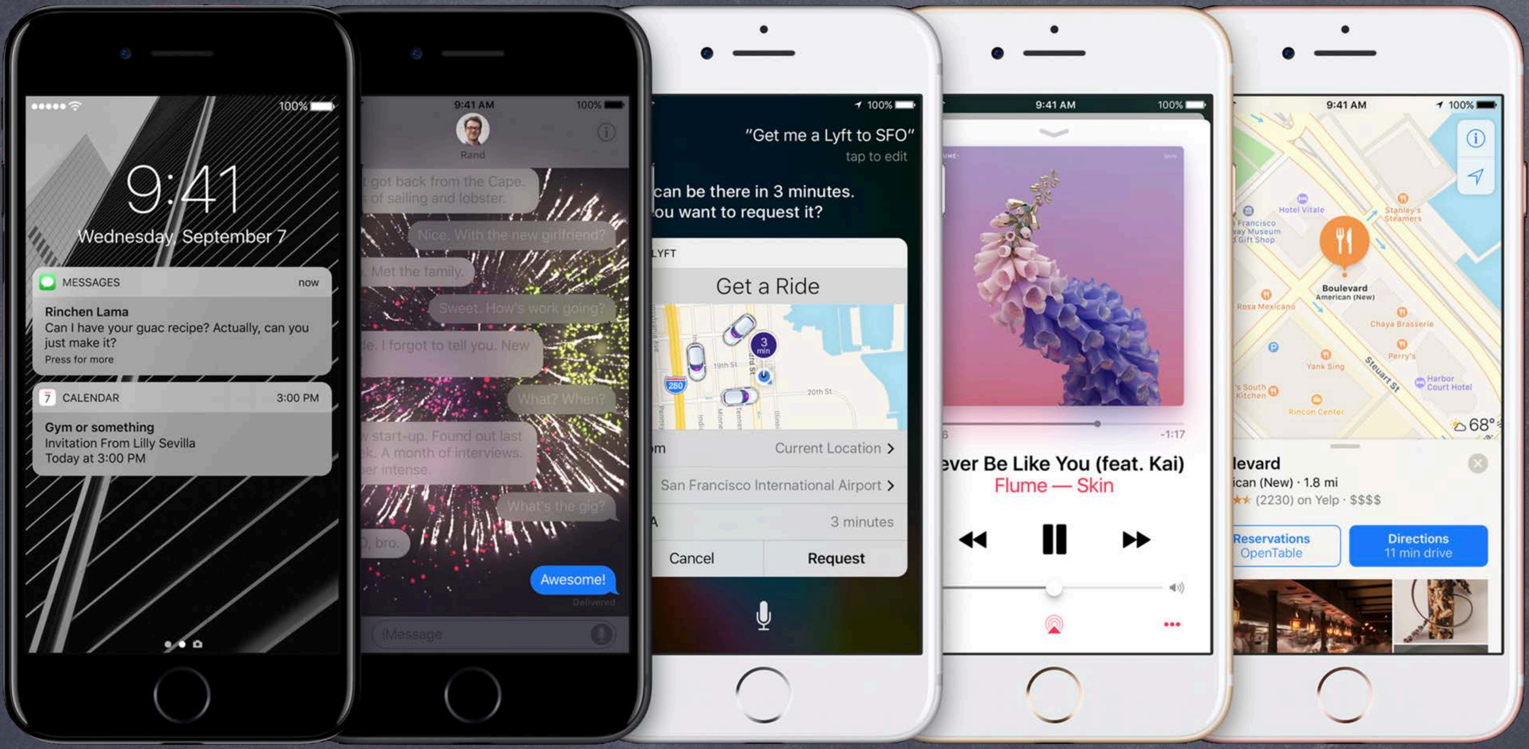

Stanford CS193p

Developing Applications for iOS
Winter 2017

Today

- More Swift & the Foundation Framework
  - What are Optionals really?
  - Tuples
  - `Range<T>`
  - Data Structures, Methods and Properties
  - `Array<T>`, `Dictionary<K,V>`, `String`, et. al.
  - Initialization
  - `AnyObject`, introspection and casting (`is` and `as`)
  - `UserDefaults`
  - `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)
}
```

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”)
```

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”)
```

```
let y = x!
```
… is …
```
switch x {
    case some(let value): y = value
    case none: // raise an exception
}
```

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")
```

```
let 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

- Optionals can be "chained"

For example, `hashValue` is a `var` in `String`.
What if we wanted to get the `hashValue` from an Optional `String`?
And what if that Optional `String` was, itself, the `text` of an Optional `UILabel`?

```
var display: UILabel? // imagine this is an @IBOutlet without the implicit unwrap !
if let temp1 = display {
    if let temp2 = temp1.text {
        let x = temp2.hashValue
        ...
    }
}
```

... with Optional chaining using `?` instead of `!` to unwrap, this becomes ...

```
if let x = display?.text?.hashValue { ... }  // x is an Int
   let x = display?.text?.hashValue { ... }  // x is an Int?
```

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) // 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
```

Range

Range

A Range in Swift is just two end points.
A Range can represent things like a selection in some text or a portion of an Array.
Range is generic (e.g. Range<T>), but T is restricted (e.g. comparable).
This is sort of a pseudo-representation of Range …

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

So, for example, a Range<Int> would be good for a range specifying a slice of an Array.

There are other, more capable, Ranges like CountableRange.
A CountableRange contains consecutive values which can be iterated over or indexed into.

Range

Range

There is special syntax for creating a Range.
Either ..< (exclusive of the upper bound) or ... (inclusive of both bounds)

```
let array = ["a","b","c","d"]
let a = array[2...3] // a will be a slice of the array containing ["c","d"]
let b = array[2..<3] // b will be a slice of the array containing ["c"]
let c = array[6...8] // runtime crash (array index out of bounds)
let d = array[4...1] // runtime crash (lower bound must be smaller than upper bound)
```

A String subrange is not Range<Int> (it's Range<String.Index>)

```
let e = "hello"[2..<4]          // this != "ll", in fact, it won't even compile
let f = "hello"[start..<end]    // this is possible; we'll explain start and end a bit later
```

Range

- Range

  If the type of the upper/lower bound is an Int, ..< makes a CountableRange.
  (Actually, it depends on whether the upper/lower bound is "strideable by Int" to be precise.)
  CountableRange is enumeratable with for in.
  For example, this is how you do a C-like for (i = 0; i < 20; i++) loop …

```
for i in 0..<20 {
}
```

  How about something like 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).

Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 of the 4 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax …

```
class ViewController: … {

}
struct CalculatorBrain {

}
enum Op {

}
```

Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 of the 4 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax …

  Properties and Functions …

```
func doit(argx argi: 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 of the 4 fundamental building blocks of data structures in Swift

- Similarities

  Declaration syntax …

  Properties and Functions …

  Initializers (again, not enum) …

```
init(arg1x arg1i: Type, arg2x arg2i: Type, …) {

}
```

Data Structures in Swift

- Classes, Structures and Enumerations

  These are the 3 of the 4 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 (function parameters are let)
  - 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?
  - Already discussed class versus struct in previous lecture (also in your Reading Assignment).
  - 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
This would almost never be done for anything but the first 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 would almost never be done for anything but the first parameter.
  If you only put one parameter name, it will be both the external and internal name.

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

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

Methods

- 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
  - Entire classes can also be marked final

Methods

- Both types and instances can have methods/properties

  Type methods and properties are denoted with the keyword `static`.
  For example, the `struct Double` has a number of `var`s and `func`s on its type.
  These are not methods or `var`s you access on an instance of a `Double` (e.g. on 53.2).
  Instead, you access them by referencing the `Double` type itself.

```
static func abs(d: Double) -> Double  { if d < 0 { return -d } else { return d } }
static var pi: Double                 { return 3.1415926 }
let d = Double.pi                     // d = 3.1415926
let d = Double.abs(-324.44)           // d = 324.44
let x: Double = 23.85
let e = x.pi                          // no! pi is not an instance var
let e = x.abs(-22.5)                  // no! abs is not an instance method
```

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: String {
    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
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]() // this appears to be winning the battle of "preferred"

let animals = ["Giraffe", "Cow", "Doggie", "Bird"]  // inferred to be Array<String>
animals.append("Ostrich")   // won't compile, animals is immutable (because of let)
let animal = animals[4]     // crash (array index out of bounds)

// enumerating an Array (it's a "sequence" just like a CountableRange is)
for animal in animals {
    print(animal)
}
```

Array

- Interesting `Array<T>` methods which take closures

  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) }  // ["1","2","3"]
```

  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
let sum = [1,2,3].reduce(0, +)     // same thing because + is just a function in Swift
```

Dictionary

- Dictionary

```
var pac12teamRankings = Dictionary<String,Int>()
```
… is the same as …
```
var pac12teamRankings = [String:Int]()

pac12teamRankings = [”Stanford”:1, ”USC”:11]
let ranking = pac12teamRankings[“Ohio State”] // ranking is an Int? (would be nil)
pac12teamRankings[“Cal”] = 12

// use a tuple with for-in to enumerate a Dictionary
for (key, value) in pac12teamRankings {
    print(“Team \(key) is ranked number \(value)”)
}
```

String

- The characters in a String

A String is made up of Unicodes, but there's also the concept of a Character.
A Character is what a human would perceive to be a single lexical character.
This is true even if it is made up of multiple Unicodes.
For example, café might be 5 Unicodes (the accent might be separate), but it's 4 Characters.

You can access any character (of type Character) in a String using [] notation.
But the indexes inside the [] are not Int, they are a type called String.Index.

```
let s: String = "hello"                      // hmm, what if we basically wanted s[0] (i.e. the "h")?
let firstIndex: String.Index = s.startIndex   // note that firstIndex's type is not an Int
let firstChar: Character = s[firstIndex]      // firstChar = the Character h
let secondIndex: String.Index = s.index(after: firstIndex)
let secondChar: Character = s[secondIndex]    // secondChar = e
let fifthChar: Character = s[s.index(firstIndex, offsetBy: 4)] // fifthChar = o
let substring = s[firstIndex...secondIndex]   // substring = "he"
```

String

- The characters in a String

  Even though String is indexable (using []), it's not a collection or a sequence (like an Array).
  Only sequences and collections can do things like for in or index(of:).
  Luckily, the characters var in String returns a collection of the String's Characters.

  With it, you can do things like …

```
for c: Character in s.characters { }  // iterate through all Characters in s
let count = s.characters.count         // how many Characters in s?
let firstSpace: String.Index = s.characters.index(of: " ")
// a String.Index into the String's characters matches a String.Index into the String
```

String

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

  So whether you can modify its characters depends on var versus let.

```
let hello = “hello”              // immutable String
var greeting = hello             // mutable String
hello += “ there”                // this is illegal because hello is immutable
greeting += “ there”             // greeting, however, is a var and thus is mutable
print(greeting)                  // “hello there”
print(hello)                     // “hello”
```

  Of course you can manipulate Strings in much more complicated ways than appending …

```
if let firstSpace = greeting.characters.index(of: “ “) {
    // insert(contentsOf:at:) inserts a collection of Characters at the specified index
    greeting.insert(contentsOf: “ you”.characters, at: firstSpace)
}
print(greeting)                  // “hello you there”
```

String

- Other String Methods

```
var endIndex: String.Index        // this is never a valid index into the String

func hasPrefix(String) -> Bool
func hasSuffix(String) -> Bool

var localizedCapitalized/Lowercase/Uppercase: String

func replaceSubrange(Range<String.Index>, with: String)
e.g., s.replaceSubrange(s.startIndex..<s.endIndex, with: “new contents”)

func components(separatedBy: String) -> [String]
e.g., let array = “1,2,3”.components(separatedBy: “,”)     // array = [“1”,”2”,”3”]
```

And much, much more. Check out the documentation.

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 (i.e., reference type)
  - `let n = NSNumber(35.5)` or `let n: NSNumber = 35.5`
  - `let intified: Int = n.intValue` // also `doubleValue`, `boolValue`, etc.
- Date
  - Value type used to find out the date and time right now or to store past or future dates
  - See also `Calendar`, `DateFormatter`, `DateComponents`
  - If you are displaying a date in your UI, there are localization ramifications, so check these out!
- Data
  - A value type "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 can have as many init methods in a class or struct as you want
  Each init will have different arguments, of course

  Callers use your init(s) by just using the name of your type and providing the args you want

```
var brain = CalculatorBrain()
var pendingBinaryOperation = PendingBinaryOperation(function: +, firstOperand: 23)
let textNumber = String(45.2)
let emptyString = String()
```

Initialization

- You get some init methods for "free"

  Free init() (i.e. an init with no arguments) given to all base classes.
  A base class has no superclass.

  If a struct has no initializers, it will get a default one with all properties as arguments

```
struct PendingBinaryOperation {
     var function: (Double,Double) -> Double
     var firstOperand: Double

     init(function: (Double,Double) -> Double, firstOperand: Double) {
          // we get this for free!
     }
}

// use of this free initializer somewhere else in our code
let pbo = PendingBinaryOperation(function: f, firstOperand: accumulator)
```

Initialization

- What can you do inside an init?
  - You can set any property's value, even those that already had default values
  - Constant properties (i.e. properties declared with let) can be set
  - You can call other init methods in your own class or struct 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 other 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 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
  - Whew!

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 ? after the word init, it returns an Optional

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

Example …

```
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
}
```

Any & AnyObject

- Any & AnyObject are special types
  - These types used to be commonly used for compatibility with old Objective-C APIs
  - But not so much anymore in iOS 10 since those old Objective-C APIs have been updated
  - Variables of type Any can hold something of any type (AnyObject holds classes only).
  - Swift is a strongly typed language, though, so you can't invoke a method on an Any.
  - You have to convert it into a concrete type first.
  - One of the beauties of Swift is its strong typing, so generally you want to avoid Any.

Any & AnyObject

- Where will you see it in iOS?

  Sometimes (rarely) it will be an argument to a function that can take different sorts of things.
  Here's a `UIViewController` method that includes a `sender` (which can be of any type).

  ```
  func prepare(for segue: UIStoryboardSegue, sender: Any?)
  ```

  The `sender` is the thing that caused this "segue" (i.e., a move to another MVC) to occur.
  The `sender` might be a `UIButton` or a `UITableViewCell` or some custom thing in your code.
  It's an Optional because it's okay for a segue to happen without a `sender` being specified.

- Where else will you see it?

  It could be used to contain a array of things with different types (e.g. `[AnyObject]`).
  But in Swift we'd almost certainly use an `Array` of an enum instead (like in `CalculatorBrain`).
  So we'd only do this to be backwards-compatible with some Objective-C API.

  You could also use it to return an object that you don't want the caller to know the type of.

  ```
  var cookie: Any
  ```

Any & AnyObject

- How do we use a variable of type Any?

  We can't usually use it directly (since we don't know what type it really is)
  Instead, we must convert it to another, known type

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

  We almost always use as? it with if let …

```
let unknown: Any = … // we can't send unknown a message because it's "typeless"
if let foo = unknown as? MyType {
    // foo is of type MyType in here
    // so we can invoke MyType methods or access MyType vars in foo
    // if unknown was not of type MyType, then we'll never get here
}
```

Casting

- By the way, casting with `as?` is not just for `Any` & `AnyObject`

  You can cast any type with `as?` into any other type that makes sense.
  Mostly this would be casting an object from one of its superclasses down to a subclass.
  But it could also be used to cast any type to a protocol it implements (more on this later).

  Example of "downcasting" from a superclass down to a subclass …

```
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`, since `vc` is typed as a `UIViewController`.

  However, if we cast `vc` to be a `CalculatorViewController`, then we can use it …

```
if let calcVC = vc as? CalculatorViewController {
    calcVC.displayValue = 3.1415 // this is okay
}
```

UserDefaults

- A very lightweight and limited database

  UserDefaults is essentially a very tiny database that persists between launchings of your app.
  Great for things like "settings" and such. Do not use it for anything big!

- What can you store there?

  You are limited in what you can store in UserDefaults: it only stores Property List data.
  A Property List is any combo of Array, Dictionary, String, Date, Data or a number (Int, etc.).
  This is an old Objective-C API with no type that represents all those, so this API uses Any.
  If this were a new, Swift-style API, it would almost certainly not use Any.
  (Likely there would be a protocol or some such that those types would implement.)

- What does the API look like?

  It's "core" functionality is simple. It just stores and retrieves Property Lists by key …

```
func set(Any?, forKey: String)         // the Any has to be a Property List (or crash)
func object(forKey: String) -> Any?  // the Any is guaranteed to be a Property List
```

UserDefaults

Reading and Writing

You don't usually create one of these databases with `UserDefaults()`.
Instead, you use the `static` (type) `var` called `standard` …

```
let defaults = UserDefaults.standard
```

<u>Setting</u> a value in the database is easy since the `set` method takes an `Any?`.

```
defaults.set(3.1415, forKey: "pi")    // 3.1415 is a Double which is a Property List type
defaults.set([1,2,3,4,5], forKey: "My Array") // Array and Int are both Property Lists
defaults.set(nil, forKey: "Some Setting") // removes any data at that key
```

You can pass anything as the first argument as long as it's a combo of Property List types.

`UserDefaults` also has convenience API for <u>getting</u> many of the Property List types.

```
func double(forKey: String) -> Double
func array(forKey: String) -> [Any]?                  // returns nil if non-Array at that key
func dictionary(forKey: String) -> [String:Any]? // note that keys in return are Strings
```

The `Any` in the returned values will, of course, be a Property List type.

UserDefaults

- Saving the database

  Your changes will be occasionally autosaved.

  But you can force them to be saved at any time with `synchronize` …

  `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)

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