# The Swift Programming Language

and Objective C

# Swift

- Swift is completely new but borrows ideas from things like Scala, Rust, et al.
    - Its what Chris Lattner likes
- Is a hybrid obect-oriented/functional language
- Is compiled, but very quickly
- Major design goal was **complete** interoperability with ObjectiveC.  Each can call or be derived from the other
    - Implications for naming conventions
    - Not everything in Swift can get over the fence into ObjC
- All current iOS and MacOS libraries are accessible from Swift

# Swift Cont'd

- Has Heap and STACK allocation
    - Different types are used for different memory allocation models
- Functions are 1st class objects
- There is no single base class
- Does NOT offer reflection (well not much, really), relies much more heavily on static typing and generics
- Does not ***require*** semi-colons

# Setting up Playgrounds

- Using code you share with a projects requires you to use a workspace
- Workspace defines search path for Frameworks
- Workspace must build items you want to consume in your Playground into a Framework
- Each Framework produces its own module
- **Be sure to make things you want to consume have access control of ‘public’**

# Playground Limitations

- Playgrounds suck at dealing with asynchronous processes
- Means of supporting asynchronous process is just to wait a specified amount of time at the end of evaluation
- No mechanism to wait, then proceed
  - import XCPlayground
  - XCPSetExecutionShouldContinueIndefinitely()

# Mutability

- Mutability/Immutability decision built into Swift language, in ObjC requires the type (class) to handle
- Two keywords: let and var
  - let means Immutable
  - var means Mutable

# Best Practices

- Immutable (i.e. let) should be the default
  - Allows compiler optimization
  - Almost required for reasoning about threads
- Needs to be coordinated with containers depending on Value or Reference type

# Simple Swift Data Types to be familiar with

- String
- Double/Float/Int/Bool
- Array
- Dictionary

# String/Int/Float/Double/Bool

- largely what you expect
- Are _internally_ Immutable
- Can be instantiated from a literal with “” for strings or just the number for the others
- Full set of functions available
- Don’t need to be unboxed for use

# Array/Dictionary

- Again largely what you expect
- Array can be instantiated with [] literal or declared with [T] or Array<T> (NB this is our first look at generics <T> means “of type T”)
- Dictionary can be instantiated with [ : ] or declared with [T:U] or Dictionary<T,U>

# Control Flow

- if / else / else if
    - VERY IMPORTANT: “if let” construct to be discussed later
- while / do { } while
    - do is also the construct for exception handling
- yawn…
- Key thing is that parens are not needed as in C/Java/C# etc

# Control Transfer

- As expected
  - break
  - continue
- New
  - fallthrough - used in switch to do what C did
  - :label - used to break/continue from within nested loops

# For-in / For

- The “for - in” loop performs a set of statements for each item in a range, sequence, collection, or progression.

- The for loop performs a set of statements until a specific condition is met, typically by incrementing a counter each time the loop ends.

- C-style for loop is doomed

Excerpt From: Apple Inc. “The Swift Programming Language.”

# Sequences

```
class FibonacciSequence : SequenceType {

    func generate() -> GeneratorOf<Int> {
        var current = 0, next = 1

        return GeneratorOf<Int> {
            var ret = current
            current = next
            next = next + ret
            return ret
        }
    }
}

let fibs = FibonacciSequence().generate()
for i in fibs {
    println(i)
    if i > 9 {
        break
    }
}
```

# Switch

- Must be exhaustive
- Does not “fall through”
- Use ‘where’ clause to do complex matching
- Use ‘let’ similarly to if let

# nil

- Shared with Objective C
- Pretty much what you expect, it’s nothing
- In ObjC [nil someMethod] ALWAYS returns nil
- In swift nil.someMethod() throws an exception
- variables which can be nil have a special data type (Optional<T>) and syntax (? and !)

# Optional Types

- Toughest thing for beginners to get used to, especially if they come from ObjC
- If an object can be missing a value, it must be declared as Optional<T> in which case it's value can be nil or an object of type T
- This is so common that it has a special syntax which gets overloaded (? and !)
- And additional syntax which you end up using all over the place (?? and if let)

# ? and ! Syntax

- var anOptionalString:String? = “x”
- var anImplicitlyUnwrappedString:String! = “y”
- var aRealString:String = “z”
- Usage
  - aRealString = anOptionalString!
  - aRealString = anImplicitlyUnwrappedString
  - aRealString = anOptionalString!.extend(“yz”)
  - anImplicitlyUnwrappedString = anOptionalString?

```swift
var aString = "A String"
var optionalIntValue = aString.toInt()
//aString = nil  // doesn't work, can't be nil

var optionalString: String? = "Hello"
optionalString == nil
optionalString = optionalString! + "... Hello..."
optionalIntValue = optionalString?.toInt()
optionalIntValue = optionalString!.toInt()
//var intValue:Int = optionalString?.toInt() // doesn't work

var otherString = optionalString ?? "A completely different string"

optionalString = nil // does work, can be nil
optionalString == nil
optionalIntValue = optionalString?.toInt()
// optionalIntValue = optionalString!.toInt()  // doesn't work

otherString = optionalString ?? "A completely different string"

var implicitlyUnwrappedString:String! = "This is implicit"
implicitlyUnwrappedString
aString = implicitlyUnwrappedString
implicitlyUnwrappedString = nil
// aString = implicitlyUnwrappedString  // doesn't work

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

```swift
var aString = "A String"
var optionalIntValue = aString.toInt()
//aString = nil  // doesn't work, can't be nil

var optionalString: String? = "Hello"
optionalString == nil
optionalString = optionalString! + "... Hello..."
optionalIntValue = optionalString?.toInt()
optionalIntValue = optionalString!.toInt()
//var intValue:Int = optionalString?.toInt() // doesn't work

var otherString = optionalString ?? "A completely different string"

optionalString = nil // does work, can be nil
optionalString == nil
optionalIntValue = optionalString?.toInt()
// optionalIntValue = optionalString!.toInt()  // doesn't work

otherString = optionalString ?? "A completely different string"

var implicitlyUnwrappedString:String! = "This is implicit"
implicitlyUnwrappedString
aString = implicitlyUnwrappedString
implicitlyUnwrappedString = nil
// aString = implicitlyUnwrappedString  // doesn't work

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

```
"A String"
nil

{Some "Hello"}
false
{Some "Hello... Hello..."}
nil
nil

"Hello... Hello..."

nil
true
nil

"A completely different string"

"This is implicit"
"This is implicit"
"This is implicit"
nil

{Some "John Appleseed"}
"Hello!"

"Hello, John Appleseed"
```

# Tuples

- Can be named or anonymous
- Accessing anonymous tuples is done by position
- Named tuples can be accessed by name as well
- Just another data type
- Can be used to return multiple values
  - Don’t be fooled into using to return (T?,Error?)
  - There’s a better way

# Functions

- Declaration Syntax(es):

  func someFunc(localName1:T, localName2:U) -> V {…stuff…}

  func someFunc(externalName1 localName1:T,
  externalName2 localName2:U) -> V {…stuff…}

- 2nd form requires invocation with the variable names
- When functions are methods, this is the default
- Review: https://developer.apple.com/library/ios/documentation/Swift/Reference/Swift_StandardLibrary_Functions/index.html#//apple_ref/doc/uid/TP40016052

# Operators

- As an aside: operators are just functions with some syntactic sugar
- This is why Array<Int>.sort(<) works
- We'll get to customer operators later but that's something to keep in mind
- For that matter, functions are just closures with a bit of syntactic sugar - so it's sugar all the way down

# Closures

- Closures take one of three forms:
    - Global functions are closures that have a name and do not capture any values
    - Nested functions are closures that have a name and can capture values from their enclosing function only (remember this when we get to classes, et al.)
    - Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context

# Closures When Not Functions

- Anonymous functions
- “Captures” variables from the environment
- This is REALLY important
- The “trick” to closures is knowing when and why they capture variables from the environment

# Variable Capture

- Occurs in two ways:
  - At Closure declaration
    - When the closure is curried (if it is curryable)
- Anything within the surrounding lexical scope is fair game
  - Things do NOT have to be passed in as an argument

# Capture Lists

- Requires in keyword and comes before arg definitions
- Examples:

```
{ [unowned self] in
    ...
}

{ [thing1 = self.grabThing(), weak thing2 = self.something] in
    ...
}

{ [unowned self] (a:A, b:B) -> ReturnType in
    ...
}
```

# Closure Declaration

### Closure Expression Syntax

Closure expression syntax has the following general form:

```
{ ( parameters ) -> return type in
    statements
}
```

Closure expression syntax can use constant parameters, variable parameters, and `inout` parameters. Default values cannot be provided. Variadic parameters can be used if you name the variadic parameter and place it last in the parameter list. Tuples can also be used as parameter types and return types.

# The Swift Type System

- Types != Classes
- Taxonomy
  - Value Types (Structs / Enums)
  - Reference Types (Classes/UnsafePointers)
  - Abstract Types (Protocols)
- MetaTypes not as easily accessible as in ObjC

# Value vs Reference Type

- Value is on the stack, Reference is on the heap
- Value is pass-by-copy, Reference is pass-by-reference
- Operations on Reference types change data in place
- All the things we have dealt with in detail so far are Value types
- In performance terms, you are trading space and time for safety - frequently its a good trade

# Class/Struct/Enum

- Implement the OOP Paradigm in Swift
- Polymorphism
- Encapsulation
- Classes also implement Inheritance

# Value Types

- Copying — the effect of assignment, initialization, and argument passing — creates an independent instance with its own unique copy of its data

- Value types by default are Immutable

- Passing a value type to a function and modifying it in the function has no side effect outside the function

- Common Value Types:

  Strings, Numbers, Arrays, Dictionaries, instances of Enums and Structs, Tuples, Closures,

- Highly recommended: https://developer.apple.com/swift/blog/?id=10

# Reference Types

- Copying a reference implicitly creates a shared instance.
- After a reference copy, there are two variables which refer to a single instance of the data, so modifying data in the second variable also affects the original
- Think pointers in C and C-like languages
- Reference types:

  Instances of Classes (i.e. objects), Value types which have been transformed via the & operator or inout keyword, Everything coming from ObjectiveC, even if it is immutable

# How to Choose

- Use a value type when:
  - Comparing instance data with == makes sense
  - You want copies to have independent state
  - The data will be used in code across multiple threads
- Use a reference type (e.g. use a class) when:
  - Comparing instance identity with === makes sense
  - You want to create shared, mutable state

# Value -> Reference
# & and inout

- If you need to operate on a Struct or Enum in place outside its lexical scope, then:
  - the function or closure doing the operation needs the inout keyword and
  - the invocation needs to prefix the argument with &

# Reference -> Value

```
extension Array {

    /// Call `body(p)`, where `p` is a pointer to the `Array`\ 's
    /// contiguous storage. If no such storage exists, it is first created.
    ///
    /// Often, the optimizer can eliminate bounds checks within an
    /// array algorithm, but when that fails, invoking the
    /// same algorithm on `body`\ 's argument lets you trade safety for
    /// speed.
    func withUnsafeBufferPointer<R>(body: (UnsafeBufferPointer<T>) -> R) -> R

    /// Call `body(p)`, where `p` is a pointer to the `Array`\ 's
    /// mutable contiguous storage. If no such storage exists, it is first created.
    ///
    /// Often, the optimizer can eliminate bounds- and uniqueness-checks
    /// within an array algorithm, but when that fails, invoking the
    /// same algorithm on `body`\ 's argument lets you trade safety for
    /// speed.
    mutating func withUnsafeMutableBufferPointer<R>(body: (inout UnsafeMutableBufferPointer<T>) -> R) -> R
}
```

# Class and Struct Similarities

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind”

# Class and Struct Differences

- Structs are Value Types, Classes are Reference Types.  All rules apply
- Structs cannot be subclassed (Question: Why?)
- Struct functions cannot be curried automatically by the language (Why?)
- Structs need the mutating keyword to change values (Why?)

# Classes and Structs

- Have or can have:
  - Properties
  - Methods
  - Initializers
  - Deinitializers
  - Subscripts

# Properties

- Come in two flavors: stored and computed
- Frequent pattern, public computed property with a private “backing” stored property
- Can, like me, be lazy.  Lazy means set at most once at first use
- Computed can have:
  - get/set
  - willSet/didSet
- Can be associated with instances or with types

# Methods

- Are funcs with a context of ‘self’
- Are why we have these data types
- Have access to the properties of self
- In classes, can inherit or override the parent’s methods
- Can be instance or type methods

# Methods cont'd

- Naming conventions are important:
  - camelCase rules
  - use external names (i.e. ObjC conventions) to preserve interoperability
    - Calling conventions specify don't call with first external name (No idea why)
- Can be curried
  - When curried, will capture self

# Initializers

- Complex set of rules
- Two Types:
  - Designated
  - Convenience

# Initializer Rules

- A designated initializer must call a designated initializer from its immediate superclass assuming it has one
- A convenience initializer must call another initializer from the same class
- A convenience initializer must ultimately call a designated initializer
- All uninitialized properties must be initialized before calling the superclass initializer

# Subscripts

- Subscripts like those used on Collections can be used by any type
- Syntax is like the computed property syntax with input parameters
- Use the subscript keyword and specify one or more input parameters and a return type
- Subscripts can be read-write or read-only

# Enums

- Not your father's Enum type
- Value type like Struct with ennumerated, discrete allowable values
- The allowed values can be:
  - of some specific other Type (the "raw" value)
  - associated with another more complex value (the "associated" value)
- Just like other Types, they have initializers, methods, subscripts, etc.

# Example Use Cases

- Anywhere you'd use and enum in C or Java, typically Ints in that case
- Message types in a wire protocol
- Return type of Success or Failure

# Extensions

- Any non-final data type can be extended
- This includes types for which you don't have the source (e.g. Apple's classes)
- Corresponding idea in ObjC is called a Category
  - NB ObjC categories have two types, one of which is called an extension.
  - There is no direct equivalent to an ObjC extension in Swift

# Extensions

- Add ***computed*** properties and ***computed*** static properties
  - Cannot add stored properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

# Protocols

- Swift and ObjC's answer to multiple inheritance (Java does something similar with Interfaces)
- Allows a type to behave as a subtype of multiple types
- Syntax is a little complicated when dealing with Generics

# Pointers

- Uses: communication protocols, interaction with C-based system APIs
- You are responsible for:
  - memory allocation
  - byte ordering
  - memory initialization
  - memory deallocation
- Many common C datatypes are directly supported as are normal operations on those types >> & | etc

| C Type | Swift Type |
| --- | --- |
| `bool` | `CBool` |
| `char`, `signed char` | `CChar` |
| `unsigned char` | `CUnsignedChar` |
| `short` | `CShort` |
| `unsigned short` | `CUnsignedShort` |
| `int` | `CInt` |
| `unsigned int` | `CUnsignedInt` |
| `long` | `CLong` |
| `unsigned long` | `CUnsignedLong` |
| `long long` | `CLongLong` |
| `unsigned long long` | `CUnsignedLongLong` |
| `wchar_t` | `CWideChar` |
| `char16_t` | `CChar16` |
| `char32_t` | `CChar32` |
| `float` | `CFloat` |
| `double` | `CDouble` |

```swift
func toByteArray<T>(var value: T) -> [Byte] {
    return withUnsafePointer(&value) {
        Array(UnsafeBufferPointer(start: UnsafePointer<Byte>($0), count: sizeof(T)))
    }
}

func fromByteArray<T>(value: [Byte], _: T.Type) -> T {
    return value.withUnsafeBufferPointer {
        return UnsafePointer<T>($0.baseAddress).memory
    }
}
```

# Pointers

- If used in comms protocols, be sure to familiarize yourself with the Swap APIs

```
struct __CFByteOrder {
    init(_ value: UInt32)
    var value: UInt32
}
var CFByteOrderUnknown: __CFByteOrder { get }
var CFByteOrderLittleEndian: __CFByteOrder { get }
var CFByteOrderBigEndian: __CFByteOrder { get }
typealias CFByteOrder = CFIndex

func CFByteOrderGetCurrent() -> CFByteOrder

func CFSwapInt16(arg: UInt16) -> UInt16

func CFSwapInt32(arg: UInt32) -> UInt32

func CFSwapInt64(arg: UInt64) -> UInt64

func CFSwapInt16BigToHost(arg: UInt16) -> UInt16

func CFSwapInt32BigToHost(arg: UInt32) -> UInt32

func CFSwapInt64BigToHost(arg: UInt64) -> UInt64

func CFSwapInt16HostToBig(arg: UInt16) -> UInt16

func CFSwapInt32HostToBig(arg: UInt32) -> UInt32

func CFSwapInt64HostToBig(arg: UInt64) -> UInt64

func CFSwapInt16LittleToHost(arg: UInt16) -> UInt16

func CFSwapInt32LittleToHost(arg: UInt32) -> UInt32

func CFSwapInt64LittleToHost(arg: UInt64) -> UInt64

func CFSwapInt16HostToLittle(arg: UInt16) -> UInt16

func CFSwapInt32HostToLittle(arg: UInt32) -> UInt32

func CFSwapInt64HostToLittle(arg: UInt64) -> UInt64
```

# Generics

- Generics allow Types to be specified as families rather than individually
- The family can be infinite in size but you can instantiate specific types on an as needed basis
- Essentially a Generic is a template for a specific type
- Generics move a lot of run time checking in ObjC back into the compiler in Swift

# Custom Operators

- Operators are top level functions
- They take prefix, postfix and infix form and hence can take either one or two arguments
- Associativity right or left is also specifiable
- Specified in a separate statement with ‘operator’ keyword
- Typically are expressed as Generics so that the compiler can determine which implementation to apply at compile time
- You can extend system operators to cover your specific types

# Access Control

- public - anyone has access
- private - only this class, not other classes, including subclasses has access
- internal - can only be accessed within the same module

# Declaration Modifiers

- dynamic - dispatch via ObjC selector methodology
- final - cannot be overriden
- lazy - initialize a property at most once, lazily
- optional - indicate that a method in a protocol does not have to be implemented, consumers will check before invoking
- required - indicate that a method in a protocol must be implemented, consumers do NOT have to check
- weak - do not bump the reference count of the property

# Type Attributes

- autoclosure - surround the property with a closure so that it can be lazily evaluated
- noreturn - do not expect a return, typically something that will thread off

# Interoperability with ObjC

- When an Objective-C API returns an id, Swift will receive AnyObject
- When an Objective-C API returns nil, Swift will receive an Optional set to the value NONE
- Because there is no guarantee that an Objective-C method won't return nil, they all return an Optional to Swift
- When an Objective-C API returns a collection it will be typed to AnyObject, e.g. [AnyObject]
- When a Swift API returns a Tuple, Objective-C will receive… NUTHIN'

# Non-Interoperability with ObjC

- When a Swift API returns a Tuple, Objective-C will receive… NUTHIN’
- Same goes for:

  Generics, Enumerations defined in Swift, Structures defined in Swift, Top-level functions defined in Swift, Global variables defined in Swift, Typealiases defined in Swift, Swift-style variadics, Nested types, and Curried functions

# Optional Chaining

- Existence of Optional Types creates long chains or if-let/else constructs
- These can be very difficult to read, maintain and reason about
- Use of functional patterns vastly improves the situation

```
if let speed = wind["speed"].object as? NSNumber {
    record.speed = speed.doubleValue
    if let direction = wind["deg"].object   as? NSNumber {
        record.direction = direction.doubleValue
        if let location = location.object as? String {
            record.location = location
            if let pressure = main["pressure"].object as? NSNumber {
                record.pressure = pressure.doubleValue
                if let humidity = main["humidity"].object as? NSNumber {
                    record.humidity = humidity.doubleValue
                    if let temp = main["temp"].object as? NSNumber {
                        record.temp = temp.doubleValue
                        if let maxTemp = main["temp_max"].object as? NSNumber {
                            record.maxTemp = maxTemp.doubleValue
                            if let minTemp = main["temp_min"].object as? NSNumber {
                                record.minTemp = minTemp.doubleValue
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
var result = Result.Success(record)
    .then { wind["speed"].object as? NSNumber >>> speedBinding     >>> $0 }
    .then { wind["deg"].object   as? NSNumber >>> directionBinding >>> $0 }
    .unbox()

if ( result is NSError ) {
    let errString = "we got an error: \(result)"
}
```

# Iteration vs Recursion

- ‘Pure’ functional languages do not use looping constructs (i.e. no for/while loops)
- Recursion can always be used instead
- Places some requirements on the language (e.g. tail recursion optimization and memoization)
- Why do this?
  - State/Side effect avoidance
  - Parallelization

# Iteration

```
func betterImperativeCount(sentence: String) -> Int {
    var count = 0
    let words = sentence.componentsSeparatedByString(" ")

    for word in words {
        let characters = countElements(word)
        if characters > 3 {
            count += characters
        }
    }

    return count
}
```

# Recursion

```
func lazyFunctionalCount(sentence: String) -> Int {
    let words = sentence.componentsSeparatedByString(" ")
    return reduce(lazy(words).map { countElements($0) }.filter { $0 > 3 }, 0, +)
}
```

# Important Foundation Classes

- NSObject
- NSThread/NSRunLoop
- NSTimer
- NSDate/NSCalendar/NSDateFormatter
- NSError
- NSString
- NSArray/NSDictionary/NSSet
- NSFileManager
- NSAutoreleasePool

# NSObject

- Base class for everything in ObjC
- Handles retain/release
- Handles reflection
- Manages forward/perform

# NSThread/NSRunLoop

- Introducing threads is the best way to waste a week or two of project time
- Will be a big topic in drawing and network performance
- NSThread manages separate threads of execution
    - Each thread has a separate stack and thread state
- NSRunLoop manages the events associated with the thread.  Threads do not have to have an NSRunLoop
- NB: Threads can save state in the form of thread variables stored in a dictionary.
- One very important use of this will be highlighted below

# NSTimer

- Mechanism for handling timed events
- Generates events onto a thread's run loop
- You get the opportunity to specify a handler
- Real life example: determining if a device has disappeared
- Real life example: animating UI movement

# NSDate/NSCalendar/ NSDateFormatter

- Just what it sounds like
- Actually it’s a Date/Time object
- Handles timezone conversions, time intervals, calendrical computations, formatting

# NSError/NSException

- NSError: handles record of an error condition
- NSException: thrown in a try/catch/finally construct
- Problem: ObjectiveC has try/catch/finally, Swift does not
- Solution: is mix and match in one module

# NSString

- Just learn the API
- Immutable
- Reference type in ObjC/Value type in Swift
- Toll-free bridge between the two, i.e. the run time will transparently treat one as the other depending on what you are doing

# NSArray/NSDictionary/ NSSet

- NSArray and NSDictionary are toll-free bridged with their equivalent types in Swift
- But they are reference types where Swift has value types so be careful when doing nesting
- NSSet has no direct equivalent in Swift but can be implemented as a subclass of Dictionary depending on desired usage

# NSFileManager

- Simple API
- Checks for permissions/existence
- Does creation
- Works with Singleton pattern

# ARC

- Automatic Retain Counting
- Compiler does static analysis on your code
- Any time a reference is saved in a variable, the retain count goes up by 1:

```
var x = “\(someNumber)” // retain count = 1

var y = x      // retain count = 2
```

- Any time a reference is forgotten retain count goes down by 1:

```
y = nil   // retain count goes back to 1

x = nil   // retain count goes to 0
```

- When retain count hits zero, object is released

# ARC cont'd

- ARC automatically manages the retain count
- You can effect the retain count indirectly, but you cannot directly increment/decrement or retrieve it in your code

# ARC Autorelease

- What happens here?

```
func someFunc() -> String {
    return 47939.description
}
```

- If retain count on the description string drops to zero it is released before the caller can retain it

- If retain count doesn’t drop to zero when will it be released?

# ARC Autorelease

- Answer: NSAutoreleasePool
- Allocating an NSAutoreleasePool associates the pool with the thread in which allocation occurs
- Instances of NSAutoreleasePool are stacked within the thread
- Returned variables are added to the current NSAutoreleasePool
- When the NSAutoreleasePool is drained every element which has been added has it's retain count decremented
- If the decrement takes the object to zero (i.e. it has not been retained elsewhere) the object is deallocated

# NSAutoreleasePool

- Key element of ARC memory management
- Generally, you don’t need your own
- Two cases where you need to use it:
    - You have a section of code that allocates a large amount of temporary memory repeatedly (happens working with images and video frequently)
    - You are using a background thread of your own
- Usage: Allocate/Drain/Release