Write a final report summarizing the programming language assigned to you. Build on the previous presentations. Focus on subroutines and parameter passing, exception handling, and data abstraction.

Judy Wang, Tino Nguruve, Linh Le
Deepak Kumar
Principles of Programming Languages
3 December 2020

Swift Final Report

Introduction

Swift is a powerful and easy to understand programming language for macOS, iOS, watchOS, tvOS. Swift is a fast, interactive and modern programming language. Swift is 8.4x faster as compared to python in performance. The advantages of using Swift are it's easy to read as it draws syntax from other languages as mentioned above. It has a concise syntax which significantly increases the programmer testing rates for the program. It also uses automatic memory management. Automatic memory management is a technique in which an operating system or application automatically manages the allocation and deallocation of memory.

Structure of a Swift Program

Similar to many other programming languages, one of the first things that needs to be done is importing the packages you will need for your program. A programmer is able to use the word "func" to begin functions, which is similar to go lang. Swift does not need semicolons, but a programmer could if they wanted to and would not get errors. In addition, brackets are very important in Swift and play a similar role in other languages like Java in order to determine scopes.

Types and Operators in Swift

Swift has built in types such integers(**int**), strings, boolean (**bools**), and floating point types. For integers, Swift is similar to go lang where you can establish how many bytes you want your integers to have ranging from Int8-Int64. There are also unsigned integers which are represented as UInt8-UInt64. For floating point types, floats have allotted 32 bytes, while doubles are allotted 64 bytes. The operators that can be used for floating point types and integers are addition (+), subtraction (-), multiplication (*), division (/) and modulo (%). For boolean values or **bool**, they are represented as true or false, which are utilized in condition statements. Strings can be represented as **String**. String literals are encapsulated by quotations. Similar to booleans, strings can also use comparator operations to compare strings.

Variables and Declarations

As with any other languages, constants and variables in Swift associate a name with a value of a particular type. Once a constant or variable of a certain type is declared, users cannot declare it again with the same name, or change it to store values of a different type. Nor can you change a constant into a variable or a variable into a constant. The names cannot begin with a number, nor can it contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, line- and box-drawing characters. Variables must either be declared to be

only a type with the syntax **var <name>: <type>** or initialized a specific value **var <name>: <type> = <value>** before they're used, while constants should be initialized with keyword **let**. Swift's type inference allows the variables to be initialized with the syntax **var <name> = <value>** without the type annotation, and instead lets the compiler infer the type of data to store based on the initial value assigned.

Control Structures in Swift

As compared to C-like languages Swift's control flow structures are considered more robust. Cases can match many different patterns, including interval matches, tuples, and casts to a specific type. Swift's control flow statements include while loops, if, guard and switch statements to execute the code based on certain conditions. It also has statements such as break and continue to transfer the flow of execution to another point in the code. To iterate over arrays, dictionaries, ranges, strings, and other sequences, Swift uses for-in loops which makes everything easier. In order to iterate through ranges, it uses the syntax 1...5 within the for loop. This means that it will iterate through 1-5 inclusive. Matched values in a switch case can be bound to temporary constants or variables for use within the case's body, and complex matching conditions can be expressed with a where clause for each case.

Subroutines and Parameter Passing

Subroutines can be represented as a function that returns values or a procedure in Swift. A function is a block of code that can take in an input or parameters and can return a value. Within that block of code, a programmer could manipulate the parameters given or return a value according to the parameters given. In Swift, a programmer can name multiple functions the same name, but have different parameters requirements. On the other hand, procedures in Swift don't need to return a value. Procedures are useful when they are needed to print statements or manipulate global variables. Below is an example of how functions and procedures are made that has no parameters:

```
1  func sayHelloWorld() -> String {
2      return "hello, world"
3  }
4  print(sayHelloWorld())
5  // Prints "hello, world"
```

In this function, "sayHelloWorld", the function returns a String which is indicated by the -> arrow. It is then printed out by called the function using "sayHelloWorld()"

Next, parameters or inputs can be passed within a function. An example is shown below:

```swift
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

Within this example, the parameter person is being passed which is a String. This example also doesn't return a value, so it is an example of a procedure. Procedures are useful when you want to do simple print statements with parameters that are passed or with global variables.

Multiple parameters can be passed within a subroutine, but when the function is called the parameters inputted must be in the same order as the subroutine that was designed. The following is an example of multiple parameters being passed.

```swift
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

An exception is that parameters can not have the same name within the same subroutine. Any object, data structure, and data types can be a parameter or be returned.

Error Handling

Errors are represented by values of types that conform to the empty Error protocol, which indicates that a type can be used for error handling. A group of related error conditions and their additional information can be modelled conveniently with Swift enumerations, for example:

```swift
enum VendingMachineError: Error {
   case invalidSelection
   case insufficientFunds(coinsNeeded: Int)
   case outOfStock
}
```

To indicate that something unexpected happened, we throw an error with a **throw** statement.

```swift
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

Error handling in Swift resembles exception handling in other languages with the use of **try, catch,** and **throw** keywords. However, as the process in other languages involves unwinding the call stack, Swift avoids this potentially computationally expensive approach and makes the performance characteristics of a **throw** statement comparable to those of a **return** statement.

There are four ways to handle errors in Swift when an error is thrown.

1) Propagate the error from a function to the code that calls that function:

To indicate that a function, method, or initializer can throw an error, the **throws** keyword is written in the function's declaration after its parameters. This is called a *throwing function*. If the function specifies a return type, the **throws** keyword stands before the return arrow (->).

For example: `func canThrowErrors() throws -> String`

2) Handle the error using a **do-catch** statement.

A **do-catch** statement is used to handle errors by running a block of code. If an error is thrown by the code in the **do** clause, it is matched against the **catch** clauses to determine which one of them can handle the error. A general form of a **do-catch** statement is as follows:

```
do {
   try expression
   statements
} catch pattern 1 {
   statements
} catch pattern 2 where condition {
   statements
} catch pattern 3, pattern 4 where condition {
   statements
} catch {
   Statements
}
```

3) Handle the error as an optional value.

try? is used to handle an error by converting it to an optional value. If an error is thrown while evaluating the **try?** expression, the value of the expression is **nil**.

4) Assert that the error will not occur.

On occasions when it is known that a throwing function or method won't throw an error at runtime, **try!** can be written before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, a runtime error will occur.

Data Abstractions in Swift

Abstract classes aren't supported in Swift but Apple's Swift programming language has the notion of "protocol". Protocol-Oriented Programming is a new programming paradigm introduced by Swift 2.0 at the WWDC 2015. It was also the first programming language to come up with this paradigm.

In Protocol oriented programming people use concepts such as protocol inheritance, protocol composition and protocol extensions. In Swift, value types are preferred over classes and inheritance cannot be applied to value types because inheritance is an object oriented programming concept. Object oriented concepts do not apply on structs and enums meaning a

struct cannot inherit from a struct. On the other hand, value types in Swift can inherit from protocols, even multiple protocols.

Work Cited

https://docs.swift.org/swift-book/LanguageGuide/Functions.html
https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html
https://www.pluralsight.com/guides/protocol-oriented-programming-in-swift