Once More

Into the breach, yada, yada…

if statements

```
var todaysDate = 8
enum Days { case Monday, Tuesday, Wednesday, Thursday,
Friday }
let todaysDay = Days.Monday

if todaysDate == 13 {
    print("be careful out there")
}

if todaysDate == 13 && todaysDay == .Friday {
    print("more caution advised")
}
```

Adding some else-es

```
let temperatureInCelsius = 32
if temperatureInCelsius < 0 {
    print("baby it’s cold outside")
} else if temperatureInCelsius > 100 {
    print("you’re dead")
} else {
    print("who can tell? it’s celsius!")
}
```

switch

- The switch is perhaps the most significant difference between Swift and its predecessors thus far
- Incredibly powerful
- Not limited to integral values!

switch on string

```
let stringValue = "foo"
switch stringValue {
    case "foo":
        print("it’s foo, of course")
    case "bar":
        print("it’s bar, none")
    case "baz":
        print("I’ve never understood baz")
    default:
        print("A string I’ve never seen before!")
}
```

Subtle (or not-so-subtle) differences

- Switches must be exhaustive
- a default case will help with this in most instances
  - Xcode has been providing this as a warning for several years now
- There is no implicit fallthrough in a switch
- break is no longer required
- Switches can contain multiple values, or ranges

Tuples in a switch

- Each element may be tested against a different value or range of values, or _ to match any possible value

- The first matching case is always used when evaluating tuples

- Tuples may bind values to temporary values for use in switch statements

Tuples in a switch

```
let cartesianPoint = (1, -1)
switch cartesianPoint {
    case let (x,y) where x == y:
        print("\(x), \(y) : this point is on a line where x == y")
    case let (x,y) where x == -y:
        print("\(x), \(y) : this point is on a line where x == -y")
    case let (x,y):
        print("\(x), \(y) : this point is not on an interesting line"
}
```

Control Transfer

- continue
- break
- return
- fallthrough
- named flow control units are possible
- allows precision as to which loop is being broken, for example

Enumerations in Swift

- Groups of related values
- Provides type safety
- Not restricted to integral values

Enumeration sample

```
var directionForTravel = CompassPoint.South
directionForTravel = .East
switch directionForTravel {
    case .North:
        print("Visiting santa, eh?")
    case .South:
        print("Antarctic research, how exciting!")
    case .East:
        print("East of what, exactly?")
    case .West:
        print("Heading to the mothership?")
}
```

Associated Values in Enumerations

```
enum Instrument {
    case Guitar(Int)
    case Drum(Double, Double)
}
//This gives us an enumeration of a guitar where we are told the number of strings, or a dr
with its diameter and depth
var eightStringGuitar = Instrument.Guitar(8)
```

Raw Values

```
enum Planet: Int {
case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn,
Uranus, Neptune
}
let marsOrder = Planet.Mars.rawValue
```

An enum is so much more!

- Remember to think of an enum as a new type of data
- It's not a placeholder for a single value, it's closely related to a class or struct
- Move switching method into an enum, rather than outside.

Thinking in Swift

- Session 3:

- Collections and Optionals

Collection Types in Swift

- Array
- Dictionary
- Set

Array

- Ordered collection of values
- Type-bridged to NSArray
- Use all existing NSArray methods, if needed
- Any Objective-C method working on an NSArray will work with a Swift Array

Mutable Arrays

- Mutability is defined at creation time:
  - let vs var
  - reminder: immutable arrays are optimized for better performance in most cases
  - Applies to all collection types

Defining an array type

```
//Defined by inference:
var someIntegers = [3,6,9,11]

//Defined explicitly
var someOtherIntegers: [Int] = []
```

Creating an Array with specifications

```
var location = [Double](count: 3, repeatedValue: 0.0)
// location in this case is of type double, and contains
three values of 0.0
// [0.0, 0.0, 0.0]
```

Arrays are strongly typed!

```
var initialArray: [Int8] = [0,0,13]
initialArray.append(2048)
// this will FAIL, as 2048 will overflow an Int8
```

But what if I NEED an array of random junk?

```
var randomJunk: [Any] = [0,0,13]
randomJunk.append("cheese")
```

"Append" is kinda gross

Simpler syntax: `+= [<Element>]`

```
randomJunk += ["old tires"]
randomJunk += [true, "Nietzsche", 14.793]
```

Array Subscripting

```
var albums = ["Darkside", "The Healer", "Aja"]
albums[0] // refers to "Darkside"
albums[1] = "War" // replaces Lonnie Smith with U2
albums[3] = "Mandatory Fun"
// FAILURE! You cannot append an item to an array using
subscript notation
```

Array Insertion

```
var artists = ["Van Gogh", "DaVinci", "Pollack"]
artists.insert("Botticelli", atIndex: 1)
// places venus between VanGogh and DaVinci
```

Array Removal

- removeAtIndex(), removeLast()
- Removal methods return the removed item

```
var consoles = [“Switch", "PS4", "Xbone"]
consoles.remove(at:0)
// removes the first item
consoles.removeLast()
// only the PS4 remains.
```

Array Iteration

- for-in iteration, as in other languages
- to get an index, use enumerate instead

```
for artist in artists {
    print(artist)
}

for (index, value) in artists.enumerate() {
    print("Artist \(index) is named \(value)")
}
```

Sets in Swift

- type-bridged to NSSet
- Distinct values of a single type with no defined ordering
- "distinct" means an item only appears once
- No shorthand form for creation

Sets in Swift

```
var guitarMakers: Set = ["Fender", "Gibson", "D’Angelico"]
// Add elements with insert
guitarMakers.insert("Rickenbacker")
// Check for any contents with isEmpty
if guitarMakers.isEmpty {
    print("what a sad world that would be")
}
```

Sets in Swift

```
// remove items using "remove"
guitarMakers.remove("D’Angelico")
// check for a particular item using "contains"
if guitarMakers.contains("Paul Reed Smith") {
    print("some folks are happy about that")
}
```

Set Iteration in Swift

- iterate using the for-in loop, just as with Arrays
- use the global sort function to iterate the set in an ordered version

```
for maker in guitarMakers {
    print(maker)
}
for maker in guitarMakers.sort() {
    print(maker)
}
```

Set operations in Swift

- Set operations are where sets become exciting in Swift

- sets can be constructed and compared using functions that are named after the mathematical foundations of set theory!

Set operations in Swift

- Set construction:
- union
- subtract
- intersect
- exclusiveOr

Set operations in Swift

- Set comparison:
- isEqual (==)
- isSubsetOf
- isSupersetOf
- isStrictSubsetOf
- isStrictSupersetOf
- isDisjointWith

Dictionaries in Swift

- Like Arrays and Sets, Dictionaries are strongly typed
- Dictionaries are type-bridged to NSDictionary
- Formal type is Dictionary<Key, Value> , but preferred notation is [Key: Value]
- You may note that that formal type is specified using generic notation

Dictionaries in Swift

```
var namesOfDoctors = [Int: String]()
// creates a dictionary where an Int is the index, and
values are Strings
namesOfDoctors[12] = "Peter"
```

Dictionary Iteration

- Dictionaries are unordered; use the global Sort method on keys or values if you need ordering

```
let airportInfo = ["DFW":"DallasFortWorth",
"DIA":"DenverInternationalAirport"]
for (airportCode, airportName) in airportInfo {
    // ...
}
for key in airportInfo.keys.sort() {
    print(airportInfo[key])
}
```

Optionals in Swift

- Optionals are a key component in Swift's safety
- Use optionals when a value may be absent
- Do not use voodoo unwrapping of just banging away on ! and ? until your code compiles
- Instead, seek to truly understand your optionals.
- Otherwise, you're looking at a code smell
- Optionals are going to save you from many hard-to-find bugs

Declaring a variable to be optional

```
var favoriteTVShow: String? = "ACL"
favoriteTVShow = nil
```

- Now there's no value for favorite TV show. This does NOT mean that you can call string operations on this nil value!

- This is NOT the same as a null value in other languages

Forced Unwrapping using an "if" statement

```
if favoriteTVShow != nil {
}
// Note that you still can’t use string operations inside that function body!
if favoriteTVShow != nil {
    // TODO use the variable here.
}
// NOTE if favoriteTVShow were nil, and untrapped by the if statement, then a runtime error would result
```

Optional binding

- The simplest form of handling optionals

```
if let favorite = favoriteTVShow {
    // TODO use the variable here.
}
// This is now safe and clear
```

Implicitly Unwrapped Optionals

- Used when an optional will always have a value, when it has been set.

```
let mightBeAString: String? = "Giants"
let forcedString: String = mightBeAString!
// From here on out, you can use forcedString without using
a ! every time.
// Will cause a runtime error if used improperly
```

How to read optionals:

- I like to read code out loud. Optionals are read as such:

Guard

- The guard statement helps us avoid rightward drift when unwrapping optionals
- It's very similar to if let, except that the constant remains in scope beyond the guard statement
- A guard statement must contain an else, and in the else you must return or throw an error
- This can be thought of as a "bouncer" pattern

Example of rightward drift

```
func functionUsingOptionals(optionalOne: Int?, optionalTwo:
Int?, optionalThree: Int?) {
    if let one = optionalOne {
        if let two = optionalTwo {
            if let three = optionalThree {
                print("\(one)\(two)\(three)")
            }
        }
    }
}
```

Clearing rightward drift using guard

```
func functionUsingOptionalsAndGuard(optionalOne: Int?,
optionalTwo: Int?, optionalThree: Int?) {
    guard let one = optionalOne else {return}
    guard let two = optionalTwo else {return}
    guard let three = optionalThree else {return}
    print("\(one)\(two)\(three)")
}
```

Cleanup using defer

- What if you allocate memory or another resource, but are going to exit early with a guard?

- Defer can take care of this cleanup for you.

```
func functionUsingOptionalsAndGuard(optionalOne: Int?,
optionalTwo: Int?, optionalThree: Int?) {
    guard let one = optionalOne else {return}
    defer{print("this prints at exit")}
    guard let two = optionalTwo else {return}
    guard let three = optionalThree else {return}
    print("\(one)\(two)\(three)")
}
functionUsingOptionalsAndGuard(1, optionalTwo: 2,
optionalThree: 3)
```

optional chaining

- What if a function returns an optional structure containing another optional structure containing an optional Int?

- No problem-- Swift has this covered.

```
let realValue =
computeStructure()?.internalStructure?.value?
```

One last word on optionals:

- Don't fall into the trap of voodoo-unwrapping your optionals. Understand them, and use them appropriately.

- Optionals are going to take some getting used to.

- Optionals are great for resolving many bugs that we commonly see, so fall in love with them!