Swift libraries

COSC346

Standard Library

- Programming languages usually come with a standard library that provides certain basic functions (such as I/O functions for instance)
- OO languages additionally provide useful classes

Swift Standard Library Reference:

https://developer.apple.com/documentation/swift

Structures versus classes

In Swift **both** structures and classes can:

- Have properties
- Have methods
- Define methods for handling subscripts
- Define initialisers
- Be extended
- Conform to protocols

Structures versus classes

What that classes have that structures do not:

- Inheritance
- Type casting
- Deinitialisers
- Reference counting
  - Recall that structures are value types: they're always copied

Structures

- Int
  - Library gives bigEndian method
- Double
- Character
- String
  - Library gives uppercased method

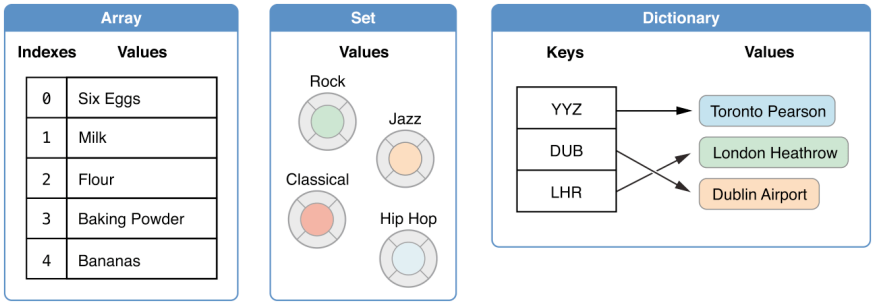
```
struct String {
    init()
}

extension String {
    public typealias Index = String.CharacterView.Index
    /// The position of the first `Character` in `self.characters` if
    /// `self` is non-empty; identical to `endIndex` otherwise.
    public var startIndex: Index { get }
    /// The "past the end" position in `self.characters`.
    ///
    /// `endIndex` is not a valid argument to `subscript`, and is always
    /// reachable from `startIndex` by zero or more applications of
    /// `successor()`.
    public var endIndex: Index { get }
    /// Access the `Character` at `position`.
    ///
    /// - Requires: `position` is a valid position in `self.characters`
    ///   and `position != endIndex`.
    public subscript (i: Index) -> Character { get }
}

extension String {
    /// A `String`'s collection of `Character`s ([extended grapheme
    /// clusters](http://www.unicode.org/glossary/#extended_grapheme_cluster))
    /// elements.
    public struct CharacterView {
        /// Create a view of the `Character`s in `text`.
        public init(_ text: String)
    }
    /// A collection of `Characters` representing the `String`'s
    /// [extended grapheme
    /// clusters](http://www.unicode.org/glossary/#extended_grapheme_cluster).
    public var characters: String.CharacterView { get }
    /// Efficiently mutate `self` by applying `body` to its `characters`.
    ///
    /// - Warning: Do not rely on anything about `self` (the `String`
    ///   that is the target of this method) during the execution of
    ///   `body`: it may not appear to have its correct value.  Instead,
    ///   use only the `String.CharacterView` argument to `body`.
    public mutating func withMutableCharacters<R>(body:
                              (inout String.CharacterView) -> R) -> R
    /// Construct the `String` corresponding to the given sequence of
    /// Unicode scalars.
    public init(_ characters: String.CharacterView)
}
```

Standard Library

Swift

Structures

- Array
- Dictionary
- Set




```
public struct Array<Element> : CollectionType, MutableCollectionType,
_DestructorSafeContainer {
    /// Always zero, which is the index of the first element when non-empty.
    public var startIndex: Int { get }
    /// A "past-the-end" element index; the successor of the last valid
    /// subscript argument.
    public var endIndex: Int { get }
    /// Access the `index`th element. Reading is O(1).  Writing is
    /// O(1) unless `self`'s storage is shared with another live array;
O(`count`) if `self` does not wrap a bridged `NSArray`; otherwise the efficiency
is unspecified..
    public subscript (index: Int) -> Element
    /// Access the elements indicated by the given half-open `subRange`.
    ///
    /// - Complexity: O(1).
    public subscript (subRange: Range<Int>) -> ArraySlice<Element>
}
```

Protocols


```
/// The common requirements for types that support integer arithmetic.
protocol IntegerArithmeticType : _IntegerArithmeticType, Comparable {

  /// Add `lhs` and `rhs`, returning a result and trapping in case of
  /// arithmetic overflow (except in -Ounchecked builds).
  func +(lhs: Self, rhs: Self) -> Self

  /// Subtract `lhs` and `rhs`, returning a result and trapping in case of
  /// arithmetic overflow (except in -Ounchecked builds).
  func -(lhs: Self, rhs: Self) -> Self

  /// Multiply `lhs` and `rhs`, returning a result and trapping in case of
  /// arithmetic overflow (except in -Ounchecked builds).
  func *(lhs: Self, rhs: Self) -> Self

  /// Divide `lhs` and `rhs`, returning a result and trapping in case of
  /// arithmetic overflow (except in -Ounchecked builds).
  func /(lhs: Self, rhs: Self) -> Self

  /// Divide `lhs` and `rhs`, returning the remainder and trapping in case of
  /// arithmetic overflow (except in -Ounchecked builds).
  func %(lhs: Self, rhs: Self) -> Self

  /// Explicitly convert to `IntMax`, trapping on overflow (except in
  /// -Ounchecked builds).
  func toIntMax() -> IntMax
}
```

- AnyObject
- CustomStringConvertible
  - Textual representation of any type to print to an output stream
- Comparable
  - Conforming types can be compared using relational operators: a strict total order
- Equatable
  - Conforming types can be compared for value equality: operators == and !=
- Hashable
  - Conforming types provide an integer hashValue: usable as Dictionary keys
- FloatingPointType
  - A set of common requirements for Swift's floating point types
- IntegerArithmeticType
  - The common requirements for types that support integer arithmetic

Protocols

- Code here shows extensions of Floats:
  - FloatingPointType
- Provides infinity, NaN, etc.

```
/// A single-precision floating-point value type.
public struct Float {
    /// Create an instance initialized to zero.
    public init()
    public init(_bits v: Builtin.FPIEEE32)
    /// Create an instance initialized to `value`.
    public init(_ value: Float)
}

extension Float : CustomStringConvertible {
    /// A textual representation of `self`.
    public var description: String { get }
}

extension Float : FloatingPointType {
    /// The positive infinity.
    public static var infinity: Float { get }
    /// A quiet NaN.
    public static var NaN: Float { get }
    /// A quiet NaN.
    public static var quietNaN: Float { get }
    /// `true` iff `self` is negative.
    public var isSignMinus: Bool { get }
    /// `true` iff `self` is normal (not zero,
subnormal, infinity, or
    /// NaN).
    public var isNormal: Bool { get }
    /// `true` iff `self` is zero, subnormal, or
normal (not infinity
    /// or NaN).
    public var isFinite: Bool { get }
    /// `true` iff `self` is +0.0 or -0.0.
    public var isZero: Bool { get }
    /// `true` iff `self` is subnormal.
    public var isSubnormal: Bool { get }
    /// `true` iff `self` is infinity.
    public var isInfinite: Bool { get }
    /// `true` iff `self` is NaN.
    public var isNaN: Bool { get }
    /// `true` iff `self` is a signaling NaN.
    public var isSignaling: Bool { get }
}
```

Functions

https://developer.apple.com/documentation/swift

```
public func print(_ items: Any...,
            separator: String = default,
            terminator: String = default)


public func assert(_ condition: @autoclosure () -> Bool,
                   _ message: @autoclosure () -> String = default,
                   file: StaticString = #file,
                   line: UInt = #line)
```

- Traditional C-style assert with an optional message.
- Previously Swift 2.0 provided mechanisms such as:

```
public func unsafeAddressOf(object: AnyObject) -> UnsafePointer<Void>
```

- Swift 4.0 (and 3.0) instead uses structures such as:

```
withUnsafePointer(to: &str) {
    print("str value \(str) has address: \($0)")
}
```

Operators

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/BasicOperators.html

- `prefix func !(_ a: Bool) -> Bool`
- `func *=(inout _ lhs: Float, _ rhs: Float)`
- `func ==(_ lhs: String, _ rhs: String) -> Bool`

Foundation Framework

NS

Swift

- The standard library for Objective-C was the Foundation Framework (despite the name, it is a library rather than a framework)
- Swift is compatible with Objective-C, hence Swift programs can use the Foundation library
- Naming convention—NS prefix (for NextStep)

- Foundation Framework Reference:
- https://developer.apple.com/documentation/foundation

Data types

NS

Swift

https://developer.apple.com/documentation/foundation/foundation_data_types

- `NSInteger`—typealias to `Int` (64-bit or 32-bit, depending on the platform)
- `NSStringEncoding`—type representing string-encoding values (*e.g.*, NSUTF8StringEncoding)
- `NSTimeInterval`—typealias to `Double`, used to specify a time interval, in seconds
- `NSRange`—a structure used to describe a portion of a series—such as characters in a string or objects in an `NSArray` object

Classes

NS

Swift

https://developer.apple.com/documentation/foundation

- `NSObject`—the root class of most Objective-C class hierarchies (other Foundation classes inherit from `NSObject`)
- `NSCoder`—abstract class declares the interface used by concrete subclasses to transfer objects and other values between memory and some other format
- `NSMutableArray`—a modifiable array of objects
- `NSArray`—subclass of `NSMutableArray` for managing ordered collections of objects
- `NSString`—immutable strings
- `NSData`—class for encapsulation of buffers—contiguous memory regions
- `NSURLConnection`—lets you load the contents of a URL by providing a URL request object

Protocols

https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/ObjC_classic/index.html#protocols

- `NSObjectProtocol`—groups methods that are fundamental to all Objective-C objects
- `NSCoding`—declares the two methods that a class must implement so that instances of that class can be encoded and decoded
- `NSCopying`—declares a method for providing functional copies of an object
- `URLSessionDataDelegate`—describes methods that should be implemented by the delegate for an instance of the `NSURLSession` class

```
public protocol NSCopying {

    public func copy(with: NSZone? = nil) -> AnyObject
}
```

Core Graphics

- There are libraries within Foundation Framework, such as the Core Graphics library
- Naming convention—`CG` prefix (for CoreGraphics)

Core Graphics Reference:

https://developer.apple.com/documentation/coregraphics

Structures

These structures are used in the AppKit API for positioning graphical elements in the application window

- **CGPoint**—represents a point in 2D space
- **CGSize**—size of a rectangular object (like a window)
- **CGRect**—represents a rectangle

```
struct CGPoint {
  var x: CGFloat
  var y: CGFloat
  init()
  init(x x: CGFloat, y y: CGFloat)
}
```

```
struct CGSize {
  var width: CGFloat
  var height: CGFloat
  init()
  init(width: CGFloat, height: CGFloat)
}
```

```
struct CGRect {
  var origin: CGPoint
  var size: CGSize
  init()
  init(origin: CGPoint, size: CGSize)
}
```

Serialisation

- **Serialisation** is the process of converting an object's state (its instance variables) into a data stream, *e.g.*,
  - Stream of bytes
  - JSON format
  - XML document
- Useful for saving the object state to a file / sending it
- Process of reconstructing an object from the byte stream is called **deserialisation**
- Only internal data gets serialised (not the methods)
  - The program performing deserialisation must know about class implementation in order to reconstruct the object

Serialisation—text format (property lists)

- NSData, NSString, NSArray, NSDictionary, NSDate, and NSNumber provide the `writeToFile` method, which serialises them and writes their state to an XML file

```
var array: [String] = ["item1", "item2"]
let arrayToSave: NSArray = array as NSArray

arrayToSave.writeToFile(toFile: "text.plist", atomically: true)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <string>item1</string>
        <string>item2</string>
</array>
</plist>
```

```
let restoredArray: NSMutableArray? = NSMutableArray(contentsOfFile: "text.txt")

if let newArray = restoredArray {
    print(newArray as NSArray)
}
```

Serialisation—binary format (archiving)

NS

Swift

- Archiving is a more flexible and powerful type of serialisation
  - It allows encoding of user defined objects

- `NSArchiver` class provides a static method that serialises object to a file in binary format
- `NSUnarchiver` class provides a static method that creates object instance from binary file
- The archived object and all its composites must adopt `NSCoding` protocol

Serialisation—binary format (archiving)

- To support archiving, classes must conform to `NSCoding` protocol and implement the coding and decoding methods

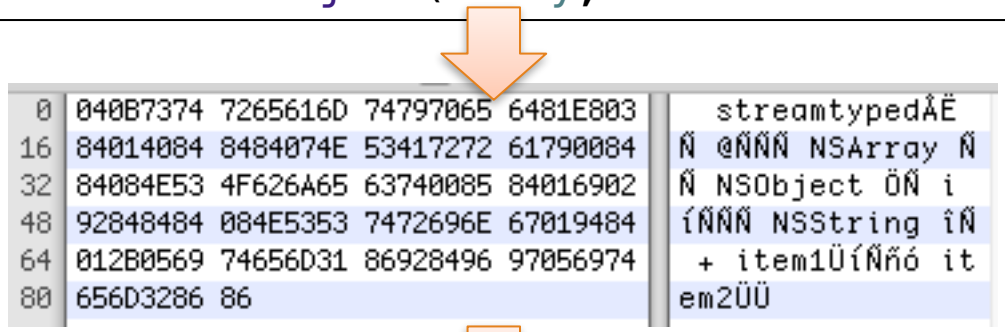
```
public protocol NSCoding {

    public func encodeWithCoder(aCoder: NSCoder)
    public init?(coder aDecoder: NSCoder) // NS_DESIGNATED_INITIALIZER
}
```

- `encodeWithCoder`—for serialising/encoding
- `init?(coder: NSCoder)`—for deserialising/ decoding
- (`NSCoder` is a class; `NSCoding` is a protocol)

Serialisation—binary format (archiving)

- NSString, NSArray, NSDictionary, NSSet, NSDate, NSNumber and NSData already conform to the NSCoding protocol

```
var array: [String] = ["item1", "item2"]

NSArchiver.archiveRootObject(array, toFile: "text.bin")
```




```
let restoredArray = NSUnarchiver.unarchiveObject(withFile: "text.bin") as!
Array<String>?
if let newArray = restoredArray {
    print(newArray)
}
```

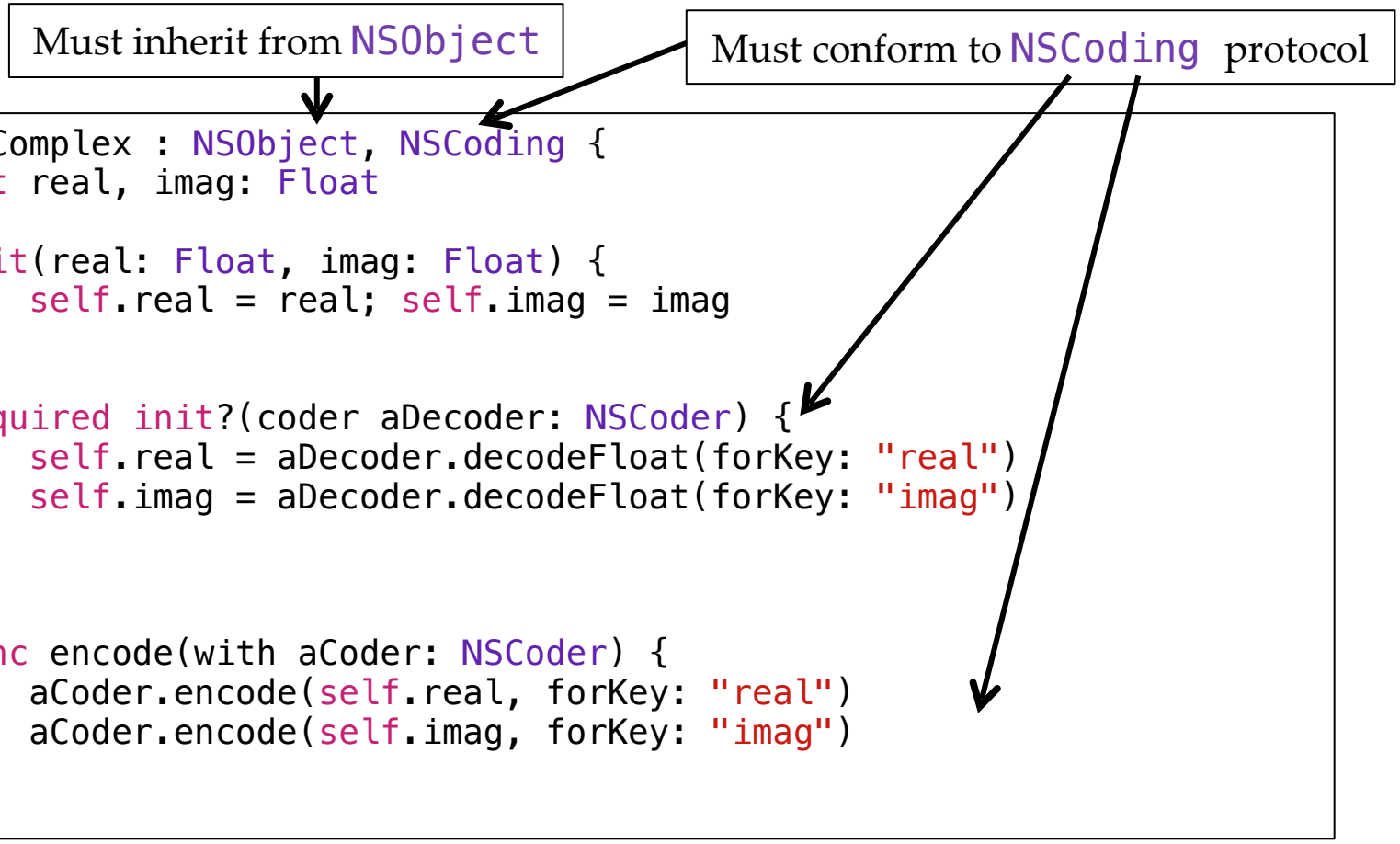
Serialisation—binary format (keyed archiving)

- The variables stored in the archive are associated with a key specified during the encoding
- Decoding does not need to follow the encoding order, since data can be retrieved by specifying its key
- `NSCoder` class provides set of functions for encoding primitive data types
  - ecode<Type>:forKey:
  - decode<Type>ForKey:

- `NSKeyedArchiver` and `NSKeyedUnarchiver` pass a reference to their coder/decoder when calling the `NSCoding` methods

Serialisation—binary format (archiving)

- Custom class archiving

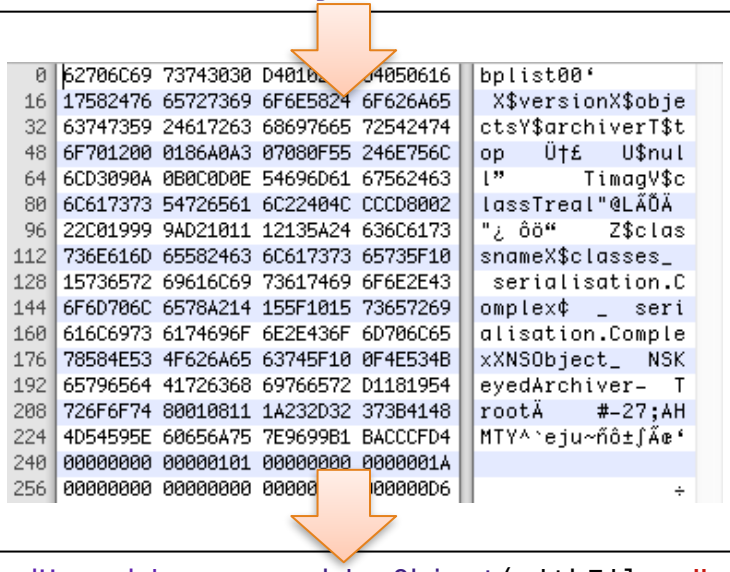



```
class Complex : NSObject, NSCoding {
    let real, imag: Float

    init(real: Float, imag: Float) {
        self.real = real; self.imag = imag
    }

    required init?(coder aDecoder: NSCoder) {
        self.real = aDecoder.decodeFloat(forKey: "real")
        self.imag = aDecoder.decodeFloat(forKey: "imag")
    }


    func encode(with aCoder: NSCoder) {
        aCoder.encode(self.real, forKey: "real")
        aCoder.encode(self.imag, forKey: "imag")
    }
}
```

Serialisation—binary format (archiving)

- NSKeyArchiver calls object's encode method when archiving it
- NSKeyUnarchiver calls object's init?(coder:) method to unarchive

```
var x = Complex(real: -2.4, imag: 3.2)

NSKeyedArchiver.archiveRootObject (x, toFile: "myarchive.bin")
```




```
let restoredX = NSKeyedUnarchiver.unarchiveObject(withFile: "myarchive.bin") as! Complex?

if let y = restoredX {
    print("xRestored=\(y.real)+\(y.imag)i")
}
```

Serialisation - Codable

Swift

- Introduced in Swift 4
- Combines Encodable and Decodable protocols

```
class Complex: Codable{
    let real, imag: Float
    init(real: Float, imag: Float){
        self.real = real
        self.imag = imag
    }
}
var c = Complex(real: -2.4 ,imag: 3.2)

let encoder = PropertyListEncoder()
encoder.outputFormat = .xml

var d = try encoder.encode(c)
print(String(data: d, encoding: .utf8)!)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
     "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
          <key>imag</key>
          <real>3.2000000476837158</real>
          <key>real</key>
          <real>-2.4000000953674316</real>
</dict>
</plist>
```

Serialisation - Codable

Swift

```
class Complex: Codable{
    let real, imag: Float
    init(real: Float, imag: Float){
        self.real = real
        self.imag = imag
    }
    enum CodingKeys: String, CodingKey {
        case real = "r"
        case imag = "i"
    }
}
var c = Complex(real: -2.4 ,imag: 3.2)

let encoder = PropertyListEncoder()
encoder.outputFormat = .xml

var d = try encoder.encode(c)
print(String(data: d, encoding: .utf8)!)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
     "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
          <key>i</key>
          <real>3.2000000476837158</real>
          <key>r</key>
          <real>-2.4000000953674316</real>
</dict>
</plist>
```

Design Pattern - Command

- Behavioural
- Encapsulate all information needed to perform an action
- Now or later (defer command execution)

- e.g. Copying text
  - GUI Button
  - Mouse right-click
  - Keyboard

CopyTextCommand
- getHighlightedRegion()
- saveToBuffer()

Command Example

Toolmaker

```
protocol MMCommandHandler{
    static func handle(_ params: [String], last: MMResultSet) throws -> MMResultSet;
}

class HelpCommandHandler: MMCommandHandler{
    static func handle(_ params: [String], last: MMResultSet) throws -> MMResultSet{
        print("quit - quit the program)
        return MMResultSet()
    }
}

class QuitCommandHandler : MMCommandHandler{
    static func handle(_ params: [String], last: MMResultSet) throws -> MMResultSet {
        exit(0)
    }
}
```

Builder

```
switch(command){
case "help":
    last = try HelpCommandHandler.handle(parts, last:last)
    break
case "quit":
    last = try QuitCommandHandler.handle(parts, last:last)
    continue
}
```

Command in the real world?

Summary?