String Theory - The Swift Edition

Mobile Computing - iOS

Objectives

- Students will be able to:
  - create String and Characters in Swift
  - know methods to perform common String operations (concatenate, insert, remove, find, divide)

Declaring Strings

- A string is a sequence of characters, represented in Swift by a <u>String</u> struct, housing a collection of Characters.

- Strings may be **immutable** (declared with **var**) or mutable (**let**)

```
let favoritePoet = "Edgar Allan Poe"        // immutable String literal
var favoriteMovie:String = "Guardians of the Galaxy" // mutable String
```

String Literals

- String literals uses " " , as in other languages
- For multiline Strings, use triple quotes """, with each """ on its own line
- Literals can include
    - special characters: **\0** (null), \\ (backslash), **\t** (horizontal tab), **\n** (line feed), **\r** (carriage return), **\"** (double quote) , **\'** (single quote)
    - Unicode scalars, written as **\u{n}** (where *n* = Unicode code point)

```
let favoriteStanza = """
I think that I shall never see
a \"poem\"
as lovely
as a \u{1F332}
"""
```

"I think that I shall never see\na "poem"\nas lovely\nas a 🌲"

Strings are Value Types

- Since Strings are structs, they are *value* types (like Ints and Doubles): if you assign a String, or pass it into or return it from a function, you get a new copy

```
var str1 = "Hello, Miller"
var str2 = str1
str1 += "!"
print(str2) // what gets printed?
```

[redacted]

Strings are Value Types

- Since Strings are structs, they are *value* types (like Ints and Doubles): if you assign a String, or pass it into or return it from a function, you get a new copy

```
var str1 = "Hello, Miller"
var str2 = str1
str1 += "!"
print(str2) // what gets printed?

// Output: Hello, Miller
```

Iterating Through a String's Characters

- Iterate through a String's characters with a for-in loop:

```
let favoriteMovie = "Guardians of the Galaxy 3"
var numGs = 0

for ch in favoriteMovie {
    if ch == "G" || ch == "g" {
        numGs += 1
    }
}
var percentageGs = Double(numGs)/Double(favoriteMovie.count) * 100.0
print(String(format: "%% of Gs = %6.2f",percentageGs))
print("This movie is \(favoriteMovie.count) characters long")

// Output: This movie is 25 characters long
```

Characters

- A Character looks like a 1-character String — a literal can be created by placing a single character in " " but it is a separate type

- A Character can represent any valid Unicode

- Q: Do you think we could get away with the statement below using type inference?

```
let gender:Character = "F" // gender is a Character, not a String
```

Techy Aside: Extended Grapheme Clusters

- A Character is an extended grapheme cluster — one *or more* Unicode scalars that when combined form a human-readable character

- There may be multiple ways to form a character, and explains why the number of Unicode scalars in a string may differ from the number of characters in a string, and string concatenation/modification may not actually change a string's count.

```
let eAcute: Character = "\u{E9}"                         // é
let combinedEAcute: Character = "\u{65}\u{301}"          // e followed by´
// eAcute is é, combinedEAcute is é
```

Techy Aside: Extended Grapheme Clusters and A Magic Trick

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

word += "\u{301}"    // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
```

A string has been concatenated to word, yet word's count remains the same

Concatenating Strings

- Strings can be concatenated with Strings using + or += as appropriate. A Character can be concatenated to a String using append(:).

```
let start = "I have a"
let end = "feeling about this!"
var middle = start + " bad " + end    // I have a bad feeling about this!

let message = "I will not text in class\n"
var assignment = ""

for i in 1 ... 100 {
    assignment += "\(i). \(message)"
}
print("\(assignment.count)") // 2892

var remarks = "Good job"
remarks.append("!")
```

```
          ...
41. I will not text in class
42. I will not text in class
43. I will not text in class
44. I will not text in class
45. I will not text in class
46. I will not text in class
47. I will not text in class
          ...
```

Counting Characters

- The length of a String is given by its **count** property
- Use **isEmpty** to determine if a String is empty

```
let MAX_NAME_LENGTH = 20
@IBOutlet weak var lastNameTF:UITextField!

@IBAction func handleTap(sender:AnyObject){

    if lastNameTF.text!.isEmpty {
        print("User forgot to enter their name")
    } else {
        if lastNameTF.text!.count > MAX_NAME_LENGTH {
            print("User entered too long a name")
        }
    }
}
```

Q: Why do we write text! rather than text. ?
Q: What if we wrote text? instead of text! ?

Counting Characters

- The length of a String is given by its **count** property
- Use **isEmpty** to determine if a String is empty

```
let MAX_NAME_LENGTH = 20
@IBOutlet weak var lastNameTF:UITextField!

@IBAction func handleTap(sender:AnyObject){

    if lastNameTF.text!.isEmpty {
        print("User forgot to enter their name")
    } else {
        if lastNameTF.text!.count > MAX_NAME_LENGTH {
            print("User entered too long a name")
        }
    }
}
```

A: We need the ! to unwrap the optional String
A: If we wrote ?, using optional chaining, isEmpty would be optional (and need to be unwrapped)

A Magic Trick: Extended Grapheme Clusters

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

word += "\u{301}"    // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
```

Accessing & Modifying Strings Using [ ]

- A String can be accessed/modified using **[ ]**, in order to access individual Characters or Substrings.

- However, the subscript type is not an Int, but rather **String.Index**

    - Index is a nested type defined in the String class, hence the **.**

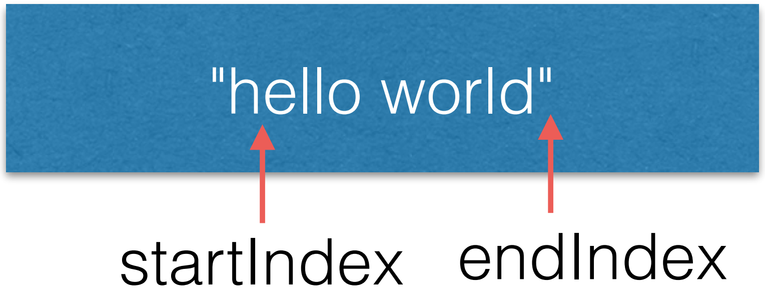
    - The need to use String.Index, rather than Ints, is a byproduct of Strings being fully Unicode compliant

Focus on String.Index

String.Index

- anyStringWhatsoever[ ]

String.Index



"hello world"

startIndex endIndex

- String defines two properties:
  - **startIndex** - position of the first character in the string
  - **endIndex** - position one beyond the last character in the string
- and 3 **index()** methods to generate other positions in the string: the one *just before*, *just after*, and a given number of positions from the first argument.
  - index(before: String.Index) -> String.Index
  - index(after: String.Index) -> String.Index
  - index(_ i: String.Index, offsetBy: Int) -> String.Index

An Example

startIndex endIndex

```
let quote = "Three to beam up"
var loc:String.Index

quote[quote.startIndex] // "T"

loc = quote.index(before:quote.endIndex) // index 15
quote[loc] // "p"

loc = quote.index(after:quote.startIndex) // index 1
quote[loc] // "h"

loc = quote.index(quote.endIndex, offsetBy: -1) // index 15
quote[loc] // "p"

loc = quote.index(quote.startIndex,offsetBy: 2) // index 2
quote[loc] // "r"
```

Iterating Through a String's Characters Using [ ]

```
let quote = "Three to beam up"
var index:String.Index

for i in stride(from:0,to:quote.count,by:2) {

    index = quote.index(quote.startIndex,offsetBy:i)
    print(quote[index], terminator:"") // Tret emu

}

// Output: Tret emu
```

Our Hero, .encodedOffset 😎

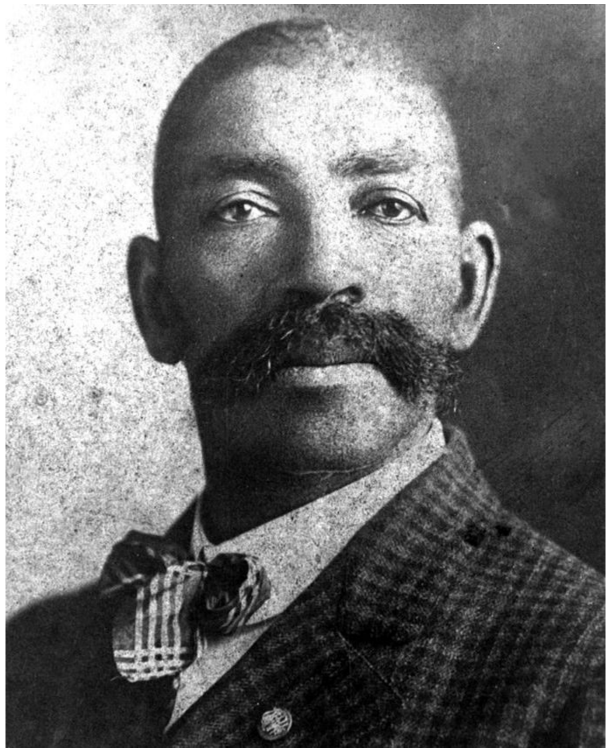
- Printing a String.Index does not get a useful value (at least when living in an ASCII world).

- Use the **.encodedOffset** property to get the offset, assuming a UTF-16 representation

```
var message = "Aim high!"

var loc = message.index(of:"m")
print(loc?.encodedOffset) // Optional(2)
```

Range



The real Lone Ranger!

- When working with more than one character at a time in a String, we need to use a **Range** struct.
- We have already used Range many times in for-in loops
- A range has two properties, **lowerBound** and **upperBound**.
- It is **generic**: you can have Range<Int>, Range<String.Index>, Range<Character>, etc.

Range Example

```
let range:Range<Int> = 0 ..< 10              // You can *store* a Range😍
print(range.lowerBound, range.upperBound)    // output: 0 10
print(range.contains(4), range.contains(10)) // output: true false


var ranger = "a" ..< "z" // Range<String> 😍 -- Range and ClosedRange
                         // work with any Comparable type

// var ranger2 = "z" ..< "a" // Fatal error: Can't form Range with
                             // upperBound < lowerBound


for i in 0 ... 10 { // 0 ... 10 is an ClosedRange instance
    print(i)
}

var closedRange:ClosedRange<Int> = 0 ... 10
print(closedRange.lowerBound, closedRange.upperBound) // output: 0 10
print(closedRange.contains(0), closedRange.contains(10)) // true, true
```

Finding a Character

- **index(of:Character) -> String.Index?**
  (nil if character is missing)

```
var phrase = "Josepheus Miller"

let spaceIndex = phrase.index(of:" ") // location of the space
let hashIndex = phrase.index(of:"#")  // nil
```

Finding a String

- **contains(_:String) -> Bool**
- **range(of:String) -> Range<String.Index>**

In hindsight, it should not be ***too*** surprising that when seeking a String's location, we get a Range <String.Index>

Finding a String Example

```
var answer = "Take two and call me in the morning"

if answer.contains("two"){
    let ranger:Range<String.Index>? = answer.range(of:"two")
    let lowerIndex:String.Index = ranger!.lowerBound
    let upperIndex:String.Index = ranger!.upperBound

    answer[lowerIndex ... upperIndex]
}
```

What will this be?

Finding a String Example

```
var answer = "Take two and call me in the morning"

if answer.contains("two"){
    let ranger:Range<String.Index>? = answer.range(of:"two")
    let lowerIndex:String.Index = ranger!.lowerBound
    let upperIndex:String.Index = ranger!.upperBound

    answer[lowerIndex ... upperIndex] // "two"
}
```

Finding Prefixes and Suffixes

- **hasPrefix(_:String) -> Bool**
- **hasSuffix(_:String) -> Bool**

Finding Prefixes and Suffixes Example

```
var ladiesAndLords:[String] = ["Sir Elton John", "Lady Gaga",
                               "Sir Humphrey Applebee", "Lady Ada", ]
var titleFreeLords:[String] = []
for var person in ladiesAndLords {
    if person.hasPrefix("Sir "){
        person.removeSubrange(person.range(of:"Sir ")!) // remove "Sir "
        titleFreeLords.append(person)
    }
}

print(titleFreeLords)
print(ladiesAndLords)
// Output:
```

What will this be?

Finding Prefixes and Suffixes Example

```
var ladiesAndLords:[String] = ["Sir Elton John", "Lady Gaga",
                               "Sir Humphrey Applebee", "Lady Ada", ]
var titleFreeLords:[String] = []
for var person in ladiesAndLords {
    if person.hasPrefix("Sir "){
        person.removeSubrange(person.range(of:"Sir ")!) // remove "Sir "
        titleFreeLords.append(person)
    }
}

print(titleFreeLords)
print(ladiesAndLords)
// Output:
// ["Elton John", "Humphrey Applebee"]
// ["Sir Elton John", "Lady Gaga", "Sir Humphrey Applebee", "Lady Ada"]
```

Inserting

- **insert(_:at:)** -- inserts a single character at String.Index
- **insert(contentsOf:at:)** -- inserts a String at String.Index

Inserting Example

```
var quote = "Ceres once in ice"

quote.insert(".", at:quote.endIndex)
// Ceres once in ice.

quote.insert(contentsOf: "was ", at: quote.index(of:"o")!)
// Ceres was once in ice.

quote.replacingOccurrences(of: "once", with: "once covered")
// Ceres was once covered in ice.
```

Removing

- **remove(at:)** -- removes 1 character at String.Index
- **removeSubrange(_:)** -- removes a range of Characters

Removing Example

```
var quote = "Remember the Cantebury!"

quote.remove(at:quote.index(before:quote.endIndex))
// Remember the Cantebury

var buryLoc = quote.range(of:"ebury")! // finds location of "ebury"
quote.removeSubrange(buryLoc)
// Remember the Cant
```

Substrings

- When you take a slice of a String, the result is a **Substring**, a distinct type that **shares String's API**: the same methods work with Substrings as Strings

```
let message  = "hello world"
var left:String.Index
var right:String.Index
var blank:String.Index? = message.index(of: " ") // find the blank, post 5
// why does index(of:) return an optional?

left = message.index(message.startIndex,offsetBy:3) // position 3
right = message.index(message.endIndex,offsetBy:-3) // position 8
message[left]
message[right]
message[left...]
message[..<blank!]
message[...blank!]
message[blank!...]

message[left...right]
```

What will this be?

Substrings

- When you take a slice of a String, the result is a **Substring**, a distinct type that **shares String's API**: the same methods work with Substrings as Strings

```
let message  = "hello world"
var left:String.Index
var right:String.Index
var blank:String.Index? = message.index(of: " ") // find the blank, post 5
// why does index(of:) return an optional?

left = message.index(message.startIndex,offsetBy:3) // position 3
right = message.index(message.endIndex,offsetBy:-3) // position 8
message[left]          // "l" (the middle one)
message[right]         // "r"
message[left...]       // lo world
message[..<blank!]     // "hello" -- up to but not including the blank
message[...blank!]     // "hello " -- up to and including the blank
message[blank!...]     // " world"

message[left...right] // "lo wor"
```

More on Substrings

- A **substring** is a slice of a string.
- When you create one, it shares storage with the original string
- It is preferred because it is faster and more efficient to use substrings rather than generate an entirely new string
- **Substring** is aka **SubSequence**. Both are within the String struct, so technically they are String.Substring and String.SubSequence.
- To convert a Substring back into a String, use the String() initializer

Miscellaneous Helpful Properties/Methods

- **Changing Case:**
  - **capitalized (String)**
  - **lowercased() -> String**
  - **uppercased() -> String**
- **Converting back & forth between primitives and Strings:**
  - "\(*primitive*)" or String(format:) or *primitive*.description
  - Int(*stringRepresentation*), Double(*stringRepresentation*), etc.
  - Remember: these initializers return Int? and Double? — you will need an ! to unwrap their values (or assign them to an Int! or Double! to skip the !).

Divide ~~and Conquer~~

- **components(separatedBy:String) -> [String]**
- **split(separator:Character) -> [Substring]**

Divide Example

```
let ageLine = agesTF.text! // e.g., 37, 42, 15, 9
let ages:[String] = ageLine.components(separatedBy: ", ") //["37","42","15","9"]
var totalAge = 0.0

for age in ages {
    if let realAge = Double(age) { // convert String to Double (why in if-let?)
        totalAge += realAge        // and sum
    }
}

if ages.count > 0 {
    averageAgeLBL.text = "\(totalAge/Double(ages.count))"
}
```

Comparing Strings

- ==   (Swift also has a === operator that checks to see if two instances of a class are in the same location, but it is not applicable to Strings. Why not?)
- func compare(String) -> ComparisonResult
- enum ComparisonResult : Int {case OrderedAscending, OrderedSame, OrderedDescending}

Comparing Strings

```
//drink and drink2 are equal, as their extended grapheme clusters are canonically equivalent -- they have the
meaning and appearance, although they are composed of different Unicode code points

var drink = "caf\u{E9}"          // "café" -- u+0059 == LATIN SMALL LETTER E WITH ACUTE
var drink2 = "caf\u{65}\u{301}" // "café" -- u+0065 == LATIN SMALL LETTER E + U0301 == COMBINING ACUTE ACCENT

print(drink.count == drink2.count) // true

if drink == drink2 {
    print("The drinks are the same “)
}

"apple".compare("orange") == .orderedAscending // true
"orange".compare("apple") == .orderedDescending // true
```

Techy Aside: Unicode

- Unicode is an internationally recognized standard for representing text. It can represent text in 139 languages, and consists of 136,755 characters.

- Each character in Unicode has a unique *name* and *number*. The latter is known as a **code point**, and written as U+*n* (where *n* is a 4-6 digit hexadecimal value)

- The entire code space, from 0 to FFFFFF, is divided up into planes, groups of 65536 code points.

- Plane 0, the basic multilingual plane (BMP), consists of code points 0000-FFFF, and includes many of the major languages. See Wikipedia for details.

Techy Aside: Unicode

- e.g., "Hello" is U+0048 U+0065 U+006C U+006C U+006F.
- In Swift, we would write:

```
let greeting = "\u{0048}\u{0065}\u{006c}\u{006c}\u{006F}" // "Hello"
```

- Beyond basic characters for all the major languages, numbers, punctuation, Emojis and symbols of all sorts, Unicode supports diacritics -- modifying character marks such as tildes (~), accents (´), etc. that can be combined with base characters to form accented ones.
    - e.g., é = "e" + "´"

Techy Aside: Unicode Encodings

- **Encoding** is how code points are represented in bits in a computer. Code points are up to 6 bytes long, so you might think that each character in a text file would require 6 bytes. However, most characters can fit in 2 bytes (the BMP plane), and English letters in just 1 byte.

- Consequently, Unicode has 3 encoding forms, that specify how many bytes are read/interpreted at a time. **UTF-8** specifies 8 bits, **UTF-16** 16 bits, and **UTF-32** 32 bits.

- All 3 forms can be used to encode any of the 1,112,064 code points in Unicode.

- UTF-8 is by far the most popular encoding form, and so we will concentrate on it.   (maps onto it)

Techy Aside: UTF-8

- What if you are using UTF-8, processing 1 byte at a time, but have a character that takes 2 bytes -- e.g., ᘑ (U+1611, CANADIAN SYLLABICS CARRIER YEE, or 3 bytes -- e.g., 🌛 (U+1F31B,FIRST QUARTER MOON WITH FACE) , OR 🃟 (U+1F0DF, PLAYING CARD WHITE JOKER)?

- Unicode uses **continuation bytes** to indicate, in UTF-8, how many bytes to group together to form a multi-byte sequence. Specifically, if a byte's first bit is 0, then only that byte needs to be read; if its first 3 bits are 110, then a second byte also needs to be read; if its first 4 bits are 1110, then 3 bytes are required.

- Since the number of high-order 1's in a byte indicate the number of bytes in a sequence, this makes it easier for programmers to process multi-byte sequences

- UTF-8 also maintains backward compatibility with ASCII. The ASCII characters (with ASCII values 0-127, i.e., all start with a 0 bit) have the same Unicode code point.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Exercises

```
var message = "It was the best of times"

// find the number of characters in message
// concatenate "!!" to message
// print the character after "I"
// print the character before the last "!"
// determine and then print the character 3 removed from the start
// determine and then print the character 6 removed from the end
// count the number of "t" characters
// print the even numbered characters (I a h e ...)
// print the location (String.Index) of "worst"
// print the location (String.Index) of "o"
// See if message contains "really"
// insert "very" just before " best"
// remove "very"
// see if message has a suffix of "times"
// remove the very first character of message
// insert "i" at the very beginning of message
// print a lowercased version of message
// print out each word of message (by dividing it)
```

String API Summary

- .count (Int)
- .isEmpty (Bool)
- index(before:) -> String.Index
- index(after:) -> String.Index
- index(_:offsetBy:) -> String.Index
- index(of:Character) -> String.Index
- contains(_:) -> Bool
- range(of:String) -> Range<String.Index>
- hasPrefix(_:) -> Bool
- hasSuffix(_:) -> Bool
- insert(_:at:)
- insert(contentsOf:at:)
- remove(at:)
- removeSubrange(_:)
- .capitalized
- lowerCased()
- upperCased()
- components(separatedBy:) -> [String]
- split(separator:) -> [Substring]

References

- https://developer.apple.com/library/tvos/documentation/Swift/Reference/Swift_String_Structure/index.html

- https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html

- https://pythonconquerstheuniverse.wordpress.com/2010/05/30/unicode-beginners-introduction-for-dummies-made-simple/