# App Development

## The Swift Programming Language

# Swift Programming Language History

- Announced at Apple WWDC June 2014, brainchild of Chris Lattner
- Published under Apache 2.0 open source license 12/2015
  - Swift language, supporting libraries, debugger, and package manager
- Governed by Swift Community
- Modern language features, including:
  - Type safety, type inference, generics, closures, tuples, protocols,
  - Automatic memory management
  - Unicode support (for character and string values as well as for identifiers).
  - Safety
    - Variables are always initialized before use, arrays and integers checked for overflow, memory is managed automatically
    - Swift objects can never be nil by default, else compile-time error. Optionals allow nil, but Swift syntax forces you to safely deal with it using ?.
    - Use of constants (immutable - read-only) to make code safer
- Can mix Swift and Objective-C in a single project and inter-call.
- Swift available for iOS, macOS, watchOS, tvOS, and Linux. Others TBD
- https://swift.org

# Swift

- Memory management: garbage collection via ARC (Automatic Reference Counting)
- C-style comments.
- Operators: Arithmetic, bitwise, and assignment similar to C
  - Under-/Overflow not allowed, compiler flags it or program terminates.
- Whitespace (spaces, tabs, LF, CR, null character) used to separate tokens and otherwise ignored.
- Statement termination
  - Semicolon ";" not expected or required unless multiple statements on single line.
- Nil pointers
- Nil-coalescing operator (a ?? b) unwraps optional a if it contains a value, or returns default value b if a is nil. Equivalent to a != nil ? a! : b
- Functions:
  - Classes passed by reference, structs passed by value (copied).
  - Both can have methods associated with them.
  - structs can't inherit from other structs, but can embed them.
- A module is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift’s import keyword.

# Naming Conventions

- Use camel case
- Names for classes, structures, enumerations, protocols, and types
  - Begin with uppercase
- Names for functions, methods, variables, and constants
  - Begin with lowercase

# Importing from other Modules

- import ModuleName
  - everything public in ModuleName is imported
  - Usually UIKit, which imports most other modules
  - Xcode: Command-Option click on ModuleName to see list of submodules this module imports or make available
- import ModuleName.SubmoduleName
- import Feature ModuleName.SymbolName
  - where Feature is the entity type (class, enum, func, protocol, struct, typealias, var)

# Literals

- Numeric literals
  - No prefix, then decimal: e.g., 12, 3.622, 1.7e3
  - 0b prefix for binary, 0o for octal, 0x for hexadecimal
  - Inferred floating point type for a variable if literal decimal point with digits on both sides or use an exponent.
  - Underscore _ can be used for readability
    - E.g., 1_000_000, 1_00_00_00, and 1000000 are equivalent
- Character literals
  - *Single* character surrounded by *double* quotes. E.g., "C" or "☺"
- String literals
  - As usual

# Types

- Declare type by appending ": " and then type after variable name
  - Types: Bool, Int, UInt, Float, Double, Character, String, Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64
  - var name: Type
  - Type can be inferred, e.g., var myInt = 21, myDouble = 12.0, myString = "Ken"
- Constants immutable via *let* keyword
  - let name: Type = expr
- Type aliases supported. E.g., typealias Byte = UInt8

  typealias FloatInFloatOut = (Float) -> Float

  var square: FloatInFloatOut = { return $0 * $0 }

  square(9.0) // returns 81.0
- Nested Types. E.g.,
  - var foo = Foo(); var bar = Bar(); foo.bar.i = 1
- Using reserved keywords as identifiers
  - Only if absolutely necessary
  - Use backticks, E.g., var `func` = 2

Nested type visibility example:

```
class C
{
   enum E
   {
      case A, B, C
   }
}
var v = C.E.B
```

# Operators

- Unary, binary, and ternary operators
- No over-/underflow allowed for + - * / %. Overflow via
  - &+ &- &* or specific type functions, e.g.:
  - let (result, overflow) = Int.add*WithOverflow*(myInt, myInt)
- Shifting signed operands preserves sign bit
- No implicit type conversion or silent casting, must explicitly convert
- == (A == B) // tests equality (same values)
- === (A === B) // tests identicality (the same objects)
- !== (A !== B) // tests unidenticality
- -2...2 ~= 2 // tests pattern match, true
- switch n { case (11...20): }
- Ternary Conditional Operator
  - expr1 ? expr2 : expr3

# Type Casting

- Type casting
  - is (check for subclass type), as (an upcast), as! (force cast with runtime error), as? (cast that returns optional value or nil)
- Closed range operator (a...b) or half-open range operator (a..<b)
  - for i in 1...10 { ... }

# Strings

- + operator to concatenate strings across lines
- += to append
- Comparison operators: ==, !=, <, <=, >, >=
- Escaped characters:
  - \0 \\ \t \n \r (Carriage return) \" \‘ \u{n} (arbitrary unicode)
- String interpolation via escape sequence \(expr)
  - Result of expressions substituted in a string literal
  - let x = "\(y) \(z)"
- String(x) to convert numeric value to string, e.g., String(10 + 30)
- Treat numeric type as function to convert from String, e.g. Int("9")

# Tuples

- Tuples use parentheses to group multiple related values separated by commas into a single compound value in a single container.
  - Inappropriate for structured, complex, or persistent data
- Can be used to return multiple values from function call
- The collection of types in order is the type of the tuple

```
let http404Error = (404, "Not Found")
// http404Error inferred type is (Int, String) & equals (404, "Not Found")
var error: (code: Int, description: String) //named tuple components
error.code = 404 // using named access on left
error.code = http404Error.0 // using position access on right
func readLine () -> (eof: Bool, readLine: String) // returning tuple
{
    …
}
```

# Arrays

- Collection of items of same type referenced via position. Passed by value.
- Array<Type> or just [Type]
- var arrayName = [Type]() // empty array
- var v = [Int](count: 10, repeatedValue: 1)
- var myArray: [String] = [“Foo", “Bar"] // array literal
- var odd = [1, 3, 5, 7, 9] // array literal
- var both = v + odd // combine arrays of same type
- var v += odd // append array
- let subset = odd[1...3] //index starts at 0, here slice created of [3,5,7]
- array.first, array.last, array.minElement, array.maxElement, array.capacity, array.count, array.isEmpty, array.append(value), array[range] = array, array.insert(value, n), array.removeAll(), array.removeAtIndex(n) (remove and return), array.removeLast(), array.reserveCapacity(n), array.contains(), array.dropFirst(number), array.elementsEqual(), array.filter(), array.forEach() { ... }, array.split(), array.sort() returns new array, array.sortInPlace() with optional trailing closure to specify how to sort { $1 < $0 }, etc.
- for *item* in *array* { ... } // iterate over each value
- for (index, item) in array.enumerate() { … } // iterate with position and value

# Slices

- Slices provide view into subset of a collection. No copy made until mutated.
- var set = [1, 2, 3, 4]
- var subset = set[1...3] // subset is 2, 3, 4

# Dictionaries

- Collection of values referenced by unique keys
- Dictionary<KeyType, ValueType> or just [KeyType:ValueType]
  - E.g., [String:Int] // key type String and value type Int
  - var kelvin: [String:Int] = [”extreme cold":0, ”freeze":273, "hot":3000]
  - let temp = kelvin[”freeze”] // temp = 273
  - kelvin[”hot”] = 4000 // replaces
- dictionary.isEmpty, dictionary.keys, dictionary.values, dictionary.popFirst(), dictionary.updateValue(), dictionary.removeValueForKey(key), dictionary.removeAll, dictionary.forEach(), dictionary.indexForKey(), dictionary.removeAtIndex(), dictionary.dropFirst(number), etc.
- Iterate via:
  - for (key, value) in dictionaryName { ... }
  - for value in dictionaryName.values { ... }
  - for key in dictionaryName.keys { ... }

# Sets

- Collection of unordered unique values of same type. Value type passed by value (copied).
- Set<Type>
  - var someSet = Set<String>()
  - var someSet: Set = [“Jack”, “Jill”, “Hill”]
- Iterate via:
  - for item in myset { ... }
- set.contains(), set.count, set.isEmpty, set.insert, set.intersect(), set.intersectInPlace(), set.exclusiveOr (), set.exclusiveOrInPlace(), set.subtractInPlace(), set.unionInPlace(), set.isDisjointWith(), set.isSubsetOf(), set.startIndex, etc.
- set.forEach() { ... }
- Option Sets provide set operations on bit-level values

# Computed Variables and Variable Observers

- Computed variable
  - Functions that act like variables
  - Contains getter and setter

```
var varName: someType = expression {
    get { // computes and returns value of someType
    }
    set(valueName) { // sets conditions
    }
}
```

- Variable observers
  - Functions attached to variables, called when variable about to change
  - willSet/didSet

```
var varName: someType = expression {
    willSet(valueName) { // called before value changed
    }
    didSet(valueName) { // called after value changed
    }
}
```

# Functions

- Function parameters are constant by default, to make them variable precede them in the function declaration with the var keyword. Use inout keyword for in-out parameters which are then passed by reference.
- func funcName(parameters) -> returnType
- {
- ...
- }
- Return optional value if reference can be nil by appending ? to return parameter or tuple
  - E.g., func foo() -> Int?
- *let binding* can be used to ensure non-nil. E.g., if let a = foo() { ... }
- addInt(paramA: x, paramB: y) //calling requires using parameter names
- Default parameter values can be specified. If parameter omitted on call, then default is used.
- Variadic parameters: variable number of parameters via "..." in declaration accessed as array. E.g., func myFunc(param: Type...) -> returnType
- Function type is expression of types of its parameters and return type, and can be used in places where other types used. E.g., pass function as parameter to other function.

# Closures

- Anonymous functions passed as arguments to other (higher-order) functions or returned. Can avoid overhead of needing a named function. Closure expression:

```
{
   (parameters) -> returnType in
      ...
}
```

- let a = pile.sort( { p1, p2 in return p1>p2 } )
- Inline closures have automatic argument names by position number ($0, $1, $2, ...)
  - pile.sort( { $0 > $1 } )
- Trailing closure: If last argument closure, then closure written after the parentheses of function's arguments. pile.sort() { $0 > $1 }
- If also no args, then can skip the (). E.g., pile.sort  { $0 > $1 }
- If returning a closure, values are captured beyond scope. By value if not modified, otherwise by reference.

# Optionals

- Handles the absence of a value – might exist or might not.
  - "there is a value, and it equals x" or "there isn’t a value at all“
- In Swift, object references are not pointers.
- They cannot normally be set to nil, unless explicitly declared to be optional values. E.g.,
  - var foo: String?
- Must be unwrapped to reveal underlying value
- Force unwrap the optional using ! If no value exists, runtime error.
  - E.g., if foo != nil { a = foo! } // unwrap and copy
- Can’t assign optional to non-optional.
- Optional binding
  - if let x = myOptional { log(x) } // x is unwrapped by “if let assignment”
- Optional chaining
  - if let x = a?.b?.c {

# Control Flow

- For loop

```
for initialization; condition; increment {
    ...
}
```

- for-in loop, use _ if index not used. Can be collection or range of numbers

```
for i in sequence { //
    ...
}
```

for i in sequence where *filter* . E.g.,

```
for i in 0...7 where (i % 2) == 0 {
    ...
}
```

# Control Flow

- while loops. Can use continue and break.

```
repeat {
    ...
} while condition
```

- if-else
- if-case. E.g., if case 13...19 = age { // teenager
- guard-else. E.g.

```
func myFunc()
{
    guard condition else {
        return // exit function
    } // continue execution
        ...
  }
```

Example:

```
guard let a = Float(str) else {
    continue  // ignore, not float
}
```

# Control Flow

```
switch expr {
    case pat1:  //  where qualifier can be used to refine
        ...
    default:
         ...
}
```

- Optional statement labels can precede switch and loops and be used by break or continue for specifying to which construct it applies.

# Error Handling

```
do { // provides scope
   ...
}
```

- Deferred execution. E.g.:

```
do {
   defer {
      ... // clean up code executed when scope exits

func myFunc() throws -> returnType
{
   ...
   if problem { throw error }

do {
   try myFunc(param)
}
catch [err] {
   ...
} catch {
 ...
}
```

# Classes

```
class NewClassName: BaseClassName
{
   ... // property and member definitions
}
```

- Can have instance properties or type properties (static keyword). Stored properties (var or let var) use memory vs. computed properties, the latter of which are methods (get/set) that look like properties.
- Property observers using willSet and didSet keyword
- self defined once initialized
- lazy keyword – initial value computed on first use
- Methods:
  - Type methods - functions associated with class. “*class* func myTypeMethod”
  - Instance methods – functions associated with each instance of a class
- *super* works. Use *override* to replace getters and setters. *final* allowed.

# Structures

- Structures - are value types, thus copied (classes are reference types). Can’t inherit. Can have computed properties, instance, type methods, initializers. No deinitializers.

```
struct name
{
}
```

- By default instance methods do not mutate properties
- For mutating methods, use the *mutating func* keyword

# Classes vs Structures

Swift automatically provides external interfaces for other code to use.

Both:

- 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

Only Classes:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

[Swift Language Guide]

# Enumerations

- A user-defined value type (copied, not referenced) of named values that allow you to work with them in a type-safe way. Can’t inherit. Can have computed properties, instance, type methods, initializers. No deinitializers.

```
enum Name {
    case A, B
}
```

- Values not assigned to enumeration members by default, but can define raw member values. If of Int type and initial one is set, auto-increments successive raw values.

```
enum Currency: Int {
    case Dollar = 100
    case Quarter = 25
    case Dime = 10
}
```

- Currency.Dime.rawValue // property access
- .Dime //can leave out Enum name if it can be inferred
- Currency(rawValue: 25) // translates raw value back to enumeration
- Associated values support different types for different cases

# Subscripts

- Classes, structures, and enumerations can define subscripts - shortcuts for accessing member elements of a collection, list, or sequence to set and retrieve values by index without needing separate methods for setting and retrieval. E.g.:

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

# Initialization

- Default values for stored properties in a new instance of a class, structure, or enumeration
- designated initializer must initialize all properties – *init(*
- convenience initializer - *convenience init(*
  - calls init with some default values
- failable initializer – can return nil *init?(*
- deinitializers – called just before deallocated *deinit*

# Access Control

- open, public, internal, fileprivate, private. Default access level of internal.
  - Open and public: enables entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework.
  - Internal:  enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app’s or a framework’s internal structure.
  - File-private: restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
  - Private: restricts use of an entity to the enclosing declaration. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.
- Open access only for classes and class members. Differs from public access:
  - Classes with public or more restrictive can be subclassed only within the module where they’re defined.
  - Class members with public or more restrictive access level can be overridden by subclasses only within the module where they’re defined.
  - Open classes can be subclassed within the module where they’re defined, and within any module that imports the module where they’re defined.
  - Open class members can be overridden by subclasses within the module where they’re defined, and within any module that imports the module where they’re defined.

# Protocol

- Defines blueprint of methods, properties, and other requirements suitable for a particular task or piece of functionality.
- Can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements.

```
protocol SomeProtocol
{
   var myProp: Double { get set } // read-write
   optional var myProp1: Float { get } // optional readonly
   static var myProp2: String { get set } // read-write
   func myFunction() -> Int
}
```

- Usage:

```
struct SomeStructure: FirstProtocol, AnotherProtocol { ...
}
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol { ...
}
```

# Global Functions

- assert
- dump
- fatalError
- max
- min
- precondition
- print
- sizeof
- swap
- etc.

# Changes between Version 2.2 to 3.x

- Swift 3 not source-compatible with 2.2 and 2.3
  - func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)  -->
  - func prepare(for segue: UIStoryboardSegue, sender: Any?)
- Changes
  - func f() { ... }                    // kein Parameter
  - func f(para: Int) { ... }           // gewöhnlicher Parameter
  - func f(_ para: Int) { ... }         // unbenannter Parameter
  - func f(ext para: Int) { ... }       // zweinamiger Parameter
  - func f(para: inout Int)  { ... }    // veränderlicher Parameter
  - func f(_ para: inout Int)  { ... }  // unbenannter ver. Parameter
  - func f(ext para: inout Int) { ... } // zweinamiger ver. Parameter
  - func f(para: Int = 0) { ... }       // optionaler Parameter
  - func f(_ para: Int = 0) { ... }     // unbenannter opt. Parameter
  - func f(ext para: Int = 0) { ... }   // zweinamiger opt. Parameter
  - func f(para: Int ...) { ... }       // variadischer Parameter
  - func f(_ para: Int ...) { ... }     // unbenannter var. Parameter
  - func f(ext para: Int ...) { ... }   // zweinamiger var. Parameter
- Foundation types no longer have NS prefix
- Functions ending with "d" return a new instance of the object. E.g., sorted()
- More detailed info: https://swift.org/blog/swift-3-0-released/

# Try Swift

- Xcode on Mac
  - "Get started with a playground" to try out fragments
- Without a Mac, various online tools:
  - E.g., IBM Swift Sandbox

# Recommended References

- Swift Language Guide
- Swift Pocket Reference: Programming for iOS and OS X by Anthony Gray. O'Reilly 2016.
  - http://www.oreilly.com/programming/free/swift-pocket-reference.csp
- Swift 3 von Michael Koffler