Swift - The Basics - Variables, Types, Functions, Etc.

Mobile Computing - iOS

Objectives

- Students will be able to:
    - Create variables and constants in Swift using type annotation and type inference
    - Generate output to the console using print(), including with String interpolation
    - Describe the fundamental types
    - Construct ranges using open (..<) and closed (...) range operators
    - Construct sequences using stride(from:through:by:) and stride(from:to:by:)
    - Declare and use arrays
    - Write simple functions (including inout and variadic parameters)
    - Generate random numbers with arc4random()

Constants and Variables

- Use **let** to declare a constant

- Use **var** to declare a variable

- Constant and variable identifiers can contain almost any character, including Unicode (really)

```
let age:Int = 15
// age *= 2 // compilation error

var weight:Double = 75.0
weight *= 2 // now weight is 150.0
```

```
var 😀:String = "This is wild!"
print(😀)         // Output: This is wild!
print("😀")       // Output: 😀
print("\(😀)")    // Output: This is wild!
```

Comments

- Comments are for you, not the compiler.
- Comments in Swift are similar to comments in Java.
- Use // for a single-line comment.
- Use /* … */ for multi-line comments.

Semicolons

- Swift doesn't require semicolons at the end of a statement: we will *not* use them in this course.
- Semicolons *can* be used to write multiple statements on a single line (not recommended).

```
var width:Double = 100.0; var height:Double = 15.0
```

Swift is Strongly Typed

- Every variable must have a type associated with it… Two ways it can happen:

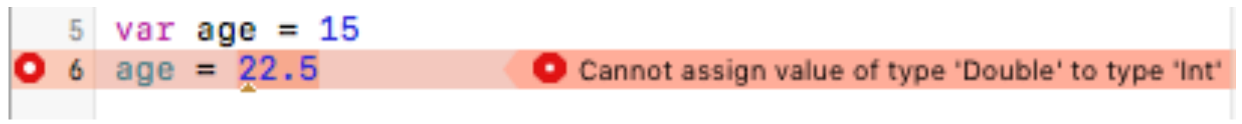
- **Type inference)** Omit the type when declaring a variable and assigning it and Swift will *infer* the type based on the assigned value.

- **Type annotation)** Explicitly declare the type of a variables or constants

- Once a variable has a type, it cannot be changed.

```
var age = 15  // type inference - age is an Int
let pi = 3.14 // pi is inferred to be Double
```

```
var age:Int = 15
let pi:Double = 3.141592654
```



```
var age = 15
age = 22.5
```

Cannot assign value of type 'Double' to type 'Int'

Swift Output and String Interpolation

- print() // prints any number of comma-delimited items
- print("Hello", "!", 124, 1.24e15)
- We can embed values inside a String by using **\\( )**
  - This is called **String interpolation**

```swift
var x = 15
print("x is \(x) and x squared is \(x*x)")
// output:  x is 15 and x squared is 225
```

Techy Aside*: Swift Output

- print() defaults to printing spaces between items and a newline character at the end. You can alter this by mentioning one or both explicitly

```
print(10, 20, 30, separator:":",terminator:"!")
//output: 10:20:30!
```

**Techy asides are good to know, but can be skipped on a first reading*

Types in Swift

- **Int**, Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64
- **Float**, Float32, Float64
- **Double**, Double32, Double64
- **Bool** (true or false)
- **String** (enclosed in " ")
- **Characters** -- uses " ", but can only include a single character in them

The bold types are the biggies — the others are variations that allow you to specify the size in bits explicitly. Most of the time you will use the bold types.

Techy Aside: Numeric Literals

Number	Prefix	Example
Decimal	N/A	8390 or 8_390
Binary	0b	0b1010
Octal	0o	0o7245
Hexadecimal	0x	0x7acd
Double	N/A	1.25, 1.25e-2

Using _ as a separator is brilliant❤️! Long live the _!

Techy Aside: Type Conversion

- To convert from one type to another, initialize a new number of the required type based on the other

```
var x:UInt32 = arc4random_uniform(10) // a random # between 0-9
var num:Int = Int(x)                  // creates a new Int, based on a UInt32
var dub:Double = Double(num)          // creates a new Double, based on an Int
var sub:Double! = Double("15.0")      // creates a Double! based on a String.
                                     // The ! means that sub is an optional
                                     // it can either store a Double or nil.
                                     // We will explore optionals later
```

Arrays

- Create an array using [ ] or an initializer.
- Elements are numbered starting at index 0
- Use **.count** to find the number of elements
- Generally we will use [*Type*] rather than Array<*Type*> when specifying an array type

```
var carsSold = [3784,2416,3925] // this is an array of Ints
var cast:[String] = ["Grant", "Danielle", "Candice", "Tom"]
var lightsOn:[Bool] = [] // an empty array
var salesForMarch:Array<Int> = [37,24,15] // FYI, Swift uses generics
var salesForYear:[Double] = Array(repeating: 0.0, count: 12)
carsSold[i] // ith element of carsSold
carsSold.count // # of elements in data
```

Some Helpful Methods for Array<T>

- init() — Init is invoked by the constructor. e.g., `var a = Array<Int>()`
- init(repeating:Int, count:T) —- changed from Swift 2.2 (argument order and names)
- mutating func append(_ newElement:T)
- mutating func insert(_ newElement:T, at:Int) —- changed from Swift 2.2 (argument name)
- mutating func remove(at:Int) -> T —- changed from Swift 2.2 (method name & parameter)
- mutating func removeLast() -> T
- mutating func removeAll()
- var count:Int { get }
- var isEmpty:Bool { get }
- mutating func sort(_ by:(T,T) -> Bool) e.g., `data.sort{$0 < $1}`

  —- changed from Swift 2.2 (argument name)
- func reversed() -> [T] —- changed from Swift 2.2 (method name)
- mutating func reverse()
- += (works only with two collections, not an individual T. e.g., `array += [2, 4]`

mutating means that the method can change the value of the object

Arrays @ Work

```swift
var values:[String] = [ ]
var values2:Array<String> = [ ]
var values3 = ["abc", "def", "ghi"]
var values4:[Int] = Array<Int>(repeating:15, count:5)

var data:[Int] = [10,20,30]
data += [40,50]
data.insert(15,at:1)
data.remove(at:0)
data = data.reverse()
data += data

var attic:[Any] = [1, 1.0, "Street"]
```

Any can be **anything**.
AnyObject can be any instance of a class.

You should always be as specific as you can about the type of item stored in your collection

for-in loops

- Use a for-in loop to iterate over an array

```
var names:[String] = ["Leela","Fry","Amy"]

for name in names { // need not declare name
    print(name)
}
```

for-in loops

- Use a for-in loop to iterate over a range of numbers
- Create ranges using the **closed range operator** (...) and the **half-open range operator** (..<)
- a ... b is a range from a to b inclusive: a, a+1, ... b-1, b
- a ..< b is a range from a to b (excluding b): a, a+1, ... b-1

```
for i in 0...4 {
    print(i, terminator:" ")    // 0 1 2 3 4

}


for i in 0..<4 {
    print(i, terminator:" ")   // 0 1 2 3
}
```

ICE: Arrays

- Declare an array, ages, of 5 Ints [13,26,19,22,35]
- Calculate their sum and maximum
- Use a for in loop to do so
- Print out the numbers from 0-100 using the range operator. Do this using both:
    - the closed range operator
    - the half open range operator

Processing an Array With an Explicit Index

- If we wish to number each element of an array, than an explicit index is required.

```
var names:[String] = ["Leela","Fry","Amy"]

for i in 0 ..< names.count {
   print("\(i): \(names[i])")
}
```

Taking Things in Stride

- The reader may be wondering ... what if we don't want to print out each element, but, say, every 2nd or every 3rd? For that, Apple provides two forms of the stride() function

- stride(from:through:by:) and stride(from:to:by:)

```
for i in stride(from: 0, to: 10, by: 2){          // 0 2 4 6 8
    print(i, terminator:" ")
}


for i in stride(from: 0, through: 10, by: 2){     // 0 2 4 6 8 10
    print(i, terminator:" ")
}
```

Mnemonic: **through** is longer than **to**, as is the resulting sequence

Techy Aside: Multidimensional Arrays

- Swift only defines a 1 dimensional array, but a 2D array can be constructed as an array of arrays (as in Java)

```
var profits:[[Int]] = [[10,20,30], [50, 20, 10], [11, 12, 13]]
print(profits[1][0]) // profits[1] is [50,20,10], so profits[1][0] is 50

var diagonalSum = 0
for j in 0 ..< profits.count {
    diagonalSum += profits[j][j]  // profits[0][0] + profits[1][1] + profits[2][2]
}

print(diagonalSum)  // 43
```

Techy Aside: Multidimensional Arrays

- Here we are calculating the sum of each row

```
var profits:[[Int]] = [[10,20,30], [50, 20, 10], [11, 12, 13]]

// calculates the sum of each row in a 2D array
// returns the row sums, as a 1D array
func rowSums(_ data:[[Int]] ) -> [Int] {

    var sumOfRows:[Int] = []

    for r in 0 ..< data.count {
        sumOfRows.append(0) // starting a new row, so add new element to sumOfRows
        for c in 0 ..< data[r].count { // for each column in that row
            sumOfRows[sumOfRows.count-1] += data[r][c] // add it to the last element
        }
    }
    return sumOfRows
}

rowSums(profits) // result: [60, 80, 36] [10+20+30, 50+20+10, 11+12+13]
```

if statements

- Swift's if statement is similar to what you have seen in other languages, with two main distinctions

  1. Parentheses are not needed around the condition: we will **not** use them.

  2. Curly brackets **are** required in all cases.

It is always handy to be able to generate random numbers ... arc4random_uniform() does the trick. Unfortunately it returns a UInt32, but we can make an Int out of it as shown.

```
var diceRoll:Int
var numOdds:Int = 0
var numEvens:Int = 0

for i in 0 ..< 1000 {
    diceRoll  = Int(arc4random_uniform(6)) + 1 // a number between 1-6

    if diceRoll == 1 || diceRoll == 3 || diceRoll == 5 {
        print("That's odd")
        numOdds += 1

    } else {
        print("That's even")
        numEvens += 1
    }
}

print("The ratio of odds to evens is \(Double(numOdds)/Double(numEvens))")
```

switch statements

- A switch statement in Swift looks like a switch statement in other languages, with a few differences:
  - The condition does not require parentheses (and we will not use them).
  - A case can have multiple comma-delimited values as well as intervals.
  - Break is not needed, as you cannot fall through to another case (unless you explicitly use the **fallthrough** keyword).

switch examples

```
var letter:Character = "a"

// A Character uses " ": Swift never uses ' '
// The closed range operator, ...,  denotes a range. Commas are used for an arbitrary list
// No breaks are required
// Notice that "q" falls through to "Q"
// Every case *must* have at least one executable statement

switch letter {
case "a" ... "m":  print("first half of alphabet")
case "A", "E", "I", "O", "U":  print("Capitalized vowels")
case "q":  fallthrough
case "Q":  print("This is the letter q")
default:  print("Some other letter")
}
```

Functions

- Functions, as all good programmers know, are named chunks of code designed to perform a particular task. Functions consist of:
  - the keyword **func**
  - the function name
  - a parameter list (in parentheses, comma-delimited, each a name:type pair)
  - -> followed by a return type
    - omit this if the return type is **Void**
  - the body of the function, in { }
- Functions are invoked by writing their name, and in parentheses, providing the arguments.

```
func helloWorld() {
    print("hello world")
}

helloWorld()

func sayHello(name:String) -> Void {
    print("Pleased to meetcha, ", name)
}

sayHello(name:"Michael")

func sayHello(name:String, inFrench:Bool) -> String {
    if inFrench {
        return "Bonjour, mon ami \(name)"
    } else {
        return "Hello, my friend \(name)"
    }
}

sayHello(name:"Michel", inFrench:true)

func mean(x:Double, y:Double, z:Double) -> Double {
    return (x + y + z) / 3.0
}

print(mean(x:25, y:15, z:35))
```

Argument Labels and Parameter Names

- Each function parameter has an argument label and a parameter name
- The **argument label (External)** is used to label an argument when the function is being called ("invoked")
  - At right, name and inFrench are labeling the arguments "Michel" and true
- By default, the argument label is the same as the **parameter name (Internal)** (name and inFrench)
- It is possible to provide explicit argument labels (to and enFrancais) by specifying them before the

```
func sayHello(name:String, inFrench:Bool) -> String {
    if inFrench {
        return "Bonjour, mon ami \(name)"
    } else {
        return "Hello, my friend \(name)"
    }
}

sayHello(name:"Michel", inFrench:true)
```

Argument labels

```
// argument labels are to and enFrancais
func sayHello(to name:String, enFrancais inFrench:Bool) -> String {

    if inFrench {
        return "Bonjour, mon ami \(name)"
    } else {
        return "Hello, my friend \(name)"
    }
}

sayHello(to:"Michel", enFrancais:true)
```

Eliminating Argument Labels

- If you do not want to use an argument label at all, write an underscore character, _, in lieu of an argument label.

```
func maximum(_ x:Double, _ y :Double) -> Double {
    return x >= y ? x : y
}

print(maximum(1000,150)) // prints 1000
```

- Often the first argument label is omitted, because it reads better if the label is incorporated into the name of the function.

```
func sayHelloTo( _ name:String, inFrench:Bool) -> String {
    if inFrench {
        return "Bonjour, mon ami \(name)"
    } else {
        return "Hello, my friend \(name)"
    }
}

sayHelloTo("Michel", inFrench:true)
```

```
func sayHello(to name:String, inFrench:Bool) -> String {
    if inFrench {
        return "Bonjour, mon ami \(name)"
    } else {
        return "Hello, my friend \(name)"
    }
}

sayHello(to:"Michel", inFrench:true)
```

To Summarize ...

```swift
func someFunction(parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}

someFunction(parameterName:25) // The argumentLabel is the parameterName.


func someFunction2(argumentLabel parameterName: Int) {
    // The argumentLabel is different from the parameterName. Inside the function,
    // use parameterName; outside, i.e., when invoking the function, use argumentLabel
}

someFunction2(argumentLabel:25)


func someFunction3(_ parameterName: Int) {
    // Use a _ as an argumentLabel to avoid having to write it in the invocation
}

someFunction3(25)
```

Working Backwards

- You can write the definition of a function based on how it is invoked. For example, if you saw:

```
var answer:Int = mystery(x:15, y:32, scramble:false)
```

- you would know that the function would look like something *like* this*

```
func mystery(x:Int, y:Int, scramble:Bool) -> Int}
```

*We say *like* this because we can't tell from the invocation if the programmer used explicit argument labels or not : e.g., this is also correct:
func mystery(x num0:Int, y num 1:Int, scramble:Bool) -> Int}

ICE: Functions

- Write the definition (first line) of these functions, based on their invocation:
  - echo(message:"hello", numTimes:15)
  - var result:Int = multiply(5, by:3)
- Write a function that is passed in an array of Ints and *returns*

  1. their sum
  2. how many are positive (call the function numPositive())
  3. if the elements are strictly increasing ([3,4,5]) or not ([3,2,5], [3,4,4,5])
  4. the minimum and maximum as an array of 2 elements
  5. an array consisting of only the positive elements (> 0).

In-Out Parameters

- For a function to modify* a parameter's value and have it be reflected in the argument, use **inout** before the parameter type in the function, and **&** before the argument in the call.

```
func doubleIt(a:inout Int){ a*=2 }
var eh = 5
print("eh was \(eh)")    // output: 5
doubleIt(a: &eh)
print("Now eh is \(eh)") // output: 10
```

*Not generally recommended. Returning a value is always available. You can even return multiple values conveniently using a tuple.

Variadic Parameters

To pass in a variable number of arguments as an array, use … after the declaration

```
func addEm(data:Int ...) ->Int {
    var sum = 0
    for num in data {
        sum += num
    }
    return sum

}

print("The sum is \(addEm(data: 3,4,18,2))")
```

Should we use data or _ data here?

We do not pass in an array, just a comma-delimited list of values. They are treated inside the function as an array.

ICE

- Write a function calcSqrt() that will be passed in a value, and change that value to be its square root
- var a:Double = 9.0
- calcSqrt(   )
- print(“Now a is \(a)”)

ICE 2

- Write a function, allEven(), that will be passed in an array of Ints, and return true if they are all even (divisible by 2), false otherwise
- Print out “They are all even” or “They are not all even” as appropriate
- Modify allEven() so that it is passed in a variable number of Ints
- Write a function, fibo(), that will be passed in a single int, n, and return an array containing the first n Fibonacci numbers.
- Write a function, multiply() that will be passed in an array an an Int. Multiply all the elements of the array by that Int, and return it

Resources

- The Swift Programming Language, Apple, Inc.
- Project Euler