Lecture 6 – String, Closure

Developing Applications for iOS

CS193p
Fall 2017-18

1

Advanced use of Protocols

"Multiple inheritance" with protocols

CountableRange/ Range

```
for n in 3..<5
{
    print(n)
}
```

```
for index in cards.indices
{
        cards[index].isFaceUp = false
        cards[index].isMatched = false
}
```

Creating a Range

Create a new range using the half-open range operator (..<).

```
static func ..< (Self, Self) -> Range<Self>
```

Returns a half-open range that contains its lower bound but not its upper bound.

```
let underFive = 0..<5
```

CS193p
Fall 2017-18

2

Advanced use of Protocols

"Multiple inheritance" with protocols

CountableRange

implements many protocols, but here are a couple of important ones …

- Sequence — makeIterator (and thus supports for in)
- Collection — subscripting (i.e. []), index(of:), etc.

```
let underFive = 0..<5
print(underFive[4])     // 4

let i = underFive.index(of:3)
print(i!)                    //3
```


3

Advanced use of Protocols

"Multiple inheritance" with protocols

CountableRange

implements many protocols, but here are a couple of important ones …

- Sequence — makeIterator (and thus supports for in)
- Collection — subscripting (i.e. []), index(of:), etc.

Why do it this way?

Because Array, for example, also implements these protocols.
So now Apple can create generic code that operates on a Collection and it will work on both!
Dictionary is also a Collection, as is Set and String.

And they don't all just inherit the fact that they implement the methods in Colle
they actually inherit an implementation of many of the methods in Collection

```
let myArray = [1 2 3 4]
for i in myArray
{
    print(i)
}
```

4

String

The characters in a String

String is also a Collection of Characters
All the indexing stuff (index(of:), etc.) is part of Collection.
A Collection is also a Sequence, so you can do things like …

for c in s { } // iterate through all Characters in s

let characterArray = Array(s) // Array<Character>
(Array has an init that takes any Sequence as an argument.)

```
let str = "test"
for c in str {
    print(c)
}
```

```
let characterArray = Array(str)   //['t','e','s','t']
print(characterArray[3])          //t
```

CS193p
Fall 2017-18

5

String

The characters in a String

A String is made up of Unicodes, but there's also the concept of a Character.
A Character is what a human would perceive to be a single lexical character.
This is true even if a single Character is made up of multiple Unicodes.

E.g., café might be 4 Unicodes (c-a-f-é) or 5 Unicodes (c-a-f-e-').
In either case, we preceive it as 4 Characters.

- there is a Unicode which is "apply an accent to the previous character".
- But there is also a Unicode which is é (the letter e with an accent on it).

CS193p
Fall 2017-18

6

String

The characters in a String

- Because of this ambiguity, the index into a String cannot be an Int.
  - Is the p in "café pesto" at index 5 or index 6?
  - Depends on the é.
- Indices into Strings are therefore of a different type …
  - String.Index.
- The simplest ways to get an index are
  - startIndex, endIndex and index(of:)
  - There are other ways (see the documentation for more)
- To move to another index, use index(String.Index, offsetBy: Int)

CS193p
Fall 2017-18

7

String

The characters in a String

```
let pizzaJoint = "café pesto"
let firstCharacterIndex = pizzaJoint.startIndex // of type String.Index
let fourthCharacterIndex = pizzaJoint.index(firstCharacterIndex, offsetBy: 3)
let fourthCharacter = pizzaJoint[fourthCharacterIndex] // é

if let firstSpace = pizzaJoint.index(of: " ") { // returns nil if " " not found
    let secondWordIndex = pizzaJoint.index(firstSpace, offsetBy: 1)
    let secondWord = pizzaJoint[secondWordIndex..<pizzaJoint.endIndex]
}
```

8

String

The characters in a String

Another way to find the second word:

pizzaJoint.components(separatedBy: “ ”)[1]

components(separatedBy:) returns an Array<String> (might be empty, though, so careful!)

```
let pizzaJoint = "café pesto"                         "café pesto"
pizzaJoint.components(separatedBy:" ")[1]             "pesto"
```

```
let pizzaJoint = "cafépesto"                          "cafépesto"
pizzaJoint.components(separatedBy:" ")[0]             "cafépesto"
```

9

Demo

Change our emojiChoices to be a String

It really doesn’t matter either way

But it’s a good opportunity to compare String and Array (which are surprisingly similar) We’ll also get a little bit of insight into the protocol-based design of the Foundation framework

```
var emojiChoices = ["🦇","🦐","👻","🎃","🦋","🐥","🌸","⭐️","🌹"]
```

```
var emojiChoices = "🦇🦐👻🎃🦋🐥🌸⭐️🌹"
```

CS193p
Fall 2017-18

12

```swift
var emojiChoices = ["🦇","🦐","👻","🎃","🦋","🐤","🌸","⭐","🌹"]

func getEmoji(for card: Card) -> String
    {
        if emojiDic[card] == nil , emojiChoices.count > 0 {
         let randomIndex = emojiChoices.count.arc4random
         emojiDic[card] =  emojiChoices.remove(at: randomIndex)
        }
        return emojiDic[card] ?? "?"
    }
```

```swift
var emojiChoices = "🦇🦐👻🎃🦋🐤🌸⭐🌹"

func getEmoji(for card: Card) -> String
    {
        if emojiDic[card] == nil , emojiChoices.count > 0 {
            let randomStringIndex = 

            emojiDic[card] = String(emojiChoices.remove(at:randomStringIndex))
        }
        return emojiDic[card] ?? "?"
    }
```



```swift
var emojiChoices = ["🦇","🦐","👻","🎃","🦋","🐤","🌸","⭐","🌹"]

func getEmoji(for card: Card) -> String
    {
        if emojiDic[card] == nil , emojiChoices.count > 0 {
         let randomIndex = emojiChoices.count.arc4random
         emojiDic[card] =  emojiChoices.remove(at: randomIndex)
        }
        return emojiDic[card] ?? "?"
    }
```

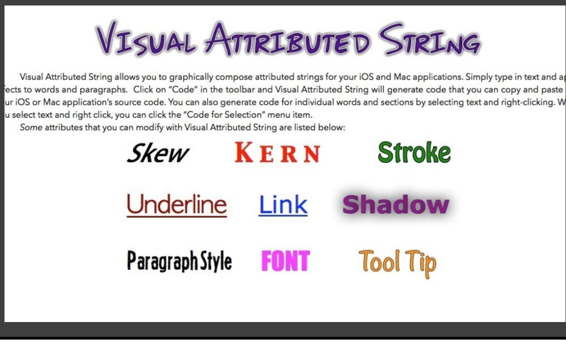
```swift
var emojiChoices = "🦇🦐👻🎃🦋🐤🌸⭐🌹"

func getEmoji(for card: Card) -> String
    {
        if emojiDic[card] == nil , emojiChoices.count > 0 {
            let randomStringIndex = emojiChoices.index(emojiChoices.startIndex,
                                              offsetBy: emojiChoices.count.arc4random)
            emojiDic[card] = String(emojiChoices.remove(at:randomStringIndex))
        }
        return emojiDic[card] ?? "?"
    }
```

NSAttributedString

A String with attributes attached to each character

Conceptually, an object that pairs a String and a Dictionary of attributes for each Character.
- The Dictionary's keys are things like "the font" or "the color", etc.
- The Dictionary's values depend on what the key is (UIFont or UIColor or whatever).



CS193p
Fall 2017-18

15

NSAttributedString

A String with attributes attached to each character

Conceptually, an object that pairs a String and a Dictionary of attributes for each Character.
- The Dictionary's keys are things like "the font" or "the color", etc.
- The Dictionary's values depend on what the key is (UIFont or UIColor or whatever).

Many times (almost always), large ranges of Characters have the same Dictionary.
Often the entire NSAttributedString uses the same Dictionary.
You can put NSAttributedStrings on UILabels, UIButtons, etc.

16

NSAttributedString

Creating and using an NSAttributedString

Here's how we'd make the flip count label have orange, outlined text …

```
let attributes: [NSAttributedStringKey : Any] = [ // note: type cannot be inferred here
    .strokeColor : UIColor.orange,
    .strokeWidth : 5.0 // negative number here would mean fill (positive means outline)
]
let attribtext = NSAttributedString(string: "Flips: 0", attributes: attributes)
flipCountLabel.attributedText = attribtext // UIButton has attributedTitle
```

CS193p
Fall 2017-18

17

NSAttributedString

```
#define FONT_SIZE 20
#define FONT_HELVETICA @"Helvetica-Light"
#define BLACK_SHADOW [UIColor colorWithRed:40.0f/255.0f green:40.0f/255.0f blue:40.0f/255.0f alpha:0.4f]

NSString*myNSString = @"This is my string.\nIt goes to a second line.";

NSMutableParagraphStyle *paragraphStyle = [[NSMutableParagraphStyle alloc] init];
paragraphStyle.alignment = NSTextAlignmentCenter;
paragraphStyle.lineSpacing = FONT_SIZE/2;

UIFont * labelFont = [UIFont fontWithName:FONT_HELVETICA size:FONT_SIZE];
UIColor * labelColor = [UIColor colorWithWhite:1 alpha:1];

NSShadow *shadow = [[NSShadow alloc] init];
[shadow setShadowColor : BLACK_SHADOW];
[shadow setShadowOffset : CGSizeMake (1.0, 1.0)];
[shadow setShadowBlurRadius : 1];

NSAttributedString *labelText = [[NSAttributedString alloc] initWithString : myNSString
                                 attributes : @{ NSParagraphStyleAttributeName : paragraphStyle,
                                                 NSKernAttributeName : @2.0,
                                                 NSFontAttributeName : labelFont,
                                                 NSForegroundColorAttributeName : labelColor,
                                                 NSShadowAttributeName : shadow }];
```

18

NSAttributedString

Peculiarities of NSAttributedString

NSAttributedString is a completely different data structure than String.
The "NS" is a clue that it is an "old style" Objective-C class.
Thus it is not really like String (for example, it's a class, not a struct).

Since it's not a value type, you can't create a mutable NSAttributedString by just using var.
To get mutability, you have to use a subclass of it called NSMutableAttributedString.
NSAttributedString was constructed with NSString in mind, not Swift's String.
NSString and String use slightly different encodings.There is some automatic bridging between old Objective-C stuff and Swift types.

But it can be tricky with NSString to String bridging because of varying-length Unicodes.
This all doesn't matter if the entire string has the same attributes.
Or if the NSAttributedString doesn't contain "wacky" Unicode characters.
Otherwise, be careful indexing into the NSAttributedString.

CS193p
Fall 2017-18

19

Demo

Make flip count outlined text

Let's apply the code from the previous slide to Concentration

```
var flipCount = 0{
        didSet{

            flipCountLabel.text = "Flips:\(flipCount)"

        }
    }
```

CS193p
Fall 2017-18

20

```
var flipCount = 0{
        didSet{
           let attributes: [NSAttributedStringKey:Any] = [
               .strokeWidth: 5.0,
               .strokeColor:UIColor.orange
           ]
           let attributedString = NSAttributedString(string: "Flips:\(flipCount)", attributes: attributes)
           flipCountLabel.attributedText = attributedString
           //flipCountLabel.text = "Flips:\(flipCount)"
       }
   }
```

21

```
var flipCount = 0{
     didSet{
        updateFlipCountLabel()
    }
}

private func updateFlipCountLabel(){

    let attributes: [NSAttributedStringKey:Any] = [
        .strokeWidth: 5.0,
        .strokeColor:UIColor.orange
    ]
    let attributedString = NSAttributedString(string: "Flips:\(flipCount)", attributes: attributes)
    flipCountLabel.attributedText = attributedString
}


@IBOutlet weak var flipCountLabel: UILabel!{
        didSet{
            updateFlipCountLabel()
        }
    }
```

22

FUNCTION & CLOSURE

23

Function Types

Function types

Functions are types too!
You can declare a variable to be of type "function"
You'll declare it with the types of the functions arguments (and return type) included

Example:

```
var operation: (Double) -> Double
    //This is a var called operation
    //It is of type "function that takes a Double and returns a Double"
```

24

Function Types

Function types

Functions are types too!
You can declare a variable (or parameter to a method or whatever) to be of type "function" You'll declare it with the types of the functions arguments (and return type) included

Example:

```
var operation: (Double) -> Double

//You can assign it like any other variable ...
operation = sqrt    // sqrt is just a function that takes a Double and returns a Double

//You can "call" this function using syntax very similar to any function call ...
let result = operation(4.0) // result will be 2.0
```

25

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt).
You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument ...

```
func changeSign(operand: Double) -> Double
{
    return -operand
}
```

We could use it instead of sqrt ...

```
var operation: (Double) -> Double
operation = changeSign
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

27

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt).
You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
func changeSign(operand: Double) -> Double
{
   return -operand
}
```

We can "in line" changeSign simply by moving the function (without its name) below …

```
var operation: (Double) -> Double
operation = changeSign
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

28

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt).
You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

~~func changeSign(operand: Double) -> Double~~
~~{~~
~~return -operand~~
~~}~~

We can "in line" changeSign simply by moving the function (without its name) below …

```
var operation: (Double) -> Double
operation = (operand: Double) -> Double { return -operand } //not finish yet
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

29

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

A minor syntactic change: Move the first { to the start and replace with in …

```
var operation: (Double) -> Double
operation = (operand: Double) -> Double { return -operand } //not finish yet
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

30

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

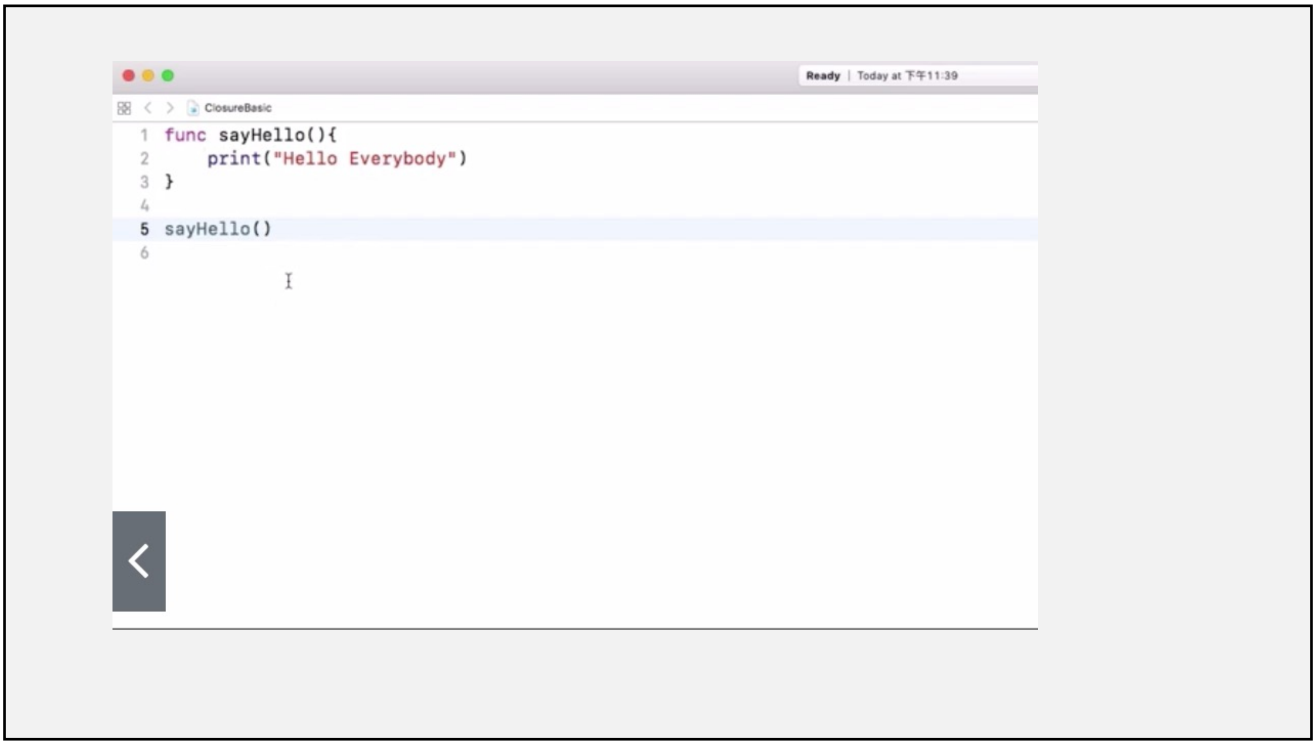
Imagine we had a function that changed the sign of its argument …

A minor syntactic change: Move the first { to the start and replace with in …

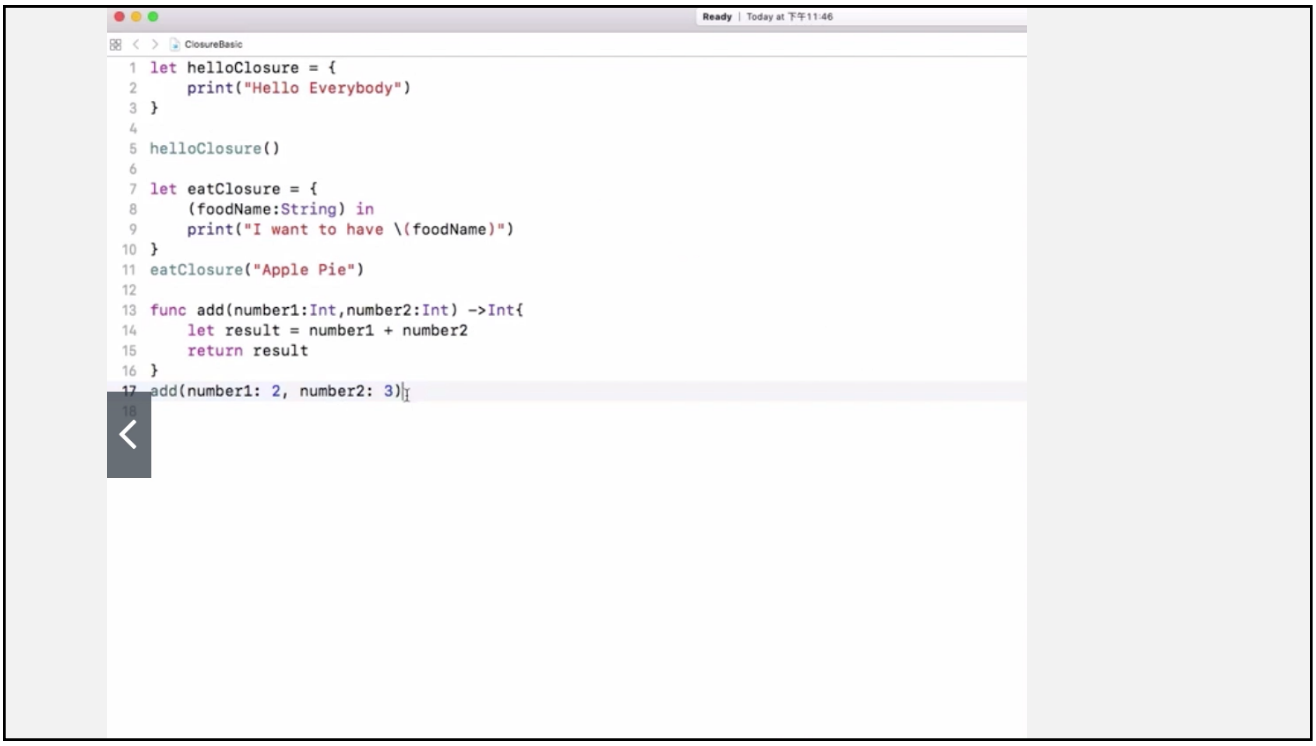
```
var operation: (Double) -> Double
operation = { (operand: Double) -> Double in return -operand }        //OK
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

31


Ready | Today at 下午11:39
ClosureBasic
func sayHello(){
print("Hello Everybody")
}
sayHello()

32


Ready | Today at 下午11:46
ClosureBasic
let helloClosure = {
print("Hello Everybody")
}
helloClosure()
let eatClosure = {
(foodName:String) in
print("I want to have \(foodName)")
}
eatClosure("Apple Pie")
func add(number1:Int,number2:Int) ->Int{
let result = number1 + number2
return result
}
add(number1: 2, number2: 3)

33

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt).
You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
func changeSign(operand: Double) -> Double
{
   return -operand
}
```

Move the first { to the start and replace with in …

```
var operation: (Double) -> Double
operation = { (operand: Double) -> Double in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

34

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt).
You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

Swift can infer that operation returns a Double

```
var operation: (Double) -> Double
operation = { (operand: Double) -> Double in return -operand }
let result = operation(4.0) // result will be -4.0
```

35

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

Swift can infer that operation returns a Double

```
var operation: (Double) -> Double
operation = { (operand: Double) ~~-> Double~~ in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

36

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

Swift can infer that operation returns a Double

```
var operation: (Double) -> Double
operation = { (operand: Double) in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

37

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

Swift can infer that operation returns a Double and that operand is a Double

```
var operation: (Double) -> Double
operation = { (operand: Double) in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

38

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

Swift can infer that operation returns a Double and that operand is a Double

```
var operation: (Double) -> Double
operation = { (operand: Double) in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

39

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

Swift can infer that operation returns a Double and that operand is a Double

```
var operation: (Double) -> Double
operation = { (operand) in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

40

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

It also knows that operation returns a value, so the return keyword is unnecessary

```
var operation: (Double) -> Double
operation = { (operand) in return -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

41

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

It also knows that operation returns a value, so the return keyword is unnecessary

```
var operation: (Double) -> Double
operation = { (operand) in -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

42

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

And finally, it'll let you replace the parameter names with $0, $1, $2, etc., and skip in …

```
var operation: (Double) -> Double
operation = { (operand) in -operand }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

43

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
operation = { (operand: Double) -> Double in return -operand }
```

And finally, it'll let you replace the parameter names with $0, $1, $2, etc., and skip in …

```
var operation: (Double) -> Double
operation = { -$0 }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

44

Function Types

Closures

Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.

Imagine we had a function that changed the sign of its argument …

```
func changeSign(operand: Double) -> Double
{
    return -operand
}
```

```
operation = { (operand: Double) -> Double in return -operand }
```

That is about as succinct as possible!

```
var operation: (Double) -> Double
operation = { -$0 }
let result = operation(4.0) // result will be -4.0
```

CS193p
Fall 2017-18

45

Closures

Where do we use closures?

Often as arguments to methods.

```
func calculate(num1:Int, num2:Int, operation: (Int, Int)->Int)
{
    print(operation(num1,num2))
}



let addClosure = {
    (number1:Int, number2:Int)->Int in
    return number1+number2
}


calculate(num1:3, num2:8, operation: addClosure)
```

CS193p
Fall 2017-18

46

Closures

Where do we use closures?

```
func giveMeMultiply() -> (Int,Int) -> String{
    return multiplyClosure
}
let doMultiply = giveMeMultiply()
doMultiply(3,5)
```

Often as arguments to methods.

```
func calculate(num1:Int, num2:Int, operation: (Int, Int)->Int)
{
    print(operation(num1,num2))
}

let addClosure = {
    (number1:Int, number2:Int)->Int in
    return number1+number2
}


calculate(num1:3, num2:8, operation: addClosure)
```

47

Closures

Where do we use closures?

Often as arguments to methods.

```
let primes = [2.0, 3.0, 5.0, 7.0, 11.0]
let negativePrimes = primes.map( { -$0 } ) // [-2.0, -3.0, -5.0, -7.0, -11.0]
```

CS193p
Fall 2017-18

48

Closures

Where do we use closures?

Array has a method called map which takes a function as an argument.
It applies that function to each element of the Array to create and return a new Array.

```
let primes = [2.0, 3.0, 5.0, 7.0, 11.0]
let negativePrimes = primes.map({ -$0 }) // [-2.0, -3.0, -5.0, -7.0, -11.0]
```

CS193p
Fall 2017-18

50

Closures

Where do we use closures?

Array has a method called map which takes a function as an argument.
It applies that function to each element of the Array to create and return a new Array.

```
let primes = [2.0, 3.0, 5.0, 7.0, 11.0]
let negativePrimes = primes.map({ -$0 }) // [-2.0, -3.0, -5.0, -7.0, -11.0]
let invertedPrimes = primes.map() { 1.0/$0 } // [0.5, 0.333, 0.2, etc.]
```

Note that
- if the last (or only) argument to a method is a closure, you can put it outside the method's parentheses that contain its arguments

CS193p
Fall 2017-18

51

Closures

Where do we use closures?

Array has a method called map which takes a function as an argument.
It applies that function to each element of the Array to create and return a new Array.

```
let primes = [2.0, 3.0, 5.0, 7.0, 11.0]
let negativePrimes = primes.map({ -$0 }) // [-2.0, -3.0, -5.0, -7.0, -11.0]
let invertedPrimes = primes.map() { 1.0/$0 } // [0.5, 0.333, 0.2, etc.]
let primeStrings = primes.map { String($0) } // ["2.0","3.0","5.0","7.0","11.0"]
```

Note that
- if the last (or only) argument to a method is a closure, you can put it outside the method's parentheses that contain its arguments
- and if the closure was the only argument, you can skip the () completely if you want.

CS193p
Fall 2017-18

52

Demo

Improve indexOfOneAndOnlyFaceUpCard implementation

We probably used more lines of code to make indexOfOneAndOnlyFaceUpCard computed
However, a better implementation using a method that takes a closure would fix that

CS193p
Fall 2017-18

55

```
    private var indexOfTheOneAndOnlyFaceUpCard: Int? {
        get {
            let faceUpCardIndices = cards.indices.filter { cards[$0].isFaceUp }
            return faceUpCardIndices.count == 1 ? faceUpCardIndices.first : nil
/               var foundIndex: Int?
/               for index in cards.indices {
/                   if cards[index].isFaceUp {
/                       if foundIndex == nil {
/                           foundIndex = index
/                       } else {
/                           return nil
/                       }
/                   }
/               }
/               return foundIndex
        }
        set {
            for index in cards.indices {
                cards[index].isFaceUp = (index == newValue)
            }
        }
    }
```

filter takes a closure as an argument returns an array

A closure
- takes an element of the sequence as its argument
- returns a Boolean value
  - indicating whether the element should be included in the returned array.

56

```
extension Collection {
    var oneAndOnly: Element? {
        return count == 1 ? first : nil
    }
}
```

```
private var indexOfTheOneAndOnlyFaceUpCard: Int? {
    get {

        return cards.indices.filter { cards[$0].isFaceUp }.oneAndOnly
          var foundIndex: Int?
          for index in cards.indices {
              if cards[index].isFaceUp {
                  if foundIndex == nil {
                      foundIndex = index
                  } else {
                      return nil
                  }
```

57

```
extension Collection {
    var oneAndOnly: Element? {
        return count == 1 ? first : nil
    }
}
```

let ch = “hello”.oneAndOnly // return nil

let ch = “h”.oneAndOnly // return ‘h’ , type of character, string is array of character

58

MORE CLOSURE

59

TYPE INFERENCE

```swift
var myLuckyNumber:Int = 7
let canWeDoThis:Bool = true
let numberArray:[Int] = [1,2,3,4,5,6,7,8]
```

```swift
let multiplyClosure = {
    (number1:Int, number2:Int) -> String in
    return "\(number1) * \(number2) = \(number1 * number2)"
}
```

```swift
let multiplyClosure:(Int,Int)->String = {
    (number1:Int, number2:Int) -> String in
    return "\(number1) * \(number2) = \(number1 * number2)"
}
```

60

TYPE INFERENCE

```swift
let eatClosure = {
    (foodName:String) in
    print("I want to have \(foodName)")
}
```

```swift
let helloClosure = {
    print("Hello Everybody")
}
```

61

TYPE INFERENCE

```swift
let eatClosure = {
    (foodName:String) in
    print("I want to have \(foodName)")
}
```

```swift
let helloClosure = {
    print("Hello Everybody")
}
```

```swift
let eatClosure:(String)->() = {
    (foodName:String) in
    print("I want to have \(foodName)")
}
```

```swift
let helloClosure:()->() = {
    print("Hello Everybody")
}
```

62

```
let multiplyClosure = {
    (number1:Int, number2: Int) -> String in
    return "\(number1)* \(number2) = \(number1*number2)"
}


func giveMeMultiply()-> (Int,Int)->String{
    return multiplyClosure
}


let doMultiply = giveMeMultiply()

How do we use doMultiply?
```

63

```
let multiplyClosure = {
    (number1:Int, number2: Int) -> String in
    return "\(number1)* \(number2) = \(number1*number2)"
}


func giveMeMultiply()-> (Int,Int)->String{
    return multiplyClosure
}


let doMultiply = giveMeMultiply()

doMultiply(3,5)
```

64

PRACTICE

```swift
func sayHello(){
    print("Hello Everybody")
}

sayHello()
```

```swift
func eat(foodName:String){
    print("I want to have \(foodName)")
}
eat(foodName: "Hamburger")
```

```swift
let helloClosure = {
    print("Hello Everybody")
}

helloClosure()
```

```swift
let eatClosure = {
    (foodName:String) in
    print("I want to have \(foodName)")
}
eatClosure("Apple Pie")
```

65

PRACTICE

- WRITE A CLOSURE ACCORDING TO THE FUNCTION

```swift
func sayHello(){
    print("Hello Everybody")
}

sayHello()
```

```swift
func eat(foodName:String){
    print("I want to have \(foodName)")
}
eat(foodName: "Hamburger")
```

66

CLOSURE

- *Closures* are self-contained blocks of functionality that can be passed around and used in your code
- Often you want to create the function "on the fly" (rather than already-existing like sqrt). You can do this "in line" using a closure.
- similar to blocks in C and Objective-C and to lambdas in other programming languages.

```
var reversedNames = names.sorted(by: { $0 > $1 })
```

67

CLOSURE

- Closures take one of three forms
  - Global functions
  - Nested functions
  - Closure expressions
- Global and nested functions, are actually special cases of closures.

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
```

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
```

```
{ s1, s2 in s1 > s2 }
```

68

Closure

The sorted(by:) method accepts a closure

- takes two arguments of the same type as the array's contents
- returns a Bool value
  - to say whether the first value should appear before or after the second value once the values are sorted.
- the sorting closure needs to return true if the first value should appear *before* the second value, and false otherwise.

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

```
e.g., sorting an array of String values
so the sorting closure needs to be a function of type
(String, String) -> Bool
```

CS193p
Fall 2017-18

70

Closure

Write a function:

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)

// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

```
e.g., sorting an array of String values so the sorting closure
needs to be a function of type
(String, String) -> Bool
```

71

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
```

Closure?

72

Closure Expression Syntax

```
{ (parameters) -> return type in
    statements
}
```

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
```

The example below shows a closure expression version of the backward(_:_:) function

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

CS193p
Fall 2017-18

73

Closure Expression Syntax

```
{ ( parameters ) -> return type in
    statements
}
```

{ (s1: String, s2: String) -> Bool in
return s1 > s2
}

// The sorted(by:) method is being called on an array of strings,
//so its argument must be a function of type (String, String) -> Bool.

```
{ (s1, s2) in return s1 > s2 }
```

// Inferring Type From Context

```
{ s1, s2 in return s1 > s2 }
```

CS193p
Fall 2017-18

74

Closure

you can still make the types explicit if you wish

- doing so is encouraged if it avoids ambiguity for readers of your code.

- In the case of the sorted(by:) method, the purpose of the closure is clear from the fact that sorting is taking place, and it is safe for a reader to assume that the closure is likely to be working with String values, because it is assisting with the sorting of an array of strings.

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

CS193p
Fall 2017-18

75

Implicit Returns from Single-Expression Closures

```
{ (parameters) -> return type in
    statements
}
```

```
{ (s1, s2) in return s1 > s2 }
```

↓

```
{ s1, s2 in return s1 > s2 }
```

↓

```
{ s1, s2 in s1 > s2 }
```

//Single-expression closures can implicitly return the result of their single expression by omitting the `return`

CS193p
Fall 2017-18

76

Shorthand Argument Names

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names $0, $1, $2, and so on.

```
{ s1, s2 in s1 > s2 }
```

↓

```
{ $0 > $1 }
```

// the number and type of the shorthand argument names will be inferred from the expected function type

```
// Not { s1 > s2 }
```

reversedNames = names.sorted(by: { $0 > $1 })

CS193p
Fall 2017-18

77

Operator Methods

An even shorter way to write the closure expression above:

String type defines the greater-than operator (>) as a method that has two parameters of type String, and returns a value of type Bool.
This exactly matches the method type needed by the sorted(by:)method.
Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: { $0 > $1 } )
reversedNames = names.sorted(by:  >  )
```

CS193p
Fall 2017-18

78

EXERCISE

```
let numbers = [10,8,20,7,56,3,2,1,99]
var finished = numbers.sorted(by:  ????   }) // write a complete function and a simplified closure

print("before sorting..")
Print(numbers)

print("\nafter sorting..")
Print(finished)
```

```
before sorting..
10 8 20 7 56 3 2 1 99

after sorting..
99 56 20 10 8 7 3 2 1
```

79

EXERCISE

```
let numbers = [10,8,20,7,56,3,2,1,99]
var finished = numbers.sorted(by: {$0 > $1} )
```

```
func backward( s1: Int,  s2: Int) -> Bool {
    return s1 > s2
}
var sortingNumbers = numbers.sorted(by: backward)
```

```
print("before sorting..")
for i in numbers{
    print("\(i)", terminator: " ")
}
```

```
before sorting..
10 8 20 7 56 3 2 1 99
```

```
print("\nafter sorting..")
for i in finished{
    print("\(i)", terminator: " ")
}
```

```
after sorting..
99 56 20 10 8 7 3 2 1
```

80

```
{ ( parameters ) -> return type in
    statements
}
```

Trailing Closures

- If you need to pass a closure expression to a function as the function's final argument, but the closure expression is long
- A trailing closure is written after the function call's parentheses, even though it is still an argument to the function

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}
```

// call this function without using a trailing closure:

```
someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})
```

// call this function with a trailing closure :

```
someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

you don't write the argument label for the closure as part of the function call.

81

Trailing Closures

- If you need to pass a closure expression to a function as the function's final argument, but the closure expression is long
- A trailing closure is written after the function call's parentheses, even though it is still an argument to the function

reversedNames = names.sorted(by: { $0 > $1 })

⬇

reversedNames = names.sorted( ) { $0 > $1 }

⬇

reversedNames = names.sorted { $0 > $1 }

82

Trailing Closures

- Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line.
- E.g., Array type has a map(_:) method which takes a closure expression as its single argument

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

// map to ["OneSix", "FiveEight", "FiveOneZero"]

83

Trailing Closures

- Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line.
- E.g.,Array type has a map(_:) method which takes a closure expression as its single argument

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

// map to ["OneSix", "FiveEight", "FiveOneZero"]

85

Trailing Closures

- Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line.
- E.g., Array type has a map(_:) method which takes a closure expression as its single argument

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

// map to ["OneSix", "FiveEight", "FiveOneZero"]

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
```

87

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
```

88

Function Types as Return Types

The return type of makeIncrementer is () -> Int

- It returns a function, rather than a simple value)
- The function it returns has no parameters, and returns an Int value each time it is called.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

CS193p
Fall 2017-18

89

Capturing Values

The incrementer() function refers to runningTotal and amount

- It does this by capturing a reference to runningTotal and amount from the surrounding function
  - ensures that runningTotal and amount do not disappear when the call to makeIncrementer ends
  - also ensures that runningTotal is available the next time the incrementer function is called.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

CS193p
Fall 2017-18

90

Capturing Values

This sets a constant called incrementByTen to refer to an incrementer function that adds 10 to its runningTotal variable each time it is called.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

Whenever you assign a function/closure to a constant/variable, you are actually setting that constant/variable to be a *reference* to the function/closure

91

Capturing Values

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

Closures Are Reference Types

IncrementByTen are constants, but the closures these constants refer to are still able to increment the runningTotal variables that they have captured. This is because functions and closures are *reference types*.

CS193p
Fall 2017-18

92

Capturing Values

If you create a second incrementer, it will have its own stored reference to a new, separate runningTotalvariable:

```
let incrementByTen = makeIncrementer(forIncrement: 10)      () -> Int
incrementByTen()                                             10
incrementByTen()                                             20
incrementByTen()                                             30

let incrementBySeven = makeIncrementer(forIncrement: 7)     () -> Int
incrementBySeven()                                           7
incrementBySeven()                                           14
incrementBySeven()                                           21
```

CS193p
Fall 2017-18

93

EXERCISE

Write a function makeDecremetor

```
let DecrementByTen = makeDecrementor(forDecrement: 10)
print(DecrementByTen())
print(DecrementByTen())
print(DecrementByTen())

let DecrementByEight = makeDecrementor(forDecrement: 8)
print(DecrementByEight())
print(DecrementByEight())
print(DecrementByEight())
```

```
90
80
70

92
84
76
```

94

EXERCISE

Write a function makeExtraDecremetor

1. Initial account: 100
2. Deduct 10 as monthly household expenses
3. Deduct extra expense if any

```
let balance = makeDecrementer(initial: 100, expenses: 10)
balance(9)
balance(8)
balance(7)
```

```
81
63
46
```

95