-C, but still offer compatibility with Objective-C code, allowing Apple to slowly phase out the old language over time. During the unveiling of Swift Apple announced they intend to eventually replace Objective-C, calling the new language "Objective-C without the baggage of C". [App14a]. Soon after the language was revealed, it reached a stable 1.0 version and Apple started accepting apps created entirely in Swift in the App store.

Swift had been a closely kept secret up until its announcement, and came as a surprise to the developer community. Since switching programming languages is a nontrivial task for individuals and organizations, Apple needed to demonstrate that Swift was worth the additional overhead. During the WWDC presentation Apple claimed that the language would run significantly faster than Objective-C, and would be a modern programming language with all the features common to other recent programming languages. They also claimed it would be extremely safe language that eliminated many classes of unsafe code. Finally the language would be easy to learn and use [App14a].

I test each of these claims as best I can in order to develop a better idea of how Swift will be used by the average programmer. I first use two different algorithms to benchmark Swift against four other languages to determine in general how fast Swift runs. I then take the major features of Swift and see if I can find other language that have the same features. Finally I create my own iOS application in Swift to demonstrate using the language for nontrivial tasks. I find that while Swift generally runs faster than Objective-C it is difficult to reproduce the exact speedup projected by Apple. I also discover that Swift has almost all the features commonly found in a modern programming language, with the exception of exception handling, native support for parallelism and concurrency, and optional types. Finally I find that Swift is easy to use, has a pleasing syntax, and eliminates many types of programming errors. However there are some difficulties in using Swift that result from interacting with Objective-C code and the Cocoa Touch APIs. Overall I conclude that Swift is a dramatic improvement from Objective-C and hopefully the rough edges will be smoothed out over time.

Chapter 2

Background

2.1 Language Details

Released in June of 2014 by Apple Swift is a statically typed language and compiled language that uses the LLVM compiler infrastructure and the Objective-C runtime [App14b]. Since Swift uses the same runtime as Objective-C the two languages can be intermixed in a single program or project, as both will compile down to native machine code. Swift can access Objective-C classes, types, functions, and variables through a "bridging header", as well as by extension C and C++ code. Similarly Objective-C can access code written in Swift, with some exceptions. This allows Swift to work with the Cocoa and Cocoa Touch frameworks and existing Objective-C apps and libraries without rewriting the large body of code that was written for iOS devices [App14c]. Swift is heavily influenced by many other languages such as Rust, Haskell, Ruby, Python, and C#, and offers many of the object-oriented and functional features found in these languages. Swift also includes a read-eval-print-loop (REPL) that can be accessed in Xcode as well as on the command line.

2.2 Previous Work in Language Evaluation

Many previous researchers have proposed methods for evaluating and comparing new programming languages. Languages are often compared against one another on a number of different criteria. Kulkarni et al. compared C++, Java, Perl, and Lisp together [KKS$^+$08] and their approach was extended to even more languages by Al-Qanhtani et al [AQPG$^+$10]. Both of these papers conclude that each language has various pros and cons and are suited to different types of tasks, with Java and C receiving the most favorable reviews. Many programming language evaluations examine a language holistically and qualitatively, although attempts have been made to be more rigorous and quantatative. AlGhamdi and Urban propose a qualitative framework for assessing languages in terms of twelve different attributes including regularity, readability, reliability, portability, and Input/Output [AU93]. This framework provides a standardized way to evaluate a language in isolation and describes the key attributes important in any language design. Although these frameworks and comparisons help unveil the important aspects of a programming language, they are too high level to be appropriately applied to Swift.

Other researchers have looked at programming languages as they apply to a specific domain. Since swift is meant to be used primarily for mobile devices, this type of research is more applicable. Gupta discusses the appropriateness of programming languages for teaching beginners or teaching, ultimately recommending qbasic or C [Gup04]. Howatt recommends evaluating a language based on how well it solves a given project or task on hand although he has doubts about the real world relevance of this approach [How95]. Oppermann and Compus discuss several popular languages used for mobile clients and server-side development [OC08]. They conclude that using single language on both the mobile client and server offers a distinct advantage and that Java and Python are the best choices for this approach. Schmanger et. all. use design patterns to evaluate the Go programming language [SCN10].They implement a subset of the HotDraw framework in Go and use their implementation to motivate a discussion of the language [SCN10]. This project was the main source of inspiration for my analysis of Swift, since the authors used a large project to demonstrate their view on a new language. While I do not use design patterns or the HotDraw framework in my Swift application, I do create a drawing application that has many similar attributes. SwiftDraw uses several design patterns including the Composite pattern and the Builder pattern, and was originally inspired by the JHotDraw Framework [Gam07].

Chapter 3

Evaluation of Swift

3.1 Speed

Apple has claimed that Swift makes apps run "lightning-fast". During the Worldwide Developer Conference they showed two slides concerning the speed of Swift. The first slide showed a comparison between Python, Objective-C and Swift for performing a complex object sort. The graph showed that Objective-C was 2.8x as fast as Python, and Swift was 3.9x as fast as Python. This makes Swift approximately 1.4x faster than Objective-C for sorting complex objects. The second slide they showed compared Python, Objective-C, and Swift for RC4 encryption. This graph showed that Objective-C was 127x faster than Python, and Swift was 220x faster than python. This means that Swift 1.73x faster than Objective-C for RC4 encryption [App14a].

While it is difficult to exactly reproduce any benchmarking results, I attempted to run these same benchmarking tests myself on my own computer. Since Apple did not specify what computer these benchmarks were run on, how the algorithms were implemented, or even what a complex object exactly is, much of the implementation was left up to my best guess. Additionally, any results achieved on my personal computer should be taken as approximate results at best, although they still should indicate general language efficiencies. I decided to expand my language comparison to include Python, Objective-C, Swift, Java, and C. While compiling Swift I used the first level of optimization by passing the compiler the "-O" flag. I did no compile optimizations for any of the other languages. For the Java benchmarking I did not warm up the Java Virtual Machine, timing the algorithm only on the first run. Java will perform better after the JVM has

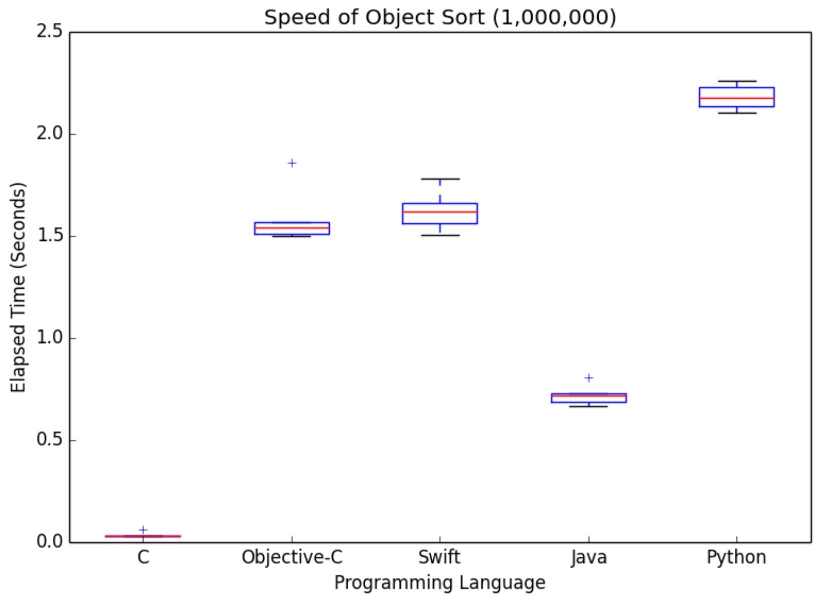


Figure 3.1: Benchmarking Complex Object Sort

been warmed up, but for approximate test this is sufficient. For the first test I wrote an algorithm in each language to sort a list of $1,000,000$ objects in ascending order. The objects were sorted in order based on a randomly generated numerical instance variable that ranged from -1000 to 1000. Since C does not support objects a struct was used instead. I ran each algorithm 25 times and plotted the results. Swift and Objective-C both performed approximately equal, running on average 1.4x faster than Python. However both languages paled in comparison to Java and C, with Java being on average twice as fast as either language.

For the next test I decided not to implement RC4 encryption because of the difficulty of implementation. Instead I substituted another computationally complex algorithm, matrix multiplication. I multiplied two 500x500 matrices together with an $O(n^3)$ algorithm in each of the five languages, again running each algorithm 25 times in each language. In this test Swift far outperformed Objective-C by a significant margin, running on average almost 10x faster than python and 4x faster than Objective-C. While still not quite as fast as C, Swift's performance was on par with Java.

While these results are approximate they still show that it is unlikely that Swift is quite as fast as projected by Apple. It is possible that the per-

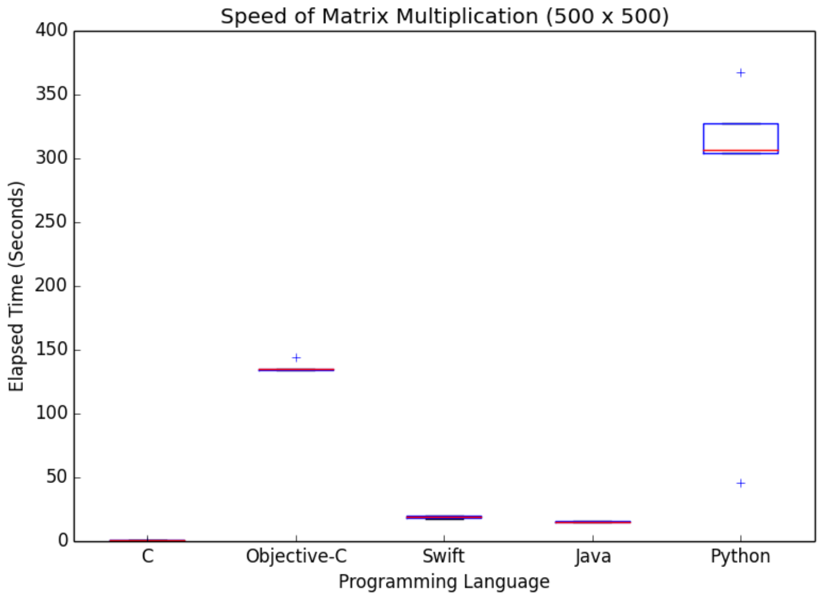


Figure 3.2: Benchmarking Matrix Multiplication

formance document in the WWDC presentation could be achieved by Swift under special circumstances, but in the general case Swift is only moderately faster than Objective-C. Swift also needs to be compiled with optimizations in order to achieve anywhere close to reasonable performance. Without any optimizations swift performed worse than both Objective-C and Python in both algorithms. It is unclear exactly why Swift performs much better than Objective-C during matrix multiplication, but no better while sorting. I predict that the difference in performance is partially because Objective-C does more work during runtime as a result of its dynamic method dispatch. Since swift mostly avoids dynamic dispatch it can execute significantly faster.

3.2 Feature Set

Swift's creator, Christ Lattner, has said that Swift was heavily influenced by many other languages, and includes many of the features commonly found in other modern programming languages [Lat14]. While Swift has too many features to fully discuss in this thesis, many of the major language constructs are drawn from other languages. I went through each feature highlighted by Apple during the WWDS presentation and tried to identify the language that had a similar implementation. In general Swift includes almost all the common features found in a modern programming language, with only a few surprises. The first surprise is the heavy use of optional types. Inspired by the Haskell Maybe monad and optional types in Rust, Swift optionals allow the programmer to declare a variable that can be null [App14b]. This feature helps to avoid null pointer references, and has been rarely seen in other widely used languages. The second surprise is that Swift uses Automatic Reference Counting (ARC) for memory management instead of a tracing garbage collector. Most modern languages, such as Java and Python, use a type of tracing garbage collector. While ARC is usually less efficient than tracing garbage collection, it can offer smoother performance and use less memory since it reclaims objects immediately [App14b]. Apple has decided that smooth performance and lower memory overhead is more important than overall efficiency. Additionally Objective-C used reference counting, and it is likely that Apple saw no need to change this while creating Swift.

Swift Feature	**Languages with similar features**
Closures	Javascript
Generics	Java
Type Inference	Haskell
Tuples	Python
Functions as First Class Objects	Javascript
Operator Overloading	C++
Pattern Matching	Scala, Haskell
Optional Types	Haskell, Rust
Automatic Reference Counting	Objective-C
Protocols	Java, C++
Read-Eval-Print-Loop (REPL)	Python

Table 3.1: Features highlighted in Apple's WWDC presentation

Swift also has a few features that are noticeably missing. Swift offers no native support for exceptions such that there is no try/catch construct in the language. Instead Swift encourages the programmer to utilize optional types and enums to catch and handle errors without turning to exceptional control flow. Initially this feature was nautically missing, since unwrapping multiple optionals littered code with if-else blocks. The latest update to

Swift now allows for unwrapping multiple optionals at once, which brings Swift closer to a traditional try/catch structure. Additionally Swift offers no native support for concurrency or parallelism, instead relying on the Cocoa and Cocoa Touch framework for utilizing threads and processes. While workarounds exist to emulate both of these features, Swift deviates from other modern languages by not including them. It is possible that in future versions Swift will add support for exception handling and native concurrency support, but currently no such features exist.

3.3 Safety

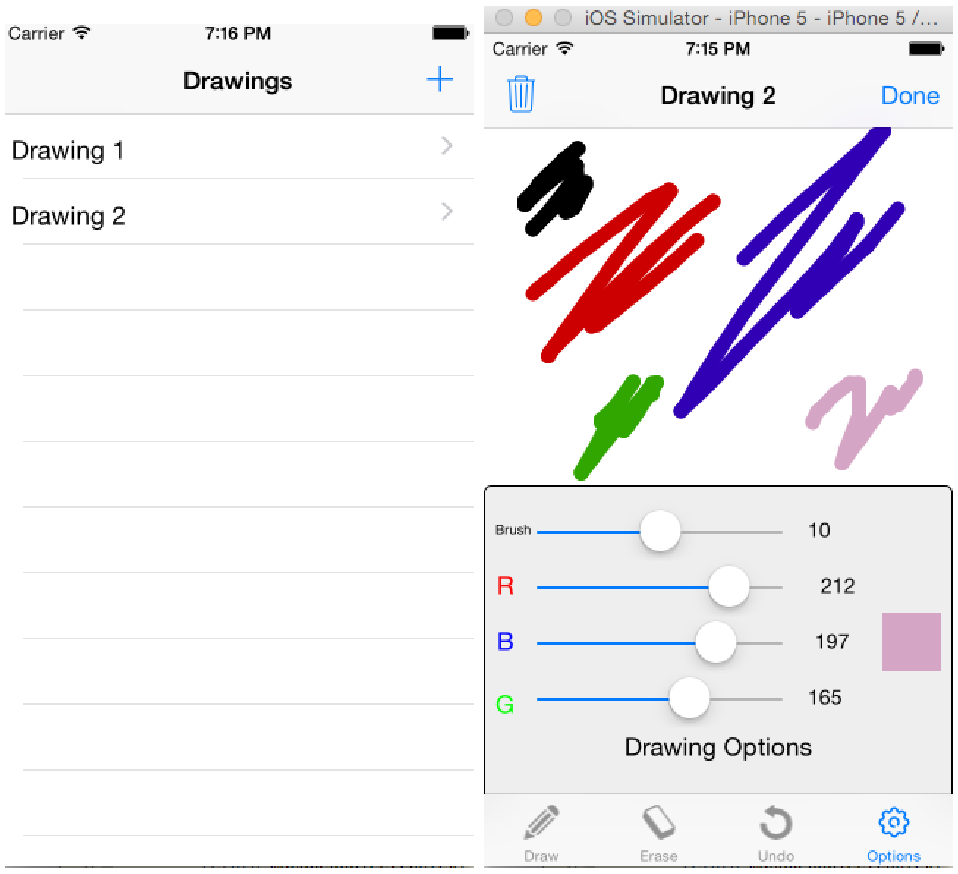
Apple has strongly declared that Swift is a "safe" language. I understand this to mean that it is a type safe language, and that many errors can be caught at compile time, rather than producing unusual behavior at runtime. This idea of catching errors at compile time is prevalent throughout Swift, and manifests in several different ways. Swift attempts to get rid of null pointer errors by requiring all variables to be initialized either right away or during the `init` method of a class. If a variable either is not initialized immediately or could eventually be null than an optional type must be used. The optional type is defined as an enumeration with two cases, `None` and `Some(T)`, which is very similar to the maybe monad from Haskell. Swift offers a syntactic sugar for optionals, allowing the programmer to write ? after a variable type to demarcate it as an optional. If a variable is an optional type then it must be unwrapped before being used which saves the programmer from accidentally trying to access a null reference. However Swift also includes implicitly unwrapped optionals that are demarcated with the ! symbol after the variable type. These variables can be null but do not need to be unwrapped before being used, acting as a normal variable. If an implicitly unwrapped optional is null and the programmer tries to access it a run-time error will occur. These types undermine the safety of optional types since a programmer using implicitly unwrapped variables still runs into null pointer errors.

Swift emphasizes explicitly declaring possible dangerous code fragments. Possibly null variables must be marked as optionals, and any subclass that overrides a method must be explicitly marked. Additionally any variables in a closure must be prefixed with self, to disambiguate between variables in the closure and those in an outer scope. While these do not help the compiler, they do inform the programmer exactly what they are doing, making accidentally accessing the wrong method or variable much less likely. Swift also has clear rules about when variables are copied or passed by reference. All classes are passed via reference, and all structs and enums are copied. Method parameters cannot be modified within a method unless they are explicitly declared to be `inout` in the method declaration. While some of these constraints can be occasionally difficult to work with, overall they help the programmer understand clearly when they are dealing with potentially dangerous areas of code.

3.4 SwiftDraw

In order to experience the experience of learning Swift and using it in an actual project I created my own application called SwiftDraw. The SwiftDraw app is a simple drawing interface for the iPhone and iPad that allows the user to create and store multiple drawings. Written entirely in Swift, SwiftDraw uses many of the features commonly found in mobile applications.

SwiftDraw has two main views. The first is a list view that allows users to create new drawings, select a previous drawing to edit, and delete a drawing entirely. The drawings are stored using the Core Data API, and are represented internally by the Drawing and Figure class. The Drawing Class stores the title of the drawing, and keeps track of the set of corresponding Figures. The Figure class keeps track of the list of point that represent the line to draw, as well as information about the line, such as color, opacity, and width. The second view is the main drawing view, where the user can draw and erase lines using the Core Drawing API. The size, color, and opacity of the line can be changed by brining up an options menu. The ability to undo and delete drawings is also functional.



Brush 10
R 212
B 197
G 165

(a) List View (b) Drawing View

Figure 3.3: Views in the SwiftDraw app

Chapter 4

Discussion

The learning process for Swift is rather painless. Programmers familiar with C type languages like Java and C++ will quickly feel comfortable in Swift, as much of the syntax and programming style is the same. I was quickly able to write basic scripts in swift and feel comfortable with most of the language concepts in a day or two. Having no prior experience in Objective-C did not seem to have any impact on my ability to pick up Swift, since Swift feels closer to Java and Scala than Objective-C. The only difficulty that arose from not knowing Objective-C is that most tutorials and guides are still written from Objective-C and it is challenging to find quality resources for Swift. The other challenge while learning to build SwiftDraw was learning to interact with the Cocoa Touch APIs. This turned out to be one of the more difficult aspects of working with Swift.

The advantage for Apple of creating a new language is that Swift can access the Cocoa APIs without having to rewrite any of the Objective-C code that already exists. The downside to this is that because the APIs were not originally written for Swift, there is additional work on the programmers part to convert them to Swift. Many of the Cocoa Touch API methods can possibly return null, or return the general type AnyObject. This requires heavy use of optionals and typecasting. While not overly difficult, this does make the code littered with optionals and question marks, making it difficult to decipher what is an optional and what is not. In many cases I ended up casting everything to an implicitly unwrapped optional so I did not have to constantly unwrap everything. Additionally some of the Cocoa Touch methods require Objective-C objects such as NSObject or AnyObject which makes them harder to use. It can be difficult to work within the rules of typecasting in Swift when you need to turn a String into an AnyObject type. Strings are not objects in Swift, and therefore must be cast to NSStings before being cast to AnyObject. Additionally there are three different type casting operators in Swift, despite all looking identical. Overall the Cocoa Touch API adds many complex layers of casting and optionals that are difficult to decipher. This can be avoided by simply force casting to an implicitly unwrapped optional, but then many of the safety guarantees in Swift are lost.

This overuse of the implicitly unwrapped optional continues because of the two stage initialization process. Every variable needs to have a value either at the top of the class or in the `init` method, or be declared an optional. Because often its desirable to have variables initialized during the `ViewDidLoad` method in order to access the self.view attribute, most variables become implicitly unwrapped optionals. This opens up a program to null pointer errors which is exactly what Swift was trying to avoid with two stage initialization.

There were also issues when dealing with protocols in Swift that I discovered while implementing the composite design pattern, which requires an array of mixed types that share a prototype. I created the protocol Shape and two classes, square and circle, which both implement Shape. I then declared an array of shapes and tried to find the first instance of a specific shape. The first difficulty was that to find an element in an array Swift uses a global method, rather than a method defined on the array class. That makes it difficult to use code completion in Xcode because you need to know exactly what the method name is before you call it. The second difficulty was that in order to find an element in an array that element must extend the Equatable protocol. This requires that the `==` method be overloaded at global scope, rather than being contained in the class. However Swift does not allow the `==` method to accept a protocol as a parameter, which means the Shape protocol cannot extend Equatable and therefore it is impossible to have an array of protocols in Swift.

There are also a few things that should work in Swift that seem to be broken. Several methods on Strings, such as containsString or length, are included in the documentation but do not work in the language. In order to access several of the methods the String must be cast to the Objective-C type NSString. Additionally Swift types are supposed to be available when defining Core Data objects, however this functionality remains unavailable. These bugs may be fixed in future updates to Swift. Several other bugs and features changed during the course of this project. When I first began writing SwiftDraw Immutable arrays initially could be modifier and there was no Set collection type despite being in the documentation. These bugs were fixed near the end of the project during a large update to Swift. Additional features were added as well such as access control, failable and forced optional type casting, and multiple optional unwrapping. This update broke some of the code I was working on, and changed how I implemented certain patterns in the language. While the changes were middle annoying, this shows that Apple is continuing to iterate and improve Swift, and the language will continue to get better.

On the positive side, writing Swift is generally a pleasure due to the many syntactic sugars in the language. Combined with the heavy use of type inference and the lack of semicolons, Swift feels closer to a scripting language than a systems language. It is a slight disappointment that list comprehensions are not included in Swift, although the map and filter syntax is easy to use. For all the trouble that optionals cause, they do make it easy to check for null variables quickly, and can even be chained together which leads to cleaner code. Swift also keeps the style of method names from Objective-C, giving each method a long descriptive name that can have internally and externally named parameters. This helps separate internal use of a variable from its meaning when called externally and is quite helpful.

Chapter 5

Conclusions

Overall Swift is a major improvement from Objective-C and a great step in the right direction for Apple. Although unlikely to be as fast as projected in most use cases, the lack of dynamic dispatch and strictness of the type system allow Swift to see significant speed improvements. Swift will never be as fast as Java or C, but for a language that feels closer to a scripting language than a systems language the efficiency is welcome if not overly impressive. Swift should be easy for most developers to learn since it has very similar syntax to Java or Scala. Most of the features found in Java, Scala, or Python are also present in Swift, and the only features that Swift lacks have little impact on the overall usefulness of the language. The only disappointment I had about Swift is that it does not have anything new or unique. Apple has created a language that feels extremely similar to the languages that are already being used today. There are a few details that are unique or original, including external parameter names, computer properties, and native variable listeners. Other than these few details Swift does not advance the field of programming languages or push developers to learn a new way of programming.

While Swift may not have many original features, it does have a focus on making sure the features that it does have are implemented correctly and intelligently. There is a major focus on eliminating programmer and runtime errors by having the compiler catch every possible sources of error. Often times the level of safety is left up to the programer such as the use of immutability and optionality. Other times Swift forces the programmer to conform to its own rules, such as when using protocols or closures. While this can sometimes be frustrating to work with, Swift significantly decreases number of errors the result from programmer mistakes. These bugs can often be hard to track down, such as when a variable or method is unknowingly modified, and Swift helps the programmer catch them before runtime. While building SwiftDraw I was pleased with the safety of Swift the vast majority of the time. However I spent a significant amount of time wrestling with the strictness of the compiler and the strange behavior that arises when working with Objective-C code and the Cocoa Touch framework. This seems to be the weakest area of Swift, and unfortunately occurs frequently while writing iOS apps. Hopefully this area becomes easy to deal with as the language matures and develops.

I believe that more and more developers will switch to using Swift over Objective-C. The learning curve is small and Swift offers many advantages in terms of syntax and guaranteeing program correctness. While Objective-C may never go away, Swift will soon become the default language for all iOS and Mac OS X applications. The language still has a few rough areas but the language will continue to improve and develop until most of the major flaws are fixed. Swift is a major step forward for Apple and should be a success for both the company and its developers.

Chapter 6

Bibliography

[App14a] Inc. Apple. Apple wwdc 2014 - swift introduction. https://www.youtube.com/watch?v=MO7Ta0DvEWA, June 2014.

[App14b] Inc. Apple. *The Swift Programming Language*. 1. Apple, Inc., 2014.

[App14c] Inc. Apple. *Using Swift with Cocoa and Objective-C*. 2. Apple, Inc., 2014.

[AQPG$^+$10] Sultan S Al-Qahtani, Pawel Pietrzynski, Luis F Guzman, Rafik Arif, and Adrien Tevoedjre. Comparing selected criteria of programming languages java, php, c++, perl, haskell, aspectj, ruby, cobol, bash scripts and scheme revision 1.0-a team cplgroup comp6411-s10 term report. 2010.

[AU93] Jarallah AlGhamdi and Joseph Urban. Comparing and assessing programming languages: basis for a qualitative methodology. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 222–229. ACM, 1993.

[Fri13] Lex Friedman. The app store turns five: A look backward and forward. http://www.macworld.com/article/2043841/the-app-store-turns-five-a-look-back-and-forward.html, July 2013.

[Gam07] Erich Gamma. Jhotdraw. `http://www.jhotdraw.org/`, November 2007.

[Gup04] Diwaker Gupta. What is a good first programming language? *Crossroads*, 10(4):7–7, 2004.

[How95] James Howatt. A project-based approach to programming language evaluation. *ACM SIGPLAn Notices*, 30(7):37–40, 1995.

[KKS+08] Prashant Kulkarni, HD Kailash, Vaibhav Shankar, Shashi Nagarajan, and DL Goutham. Programming languages: A comparative study. 2008.

[Koc99] Stephen G. Kochan. *Programming in Objective-C*. Sams Publishing, 1999.

[Lat14] Christ Lattner. Christ lattner's homepage. `http://nondot.org/sabre/`, September 2014.

[OC08] Leif Oppermann and Jubilee Campus. On the choice of programming languages for developing location-based mobile games. *GI Jahrestagung (1)*, pages 481–488, 2008.

[SCN10] Frank Schmager, Nicholas Cameron, and James Noble. Gohotdraw: evaluating the go programming language with design patterns. *Evaluation and Usability of Programming Languages and Tools*, page 10, 2010.