# Stanford CS193p

## Developing Applications for iOS
## Fall 2017-18

# Today

- Mostly more Swift but some other stuff too
  - Quick demo of `mutating`
  - `protocols`
  - `String`
  - `NSAttributedString`
  - Closures (and functions as types in general)

# Data Structures

- Four Essential Data Structure-building Concepts in Swift
  - `class`
  - `struct`
  - `enum`
  - `protocol`
- `struct`
  - Value type (`structs` don't live in the heap and are passed around by copying them)
  - Very efficient "copy on write" is automatic in Swift
  - This copy on write behavior requires you to mark `mutating` methods
  - No inheritance (of data)
  - Mutability controlled via `let` (e.g. you can't add elements to an `Array` assigned by `let`)
  - Supports functional programming design
  - Examples: `Card`, `Array`, `Dictionary`, `String`, `Character`, `Int`, `Double`, `UInt32`

Let's jump over to Concentration and see what happens if we make `Concentration` a `struct` …

# Data Structures

- Four Essential Data Structure-building Concepts in Swift
  - `class`
  - `struct`
  - `enum`
  - `protocol`
- `protocol`
  - A type which is a declaration of functionality only
  - No data storage of any kind (so it doesn't make sense to say it's a "value" or "reference" type)
  - Essentially provides multiple inheritance (of functionality only, not storage) in Swift
  - We'll "ease into" learning about protocols since it's new to most of you
  - Let's dive a little deeper into `protocols` ...

# Protocols

- Protocols are a way to express an API more concisely
  - Instead of forcing the caller of an API to pass a specific `class`, `struct`, or `enum`,
    an API can let callers pass any `class`/`struct`/`enum` that the caller wants
    but can require that they implement certain methods and/or properties that the API wants.
  - The API expresses the functions or variables it wants the caller to provide using a `protocol`.
  - So a `protocol` is simply a collection of method and property declarations.
- What are `protocol`s good for?
  - Making API more flexible and expressive
  - Blind, structured communication between View and Controller (delegation)
  - Mandating behavior (e.g. the keys of a `Dictionary` must be hashable)
  - Sharing functionality in disparate types (`String`, `Array`, `CountableRange` are all `Collection`s)
  - Multiple inheritance (of functionality, not data)

# Protocols

- A `protocol` is a TYPE

  It can be used almost anywhere any other type is used: vars, `func`tion parameters, etc.

# Protocols

- There are three aspects to a protocol
  1. the `protocol` declaration (which properties and methods are in the `protocol`)
  2. a `class`, `struct` or `enum` declaration that makes the claim to implement the `protocol`
  3. the code in said `class`, `struct` or `enum` (or `extension`) that implements the `protocol`

# Protocols

- Optional methods in a protocol
  - Normally any `protocol` implementor must implement all the methods/properties in the protocol.
  - However, it is possible to mark some methods in a protocol `optional`
    (don't get confused with the type Optional, this is a different thing).
  - Any `protocol` that has `optional` methods must be marked `@objc`.
  - And any `class` that implements an `optional protocol` must inherit from `NSObject`.
  - These sorts of `protocols` are used often in iOS for delegation (more later on this).
  - Except for delegation, a `protocol` with `optional` methods is rarely (if ever) used.
  - As you can tell from the `@objc` designation, it's mostly for backwards compatibility.

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and 2

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and 2

You must specify whether a property is `get` only or both `get` and `set`

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and 2

You must specify whether a property is `get` only or both `get` and `set`

Any functions that are expected to mutate the receiver should be marked `mutating`

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : class, InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and `2`

You must specify whether a property is `get` only or both `get` and `set`

Any functions that are expected to mutate the receiver should be marked `mutating`
(unless you are going to restrict your protocol to class implementers only with `class` keyword)

# Protocols

- Declaration of the `protocol` itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and `2`

You must specify whether a property is `get` only or both `get` and `set`

Any functions that are expected to mutate the receiver should be marked `mutating`
(unless you are going to restrict your protocol to class implementers only with `class` keyword)

You can even specify that implementers must implement a given `init`ializer

# Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a `class`

# Protocols

- How an implementer says "I implement that protocol"

```
enum SomeEnum : SomeProtocol, AnotherProtocol {
    // implementation of SomeEnum here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a `class`
(obviously, `enum`s and `struct`s would not have the superclass part)

# Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a `class`
(obviously, `enum`s and `struct`s would not have the superclass part)

# Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a `class`
(obviously, `enum`s and `struct`s would not have the superclass part)
Any number of protocols can be implemented by a given `class`, `struct` or `enum`

# Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here, including ...
    required init(…)
}
```

Claims of conformance to protocols are listed after the superclass for a `class`
(obviously, `enum`s and `struct`s would not have the superclass part)
Any number of protocols can be implemented by a given `class`, `struct` or `enum`
In a `class`, `init`s must be marked `required` (or otherwise a subclass might not conform)

# Protocols

- How an implementer says "I implement that protocol"

```
extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties though
}
```

Claims of conformance to protocols are listed after the superclass for a `class`
(obviously, `enum`s and `struct`s would not have the superclass part)
Any number of protocols can be implemented by a given `class`, `struct` or `enum`
In a `class`, `init`s must be marked `required` (or otherwise a subclass might not conform)
You are allowed to add `protocol` conformance via an `extension`

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
thingToMove.changeOil()
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
thingToMove.changeOil()
thingToMove = square
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(slider: Moveable) {
    let positionToSlideTo = …
    slider.move(to: positionToSlideTo)
}
slide(prius)
slide(square)
```

# Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

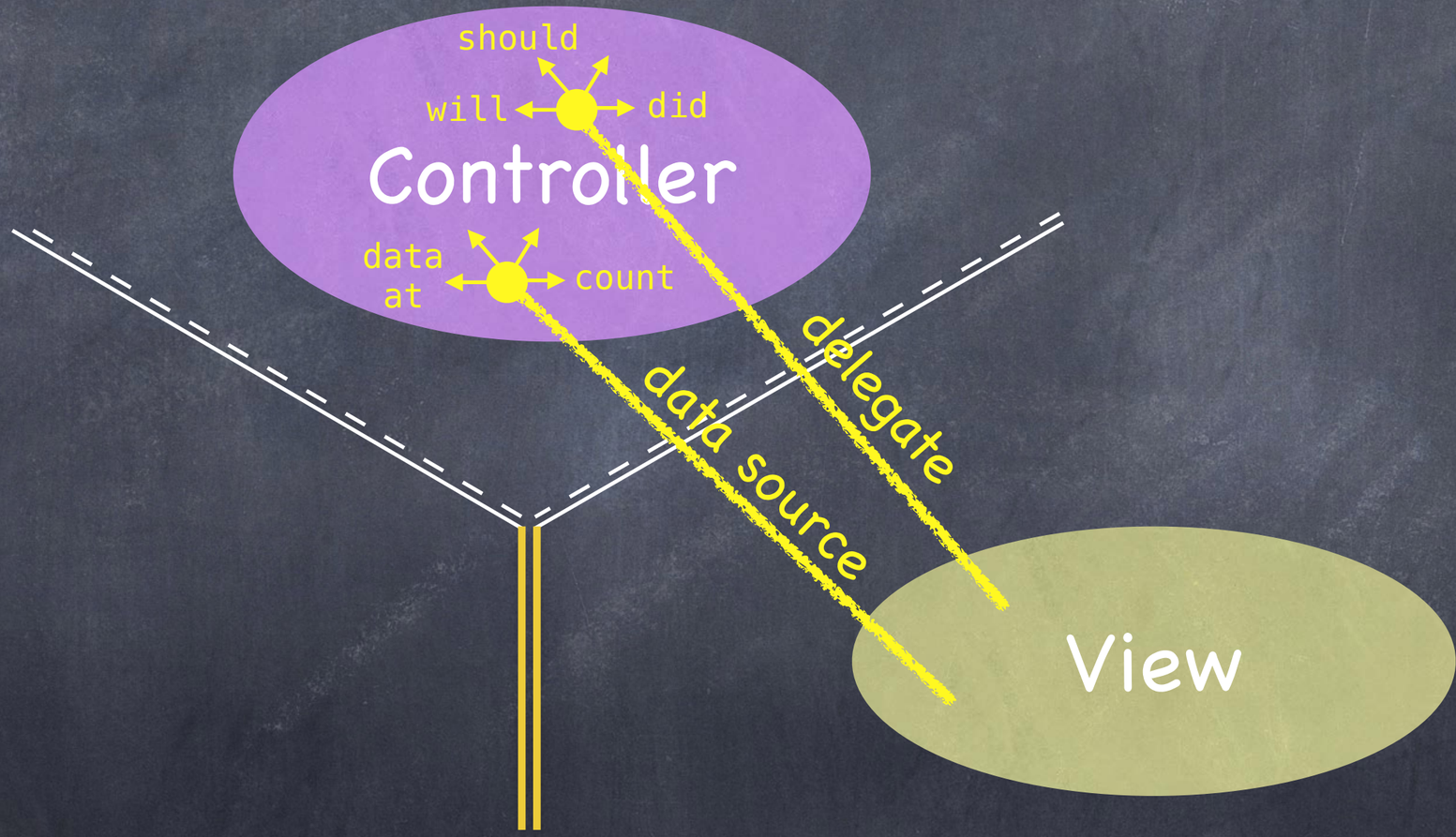
```
var thingToMove: Moveable = prius
thingToMove.move(to: …)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(slider: Moveable) {
    let positionToSlideTo = …
    slider.move(to: positionToSlideTo)
}
slide(prius)
slide(square)
func slipAndSlide(x: Slippery & Moveable)
slipAndSlide(prius)
```

# Delegation

- A very important (simple) use of `protocols`

  It's a way to implement "blind communication" between a View and its Controller



# Delegation

- A very important (simple) use of `protocol`s
  - It's a way to implement "blind communication" between a View and its Controller
- How it plays out ...
  1. A View declares a delegation `protocol` (i.e. what the View wants the Controller to do for it)
  2. The View's API has a `weak` `delegate` property whose type is that delegation `protocol`
  3. The View uses the `delegate` property to get/do things it can't own or control on its own
  4. The Controller declares that it implements the `protocol`
  5. The Controller sets `delegate` of the View to `itself` using the property in #2 above
  6. The Controller implements the `protocol` (probably it has lots of `optional` methods in it)
- Now the View is hooked up to the Controller
  - But the View still has no idea what the Controller is, so the View remains generic/reusable
- This mechanism is found throughout iOS
  - However, it was designed pre-closures in Swift. Closures are sometimes a better option.
  - We'll learn about closures soon.

# Delegation

- Example

  UIScrollView (which we'll talk about next week) has a delegate property …

```
weak var delegate: UIScrollViewDelegate?
```

  The UIScrollViewDelegate protocol looks like this …

```
@objc protocol UIScrollViewDelegate {
    optional func scrollViewDidScroll(scrollView: UIScrollView)
    optional func viewForZooming(in scrollView: UIScrollView) -> UIView
   … and many more …
}
```

  A Controller with a UIScrollView in its View would be declared like this …

```
class MyViewController : UIViewController, UIScrollViewDelegate { … }
```

  … probably in the @IBOutlet didSet for the scroll view, the Controller would do …

```
scrollView.delegate = self
```

  … and the Controller then would implement any of the protocol's methods it is interested in.

# Another use of Protocols

- Being a key in a `Dictionary`

  To be a key in a `Dictionary`, you have to be able to be unique.
  A key in a `Dictionary` does this by providing an `Int` that is very probably unique (a hash)
  and then also by implementing equality testing to see if two keys are, in fact, the same.

  This is enforced by requiring that a `Dictionary`'s keys implement the `Hashable` protocol.
  Here's what that `protocol` looks like ...

```
protocol Hashable: Equatable {
    var hashValue: Int { get }
}
```

  Very simple. Note, though, that `Hashable` inherits from `Equatable` ...

# Another use of Protocols

- Being a key in a `Dictionary`

  That means that to be `Hashable`, you also have to implement `Equatable`.

  The `Equatable` protocol looks like this …

```
protocol Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool
}
```

  Types that conform to `Equatable` have to have a type function (note the `static`) called ==

  The arguments to == are both of that same type (i.e. `Self` of the type is the type itself)

  The == operator also happens to look for such a `static` method to provide its implementation!

# Another use of Protocols

- Being a key in a `Dictionary`

  `Dictionary` is then declared like this: `Dictionary<Key: Hashable, Value>`
  This restricts keys to be things that conform to `Hashable` (there's no restriction on values)

  Let's go make `Card` be `Hashable`.
  Then we can use it directly as the key into our `emoji` `Dictionary`.
  As a bonus, we'll be able to compare `Cards` directly since they'll be `Equatable`.
  This will even allow us to make `identifier` be `private` in `Card`, which makes a lot of sense.

# Demo

- Make `Card` `struct` `Hashable` and `Equatable`
  - Doing this allows us to make `Card`'s `identifier` `private`
  - It also lets us look up `Card`s directly in a `Dictionary` (rather than with its identifier)
  - And we can use `==` to compare `Card`s directly
  - And for your homework, supporting `Equatable` lets you use `index(of:)` on an `Array` of things

# 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(offsetBy:), 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 Collection,
  they actually inherit an implementation of many of the methods in Collection, because …

# protocol & extension

- Using `extension` to provide `protocol` implementation

  We said that `protocol` implementation is provided by implementing types (`struct`, `enum`, `class`)
  However, an `extension` can be used to add default implementation to a `protocol`.
  Since there's no storage, said implementation has to be in terms of other API in the `protocol`
  (and any API in any `protocol` that that `protocol` inherits from, of course).

  For example, for the `Sequence` `protocol`, you really only need to implement `makeIterator`.
  (An iterator implements the `IteratorProtocol` which just has the method `next()`.)
  If you do, you will automatically get implementations for all these other methods in `Sequence`:
  `contains()`, `forEach()`, `joined(separator:)`, `min()`, `max()`, even `filter()` and `map()`, et. al.
  All of these are implemented via an extension to the `Sequence` `protocol`.
  This `extension` (provided by Apple) uses only `Sequence` `protocol` methods in its implementation.

```
extension Sequence {
    func contains(_ element: Element) -> Bool { }
    // etc.
}
```

# Advanced use of Protocols

- Functional Programming

  By combining `protocols` with generics and `extensions` (default implementations),
  you can build code that focusses more on the behavior of data structures than storage.

  This approach to development is called "functional programming."

  It is different than "object-oriented programming" (it's sort of an evolution thereof).

  We don't have time to teach functional programming, but you are getting a taste of it.

  What's great about Swift is that it supports both paradigms.

# 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.

  For example, 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).
  So the string café might be 4 Unicodes (c-a-f-é) or 5 Unicodes (c-a-f-e-').
  In either case, we preceive it as 4 `Character`s.
  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 `String`s 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)`.

# 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]
}
```

Note the `..<` above.

This is a `Range` of `String.Index`.

`Range` is a generic type (like `Array` is). It doesn't have to be a range of `Int`s.

Another way to find the second word: `pizzaJoint.components(separatedBy: “ “)[1]`

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

# String

- The characters in a String

  String is also a Collection (in the same sense that an Array is 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.)

- A String is a value type (it's a struct)

  We usually work with immutable Strings (i.e. let s = …).

  But there are mutating methods on String as well, for example …

```
var s = pizzaJoint // makes a mutable copy of pizzaJoint (because it's a value type!)
s.insert(contentsOf: " foo", at: s.index(of: " ")!) // café foo pesto
```

  The type of contentsOf: argument is any Collection of Character (which String is).

# String

- Other `String` Methods

```
func hasPrefix(String) -> Bool
func hasSuffix(String) -> Bool
var localizedCapitalized/Lowercase/Uppercase: String

func replaceSubrange(Range<String.Index>, with: Collection of Character)
```

e.g., `s.replaceSubrange(..<s.endIndex, with: “new contents”)`

Note the `..<` Range appears to have no start! It defaults to the start of the `String`.

And much, much more. Check out the documentation.

# 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

# 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 (like in your homework), the entire NSAttributedString uses the same Dictionary.
  - You can put NSAttributedStrings on UILabels, UIButtons, etc.
  - Next week we'll also learn how to draw an NSAttributedString on the screen directly.

# 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
```

# 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 (like in your homework).
  - Or if the NSAttributedString doesn't contain "wacky" Unicode characters.
  - Otherwise, be careful indexing into the NSAttributedString.

# Demo

- Make flip count outlined text

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

# Function Types

- Function types
  - Functions are people* too! (* er, types)
  - 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
  - You can do this anywhere any other type is allowed

  Example ...

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

  This is a `var` called `operation`

  It is of type "function that takes a `Double` and returns a `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
```

# 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
```

# 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
```

# 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 …

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

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

# 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 }
let result = operation(4.0) // result will be -4.0
```

# 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 }
let result = operation(4.0) // result will be -4.0
```

# 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`

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

# 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`

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

# 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
```

# 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) in return -operand }
let result = operation(4.0) // result will be -4.0
```

# 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 ...

  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
```

# 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 ...

  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
```

# 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 …

  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
```

# 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 …

  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
```

# 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 ...

  That is about as succinct as possible!

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

# Closures

- Where do we use closures?
  - Often as arguments to methods.
  - Many times a method wants to know "what to do" and providing a function tells it what to do.
  - For example, what to do when there's an error or when something asynchronous finishes.
  - Or maybe you want to ask some method to repeatedly perform a function ...

# 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.

# Closures

- Closures with property initialization

  You can also execute a closure to do initialization of a property if you want …

```
var someProperty: Type = {
    // construct the value of someProperty here
    return <the constructed value>
}()
```

  This is especially useful with `lazy` property initialization.

# Closures

- Capturing

  Closures are regular types, so they can be put in `Arrays`, `Dictionarys`, etc.
  When this happens, they are stored in the heap (i.e. they are reference types).

  What is more, they "capture" variables they use from the surrounding code into the heap too.
  Those captured variables need to stay in the heap as long as the closure stays in the heap.

```
var ltuae = 42
operation = { ltuae * $0 } // "captures" the ltuae var because it's needed for this closure
arrayOfOperations.append(operation)
// if we later change ltuae, then the next time we evaluate operation it will reflect that
// even if we leave the scope (function or whatever) that this code is in!
```

  This can create a memory cycle though.
  We'll see this later in the quarter and how to avoid that.

# 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