Matthew King

12/12/2022

Connect 4 – Swift

Introduction

Swift is a general purpose, compiled, multi-paradigm language (“About Swift.”). After beginning development in 2010 and being released in 2014, it is one of the most recent major releases of any programming language. The project was spearheaded by Chris Lattner (developer of the Clang compiler, LLVM, etc.) and was quickly backed by Apple after they decided it would be the replacement for their previous language, Objective-C. Released in 1984 alongside the original Macintosh, Objective-C had remained mostly the same since the 1980’s. It was an extension of the C language, with the most notable addition being messages in the vein of Smalltalk, a language construct that allows data to be stored and manipulated by “sending itself” to variables or objects. The main goal of Swift was to keep the readability of Objective-C syntax fully intact while pulling modern language features from “Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list" (Lattner).

As stated above, Swift is a multi-paradigm language. It has extensive support for Object Oriented, Functional, and Protocol Oriented programming, which can be thought of as Apple’s take on interfaces. While this flexibility makes Swift a viable choice for virtually any project, app design is by far the most common use of the language. Swift is included with and very closely tied to Apple’s own IDE, XCode, which includes many powerful tools for iOS and MacOS app development (“Xcode IDE.”). Tools like a drag and drop graphical interface builder and a built-in simulator for iOS, watchOS, and tvOS, make Apple-centric development the domain Swift is most used in.

Data Abstractions

Swift has support for a large variety of data types, but there are 6 primitives that are included in the language: `Int` (Integer), `String`, `Float`, `Double`, `Bool` (Boolean), and `Character` (“The Basics.”). Integers can be either signed or unsigned using Int or UInt declarations. Swift can store these variables in Tuples and Arrays. Tuples are collections of objects, similar to dictionaries in other languages. Each element in a tuple can be its own data type, and each can have a label. Tuples are of fixed length, so elements cannot be added or removed. Arrays are lists of variable length that can hold a single data type but can be heavily modified.

Swift also has support for a rather interesting data type: `Optional` (“Optional.”). `Optional`s can be thought of as wrappers for a value that indicates if the value exists. `Optional`s have two states: it either contains a value or is `Nil` if the value is missing. Any value can be wrapped into an `Optional` by using a `?` and can be unwrapped with a `!`, making it a very orthogonal feature.

```
var name: String? = "Alonzo" // Optional variable containing string
print(name!)                 // "Alonzo”, exclamation point unwraps name
 name = nil print(name!)                  // Runtime error, forced
unwrap of nil


if (name != nil)             // Check if name contains variable
print(name!)
```

Variables and constants (both referenced as variables for simplicity) in Swift are strongly/explicitly, statically typed. Swift will throw an error if you assign a value to a variable without declaring it first. Variables also cannot change their type after declaration. If one were to initialize a variable with a `String`, and later assign an `Int` to it, Swift would throw an error at compile time. Functions work similarly, if a function expects an `Int` parameter and a `String` is passed, an error will be thrown. All variables must stay the same type throughout the program. Those values do not always need to be explicitly declared, however. Swift supports type inference, which is a big help in terms of writability.

```
let columns: Int    // Explicit int declaration
let empty = ". "    // Inferred string var
player = 1      // Inferred int
```

Swift does not have implicit type coercion. If a data type needs to be changed, the user must do so explicitly.

```
let column = Int(Character(input.uppercased()).asciiValue!) - 65
```

The above line of code shows explicit type conversion from a `String` to a `Character` to an `Int`. Swift cannot do this on its own without it being explicitly written. Even for simple arithmetic between an `Int` and a `Float`, Swift does not have implicit type coercion.

```
var num1 = 5 var num2 = 7.5 return num1 + num2          //
Compiler error  return Float(num1) + num2   // Explicit type
conversion into 5.0
```

Control Abstractions

Swift was created to be as elegant and readable as possible, and the expressions used in the language strongly embody that design aspect. No semicolons are needed, as Swift uses line breaks to determine where statements end. Arithmetic is very similar to most other C-based languages and mathematic syntax in general. Swift uses the common `+-*/%^` set of operators for arithmetic, and are almost all infix (Figure 1).

Like most languages, parenthesis will prioritize operations encapsulated inside them. If operands have equal precedence, swift prioritizes them from left to right.

```
return 5 - 3 + (7 + 4) - 4        // ((5 - 3) + (7 + 4)) - 4
```

Swift has many typical selection constructs found in general purpose languages. `if-else` statements and `while` loops are as one would expect, executing when a condition is true (“Statements.”). `do-while` loops are implemented in the form of `repeat-while`, but function identically to their C++/Java counterparts. Swift also has `where` clauses, adding a simple and readable extension to loops and switch statements.

```
for piece in board where piece == empty {
```

Switch statements are standard as well, but Swift does not require break statements after each case. Instead, cases are automatically exited once they’re executed, only continuing to the next case if a `fallthrough` statement is included. If a case has no statements inside it, it will also `fallthrough` until a case with instructions is arrived at (Figure 2).

Guard statements are like **if-else** statements but will only execute if the guarded statement is **false**. They are a common pattern in other languages, but Swift takes the unique approach of adding new syntax. These are used to ensure certain conditions are met and can assist in managing nested **if-else** statements to improve readability. **Guard** statements must also end with **break**, **return**, **throw**, or **continue**. The below code shows a **guard** statement in action.

```
guard (input.count == 1) else { return false
}
```

Swift provides a **for-in** loop to iterate over data types that are sequential, like arrays, strings, sets, dictionaries, or any object that conforms to the Sequence protocol ("Control Flow."). **for-in** loops take two parameters: the name of each accessed item and the object to be iterated through. Swift does not include typical C++/Java style **for** loops, but there are multiple ways to create loops with an index. One method is to iterate through a range, shown below, or by enumerating a sequence (Figure 3).

```
for j in 0...(winLength - 1) {              // Range is 0 to (winLength - 1)
row.append(board[i + (iterate * j)])  // j increases by 1 each iteration }
```

Function parameters need to be explicitly declared in Swift. Each parameter needs a type, and like declaring variables, Swift adopts the Smalltalk syntax Apple used back in Objective-C. Any data type can be passed as a parameter, even other functions (Figure 4) (Finn). Functions can also have an undefined number of parameters by storing each one in an array accessible inside the function (Figure 5).

Scope in Swift follows the same rules one would expect from a language like C++/Java. Variables aren't usable in scopes above the ones they're declared in, and they must be unique within their scope. **Break** statements only exit the current scope level but can be tied to different scopes by using labels.

Swift comes standard with a built-in Package Manager, which automatically adds dependencies to frameworks and libraries ("Package Manager."). Swift programs are organized into modules. Modules each have a namespace and control where and how code can be accessed outside the module. These modules and their functions can be imported into programs with the **import** keyword. Packages are extremely similar, containing a Swift source file and a manifest file naming and distributing the contents of the source file. They can also be imported using **import**.

Error handling is mainly handled with `do-catch` statements (“Error Handling.”). These statements execute code with the `try` keyword , and if an error is thrown within the statement, the program enters the `catch` clause. `Catch` clauses are structured like `switch` statements, where the type of error is matched to the correct case (Figure 6). Errors can also be wrapped into optional statements to improve readability and reliability. Using `try?` Inside of an `if` statement will enter the statement if the optional value returned is not `nil`, shown below.

```
let regex = try? NSRegularExpression(pattern: #"(\d+)x(\d+)"#)
```

Object-Oriented Programming

Swift has extensive support for object-oriented design. Classes are defined with the `class` keyword, and properties are defined as variables within the scope of the class (“Structures and Classes.”). Initializers take the form of `init` functions and take in parameters for the object created by the class. Functions are defined inside of the class, and can reference their own class with the `self` keyword. An example class is shown in Figure 7.

Swift does not support interfaces. Instead, Swift uses protocols. Like interfaces, protocols allow developers to write out blueprints of functions and properties without defining the actions or data inside them. Then, other classes can implement those functions and properties best suited for their own needs. An example of a class inheriting a function from a protocol is shown below.

```
protocol Job {     func
isSalary() -> Bool
}
class Job : Cashier {
func isSalary() -> Bool {
return true
   }
}
```

Swift has many ways to control and protect who has access to data (“Access Control.”). There are four access controls in swift: `public`, which can be seen anywhere, `private`, which can only be seen within the current class, `fileprivate`, which can only be seen within the current file, and `internal`, which is the default declaration control level. When a type is `internal`, it can only be accessed within the same module.

Overall Evaluation

Swift is an extremely readable language. Its lack of semi-colons make it feel closer to reading a dialect and encourages proper formatting, due to Swift basing syntax on line breaks rather than an explicit delimiter. The mathematical operators and their infix placement makes arithmetic familiar to read. Swift also does a good job of removing unnecessary and redundant functions. For example, there is no shortcut for iteration in Swift. To iterate a variable, one must write `num += 1` rather than `num++`. Choices like this make Swift more universal to read, as a user doesn't need to know all the ways one thing can be achieved.

Swift has full support for Functional, Object Oriented, and Protocol based approaches, leaving developers with a lot of room to write how they want. This large selection of tools lends itself to writability, as developers rarely have to solve problems in ways they aren't comfortable with. Swift also removes a lot of cumbersome code. `Switch` statements do not require `break`s, which are almost always present in other languages. Reliability is a strong priority in Swift, due to the static type checking and error handling it provides via `optional`s, giving programmers many opportunities to make sure their programs are safe. Orthogonality is also strong in Swift. For example, any data type can be contained in an `optional` value using the same syntax, and any data type following the sequence protocol can be iterated through in the same way. There are virtually no exceptions to the rules in Swift.

Swift isn't perfect, however. While the language does a good job of eliminating redundant code in some areas, others are very cumbersome. Every function call with a parameter requires you to specify the type, while also requiring you to provide them in the right order. While this helps readability, these two qualities serve the same purpose while writing, making them annoying to use. Swift is also statically typed, meaning developers must create new variables entirely if they want to change the data type, reducing writability. Swift has a very strong type inference system, but if it were to "guess" the wrong type, the integrity of the program could be jeopardized, resulting in less reliability.

The biggest tradeoff that Swift has is reliability and writability. Because Swift is a compiled, statically typed language, almost all errors can be checked before the program ever compiles. As mentioned before, however, this leads to restrictions that can annoy programmers, like specifying types in function calls.

Swift is a very suitable choice for Connect 4, although it's not a project that showcases a lot of what Swift can do. As stated in the introduction, Swift is mostly used for app development. This is in part due to the powerful UI features it has, as well as XCode's tools for iOS deployment. A command line program uses none of these cool features. Swift also struggles with regex, and it's a large hassle to match regex literals. Swift requires regex pattern objects to be formed out of regex literals, which is hard to read and write. Because these are important for parsing the command line arguments for the program, this aspect of Swift is not well suited to the project.

As a general-purpose language, however, Swift is still more than capable of pulling off a command line program. It's even included as a blueprint in XCode. Because of the multiple paradigms offered and the ease of iterating through arrays, managing a Connect4 board is trivial. The language is full of modern features while still being extremely performance friendly. However, the language's infancy shows up in some areas, like the unnecessary syntax of parameters and the painfully confusing regex support. Overall, the Swift is a joy to work in, even if it's a bit of a head-scratcher sometimes.

Appendix

Figure 1

A modified and condensed version of Programiz's operator precedence list from top to bottom

Bitwise shift	>> <<
Multiplicative	% * /
Additive	| - + - ^
Comparison	!= > < >= <= === ==
Logical AND/OR	&& / ||

("Swift Operator Precedence and Associativity.")

Figure 2

Swift switch case

```
switch num {
    case 76 :               // If num = 76
        // Do something
    case 100,254 :          // If num = 100 or 254
        // Do something
        fallthrough         // Enters next case
case 7 :
        // Do something
    default :               // If none of the above
        // Do something
}
```

Figure 3

Swift for-in loop with enumerated array

```
// Board is an array, turned into pair by .enumerated //
Index is the pair value representing current location
for (index, piece) in board.reversed().enumerated() {
print(piece, terminator: "")


    if ((index + 1) % columns == 0) {
print()
    }
}
```

Figure 4

Swift function accepting a function as a parameter Modified from https://www.aidanf.net/learn-swift/functions (Finn)

```
func capitalize(data:String) -> String {
return data.capitalizedString()
}
// Takes other function as param and uses it inside function func
applyToString(data:String, function:(String)->String) -> String {
return(function(data))
}
applyToString("hello", capitalize)  // Prints "Hello", uses capitalize function
```

Figure 5

Swift function with an undefined number of parameters

```
func placePiece(columns:Int...) -> Bool{ // Takes Int, returns Bool
    for piece in columns {               // ... indicates multiple parameters
if (successful(piece)) {             return false
        }
}
    return true
}
placePiece(3, 5, 7, 8)                   // Arguments stored in columns array
```

Figure 6

Swift do-try-catch function

```
do {      data = try getData()              // If error is thrown, catch is
entered
} catch
{
    print("Failed to get data!")  }
```

Figure 7

Swift simple class

```
class Job {
    var pay: Int                // Class variable

    init(pay: Int) {            // Initializer
        self.pay = pay
    }       func getPay() -
> Int {          return
self.pay
    }
}
```

References

"About Swift." *Swift.org*, Apple Inc., https://www.swift.org/about/.

"Access Control." *Swift.org*, https://docs.swift.org/swiftbook/LanguageGuide/AccessControl.html.

"The Basics." *Swift.org*, https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html

"Control Flow." *Swift.org*, https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html.

"Error Handling." *Swift.org*, https://docs.swift.org/swiftbook/LanguageGuide/ErrorHandling.html.

Finn, Aidan. "Chapter 11 Functions and Closures ." *Adianf*, https://www.aidanf.net/learnswift/functions.

Lattner, Chris. "Chris Lattner's Homepage." *Nondot*, https://nondot.org/sabre/.

"Optional." *Apple Developer Documentation*, https://developer.apple.com/documentation/swift/optional.

"Package Manager." *Swift.org*, https://www.swift.org/package-manager/.

"Statements." *Swift.org*, https://docs.swift.org/swift-book/ReferenceManual/Statements.html.

"Structures and Classes." *Swift.org*, https://docs.swift.org/swiftbook/LanguageGuide/ClassesAndStructures.html.

"Swift Operator Precedence and Associativity." *Programiz*, https://www.programiz.com/swiftprogramming/operator-precedence-associativity.

"Xcode IDE." *Apple Developer*, Apple Inc., https://developer.apple.com/xcode/features/.