# Stanford CS193p

## Developing Applications for iOS
## Spring 2016

# Today

- More Swift & the Foundation Framework
  - What are Optionals really?
  - Access Control
  - Tuples
  - Range<T>
  - Data Structures in Swift
  - Methods
  - Properties
  - Array<T>, Dictionary<K,V>, String, et. al.
  - Initialization
  - AnyObject, introspection and casting (is and as)
  - Property List
  - NSUserDefaults
  - AnyObject in CalculatorBrain
  - assert

# Optional

- An Optional is just an enum

In other words …

```
enum Optional<T> {  // the <T> is a generic like as in Array<T>
    case None
    case Some(T)
}
```

```
let x: String? = nil
```

… is …

```
let x = Optional<String>.None
```

```
let x: String? = “hello”
```

… is …

```
let x = Optional<String>.Some(“hello”)
```

```
var y = x!
```

… is …

```
switch x {
    case Some(let value): y = value
    case None: // raise an exception
}
```

# Optional

- An Optional is just an `enum`

```
let x: String? = ...
if let y = x {
    // do something with y
}
```

… is …

```
switch x {
    case .Some(let y):
        // do something with y
    case .None:
        break
}
```

# Optional

- Optionals can be "chained"

  For example, `hashValue` is a `var` in `String` which is an `Int`
  What if we wanted to get the `hashValue` from something which was an Optional `String`?
  And what if that Optional `String` was, itself, contained in an Optional `UILabel` `display`?

```
var display: UILabel? // imagine this is an @IBOutlet without the implicit unwrap !
if let label = display {
    if let text = label.text {
        let x = text.hashValue
        ...
    }
}
```

  … or …

```
if let x = display?.text?.hashValue { ... }
```

# Optional

- There is also an Optional "defaulting" operator ??

  What if we want to put a `String` into a `UILabel`, but if it's `nil`, put " " (space) in the `UILabel`?

```
let s: String? = ... // might be nil
if s != nil {
    display.text = s
} else {
    display.text = " "
}
```

  ... can be expressed much more simply this way ...

```
display.text = 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)
let (word, number, value) = x // tuple elements named 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 …

```
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

- Returning multiple values from a function

```
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
```

# Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax ...

```
class CalculatorBrain {

}
struct Vertex {

}
enum Op {

}
```

# Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax ...

  Properties and Functions ...

```
func doit(argument: Type) -> ReturnValue {

}

var storedProperty = <initial value> (not enum)

var computedProperty: Type {
    get {}
    set {}
}
```

# Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax …

  Properties and Functions …

  Initializers (again, not enum) …

```
init(argument1: Type, argument2: Type, …) {

}
```

# Data Structures in Swift

- Classes, Structures and Enumerations
  - These are the 3 fundamental building blocks of data structures in Swift
- Similarities
  - Declaration syntax ...
  - Properties and Functions ...
  - Initializers (again, not `enum`) ...
- Differences
  - Inheritance (`class` only)
  - Value type (`struct`, `enum`) vs. Reference type (`class`)

# Value vs. Reference

- Value (`struct` and `enum`)
  - Copied when passed as an argument to a function
  - Copied when assigned to a different variable
  - Immutable if assigned to a variable with `let`
  - Remember that function parameters are constants
  - You must note any `func` that can mutate a `struct/enum` with the keyword `mutating`
- Reference (`class`)
  - Stored in the heap and reference counted (automatically)
  - Constant pointers to a class (`let`) still can mutate by calling methods and changing properties
  - When passed as an argument, does not make a copy (just passing a pointer to same instance)
- Choosing which to use?
  - Usually you will choose `class` over `struct`. `struct` tends to be more for fundamental types.
  - Use of `enum` is situational (any time you have a type of data with discrete values).

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

```
func foo(externalFirst first: Int, externalSecond second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

```
func foo(externalFirst first: Int, externalSecond second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

  The external name is what callers use when they call the method

```
func foo(externalFirst first: Int, externalSecond second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(externalFirst: 123, externalSecond: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

  The external name is what callers use when they call the method

  You can put _ if you don't want callers to use an external name at all for a given parameter

```
func foo(_ first: Int, externalSecond second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(123, externalSecond: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

  The external name is what callers use when they call the method

  You can put _ if you don't want callers to use an external name at all for a given parameter

  This is the default for the first parameter (except in initializers!)

```
func foo(first: Int, externalSecond second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(123, externalSecond: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

  The external name is what callers use when they call the method

  You can put _ if you don't want callers to use an external name at all for a given parameter

  This is the default for the first parameter (except in initializers!)

  For other (not the first) parameters, the internal name is, by default, the external name

```
func foo(first: Int, second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(123, second: 5.5)
}
```

# Methods

- Parameters Names

  All parameters to all functions have an internal name and an external name

  The internal name is the name of the local variable you use inside the method

  The external name is what callers use when they call the method

  You can put _ if you don't want callers to use an external name at all for a given parameter

  This is the default for the first parameter (except in initializers!)

  For other (not the first) parameters, the internal name is, by default, the external name

  Any parameter's external name can be changed (even forcing the first parameter to have one)

```
func foo(forcedFirst first: Int, _ second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(forcedFirst: 123, 5.5)
}
```

# Methods

- Parameters Names
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers use when they call the method
  - You can put _ if you don't want callers to use an external name at all for a given parameter
  - This is the default for the first parameter (except in initializers!)
  - For other (not the first) parameters, the internal name is, by default, the external name
  - Any parameter's external name can be changed (even forcing the first parameter to have one)
  - It is generally "anti-Swift" to force a first parameter name or suppress other parameters names

```
func foo(first: Int, second: Double) {
    var sum = 0.0
    for _ in 0..<first { sum += second }
}

func bar() {
    let result = foo(123, second: 5.5)
}
```

# Methods

- Obviously you can override methods/properties in your superclass
  - Precede your `func` or `var` with the keyword `override`
  - A method can be marked `final` which will prevent subclasses from being able to override
  - Classes can also be marked `final`
- Both types and instances can have methods/properties
  - For this example, lets consider using the `struct Double` (yes, `Double` is a `struct`)

```
var d: Double = …
if d.isSignMinus {
    d = Double.abs(d)
}
```

  - `isSignMinus` is an instance property of a `Double` (you send it to a particular `Double`)
  - `abs` is a type method of `Double` (you send it to the type itself, not to a particular `Double`)
  - You declare a type method or property with a `static` prefix …

```
static func abs(d: Double) -> Double
```

# Properties

- Property Observers

  You can observe changes to any property with `willSet` and `didSet`

  Will also be invoked if you mutate a `struct` (e.g. add something to a dictionary)

  One very common thing to do in an observer in a Controller is to update the user-interface

```
var someStoredProperty: Int = 42 {
    willSet { newValue is the new value }
    didSet { oldValue is the old value }
}

override var inheritedProperty {
    willSet { newValue is the new value }
    didSet { oldValue is the old value }
}

var operations: Dictionary<String, Operation> = [ ... ] {
    willSet { will be executed if an operation is added/removed }
    didSet { will be executed if an operation is added/removed }
}
```

# Properties

- Lazy Initialization

  A `lazy` property does not get initialized until someone accesses it

  You can allocate an object, execute a closure, or call a method if you want

```
lazy var brain = CalculatorBrain() // nice if CalculatorBrain used lots of resources

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

lazy var myProperty = self.initializeMyProperty()
```

  This still satisfies the "you must initialize all of your properties" rule

  Unfortunately, things initialized this way can't be constants (i.e., `var` ok, `let` not okay)

  This can be used to get around some initialization dependency conundrums

# Array

- Array

```
var a = Array<String>()
… is the same as …
var a = [String]()

let animals = [“Giraffe”, “Cow”, “Doggie”, “Bird”]
animals.append(“Ostrich”) // won’t compile, animals is immutable (because of let)
let animal = animals[5] // crash (array index out of bounds)

// enumerating an Array
for animal in animals {
    println(“\(animal)”)
}
```

# Array

- Interesting `Array<T>` methods

  This one creates a new array with any "undesirables" filtered out
  The function passed as the argument returns `false` if an element is undesirable

```
filter(includeElement: (T) -> Bool) -> [T]
let bigNumbers = [2,47,118,5,9].filter({ $0 > 20 }) // bigNumbers = [47, 118]
```

  Create a new array by transforming each element to something different
  The thing it is transformed to can be of a different type than what is in the Array

```
map(transform: (T) -> U) -> [U]
let stringified: [String] = [1,2,3].map { String($0) }
```

  Reduce an entire array to a single value

```
reduce(initial: U, combine: (U, T) -> U) -> U
let sum: Int = [1,2,3].reduce(0) { $0 + $1 }  // adds up the numbers in the Array
```

# Dictionary

- Dictionary

```
var pac10teamRankings = Dictionary<String, Int>()
… is the same as …
var pac10teamRankings = [String:Int]()

pac10teamRankings = [”Stanford”:1, ”Cal”:10]
let ranking = pac10teamRankings[“Ohio State”] // ranking is an Int? (would be nil)

// use a tuple with for-in to enumerate a Dictionary
for (key, value) in pac10teamRankings {
    print(“\(key) = \(value)”)
}
```

# String

- The characters in a String

  The simplest way to deal with the characters in a string is via this property …

  ```
  var characters: String.CharacterView { get }
  ```

  You can think of this as an `[Character]` (it's not actually that, but it works like that).
  A `Character` is a "human understandable idea of a character".
  That will make it easier to index into the characters.

  Indexing into a `String` itself is quite a bit more complicated.
  Your reading assignment goes over it all.

# String

- Other `String` Methods

  String is automatically "bridged" to the old Objective-C class `NSString`
  So there are some methods that you can invoke on `String` that are not in `String`'s doc
  You can find them in the documentation for `NSString` instead.
  Here are some other interesting `String` methods ...

```
startIndex -> String.Index
endIndex -> String.Index
hasPrefix(String) -> Bool
hasSuffix(String) -> Bool
capitalizedString -> String
lowercaseString -> String
uppercaseString -> String
componentsSeparatedByString(String) -> [String] // "1,2,3".csbs(",") = ["1","2","3"]
```

# Other Classes

- NSObject
  - Base class for all Objective-C classes
  - Some advanced features will require you to subclass from `NSObject` (and it can't hurt to do so)
- NSNumber
  - Generic number-holding class

```
let n = NSNumber(35.5)
let intversion: Int = n.intValue // also doubleValue, boolValue, etc.
```

- NSDate
  - Used to find out the date and time right now or to store past or future dates.
  - See also `NSCalendar`, `NSDateFormatter`, `NSDateComponents`
  - If you are displaying a date in your UI, there are localization ramifications, so check these out!
- NSData
  - A "bag o' bits". Used to save/restore/transmit raw data throughout the iOS SDK.

# Initialization

- When is an `init` method needed?
  - `init` methods are not so common because properties can have their defaults set using `=`
  - Or properties might be Optionals, in which case they start out `nil`
  - You can also initialize a property by executing a closure
  - Or use `lazy` instantiation
  - So you only need `init` when a value can't be set in any of these ways
- You also get some "free" `init` methods
  - If all properties in a base `class` (no superclass) have defaults, you get `init()` for free
  - If a `struct` has no initializers, it will get a default one with all properties as arguments

```
struct MyStruct {
    var x: Int
    var y: String
}

let foo = init(x: 5, y: "hello") // free init() method!
```

# Initialization

- What can you do inside an `init`?
  - You can set any property's value, even those with default values
  - Constant properties (i.e. properties declared with `let`) can be set
  - You can call other `init` methods in your own class using `self.init(<args>)`
  - In a class, you can of course also call `super.init(<args>)`
  - But there are some rules for calling `init`s from `init`s in a `class` …

# Initialization

- What are you required to do inside `init`?
  - By the time any `init` is done, all properties must have values (optionals can have the value `nil`)
  - There are two types of `init`s in a `class`: `convenience` and designated (i.e. not `convenience`)
  - A designated `init` must (and can only) call a designated `init` that is in its immediate `super`class
  - You must initialize all properties introduced by your class before calling a superclass's `init`
  - You must call a superclass's `init` before you assign a value to an inherited property
  - A `convenience` `init` must (and can only) call an `init` in its own class
  - A `convenience` `init` must call that `init` before it can set any property values
  - The calling of other `init`s must be complete before you can access properties or invoke methods
  - Whew!

# Initialization

- Inheriting `init`
  - If you do not implement any designated `init`s, you'll inherit all of your superclass's designateds
  - If you `override` all of your superclass's designated `init`s, you'll inherit all its `convenience init`s
  - If you implement no `init`s, you'll inherit all of your superclass's `init`s
  - Any `init` inherited by these rules qualifies to satisfy any of the rules on the previous slide
- Required `init`
  - A class can mark one or more of its `init` methods as `required`
  - Any subclass must implement said `init` methods (though they can be inherited per above rules)

# Initialization

- Failable init

  If an init is declared with a ? (or !) after the word init, it returns an Optional

```
init?(arg1: Type1, …) {
    // might return nil in here
}
```

  These are rare.

```
let image = UIImage(named: “foo”) // image is an Optional UIImage (i.e. UIImage?)
```

  Usually we would use if-let for these cases …

```
if let image = UIImage(named: “foo”) {
    // image was successfully created
} else {
   // couldn’t create the image
}
```

# Initialization

- Creating Objects

  Usually you create an object by calling it's initializer via the type name …

```
let x = CalculatorBrain()
let y = ComplicatedObject(arg1: 42, arg2: “hello”, …)
let z = [String]()
```

  Obviously sometimes other objects will create objects for you.

# AnyObject

- `AnyObject` is a special type (actually it's a protocol)
  - Used to be commonly used for compatibility with old Objective-C APIs
  - But not so much anymore in iOS9 since those old Objective-C APIs have been updated
  - A variable of type `AnyObject` can point to any `class`, but you don't know which
  - A variable of type `AnyObject` cannot hold a `struct` or an `enum`
  - There is another type, `Any`, which can hold anything (very, very rarely used)
- Where will you see it?
  - Sometimes (rarely) it will be an argument to a function that can actually take any `class` …

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject)
```

```
func touchDigit(sender: AnyObject)
```

  - Or when you want to return an object and you don't want the caller to know its `class` …

```
var cookie: AnyObject
```

# AnyObject

- How do we use a variable of type `AnyObject`?

  We can't usually use it directly (since we don't know what `class` it is)
  Instead, we must convert it to another, known `class` (or protocol)

  This conversion might not be possible, so the conversion generates an Optional
  Conversion is done with the `as?` keyword in Swift (or `as!` to force unwrap the Optional)
  You can also check to see if something can be converted with the `is` keyword (`true`/`false`)

  We usually use `as?` it with `if let` …

```
let ao: AnyObject = …
if let foo = ao as? SomeClass {
    // we can use foo and know that it is of type SomeClass in here
}
```

# AnyObject

- Example …

  Remember when we wired up our Actions in our storyboard?
  The default in the dialog that popped up was `AnyObject`. We changed it to `UIButton`.
  But what if we hadn't changed it to `UIButton` because `UISliders` could also send it?
  (Let's imagine the slider slides through all the numbers 0 through 9 … silly.)
  How would we have implemented `touchDigit`?

```
@IBAction func touchDigit(sender: AnyObject) {
    if let sendingButton = sender as? UIButton {
        let digit = sendingButton.currentTitle!
        …
    } else if let sendingSlider = sender as? UISlider {
        let digit = String(Int(sendingSlider.value))
        …
    }
}
```

# Property List

- Another use of `AnyObject`: Property List

  Property List is really just the definition of a term
  It means an `AnyObject` which is known to be a collection of objects which are ONLY one of …
  `String`, `Array`, `Dictionary`, a number (`Double`, `Int`, etc.), `NSData`, `NSDate`
  e.g. a `Dictionary` whose keys were `String` and values were `Array` of `NSDate` is one

  But wait! `String`, `Array`, `Dictionary`, `Double` … these are all `structs`, not `classes`!
  Yes, but they are automatically "bridged" to Objective-C counterparts which are `classes`.
  Objective-C is almost invisible now in the Swift API to iOS, so we'll skip talking about bridging.
  But we will talk about Property List and even give an example of it.

  Property Lists are used to pass generic data structures around "blindly".
  The semantics of the contents of a Property List are known only to its creator.
  Everyone else just passes it around as `AnyObject` and doesn't know what's inside.
  Let's look at an iOS API that does this: `NSUserDefaults` …

# NSUserDefaults

- A storage mechanism for Property List data

  NSUserDefaults is essentially a very tiny database that stores Propery List data.
  It persists between launchings of your application!
  Great for things like "settings" and such.
  Do not use it for anything big!

  It can store/retrieve entire Property Lists by name (keys) ...

```
setObject(AnyObject, forKey: String)
objectForKey(String) -> AnyObject?
arrayForKey(String) -> Array<AnyObject>? // returns nil if you setObject(not-an-array)
```

  It can also store/retrieve little pieces of data ...

```
setDouble(Double, forKey: String)
doubleForKey(String) -> Double
```

# NSUserDefaults

- Using NSUserDefaults

  Get the defaults reader/writer …

```
let defaults = NSUserDefaults.standardUserDefaults()
```

  Then read and write …

```
let plist = defaults.objectForKey("foo")
defaults.setObject(plist, forKey: "foo")
```

  Your changes will be automatically saved.
  But you can be sure they are saved at any time by synchronizing …

```
if !defaults.synchronize() { // failed! but not clear what you can do about it }
```

  (it's not "free" to synchronize, but it's not that expensive either)

# Property List

- Another example of Property List

  What if we wanted to export the sequence of events that was input to the `CalculatorBrain`.
  We could consider this the `CalculatorBrain`'s "program".

  ```
  var program: AnyObject
  ```

  This would be `get`-able and `set`-able.
  The `program` will be a Property List containing all the operands/operations done.
  Only the `CalculatorBrain` will be able to interpret the program.
  So it's only good for getting it, holding onto it, then giving it back to the `CalculatorBrain` later.
  In assignment 2, we'll use this to create a "programmable calculator" with variables.

  To give you a head start on your assignment, let's implement this `var` in `CalculatorBrain` ...

# Casting

- By the way, casting is not just for AnyObject

  You can cast with as (or check with is) any object pointer that makes sense

  For example …

```
let vc: UIViewController = CalculatorViewController()
```

  The type of vc is UIViewController (because we explicitly typed it to be)

  And the assignment is legal because a CalculatorViewController is a UIViewController

  But we can't say, for example, vc.displayValue

  However, if we cast vc, then we can use it …

```
if let calcVC = vc as? CalculatorViewController {
    let x = calcVC.displayValue // this is okay
}
```

  Or we could force the cast (might crash) by using as! (with no if let) rather than as?

# Assertions

- Debugging Aid

  Intentionally crash your program if some condition is not true (and give a message)

  `assert(() -> Bool, "message")`

  The function argument is an "autoclosure" however, so you don't need the `{ }`

  e.g. `assert(validation() != nil, "the validation function returned nil")`

  Will crash if `validation()` returns `nil` (because we are asserting that `validation()` does not)

  The `validation() != nil` part could be any code you want

  When building for release (to the AppStore or whatever), `asserts` are ignored completely