Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- More Swift & the Foundation Framework
  - Optionals and enum
  - Array<T>, Dictionary<K,V>, Range<T>, et. al.
  - Data Structures in Swift
  - Methods
  - Properties
  - Initialization
  - AnyObject, introspection and casting (is and as)
  - Helpful Methods and Functions
  - String vs. NSString, Array vs. NSArray, Dictionary vs. NSDictionary
  - Property List
  - NSUserDefaults

Optional

- An Optional is just an enum

Conceptually it is like this (the <T> is a generic like as in Array<T>) …

```
enum Optional<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
}
```

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)”)
}
```

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 {
    println("\(key) = \(value)")
}
```

Range

- Range

  A Range in Swift is just two end points of a sensible type (not gonna explain right now)

  Range is generic (e.g. Range<T>)

  This is sort of a pseudo-representation of Range:

```
struct Range<T> {
    var startIndex: T
    var endIndex: T
}
```

  An Array's range would be a Range<Int> (since Arrays are indexed by Int)

  Warning: A String subrange is <u>not</u> Range<Int> (it is Range<String.Index> … we'll talk later!)

  There is special syntax for specifying a Range: either ... (inclusive) or ..< (open-ended)

```
let array = ["a","b","c","d"]
let subArray1 = array[2...3]  // subArray1 will be ["c","d"]
let subArray2 = array[2..<3]  // subArray2 will be ["c"]
for i in 27...104 { }         // Range is enumeratable, like Array, String, Dictionary
```

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 = 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.

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)
  - Introspection and casting (`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, by default, constants
  - You can put the keyword `var` on an parameter, and it will be mutable, but it's still a copy
  - 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

- 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 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 (or class in a class) …

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

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(external internal: Int) {
    let local = internal
}

func bar() {
    let result = foo(external: 123)
}
```

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 will use to call the method

```
func foo(external internal: Int) {
    let local = internal
}

func bar() {
    let result = foo(external: 123)
}
```

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 will use to 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(_ internal: Int) {
    let local = internal
}

func bar() {
    let result = foo(123)
}
```

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 will use to call the method

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

  An _ is the default for the first parameter (only) in a method (but not for init methods)

```
func foo(internal: Int) {
    let local = internal
}

func bar() {
    let result = foo(123)
}
```

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 will use to call the method
  You can put _ if you don't want callers to use an external name at all for a given parameter
  An _ is the default for the first parameter (only) in a method (but not for init methods)
  You can force the first parameter's external name to be the internal name with #

```
func foo(#internal: Int) {
    let local = internal
}

func bar() {
    let result = foo(internal: 123)
}
```

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 will use to call the method

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

An _ is the default for the first parameter (only) in a method (but not for `init` methods)

You can force the first parameter's external name to be the internal name with #

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

```
func foo(first: Int, second: Double) {
    let local = internal
}

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 will use to call the method
You can put _ if you don't want callers to use an external name at all for a given parameter
An _ is the default for the first parameter (only) in a method (but not for init methods)
You can force the first parameter's external name to be the internal name with #
For other (not the first) parameters, the internal name is, by default, the external name
Any parameter's external name can be changed

```
func foo(first: Int, externalSecond second: Double) {
    let local = internal
}

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 will use to call the method
  You can put _ if you don't want callers to use an external name at all for a given parameter
  An _ is the default for the first parameter (only) in a method (but not for init methods)
  You can force the first parameter's external name to be the internal name with #
  For other (not the first) parameters, the internal name is, by default, the external name
  Any parameter's external name can be changed
  Or even omitted (though this would be sort of "anti-Swift")

```
func foo(first: Int, _ second: Double) {
    let local = internal
}

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

Properties

- Property Observers

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

```
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 }
}
```

Properties

Property Observers

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

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

Lazy Initialization

A `lazy` property does not get initialized until someone <u>accesses</u> 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

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 = 42
    var y: String = "moltuae"

    init(x: Int, y: String) // comes for free
}
```

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 inits from inits 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 inits 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 superclass
  - 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 a designated init in its own class
  - A convenience init may call a designated init indirectly (through another convenience init)
  - A convenience init must call a designated init before it can set any property values
  - The calling of other inits must be complete before you can access properties or invoke methods

Initialization

- Inheriting `init`
  - If you do not implement <u>any</u> 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.

  Note: The documentation does not seem to properly show these inits!

  But you'll be able to tell because the compiler will warn you about the type when you access it.

```
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]()
```

But sometimes you create objects by calling type methods in classes …

```
let button = UIButton.buttonWithType(UIButtonType.System)
```

Or obviously sometimes other objects will create objects for you …

```
let commaSeparatedArrayElements: String = ",".join(myArray)
```

AnyObject

- Special "Type" (actually it's a Protocol)

  Used primarily for compatibility with existing Objective-C-based APIs

- Where will you see it?

  As properties (either singularly or as an array of them), e.g. …

```
var destinationViewController: AnyObject
var toolbarItems: [AnyObject]
```

  … or as arguments to functions …

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject)
func addConstraints(constraints: [AnyObject])
func appendDigit(sender: AnyObject)
```

  … or even as return types from functions …

```
class func buttonWithType(buttonType: UIButtonType) -> AnyObject
```

AnyObject

- How do we use `AnyObject`?
  - We don't usually use it directly
  - Instead, we convert it to another, known type
- How do we convert it?
  - We need to create a new variable which is of a known object type (i.e. not `AnyObject`)
  - Then we assign this new variable to hold the thing that is `AnyObject`
  - Of course, that new variable has to be of a compatible type
  - If we try to force the `AnyObject` into something incompatible, crash!
  - But there are ways to check compatibility (either before forcing or while forcing) …

AnyObject

- Casting AnyObject

  We "force" an `AnyObject` to be something else by "casting" it using the `as` keyword …

  Let's use `var destinationViewController: AnyObject` as an example …

  ```
  let calcVC = destinationViewController as CalculatorViewController
  ```

  … this would crash if `dvc` was not, in fact, a `CalculatorViewController` (or subclass thereof)

  To protect against a crash, we can use `if let` with `as?` …

  ```
  if let calcVC = destinationViewController as? CalculatorViewController { … }
  ```

  … `as?` returns an Optional (`calcVC = nil` if `dvc` was not a `CalculatorViewController`)

  Or we can check before we even try to do `as` with the `is` keyword …

  ```
  if destinationViewController is CalculatorViewController { … }
  ```

AnyObject

- Casting Arrays of AnyObject

  If you're dealing with an [AnyObject], you can cast the elements or the entire array …

  Let's use var toolbarItems: [AnyObject] as an example …

```
for item in toolbarItems {                                   // item's type is AnyObject
    if let toolbarItem = item as? UIBarButtonItem {
        // do something with the toolbarItem (which will be a UIBarButtonItem here)
    }
}
```

  … or …

```
for toolbarItem in toolbarItems as [UIBarButtonItem] { // better be so, else crash!
    // do something with the toolbarItem (which will be a UIBarButtonItem)
}
```

  // can't do as? here because then it might be "for toolbarItem in nil" (makes no sense)

AnyObject

- Another 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?
  How would we have implemented appendDigit?

```
@IBAction func appendDigit(sender: AnyObject) {
    if let sendingButton = sender as? UIButton {
        let digit = sendingButton.currentTitle!
        …
    }
}
```

AnyObject

- Yet another example …

  It is possible to create a button in code using a UIButton type method …

```
let button: AnyObject = UIButton.buttonWithType(UIButtonType.System)
```

  The type of this button is AnyObject (for historical reasons only)
  To use it, we'd have to cast button to UIButton

  We can do this on the fly if we want …

```
let title = (button as UIButton).currentTitle
```

  Again, this would crash if button was not, in fact, a UIButton

Casting

- 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.enter()`

  ```
  if let calcVC = vc as? CalculatorViewController {
      // in here we could say calcVC.enter() if we wanted to
  }
  ```

Functions

- Some Array<T> Methods

```
+= [T]          // not += T
first -> T?     // note optional
last -> T?      // note optional

var a = [a,b,c]  // assume a, b, c are of some type (the same type)

append(T)
insert(T, atIndex: Int)          // a.insert(d, atIndex:1), a = [a,d,b,c]
splice(Array<T>, atIndex: Int)   // a.splice([d,e], atIndex:1), a = [a,d,e,b,c]

removeAtIndex(Int)  // a.removeAtIndex(1), a = [a,c]
removeRange(Range)  // a.removeRange(0..<2), a = [c]
replaceRange(Range, [T]) // a.replaceRange(0...1, with: [x,y,z]), a = [x,y,z,b]

sort(isOrderedBefore: (T, T) -> Bool) // e.g., a.sort { $0 < $1 }
```

Functions

- More 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]
  ```

  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 { "\($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
  ```

String

- `String.Index`

  In Unicode, a given glyph might be represented by multiple Unicode characters (accents, etc.)
  As a result, you can't index a `String` by `Int` (because it's a collection of characters, not glyphs)

  So a lot of native Swift `String` functions take a `String.Index` to specify which glyph you want
  You can get a `String.Index` by asking the string for its `startIndex` then advancing forward
  You advance forward with the function (not method) `advance(String.Index, Int)`

```
var s = "hello"
let index = advance(s.startIndex, 2)  // index is a String.Index to the 3rd glyph, "l"
s.splice("abc", index)                // s will now be "heabcllo" (abc inserted at 2)
let startIndex = advance(s.startIndex, 1)
let endIndex = advance(s.startIndex, 6)
let substring = s[index..<endIndex]   // substring will be "eabcl"
```

String

- String.Index

The method rangeOfString returns an Optional Range<String.Index>
As an example, to get whole number part of a string representing a double …

```
let num = "56.25"
if let decimalRange = num.rangeOfString(".") { // decimalRange is Range<String.Index>
    let wholeNumberPart = num[num.startIndex..<decimalRange.startIndex]
}
```

We could remove the whole number part using this method …

```
s.removeRange([s.startIndex..<decimalRange.startIndex])
```

(there are other (defaulted) arguments to removeRange, but I'm not going to cover them now)

There's also replaceRange(Range, String)

String

- Other String Methods

```
description -> String                                      // Printable
endIndex -> String.Index
hasPrefix(String) -> Bool
hasSuffix(String) -> Bool
toInt() -> Int?                                            // no toDouble (# of sig digits? etc.)
capitalizedString -> String
lowercaseString -> String
uppercaseString -> String
join(Array) -> String                                      // “,”.join([“1”,”2”,”3”]) = “1,2,3”
componentsSeparatedByString(String) -> [String] // “1,2,3”.csbs(“,”) = [“1”,”2”,”3”]
```

Type Conversion

- Conversion between types with `init()`

  A sort of "hidden" way to convert between types is to create a new object by converting

```
let d: Double = 37.5
let f: Float = 37.5
let x = Int(d)                     // truncates
let xd = Double(x)
let cgf = CGFloat(d)               // we'll see CGFloat later in the quarter

let a = Array("abc")               // a = ["a","b","c"], i.e. array of Character
let s = String(["a","b","c"])      // s = "abc" (the array is of Character, not String)

let s = String(52)                 // no floats
let s = "\(37.5)"                  // but don't forget about this mechanism
```

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