TODAY

• More protocols
• More enums
• Error handling
• Vestiges of Objective-C
• Persistence
MORE PROTOCOLS

Dictionaries require keys to be *unique*
- hashes are most likely unique
- equality testing

Dictionary keys required to implement `Hashable`:

```swift
protocol Hashable: Equatable {
    var hashValue: Int { get }
}
```

```swift
protocol Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool
}
```

MORE ENUMS

- Already discussed
  - `rawValues`
  - `CaseIterable` protocol, `allCases()`
  - `CustomStringConvertible` protocol
    - `var description: String { return …. }`
MORE ENUMS

class Card : CustomStringConvertible {
    var color : Color
    var shape : Shape
    var number : Number
    var shade : Shade

    init(c: Color, s: Shape, d: Shade, n: Number) {
        color = c
        number = n
        shape = s
        shade = d
    }

    var description: String { return "\(number) \(shade) \(shape.rawValue) \(color)" }
}

enum Color : String, CaseIterable {
    case green
    case red
    case blue
}

enum Shade : String, CaseIterable {
    case solid
    case clear
    case partial
}

enum Shape : String, CaseIterable {
    case square = "■"
    case triangle = "▲"
    case oval = "●"
}

enum Number : Int, CaseIterable {
    case one = 1, two, three
}

MORE ENUMS

• Associated Data! Each case can have associated state.

eenum Motorcycle {
    case tourer(numCylinders: Int)
    case cruiser(loudness: Int, type: ObnoxiousCycle)
    case crotchRocket(String, topSpeed: Int)
}

eenum ObnoxiousCycle {
    case harley
    case canam
}
MORE ENUMS

• Setting enum values:
  • Must specify associated values
  • Type inference works on one side or the other, not both.

```swift
let mine = Motorcycle.tourer(numCylinders: 4)
let yours : Motorcycle = .crotchRocket("I will soon die",
  topSpeed: 164)
let theirs = Motorcycle.cruiser // fail
```

MORE ENUMS

• Associated data can be ignored when reading

```swift
let theHonda = Motorcycle.tourer(numCylinders: 4)

switch (theHonda) {
  case .cruiser:
    print("Get your motor running... ")
  case .tourer:
    print("Head out on the highway...")
  case .crotchRocket:
    print("Looking for adventure... ")
}
```
MORE ENUMS

• Associated values can be read, renamed etc in switch:
  • cruiser type renamed to noisemaker
  • crotchRocket’s unnamed data labeled catchPhrase

```swift
switch theHonda {
  case .tourer(let numCylinders):
    print("\(numCylinders) is better than none")
  case .cruiser(let noisemaker):
    print("My \(noisemaker) is louder than yours")
  case .crotchRocket(let catchPhrase, let speedGoal):
    print("zzzzZOOOOmmmm: \(catchPhrase) at \(speedGoal)")
}
```

ERROR HANDLING

• propagate upwards
  • by calling function also throwing
• handle using do-catch
• concert to optional value
• assert that it just won’t happen
**ERROR HANDLING**

- you will know because:
  ```swift
  func foo() throws -> Int { …
  ```
- Calls to these funcs must be caught:
  ```swift
  do {
      try func foo()
  } catch let error {
      // something that knows the Error protocol
  }
  ```
- If you are certain they won’t err, you can force:
  ```swift
  try! foo()     // crash on throw…
  ```
- Or conditionally try, getting an optional:
  ```swift
  let bar = try? foo()  // bar is Int?
  ```

**VESTIGES OF OBJECTIVE-C**

- Any
  - can hold any type of object
- AnyObject
  - any class

Swift is strongly typed, so you can use directly.

Need a concrete type…
VESTIGES OF OBJECTIVE-C

• Where is it used?
  
  ```
  let attributes: [NSAttributedStringKey: Any] = ...
  ```

• Attributes can be different things: UIFont, UIColor, etc
  
  ```
  func prepare(for segue: UIStoryboardSegue, sender: Any?)
  ```

• sender (UIButton ..) caused the “segue”

• optional because sender need not be specified

Might use `enum` with `associated data` in Swift.

VESTIGES OF OBJECTIVE-C

• How to use a var of type Any?
  
  Must be converted, as don’t know what it is…

  Conversion with Swift’s `as?` keyword, makes optional.
  
  ```
  let foo: Any = [0, 1, 2]
  print(foo.count())
  if let bar = foo as? [Int] { …
  ```

  Can also check it can be converted with `is`.
  
  ```
  if foo is [Int] { …
  ```
VESTIGES OF OBJECTIVE-C

```swift
var things = [Any]()

    things.append(0)
    things.append(0.0)
    things.append(42)
    things.append(3.14159)
    things.append("hello")
    things.append((3.0, 5.0))

puts("\(things)")

for thing in things {
    switch thing {
        case 0 as Int:
            print("zero as an Int")
        case 0 as Double:
            print("zero as a Double")
        case let someInt as Int:
            print("an integer value of \(someInt)")
        case let someDouble as Double where someDouble > 0:
            print("a positive double value of \(someDouble)")
        case is Double:
            print("some other double value that I don't want to print")
        case let someString as String:
            print("a string value of \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("an \(x), \(y) point at \((x), \(y))")
        case let stringConverter as (String) -> String:
            print(stringConverter("Michael"))
        default:
            print("something else")
    }
}```

CASTING

- Casting also for:
  - casting from `superclass` to `subclass`
  - object to `protocol` it implements

```swift
class Vehicle {
    var numWheels : Int = 4
}

class Car : Vehicle {
    var maker: String = "Honda"
}

var mover : Vehicle = Car()

if let wheels = mover as? Car {
    print(wheels.maker, wheels.numWheels)
}

Honda 4
PERSISTANCE

- UserDefaults
- PropertyLists
- Archiving and Codable
- Filesystem
- Core Data
- Cloud Kit

USERDEFAULTS

- Essentially a persistent dictionary
  - Can only store Property List: any combination of Array, Dictionary, String, Date, Data, or numbers
  - Small amounts of data
    ```swift
    func set<Any?, forKey: String> // must be property list
    func object(forKey: String) -> Any?
    ```
  - Keys are Strings
    ```swift
    let defaults = UserDefaults.standard
    defaults.set(42, forKey: "the answer")
    defaults.set(["The Como", "The LHT", "The Ogre", "The Honda"], forKey: "bikes")
    ```
  - Many convenience funcs:
    ```swift
    func array(forKey: String) -> [Any]?
    func int(forKey: String) -> Int
    ...```
USER DEFAULTS

- Persistence
  - Changes auto-saved
  - Or:

```swift
if !defaults.synchronize() {  // what to do?
```

ARCHIVING

- Old: NSCoder
  - store all data in coder’s dictionary
  - each object type needs custom serialization code

- New: Codable
  - serialized to JSON or Property Lists
  - most types already supported, easy to extend
ARCHIVING

Given Codable data, convert to JSON or Property List:

```swift
let foo: MyObj = ...
let jsonData: Data? = try? JSONEncoder().encode(foo)
```

Encoding throws (next slide). Convert to string:

```swift
let jsonString = String(data: jsonData, encoding: .utf8)
```

Decode:

```swift
let decoder = JSONDecoder()
decoder.dataDecodingStrategy = .secondsSince1970

The thrown thing is an `enum` w/ associated values

```swift
do {
    let obj = try JSONDecoder().decode(MyObj, from: jsonData!)
    // no throw, do stuff
} catch DecodingError.ValueNotFound(let type, let context) {
    print(type, context)
}
```

ARCHIVING

• How to make your types Codable? Easy…

```swift
struct peteClass : Codable {
    var howMany: Int
    var title: String
    var dept: String
}
```

• as long as all your properties also Codable.

• JSON output:

```swift
{
    "howMany": 156,
    "title": "iOS Programming",
    "dept": "Computer Science"
}
```
ARCHIVING

- Guide the process through a private enum `CodingKeys`
  - change names
  - omit properties

```swift
private enum CodingKeys: String, CodingKey {
    case howMany = "how_many" // changed
    case title = "title" // unchanged
    // howMany omitted
}
```

So I have a JSON string (or a Property List).

Now what?