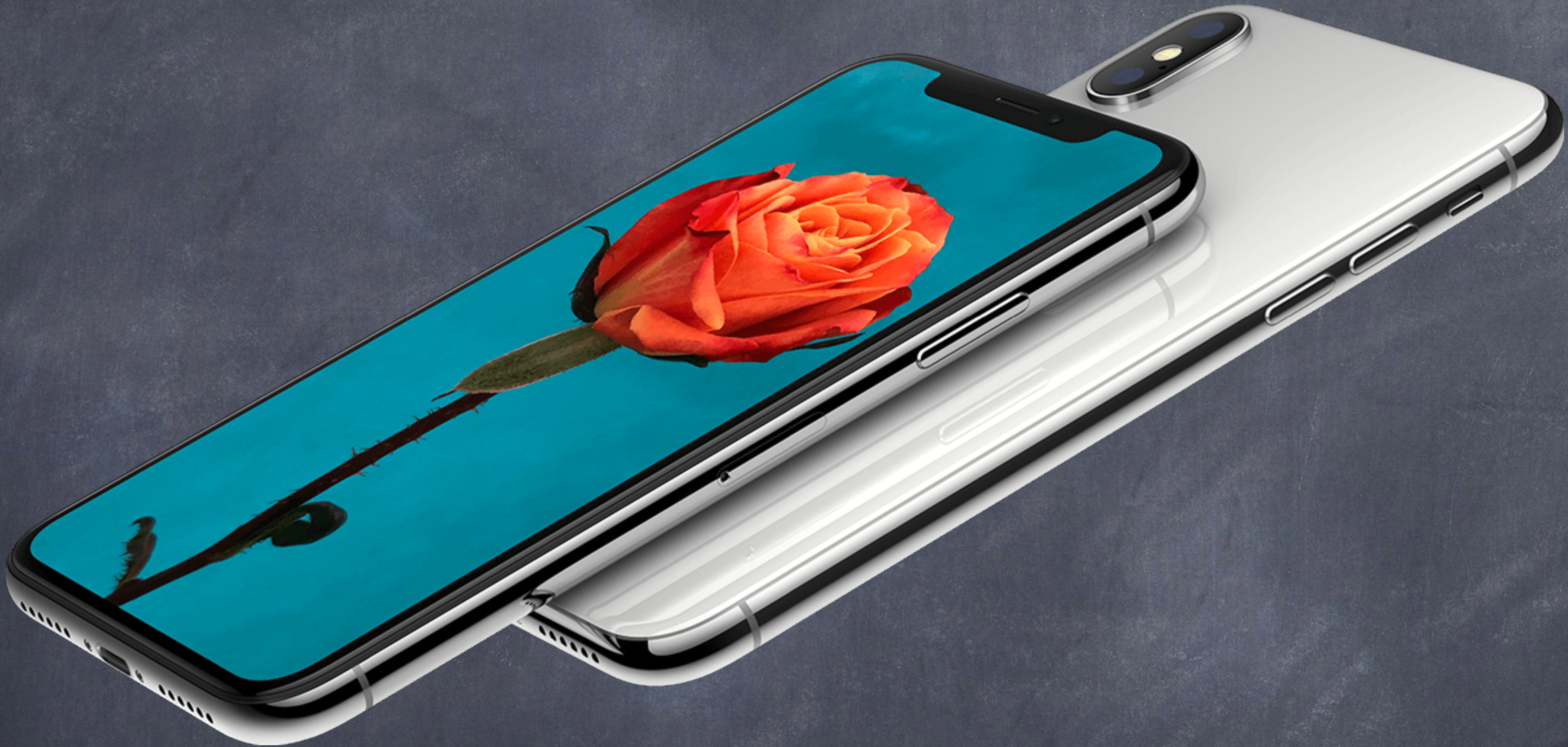

Stanford CS193p

Developing Applications for iOS

Fall 2017-18

Today

- Mostly Swift but some other stuff too
  - Autolayout teaser
  - Quick review of what we learned in Concentration
  - `CountableRange` of floating point numbers
  - Tuples
  - Computed Properties
  - Access Control
  - `assertions`
  - `extensions`
  - `enum`
  - Optionals are `enums`
  - Data structure review (including memory management)
  - `protocols` (time permitting)

Demo

- Making Concentration's button layout dynamic

  We want our UI to work on different iPhones and in both landscape and portrait

  We'll do this using `UIStackView` and autolayout

  This is only a taste of what's to come on this front in a couple of weeks

Review

- Brief Review of Week 1
  - Target/Action and Outlets and Outlet Collections
  - Methods and Properties (aka instance variables)
  - Property Observer (`didSet`)
  - `Array<Element>`
  - MVC
  - Value types (`struct`) versus reference types (`class`)
  - `init`ializers
  - Type (`static`) methods
  - `lazy` properties
  - `for in` loops
  - `Dictionary<Key,Value>`
  - Type conversion (e.g. `UInt32(anInt)`)
  - And, of course, Optionals, which we're going to revisit in more detail a bit later in this lecture

Range

- Floating point `CountableRange`

  How do you do `for (i = 0.5; i <= 15.25; i += 0.3)`?
  Floating point numbers don't stride by `Int`, they stride by a floating point value.
  So `0.5...15.25` is just a `Range`, not a `CountableRange` (which is needed for `for in`).

  Luckily, there's a global function that will create a `CountableRange` from floating point values!

```
for i in stride(from: 0.5, through: 15.25, by: 0.3) {

}
```

  The return type of `stride` is `CountableRange`
  (actually `ClosedCountableRange` in this case because it's `through:` instead of `to:`).
  As we'll see with `String` later, `CountableRange` is a generic type (doesn't have to be `Int`s).

Tuples

- What is a tuple?

It is nothing more than a grouping of values.
You can use it anywhere you can use a type.

```
let x: (String, Int, Double) = (“hello”, 5, 0.85) // the type of x is “a tuple”
let (word, number, value) = x // this names the tuple elements when accessing the tuple
print(word)    // prints hello
print(number)  // prints 5
print(value)   // prints 0.85
```

… or the tuple elements can be named when the tuple is declared (this is strongly preferred) …

```
let x: (w: String, i: Int, v: Double) = (“hello”, 5, 0.85)
print(x.w) // prints hello
print(x.i) // prints 5
print(x.v) // prints 0.85
let (wrd, num, val) = x // this is also legal (renames the tuple’s elements on access)
```

Tuples

- Tuples as return values

You can use tuples to return multiple values from a function or method …

```
func getSize() -> (weight: Double, height: Double) { return (250, 80) }

let x = getSize()
print("weight is \(x.weight)") // weight is 250
… or …
print("height is \(getSize().height)") // height is 80
```

Computed Properties

- The value of a property can be computed rather than stored

  A typical stored property looks something like this ...

  ```
  var foo: Double
  ```

  A computed property looks like this ...

  ```
  var foo: Double {
      get {
          // return the calculated value of foo
      }
      set(newValue) {
          // do something based on the fact that foo has changed to newValue
      }
  }
  ```

  You don't have to implement the set side of it if you don't want to.
  The property then becomes "read only".

Computed Properties

- Why compute the value of a property?

  Lots of times a "property" is "derived" from other state.

  For example, in Concentration, we can derive this var easily from looking at the cards …

```
var indexOfOneAndOnlyFaceUpCard: Int?
```

  In fact, properly keeping this var up-to-date is just plain error-prone. This would be safer …

```
var indexOfOneAndOnlyFaceUpCard: Int? {
    get {
        // look at all the cards and see if you find only one that's face up
        // if so, return it, else return nil
    }
    set {
        // turn all the cards face down except the card at index newValue
    }
}
```

  Let's go to Concentration and make this change …

Demo

- Make `indexOfOneAndOnlyFaceUpCard` be computed
  - That way it will always be in sync
  - And we'll see that the rest of our code gets even simpler because of it

Access Control

- Protecting our internal implementations

  Likely most of you have only worked on relatively small projects
  Inside those projects, any object can pretty much call any function in any other object
  When projects start to get large, though, this becomes very dicey
  You want to be able to protect the INTERNAL implementation of data structures
  You do this by marking which API* you want other code to use with certain keywords

  * i.e. methods and properties

Access Control

- ## Protecting our internal implementations

Swift supports this with the following access control keywords …

`internal` - this is the default, it means "usable by any object in my app or framework"

`private` - this means "only callable from within this object"

`private(set)` - this means "this property is readable outside this object, but not settable"

`fileprivate` - accessible by any code in this source file

`public` - (for frameworks only) this can be used by objects outside my framework

`open` - (for frameworks only) public and objects outside my framework can subclass this

We are not going to learn to develop frameworks this quarter, so we are only concerned with …

`private`, `private(set)`, `fileprivate` and `internal` (which is the default, so no keyword)

A good strategy is to just mark everything `private` by default.

Then remove the `private` designation when that API is ready to be used by other code.

Concentration needs some access control …

Demo

- Add access control to Concentration

  Even though our app is small, we want to learn to develop like we're on a big team

  Also, let's protect our API with assertions too

Extensions

- Extending existing data structures

  You can add methods/properties to a `class/struct/enum` (even if you don't have the source).

- There are some restrictions

  You can't re-implement methods or properties that are already there (only add new ones).
  The properties you add can have no storage associated with them (computed only).

- This feature is easily abused

  It should be used to add clarity to readability not obfuscation!
  Don't use it as a substitute for good object-oriented design technique.
  Best used (at least for beginners) for very small, well-contained helper functions.
  Can actually be used well to organize code but requires architectural commitment.
  When in doubt (for now), don't do it.

  Let's add a simple `extension` in Concentration …

Demo

- Make `arc4random` code a lot cleaner

  In your homework you are using it at least twice

  And it's easy to imagine using it even more often in Concentration and beyond

Optionals

- Optional

  A completely normal type in Swift

  It's an enumeration

  Let's take a moment and learn about enumerations and then we'll look at Optionals a little closer

enum

- Another variety of data structure in addition to `struct` and `class`

  It can only have discrete states …

```
enum FastFoodMenuItem {
    case hamburger
    case fries
    case drink
    case cookie
}
```

  An `enum` is a VALUE TYPE (like `struct`), so it is copied as it is passed around

enum

- Associated Data

  Each state can (but does not have to) have its own "associated data" …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. "Coke"
    case cookie
}
```

  Note that the drink case has 2 pieces of associated data (one of them "unnamed")
  In the example above, FryOrderSize would also probably be an enum, for example …

```
enum FryOrderSize {
    case large
    case small
}
```

enum

- Setting the value of an enum

  Assigning a value to a variable or constant of type enum is easy …

```
let menuItem: FastFoodMenuItem =
var otherItem: FastFoodMenuItem =
```

enum

- Setting the value of an enum

  Just use the name of the type along with the case you want, separated by dot …

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```

enum

- Setting the value of an enum

  When you set the value of an enum you must provide the associated data (if any) …

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```

enum

- Setting the value of an enum

  Swift can infer the type on one side of the assignment or the other (but not both) …

```swift
let menuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = .cookie
var yetAnotherItem = .cookie // Swift can't figure this out
```

enum

- Checking an enum's state

An enum's state is checked with a `switch` statement ...

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case FastFoodMenuItem.hamburger: print("burger")
    case FastFoodMenuItem.fries: print("fries")
    case FastFoodMenuItem.drink: print("drink")
    case FastFoodMenuItem.cookie: print("cookie")
}
```

Note that we are ignoring the "associated data" above ... so far ...

enum

- **Checking an enum's state**

  An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case FastFoodMenuItem.hamburger: print("burger")
    case FastFoodMenuItem.fries: print("fries")
    case FastFoodMenuItem.drink: print("drink")
    case FastFoodMenuItem.cookie: print("cookie")
}
```

  This code would print "burger" on the console

enum

- Checking an enum's state

  An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case .hamburger: print("burger")
    case .fries: print("fries")
    case .drink: print("drink")
    case .cookie: print("cookie")
}
```

  It is not necessary to use the fully-expressed FastFoodMenuItem.fries inside the switch
  (since Swift can infer the FastFoodMenuItem part of that)

enum

- `break`

  If you don't want to do anything in a given case, use break …

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    case .drink: print("drink")
    case .cookie: print("cookie")
}
```

  This code would print nothing on the console

enum

- `default`

  You must handle ALL POSSIBLE CASES (although you can `default` uninteresting cases) …

```
var menuItem = FastFoodMenuItem.cookie
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    default: print("other")
}
```

enum

- default

You must handle ALL POSSIBLE CASES (although you can default uninteresting cases) …

```
var menuItem = FastFoodMenuItem.cookie
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    default: print("other")
}
```

If the menuItem were a cookie, the above code would print "other" on the console

enum

- Multiple lines allowed

Each case in a switch can be multiple lines and does NOT fall through to the next case …

```
var menuItem = FastFoodMenuItem.fries(size: FryOrderSize.large)
switch menuItem {
    case .hamburger: print(“burger”)
    case .fries:
        print(“yummy”)
        print(“fries”)
    case .drink:
        print(“drink”)
    case .cookie: print(“cookie”)
}
```

The above code would print “yummy” and “fries” on the console, but <u>not</u> “drink”

enum

- Multiple lines allowed

  By the way, we can let Swift infer the enum type of the size of the fries too …

```
var menuItem = FastFoodMenuItem.fries(size: .large)
switch menuItem {
    case .hamburger: print(“burger”)
    case .fries:
        print(“yummy”)
        print(“fries”)
    case .drink:
        print(“drink”)
    case .cookie: print(“cookie”)
}
```

enum

- What about the associated data?

  Associated data is accessed through a `switch` statement using this `let` syntax …

```
var menuItem = FastFoodMenuItem.drink(“Coke”, ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount): print(“a burger with \(pattyCount) patties!”)
    case .fries(let size): print(“a \(size) order of fries!”)
    case .drink(let brand, let ounces): print(“a \(ounces)oz \(brand)”)
    case .cookie: print(“a cookie!”)
}
```

enum

- What about the associated data?

Associated data is accessed through a `switch` statement using this `let` syntax …

```
var menuItem = FastFoodMenuItem.drink(“Coke”, ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount): print(“a burger with \(pattyCount) patties!”)
    case .fries(let size): print(“a \(size) order of fries!”)
    case .drink(let brand, let ounces): print(“a \(ounces)oz \(brand)”)
    case .cookie: print(“a cookie!”)
}
```

The above code would print “a 32oz Coke” on the console

enum

- What about the associated data?

  Associated data is accessed through a `switch` statement using this `let` syntax …

```
var menuItem = FastFoodMenuItem.drink("Coke", ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount): print("a burger with \(pattyCount) patties!")
    case .fries(let size): print("a \(size) order of fries!")
    case .drink(let brand, let ounces): print("a \(ounces)oz \(brand)")
    case .cookie: print("a cookie!")
}
```

  Note that the local variable that retrieves the associated data can have a different name

  (e.g. `pattyCount` above versus `patties` in the `enum` declaration)

  (e.g. `brand` above versus not even having a name in the `enum` declaration)

enum

- Methods yes, (stored) Properties no

An enum can have methods (and computed properties) but no stored properties …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int)
    case cookie

    func isIncludedInSpecialOrder(number: Int) -> Bool { }
    var calories: Int { // calculate and return caloric value here }
}
```

An enum's state is entirely which case it is in and that case's associated data.

enum

- Methods yes, (stored) Properties no

  In an enum's own methods, you can test the enum's state (and get associated data) using `self` ...

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a drink and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

enum

- Methods yes, (stored) Properties no

In an enum's own methods, you can test the enum's state (and get associated data) using `self` …

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a drink and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

Special order 1 is a single patty burger, 2 is a double patty (3 is a triple, etc.?!)

enum

- Methods yes, (stored) Properties no

In an enum's own methods, you can test the enum's state (and get associated data) using self …

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a drink and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

Notice the use of _ if we don't care about that piece of associated data.

enum

- Modifying `self` in an `enum`

  You can even reassign `self` inside an `enum` method …

```
enum FastFoodMenuItem {
    . . .
    mutating func switchToBeingACookie() {
        self = .cookie // this works even if self is a .hamburger, .fries or .drink
    }
}
```

enum

- Modifying self in an enum

  You can even reassign self inside an enum method …

```
enum FastFoodMenuItem {
    . . .
    mutating func switchToBeingACookie() {
        self = .cookie // this works even if self is a .hamburger, .fries or .drink
    }
}
```

  Note that mutating is required because enum is a VALUE TYPE.

Optionals

Optional

So an Optional is just an enum

It essentially looks like this …

```
enum Optional<T> { // a generic type, like Array<Element> or Dictionary<Key,Value>
    case none
    case some(<T>) // the some case has associated data of type T
}
```

But this type is so important that it has a lot of special syntax that other types don’t have …

Optionals

- Special Optional syntax in Swift

  The "not set" case has a special keyword: `nil`

  The character `?` is used to declare an Optional, e.g. `var indexOfOneAndOnlyFaceUpCard: Int?`

  The character `!` is used to "unwrap" the associated data if an Optional is in the "set" state …

  e.g. `let index = cardButtons.index(of: button)!`

  The keyword `if` can also be used to conditionally get the associated data …

  e.g. `if let index = cardButtons.index(of: button) { … }`

  An Optional declared with `!` (instead of `?`) will implicitly unwrap (add `!`) when accessed …

  e.g. `var flipCountIndex: UILabel!` enables `flipCountIndex.text = "…"` (i.e. no `!` here)

  You can use `??` to create an expression which "defaults" to a value if an Optional is not set …

  e.g. `return emoji[card.identifier] ?? "?"`

  You can also use `?` when accessing an Optional to bail out of an expression midstream …

  this is called Optional Chaining

  we'll take a closer look at it in a few slides

Optionals

Optional

Declaring and assigning values to an Optional …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String?                  var hello: Optional<String> = .none
var hello: String? = "hello"        var hello: Optional<String> = .some("hello")
var hello: String? = nil            var hello: Optional<String> = .none
```

Optionals

Optional

Note that Optionals always start out `nil` …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String?                  var hello: Optional<String> = .none
var hello: String? = “hello”        var hello: Optional<String> = .some(“hello”)
var hello: String? = nil            var hello: Optional<String> = .none
```

Optionals

Optional

Unwrapping …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let hello: String? = …
print(hello!)
```

```
switch hello {
    case .none: // raise an exception (crash)
    case .some(let data): print(data)
}
```

```
if let greeting = hello {
    print(greeting)
} else {
    // do something else
}
```

```
switch hello {
    case .some(let data): print(data)
    case .none: { // do something else }
}
```

Optionals

Optional

Implicitly unwrapped Optional …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String!
hello = …
print(hello)
```

```
var hello: Optional<String> = .none

switch hello {
    case .none: // raise exception (crash)
    case .some(let data): print(data)
}
```

Optionals

Optional

Implicitly unwrapped Optional (these start out nil too) …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String!
hello = …
print(hello)
```

```
var hello: Optional<String> = .none

switch hello {
    case .none: // raise exception (crash)
    case .some(let data): print(data)
}
```

Optionals

- Optional

Nil-coalescing operator (Optional defaulting) …

```
enum Optional<T> {
     case none
     case some(<T>)
}
```

```
let x: String? = …
let y = x ?? "foo"
```

```
switch x {
     case .none: y = "foo"
     case .some(let data): y = data
}
```

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Optionals

- Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Optionals

- Optional

  Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = …
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

Data Structures

- Four Essential Data Structure-building Concepts in Swift

  `class`
  `struct`
  `enum`
  `protocol`

- `class`

  Supports object-oriented design
  Single inheritance of both functionality and data (i.e. instance variables)
  Reference type (`class`es are stored in the heap and are passed around via pointers)
  Heap is automatically "kept clean" by Swift (via reference counting, not garbage collection)
  Examples: `ViewController`, `UIButton`, `Concentration`

Memory Management

- ## Automatic Reference Counting

  Reference types (`classes`) are stored in the heap.
  How does the system know when to reclaim the memory for these from the heap?
  It "counts references" to each of them and when there are zero references, they get tossed.
  This is done automatically.
  It is known as "Automatic Reference Counting" and it is NOT garbage collection.

- ## Influencing ARC

  You can influence ARC by how you declare a reference-type var with these keywords …
  `strong`
  `weak`
  `unowned`

Memory Management

- strong

  strong is "normal" reference counting

  As long as anyone, anywhere has a strong pointer to an instance, it will stay in the heap

- weak

  weak means "if no one else is interested in this, then neither am I, set me to nil in that case"

  Because it has to be nil-able, weak only applies to Optional pointers to reference types

  A weak pointer will NEVER keep an object in the heap

  Great example: outlets (strongly held by the view hierarchy, so outlets can be weak)

- unowned

  unowned means "don't reference count this; crash if I'm wrong"

  This is very rarely used

  Usually only to break memory cycles between objects (more on that in a little while)

Data Structures

- Four Essential Data Structure-building Concepts in Swift

  `class`
  `struct`
  `enum`
  `protocol`

- `struct`

  Value type (`struct`s 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` ...

Demo

- Make `Concentration` into a `struct`
  - We'll see that there's little difference in how it's declared

Data Structures

- Four Essential Data Structure-building Concepts in Swift

  `class`
  `struct`
  `enum`
  `protocol`

- `enum`

  Used for variables that have one of a discrete set of values
  Each option for that discrete value can have "associated data" with it
  The associated data is the only storage that an `enum` can have (no instance variables)
  Value type (i.e. passed around by copying)
  Can have methods and computed (only) properties
  Example: we'll create a `PlayingCard` `struct` that uses `Rank` and `Suit` `enum`s

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` …