Lecture 3

Optionals

- An Optional is just an enum

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

```swift
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
}
//---------------------------
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 Chaining

- 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?
  - 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 EQUIVALENTLY...

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

Optional defaulting operator (??)

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

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

// OR EQUIVALENTLY...
display.text = s ?? " "
```

Tuples

- A grouping of values
- You can use it anywhere you can use a type

```
let x: (String, Int, Double) = ("hello", 5, 0.85)
// tuple elements named when accessing the tuple
let (word, number, value) = x
print(word) // hello
print(number) // 5
print(value) // 0.85

// tuple elements named when tuple declared
let x: (w: String, i: Int, v: Double) = ("hello", 5, 0.85)
print(x.w) // hello
print(x.i) // 5
print(x.v) // 0.85
```

- Very useful for 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

- **THREE FUNDAMENTAL BUILDING BLOCKS OF DATA STRUCTURES**
  a. Classes
  b. Structs
  c. Enumerations

Similarities

- Declaration syntax

```
class CalculatorBrain {}
struct Vertex {}
enum Op {}
```

- Properties and functions

```
func doIt(argument: Type) -> ReturnValue {}
var storedProperty = <initial value>
// enums cannot have stored properties
var computedProperty: Type {
  get{}
  set{}
}
```

- Initializers
  - except enums

```
init(argument1: Type, argument2: Type, ...) {}
```

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`
        - If you assign a struct/enum to a variable (e.g. `let x = myStruct`), you can no longer change `myStruct`
    - Remember that function parameters are constant
    - 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) (ARC)
    - 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 the same instance
- Which to use?
    - Usually choose `class` over `struct`
        - `struct` tends to be for more fundamental types
    - Use `enum` any time you have a type of data with discrete values

---

Methods

Parameter Names

- All parameters to all functions have an **internal** name and an external name
    - **Internal** name is the name of the local variable you use inside the method
    - 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) { … }`
        - This is the default for first parameter (except in initializers)
    - For other (non-first) parameters, 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(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)
}
```

- Obviously you can override methods/properties from 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` (preventing subclassing)
- Both types and instances can have methods/properties
  - For this example, let's 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 value-type 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 UI

```
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 about to be added/removed
  }
  didSet {
    // will be executed after an operation is added/removed
  }
}
```

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. `lazy var` okay, `lazy let` not okay
- This can be used to get around some initialization dependency conundrums

Array

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

let animals = ["Giraffe", "Cow", "Dog", "Bird"]
animals.append("Ostrich")
// won't compile, animals is immutable (because animals is constant)
let animal = animals[5] // crash (out of bounds index)

// enumerating an array
for animal in animals {
  println("\(animal)")
}
```

Interesting Array<T> Methods

- `filter(includeElement: (T) -> Bool) -> [T]`
  - Creates a new array with any “undesirables” filtered out
  - The function passed as the argument returns flase if an element is undesirable

```
// Filters out any integers under 21
let bigNumbers = [2,47,118,5,9].filter({ $0 > 20 })
```

- `map(transform: (T) -> U) -> [U]`
  - The thing it is transformed to can be of a different type than what is in the input array

```
// Converts integer array to string array
let stringified: [String] = [1,2,3].map { String($0) }

// No parentheses around closure because:
// () are optional when closure is last parameter of function
// TRAILING-CLOSURE SYNTAX
```

- `reduce(initial: U, combine: (U, T) -> U) -> U`
  - Reduce an entire array to a single value

```
// Adds up the nubmers in the Array
let sum: Int [1,2,3].reduce(0) { $0 + $1 }
```

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 type Int? (would be nil in this case)

// 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 a `[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
  - Check reading assignment for more details

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 the docs
  - You can find them in the `NSString` docs instead
- 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(",") will return ["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: NSCalender, 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!
```

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`

What are you required to do inside `init`?

- By the time `init` is done, all properties must have values (optionals may be `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 an `init` in its own class
- A `convenience init` must call that `init` before it can set any property values
- The calling of other `inits` must be complete before you can access properties or invoke methods

Inheriting `init`

- If you do not implement any designated `inits`, you'll inherit all of your superclass's designated `inits`
- If you override all of your superclass's designated `inits`, you'll inherit all its `convenience inits`
- If you implement no `inits`, you'll inherit all of your superclass's inits
- 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)

Failable `init`

- If an `init` is declared with a ? (or !) after the `init` keyword, 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

// 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
}
```

Creating Objects

- Usually you create an object by calling its initializer via the type name
  - `let x = CalculatorBrain()`
  - `let z = [String]()`
- Obviously, sometimes other objects will create objects for you

---

AnyObject

- `AnyObject` is a special type (actually a protocol)
    - 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
```

- 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 conversion returns Optional
    - Conversion is done with as? keyword in Swift (or as! to force unwrap)
    - You can also check to see if something can be converted with the is keyword (true/false)

```
// We usually use as? 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
}
```

Property List (another use of `AnyObject`)

- 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
- 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 Property 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>?`
- It can also store/retrive little pieces of data
    - `setDouble(Double, forKey: String)`
    - `doubleForKey(String) -> Double`
- User **`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 unclear what you can do about it }`
        - it’s not FREE to sync but not that expensive either

Another example of Property List

- What if we wanted to export th