CS 334 Final Project: Swift

Zack Romrell and Will Swindell

May 2021

Overview:

Developed by Apple and officially introduced in 2014, the Swift programming language has quickly become a widely used programming language which continues to grow in popularity. At a low level, Swift was designed primarily for Application programming, which includes mobile and web apps, but the language can be extended and used for systems programming, cloud services, and more. In 2015, Swift became open source, allowing for ease of use with all Apple platforms– iOS, macOS, watchOS and tvOS.

The designers of Swift intended for the programming language to be easy to learn and implement, which is facilitated through the development of Xcode, the intended programming platform which is available on the App Store or can be easily downloaded online. While Swift can be written, compiled, and run in Terminal and other editing softwares, Xcode makes development much easier by immediately showcasing the result of your code, as well as adding tutorials and easy implementation of UX through SwiftUI. Similar to Scala's ability to run Java packages and modules, Swift has the Objective C API built in, so it is able to run files written in Objective C to minimize issues with legacy programming and older projects.

Swift is a general purpose object oriented programming language with a design focused around internal data and methods. By design, Swift is somewhat of a hybrid language, as it supports older procedural programming through the incorporation of structs (comparable to C based languages), yet it also has a modern class-based approach (similar to Java and Scala) that allows the programmer to make use of subclasses, protocols, enumerations, and other advanced object-oriented language features.

Types:

Built-in Types

Swift includes all the standard types included in C and Objective C, which include the following:

```
//Built-in Data Types
Int, Float, Double, Bool, String, Character, Optional (represents a
variable that can hold either a value or no value), Tuples
```

Similar to "var" and "val" in Scala, Swift allows for the creation of mutable and immutable objects using "var" and "let" respectively.

```
//Constant Declaration (Immutable data)
let constant_name: type = expression //type annotation is
//optional when the type can be inferred
//Example
let num: Int = 4; or let num = 4;

//Variable Declaration (Mutable data)
var constant_name: type = expression //if no initializer expression is present, var declaration must
    include an explicit type annotation

//Examples
var num: Int = 4; or var num = 4; or var num: Int; num = 4;
```

Swift also includes 3 built in collection types: Arrays, Sets, and Dictionaries.

Additionally, Swift allows for user defined types through class and struct implementations.

Type Safety

Like many other languages, Swift is a statically type-safe programming language, meaning type safety is checked for at compile time to prevent runtime errors and unexpected behavior. This approach has no runtime cost and ensures type safety on all possible paths. Swift, like Scala, also supports type inference, meaning the programmer is not required to explicitly specify types when declaring variables. Swift, however, does require that the programmer specify the return type of functions (if the function is void, no return type is needed).

```
//Function Declaration and Calling (parameter type cannot be inferred)
//Non-Void return types
func func_name(p_name: p_type) -> return_type { body }
func_name(p_name: value)

//Void return type
func func_name(p_name: p_type) { body }
func_name(p_name: value)
```

Collection Types

As stated above, Swift includes three built in collection types: Arrays, Sets, and Dictionaries.

An Array in Swift functions very similar to a Vector in Java or a list in Scala. When allocated, Arrays have a specific amount of allocated memory, but since memory is managed automatically in Swift, capacity adjustments are done automatically. To improve performance, Arrays have a reserve capacity method which allows the allocation of a specific size. See below for examples of standard Array use in Swift.

```
//Declaration (can be both mutable or immutable)
var array_name = [specified_type]();
var array_name = [specified_type](count: initial#OfElements, repeatedValue:
    intializationOfEveryElement);

//Accessing and Modifying
array_name[n] //returns nth element
array_name.append(variable of array_type) //adds element to end of array
array_name += [variable of array_type] //same as above
array_name[n] = new variable of array_type //modifying nth element

//Additional Functionality
for any_name in array_name{ body } //iterates through array
print(array_name.enumerated()) //prints index along with value
var new_array = array1 + array2 //adds two arrays
array_name.count //returns the size of array
array_name.isEmpty //bool value

//Arrays and higher order functions examples below
```

Sets in Swift function very similarly to Arrays, however they only allow for one instance of each element and have no explicit ordering. In order to be used in a set, a type must be hashable, meaning user defined types

must implement a hash function in order to use this type in a set. By only allowing one instance of each element, Sets support Set specific methods such as Union and Intersection, as well as more advanced Join operations. See below for examples of standard Set use in Swift.

```
//Declaration (can be both mutable or immutable)
var set_name = Set<data_type>()

//Accessing and Modifying
set_name.insert(...)
set_name.remove(...)

//Additional Functionality
set_name.count //returns number of elements
set_name.isEmpty //bool
set_name.contains(...) //bool
set_name.sorted() //orders the set (sets are not automatically ordered)
set_1.union(set_2) //performs the union operator
set_1.intersection(set_2) //performs the intersection operator
set_1.subtracting(set_2) //performs set-minus operator
//iterate over the set the same way as array (sets are not automatically ordered)

//others... "equal to", "isSubset(of:)", "isSuperset(of:)", etc.
```

A Swift Dictionary functions similarly to a HashMap in Java or a map in Scala. Dictionaries store a list of key/value pairs which can be looked up by the user. Like Sets, the keys in a Swift Dictionary must be hashable. See below for examples of standard Array use in Swift.

```
//Declaration (can be both mutable or immutable)
var dict_name = [key_type : value_type]() //empty dictionary
var election:[Int:String] = [42:"Will", 120:"Zack", 3:"Ben"]
//you can also combine two arrays of the same size to make a Dictionary (helpful when working with a
    constant)
let dict_name = Dictionary(uniqueKeysWithValues: zip(array1, array2))

//Accessing and Modifying
dict_name.updateValue(new_value, forKey: specific_key)
dict_name.removeValue(forkey: specific_key)
let winner = election[120]   //winner stores the value "Zack"
election[3] = "Jason"        //value for key 3 updated from "Ben" to "Jason"
//you can assign dict_values to nil to remove a key-value pair
election[3] = nil            //remove key 3 from dictionary
election[50] = "Jack"        //key-value pair [50: "Jack"] added to dictionary

//Additional Functionality
for (key, value) in dict_name.enumerated()) { body } // iterates through dictionary... Note: dicts
    are not ordered so calling dict_name.enumerated() orders the dictionary respectively
dict_name.count //returns size
dict_name.isEmpty //bool
dict_name.filter {$0.value < bound} //dicts support some HOF like filter

//Example
let dict: [String:Int] = ["Three": 3, "Seven": 7, "Forty": 40]
//Return an Array of keys or values
var keys = dict.keys()       //Returns ["Three", "Seven", "Forty"]
var values = dict.values()   //Returns [3, 7, 40]
```

```
//Iterating through Dictionaries
for (votes, name) in election {some code}
```

Generics

Like Java and Scala, Swift supports the use of generic types and functions, signified by the generic type T, and the underscore _ placeholder when accessing elements. Additionally, the generic type keywords 'Any' and 'Element' can be used as placeholders until a type is dynamically determined. Swift also allows for users to create their own generic types, such as Stacks that can contain any data type. In this way, Swift supports polymorphism at compile time, as generic type placeholders are determined dynamically at run time.

Operators

Swift supports all basic operators. While Swift does not support operator overloading, custom operators are allowed.

```
//All basic operators exist
Arithmetic: +, -, *, /, %
Comparison: ==, !=, >, <, >=, <=
Logical: &&, ||, !
Bitwise: &, |, ^, ~, <<, >>
Assignment: =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
Identity (reference checking): ===, !==
Protocol Conformance: 'is' (returns true if an instance conforms to a
protocol and false otherwise), 'as?', and 'as!'
```

Control flow operators:

Swift supports for-in loops, while loops, break, repeat-while loops (analogous to do-while loops in other languages).

```
// For loops
for object in iterable{some code} // Iterate through an iterable type
for (key, value) in dictionary{some code} // Iterate through a Dictionary
for i in 1...10{some code} // Repeat something 10 times

// While loops, repeat-while loops
while true{some code} // Repeats indefinitely
repeat{some code} while true // Repeats indefinitely
```

In addition, Swift supports switch statements, usually in conjunction with if let operators and optional types. Optional types are specified using "?" and "!", which specify that an object may contain a nil value. If an object specified with ? is found to be nil, Swift will avoid using it. If an object specified with ! is nil, Swift will still allow it to be used, which may cause run time errors. See the example below for an example usage of optionals, switch statements, and if let operations.

```
// Switch, if let, optionals demonstration
// Below program will print "Move to the Right"
```

```
let movement = "Right"
var optionalMove = String?
switch move {
case "Right":
    optionalMove = "Right"
case "Left":
    optionalMove = "Left"
}

if let direction = move{
    print("Move to the \(direction)")
}
else{
    print("Did not move")
}
```

Reliability and Safety:

Overview

Swift provides for many different features that help the programmer detect and avoid/handle errors. One subtle but helpful feature that Swift provides is the ability to define values as mutable or immutable. Using the "let" keyword creates an immutable value that cannot be altered (similar to "val" in Scala) and the "var" keyword creates a mutable value that can be altered later on (similar to Scala's "var"). This feature is quite useful and allows for the programmer to dictate if they wants their program to have no assignable state which implies no state changes and no side effects. Also as discussed above, Swift is statically typed meaning that it catches type errors at compile time, eliminating run-time overhead and having full coverage over all paths of the code.

Error Handling

Swift also provides first-class support when it comes to catching, throwing, and exploiting recoverable errors at runtime. Swift allows for 4 different approaches with handling errors. You can propagate the error from a function to the code that calls that function, control errors with a do-catch statement, handle the error as an optional value, or use assertions for specified errors. The do-catch statement is essentially a bunch of conjoined try-catch cases that let the programmer define specific cases for all possible exceptions. Similar to the "finally block" in ML the do-catch's last catch allows for a final case protecting against any unhandled exceptions that may exist. Overall, the main goal of this feature is to create a clean structured exit from the middle of potentially problematic code.

```
do {
    try expression
    body
} catch pattern_1 {
    body
} catch pattern_2 where condition {
    body
```

```
} catch pattern_3, pattern_4 where condition {
    body
} catch {
    body
}
```

Memory Management

When it comes to memory safety Swift has your back. Swift has implemented a couple key features that prevent unsafe behavior from happening in your code. First, Swift manages memory automatically so for the most part it isn't necessary to think about accessing memory yourself. Swift also ensures that all variables are initialized prior to being used in your code. It also has procedures in the way to make sure memory isn't accessed after it has been deallocated. Another cool feature many other languages lack is that Swift statically checks array indices and makes sure there are no out-of-bounds errors. Swift has a built in property that helps manage data race conditions. It does this by requiring memory modifying code to have exclusive access to the specific memory location. If a program contains conflicting accesses to memory the program will throw a compile or run time error.

Parameter Passing

When it comes to passing parameters Swift automatically converts parameters to constants to help prevent any unwanted side-effects (passing by value). It will result in a compile-time error if you try to modify the value of a parameter in the body of the function. However, you can also write the parameters to be "inout" parameters by placing the inout keyword in front of the parameter type. An inout parameter (passing by reference) allows for you to modify the parameter's value which will persist after the function call has ended. When calling a function that has inout parameters one must place the ampersand & symbol directly before the respective variable's name that refers to the inout parameter.

Abstraction:

Scoping (Access Control)

With regard to abstraction and information hiding, Swift has several scoping keywords that can be applied to classes, functions and variables that allow for varying degrees of abstraction. Data abstraction in Swift works similar to the private, public, and protected keywords in Java, but the keywords and levels of access control are slightly different. Ranging from least secure to most secure, the levels of access control are as follows:

Open access allows for any and all modules to access the declared class, function or variable, whether it be within the module which it is declared, or in another module that imports the module where it is declared. Classes declared as Open can be subclassed in any module, and Open methods can be overridden in subclasses that are part of any module.

Public access is similar to Open access, but subclassing and overriding Public classes and methods can only be done in the local module.

Internal access is comparable to protected access in Java. Classes and methods defined as Internal can only be accessed within the module where they are declared.

File-private access is one step more secure than Internal access. Entities declared as File-private can only be accessed within the actual file that they are declared. This differs from Java's protected keyword as objects declared as protected can be accessed within the module, so the File-private keyword allows for more secure data hiding.

Private entities work like private entities in Java. Only classes in which private methods are declared can access these methods, as well as subclasses that are declared within the same file.

Additional Abstraction Support

Some aspects of Swift, by nature, allow for a certain level of abstraction and data hiding. While not as explicit as the scoping keywords from above, features such as type-aliasing, closures, enumerations, and nested functions support abstraction in Swift programming.

Type-aliasing allows a programmer to create user-defined types which essentially act as a wrapper for another data type, hiding the original

data type of specific elements. See the example of type-aliasing below.

```
//Type-Alias example
typealias IntMap<Value> = Dictionary<Int, Value>
// The following dictionaries have the same type
var dictionary1: IntMap<String> = [:]
var dictionary2: Dictionary<Int, String> = [:]
```

Closures support abstraction in a similar manner as type-aliasing, but for functions instead of data types. Using closures allows for a programmer to add extra functionality to a program which cannot be accessed outside of the function where it is used. An example of closure usage can be seen in the following section about higher-order functions.

Enumerations support abstraction by having multiple cases, and allowing elements of any type to be stored in their case values, which makes it difficult to extrapolate the contents of an enumeration without explicit access. Enumerations add an extra layer of complexity to the concept of switch statements from above.

Swift also supports nested functions which are only usable in the function in which they are declared, which is a very effective way to hide functionality within a method. See below for an example of nested function usage.

```
// Nested function that adds two Ints
func operation(x: Int, y: Int) -> Int{
    func addInts(x: Int, y: Int) -> Int { return x+y }
    return addInts(x, y)
}
```

Higher Order Funtions:

Built-in Functions

Swift has several built-in uses for higher order functions. A few useful examples are the "map" and "sorted" methods, which both work with Swift Arrays. The map function takes a function as an argument and applies

it to each element of an Array. The sorted function takes a comparison function as an argument and sorts an Array according to this function. Example usage of map and sorted can be seen below.

```
// Map usage
var someInts: Array[Int] = [1, 2, 3]
func square(x: Int) -> Int{
    return x*x
}
for x in someInts.map($0.square()){print(x)}
// Prints 1, 4, 9

// Sorted usage
var someStrings: Array[String] = ["Australia", "Bolivia", "Chad"]
func sortStrings(s1: String, s2: String) -> Bool{
    return s1.count < s2.count   // Sort by String length
}
for x in someStrings.sorted(by: sortStrings){print(x)}
// Prints Chad, Bolivia, Australia
```

User-Defined Functions

Swift also allows for functions to take another function as a parameter value, as long as the data type of the function is specified correctly. For example, passing a function that adds two integers would need to be done as follows:

```
// Pass a function addInts to another function
func addInts(x: Int, y: Int) -> Int{
    return x+y
}
func something(addInts: (Int, Int) -> Int) { some code }
```

Additionally, Swift functions can also return other functions as return values. Like above, it is necessary that the programmer specify the function return type. For example, a function that returns another function that adds two ints would need to be declared as follows:

```
func something(params) -> Int -> Int { some code }
```

Swift also supports nested functions, which can be easily done by declaring a new function as usual within the body of another function. As described in the section on Abstraction, nested functions can only be used in the function which they are declared.

Closures

Closures are Swift's version of lambda functions. Closures allow for the use of nested functions without the need to declare and name a new function. Like lambda functions, closures are declared temporarily within a line of code which allows for efficient programming without loss of functionality. See below for an example of using Swift closures to simplify the sort example from above

```
// Sort an Array using a closure
```

```
var someStrings: Array[String] = ["Australia", "Bolivia", "Chad"]
for x in someStrings.sorted(by: { (s1: String, s2: String) -> Bool in
return s1.count < s2.count}){print(x)}
// Prints Chad, Bolivia, Australia
```

Classes and Protocols:

Swift provides structures and classes which are the flexible constructs that act as the building blocks of the programmer's code. In general, class and struct declarations are as follows:

```
// Class declaration
class Class_name: super_class, adopted protocols {
    body
}
// Struct declaration
struct Structure_name: adopted protocols {
       declarations
}
```

Classes have the option to inherit methods, properties (associated values with a specific class), and other characteristics from another class. When a class inherits from another it is referred to as the subclass, while the parent class is also known as the superclass. Subclasses in Swift can "call and access methods, properties, and subscripts belonging to their superclass" and also can override and provide its own version of any specified method, property or subscript. Just like in Java you can also prevent any form of overriding by marking the specific method, property, or subscript as final. Swift does not support multiple inheritance and a class type can only inherit from one parent class. Class declaration and extension can be seen in the example below.

```
// Superclass Declaration
class Point {
    var x = 0
    var y = 0
}
// Sub-class declaration
class ColorPoint: Point{
    var color = "Red"
}
```

Both classes and structures have the option to conform to specific protocols. A protocol is similar to an interface, it defines a blueprint for a specific task or piece of functionality. A protocol lays out the necessary methods, properties, and other requirements needed for a class or structure to adopt it. However, a protocol doesn't actually implement any functionality itself. One cool feature of Swift is that one can use protocols as types. An example of protocol usage can be seen below, using the Point and ColorPoint example from above.

```
// Superclass definition
class Point{
    var x = 0
    var y = 0
}
// Protocol definition
protocol Colored{
```

```
    var color: String { get set }
}
// Subclass that conforms to protocol
class ColorPoint: Point, Colored{
    var color: String
}
```

Like Java interfaces and similar to Scala traits, protocols can inherit from as many other protocols as desired. This allows for one to effectively build an efficient complex structure of protocols that conform to higher protocols. There exist many additional new features for protocols such as optional protocol requirements, protocol extensions, and operators that check for protocol conformance.

```
protocol protocol_name {
    // protocol definition
}
```

Some final optimizations for classes and structures. Classes are always reference typed which means they are referred to (not copied) when assigned to a variable or passed as a parameter to a function call. On the other hand, structures are value typed so they are passed by value, meaning a copy is made when assigned to a variable or passed as an argument to a function call.

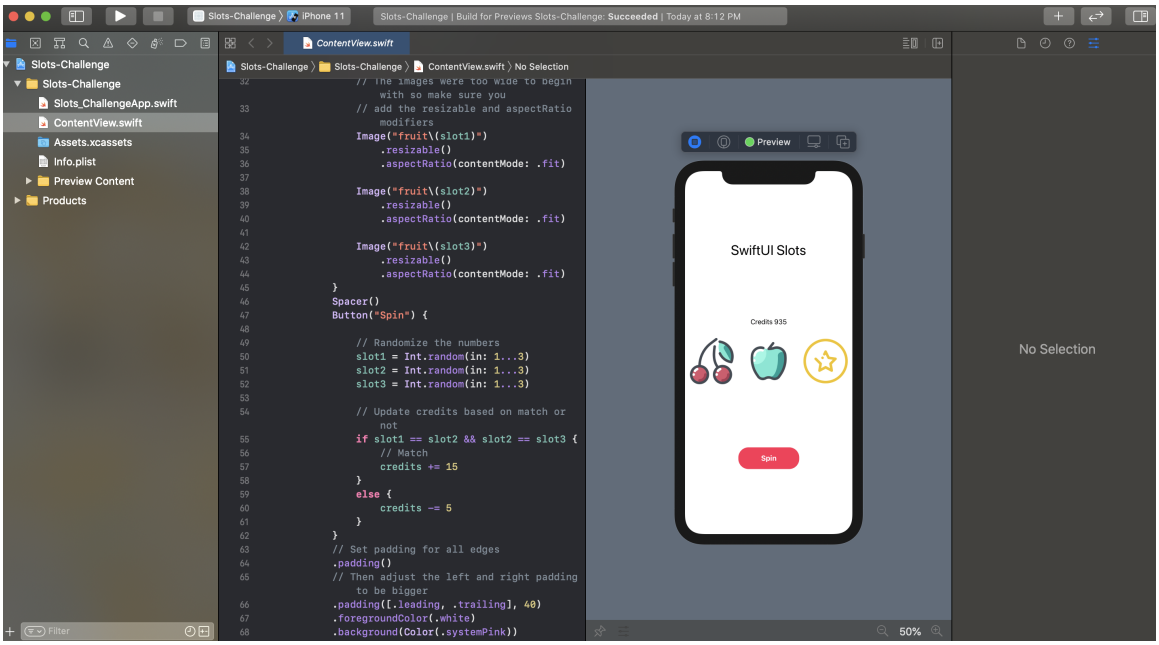
Concurrency:

By default, Swift programs execute serially, but Swift does support a few special methods which allow us to take advantage of parallel programming and concurrent threads. Swift 5.5 introduced the function parameters "async" and "await", which work similarly to the synchronized keyword in Java with slightly more complexity. If a function is labeled as an async function, it will pause if execution is interrupted. Hence, any async function must also use the "throws" keyword to specify that it will throw an error. In order to handle an interruption, we must use the await keyword when calling an async function. An example of concurrent programming in Swift can be seen below.

```
// Concurrent Threads
var x = 0
var y = 0
func difference() async throws -> Int {
    return x-y
}
func incrementBoth(x: Int, y: Int) {
    x++
    y++
}
func loopIncrement(x: Int, y: Int) async throws {
    for 1...10{
      incrementBoth(x, y)
       try await difference(x, y)
    }
}
```

SwiftUI

SwiftUI is a simple set of tools that help the programmer describe and control the user interface. The declarative syntax helps make SwiftUI easy to read and write for programmers of all levels. Xcode includes a lot of neat features that make it interactive and fun to code in.



The Xcode environment allows for you to drag and drop controls onto the canvas and open inspector tools like color selection, alignment, font, etc. The Swift compiler is fully embedded throughout Xcode, constantly building and running your app. One neat feature is that the design canvas is your actual live app. You can test your app on many different iphone models and are able to configure almost everything your user might see.

Conclusion

Overall Swift is a very large and general programming language and may feel overwhelming at first. The approach to problem solving when building apps on Xcode using the SwiftUI library takes on the form of declarative programming. Like functional programming, this approach limits side effects, is confluent, and the expressions always evaluate. This approach is especially easy for beginner programmers because you do not need too much background knowledge to build complex programs. However, this form of declarative programming makes it quite difficult to understand the inner workings of a program. This makes for a steep learning curve for programmers trying to build a large and complex application. Since the language of Swift itself provides a wide variety of similar features to other programming languages, knowledgeable programmers will easily catch on to the imperative side of Swift. However, learning how to incorporate imperative Swift with the SwiftUI will take time to learn effectively.

References

https://swift.org/

https://developer.apple.com/documentation/swift

https://www.javatpoint.com/history-of-swift

https://forums.swift.org/t/swift-concurrency-roadmap/41611

https://betterprogramming.pub/async-and-await-in-swift-5-5-5c8abb9f4f85

https://developer.apple.com/xcode/swiftui/