COCOAHEADS AC MAY 2016

ONE LESS THING – WHAT TO EXPECT OF SWIFT 3

A REMINDER

SWIFT 2.2

SE-0020 – SWIFT LANGUAGE VERSION BUILD CONFIGURATION

Swift 2.2

```
#if swift(>=3)
  // Your Swift 3 code here
#else
  // Fallback to Swift 2.2
#endif
```

SE-0021 – NAMING FUNCTIONS WITH ARGUMENT LABELS

Swift 2.0

```
class Foo {
  func bar(a: Int) {}
  func bar(b: Double) {}
}

let foo = Foo()
let a = foo.bar as (Int) -> Void
let b = foo.bar as (Double) -> Void
```

Swift 2.2

```
class Foo {
  func bar(a: Int) {}
  func bar(b: Double) {}
}

let foo = Foo()
let a = foo.bar(a:)
let b = foo.bar(b:)
```

SE-0022 – REFERENCING THE OBJECTIVE-C SELECTOR OF A METHOD

Swift 2.0

```
class Foo: NSObject {
  @objc func bar(arg: NSString) {}
}

let sel = Selector("bar:")
```

Swift 2.2

```
class Foo: NSObject {
  @objc func bar(arg: NSString) {}
}

let sel = #selector(Foo.bar)
```

Rationale

Stringly typing very error prone, checks only at runtime

SE-0011 – REPLACE TYPEALIAS KEYWORD WITH ASSOCIATEDTYPE FOR ASSOCIATED TYPE DECLARATIONS

Swift 2.0

```
protocol Prot {
    typealias Container
            : SequenceType
}
```

Swift 2.2

```
protocol Prot {
    associatedtype Container
                : SequenceType
}
```

Rationale

Confusion with typealiases used as an abreviation

SE-0028 – MODERNIZING SWIFT'S DEBUGGING IDENTIFIERS

Swift 2	Swift 3
`__FILE__`	`#file`
`__LINE__`	`#line`
`__COLUMN__`	`#column`
`__FUNCTION__`	`#function`
`__DSO_HANDLE__`	`#dsohandle`

Rationale

Old names owe their syntax to the C preprocessor

Swift compiler magic is done via #

SE-0015 – TUPLE COMPARISON OPERATORS

Swift 2.2

```
let x = (2, 3)
let y = (2, 3)

if x == y {
  ...
}
```

THE MANUAL

API DESIGN GUIDELINES

API DESIGN GUIDELINES

- Provide a standardised way of naming methods and properties in Swift
- Will be adopted in the Swift 3 standard library
- Often obvious, sometimes good hints
- Read them!
- https://swift.org/documentation/api-design-guidelines

A SHORT OVERVIEW

- Write doc comments
    - Help yourself to understand what you're doing
- Mutating methods: append, non-mutationg: appending
- Lower-case enum elements
- Use first argument label

THE NEW TOYS

NEW FEATURES

SE-0046 – ESTABLISH CONSISTENT LABEL BEHAVIOR ACROSS ALL PARAMETERS INCLUDING FIRST LABELS

Swift 2

```
func foo(x: Int, y: Int) { }

class Bar {
  func foo(x: Int, y: Int) { }
}

foo(x: 1, y: 2)
// but
let bar = Bar()
bar.foo(1, y: 2)
```

Swift 3

```
func foo(x: Int, y: Int) { }

class Bar {
  func foo(x: Int, y: Int) { }
}

foo(x: 1, y: 2)

let bar = Bar()
bar.foo(x: 1, y: 2)
```

Rationale

New naming guidelines encourage usage of first arg label

Eliminate inconsistency between functions and methods

SE-0025 – SCOPED ACCESS LEVEL

- Currently: `private` declarations are accessible from the current file only
- New: `private` declarations are only visible from the declaring type
  - No extensions!
- New `fileprivate` modifier behaves like old `private`
- Manual conversion if needed

SE-0071 – ALLOW (MOST) KEYWORDS IN MEMBER REFERENCES

Swift 2

```
let cell = UITableViewCell(
             style: .`default`,
             reuseIdentifier: nil)
```

Swift 3

```
let cell = UITableViewCell(
             style: .default,
             reuseIdentifier: nil)
```

SE-0001 – KEYWORDS AS ARGUMENT LABELS

Swift 2

```
indexOf(value, `in`: collection)
```

Swift 3

```
indexOf(value, in: collection)
```

SE-0048 – GENERIC TYPE ALIASES

Swift 3

```
typealias StringDictionary<T> = Dictionary<String, T>
```

SE-0092 – TYPEALIASES IN PROTOCOLS AND PROTOCOL EXTENSIONS

Swift 3

```
protocol Sequence {
  typealias Element = Iterator.Element
  ...
}
```

SE-0043 – DECLARE VARIABLES IN CASE LABELS WITH MULTIPLE PATTERNS

Swift 3

```
switch value {
  case let .Case1(x), let .Case2(x):
  ...
}
```

SE-0047 – DEFAULTING NON-VOID FUNCTIONS SO THEY WARN ON UNUSED RESULTS

- Swift 2: You could add `@warn_unused_result` to functions to create a warning if the result was unused
- Result can be explicitly discarded using `_ = foo()`
- Swift 3: Non-void functions always warn on unused result, unless `@discardableResult` is added

SE-0061 – ADD GENERIC RESULT AND ERROR HANDLING TO AUTORELEASEPOOL()

Swift 2

```
var result: Result? = nil
 var error: ErrorProtocol? = nil
 autoreleasepool {
   do { result = ... }
   catch let e { error = e }
 }
 guard let result = result else {
   throw error!
 }
 return result!
```

Swift 3

```
return try autoreleasepool {
  ...
}
```

CLEANING UP

REMOVALS

SE-0007 – REMOVE C-STYLE FOR-LOOPS

Swift 2

```
var primes = [2,3,5,7,11,13]
for var i=0; i<primes.count; i++ {
  ...
}

for var i=0; i < 10; i++ {
  ...
}
```

Swift 3

```
var primes = [2,3,5,7,11,13]
for prime in primes {
  ...
}

for i in stride(from: 0, through:
                  10, by: 1) {
  ...
}
```

Rationale

Carry-over from C
for-in and stride provide equivalent behaviour

SE-0004 – REMOVE THE ++ AND -- OPERATORS

Swift 2

```
var x: Int
x++

let a = ++x
```

Swift 3

```
var x: Int
x += 1

let a = x + 1
x += 1
```

Rationale

Carry-over from C

Mostly used to iterate something, for-in is better there

SE-0002 – REMOVE CURRYING FUNC DECLARATION SYNTAX

Swift 2

```
func curried(x: Int)(y: String)
       -> Float {
  ...
}
```

Swift 3

```
No longer supported
```

Rationale

Rarely used and a lot of language complexity

SE-0003 – REMOVE VAR FROM FUNCTION PARAMETERS

Swift 2

```
func foo(var i: Int) {
  i += 1
}
```

Swift 3

```
func foo(argI: Int) {
  var i = argI
  i += 1
}
```

Rationale

Confusion about inout parameters

SE-0053 – REMOVE EXPLICIT USE OF LET FROM FUNCTION PARAMETERS

Swift 2

```
func foo(let x: Int) {
  ...
}
```

Swift 3

```
func foo(x: Int) {
  ...
}
```

Rationale

All parameters are let since var parameters are removed

SE-0029 – REMOVE IMPLICIT TUPLE SPLAT BEHAVIOR FROM FUNCTION APPLICATIONS

Swift 2

```
func foo(a: Int, b: Int) {}

let x = (a: 1, b: 1)

foo(x)
```

Swift 3

```
No longer possible
```

Rationale

Internal modelling of functions as tuple to tuple changed

Confusing to newcomers; buggy anyway

SE-0036 – REQUIRING LEADING DOT PREFIXES FOR ENUM INSTANCE MEMBER IMPLEMENTATIONS

Swift 2

```
enum Coin {
  case heads, tails

  func f() {
    if self == heads {
      ...
    }
  }
}
```

Swift 3

```
enum Coin {
  case heads, tails

  func f() {
    if self == .heads {
      ...
    }
  }
}
```

Rationale

Leading dot required almost everywhere else

Enum cases semantially closest to static properties

SE-0054 – ABOLISH IMPLICITLYUNWRAPPEDOPTIONAL TYPE

- Implicitly Unwrapped Optionals (e.g. `Int!`) are no longer a type in the Standard Library but a compiler attribute
- Changes type inferrence

```
let x: Int! = 5

let y = x
```

- y has type `Int?` and not `Int!`

SE-0060 – ENFORCING ORDER OF DEFAULTED PARAMETERS

Swift 2

```
func foo(x: Int = 0, y: Int = 0) {
}

foo(y: 1, x: 1)
```

Swift 3

```
No longer supported
```

Rationale

Very rarely used

Complicates language for little benifit

SE-0066 – STANDARDIZE FUNCTION TYPE ARGUMENT SYNTAX TO REQUIRE PARENTHESES

Swift 2

```
let x: Int -> String

let y: (Int, Int) -> String

// Does y take a single tuple as
// argument or two Int's?
// Answer: Two Int's
```

Swift 3

```
let x: (Int) -> String
```

Rationale

Ambiguity between single-argument tuple and multiple args

NITPICKING

RENAME

SE-0068 – EXPANDING SWIFT SELF TO CLASS MEMBERS AND VALUE TYPES

Swift 2

```
self.dynamicType.staticMethod()
```

Swift 3

```
Self.staticMethod()
```

Rationale

Shorter, clearer intent matches self

SE-0031 – ADJUSTING INOUT DECLARATIONS FOR TYPE DECORATION

Swift 2

```
func foo(inout arg: Int) {

}
```

Swift 3

```
func foo(arg: inout Int) {

}
```

Rationale

Allows inout as parameter label

Allows specifying inout in a function's type

SE-0049 – MOVE @NOESCAPE AND @AUTOCLOSURE TO BE TYPE ATTRIBUTES

Swift 2

```
func foo(@noescape fn: () -> ()) {}
```

Swift 3

```
func foo(fn: @noescape () -> ()) {}
```

Rationale

You weren't able to specify the type of foo previously

Issues with manual currying

SE-0040 – REPLACING EQUAL SIGNS WITH COLONS FOR ATTRIBUTE ARGUMENTS

Swift 2

```
@available(*, unavailable,
  renamed= "MyRenamedProtocol")
```

Swift 3

```
@available(*, unavailable,
  renamed: "MyRenamedProtocol")
```

Rationale

Colon aligns better with the existing syntax of function calls

SE-0039 – MODERNIZING PLAYGROUND LITERALS

Swift 2

```
[#Color(colorLiteralRed: red,
  green: green, blue: blue,
  alpha: alpha)#]
```

```
[#Image(imageLiteral: imageName)#]
```

```
[#FileReference(
  fileReferenceLiteral: fileName)#]
```

Swift 3

```
#colorLiteral(red: red,
  green: gree, blue: blue,
  alpha: alpha)
```

```
#imageLiteral(resourceName:
                imageName)
```

```
#fileLiteral(resourceName:
               fileName)
```

MAKING THE TRANSITION EASIER THAN EVER

OBJ-C INTEROP

SE-0033 – IMPORT OBJECTIVE-C CONSTANTS AS SWIFT TYPES

Swift 3

```
typedef NSString * HKQuantityTypeIdentifier
__attribute__((swift_wrapper(enum));

HK_EXTERN HKQuantityTypeIdentifier const HKQuantityTypeIdentifierHeight;
HK_EXTERN HKQuantityTypeIdentifier const HKQuantityTypeIdentifierBodyMass;
HK_EXTERN HKQuantityTypeIdentifier const HKQuantityTypeIdentifierLeanBodyMass;

// imports as
enum HKQuantityTypeIdentifier : String {
    case Height
    case BodyMass
    case LeanBodyMass
}
```

SE-0055 – MAKE UNSAFE POINTER NULLABILITY EXPLICIT USING OPTIONAL

- `UnsafePointer` and friends can no longer be `nil`
- Handled using optional pointers, e.g. `UnsafePointer?`
- `ptr?.pointee = newValue`

SE-0057 – IMPORTING OBJECTIVE-C LIGHTWEIGHT GENERICS

- You could always write your own ObjC lightweight generics
- Now they are also imported into Swift 🎉

```
@interface MySet<T : id<NSCopying>> : NSObject
-(MySet<T> *)unionWithSet:(MySet<T> *)otherSet;
@end

class MySet<T : NSCopying> : NSObject {
  func unionWithSet(otherSet: MySet<T>) -> MySet<T>
}
```

SE-0062 – REFERENCING OBJECTIVE-C KEY-PATHS

Swift 2

```
class City: NSObject {
  dynamic var name: String = ""
}

let ac = City()
ac.setValue("Aachen",
            forKeyPath: "name")
```

Swift 3

```
class City: NSObject {
  dynamic var name: String = ""
}

let ac = City()
ac.setValue("Aachen", forKeyPath:
              #keyPath(City.name))
```

Rationale

Stringly typing unsafe and errors are only caught at runtime

SE-0064 – REFERENCING THE OBJECTIVE-C SELECTOR OF PROPERTY GETTERS AND SETTERS

Swift 3

```
class City: NSObject {
  dynamic var name: String = ""
}

let nameSetter = #selector(setter: City.name)
```

SE-0070 – MAKE OPTIONAL REQUIREMENTS OBJECTIVE-C-ONLY

Swift 2

```
@objc protocol MyProtocol {
  optional func myOptFunc()
}
```

Swift 3

```
@objc protocol MyProtocol {
  @objc optional func myOptFunc()
}
```

Rationale

Making optional requirements first class in Swift suggested several times, make clear that its a ObjC interop feature only

SE-0044 – IMPORT AS MEMBER

- E.g. `CGPathAddLineToPoint` will be imported as a member on `CGPath`
- Changes
  ```
  CGPathAddLineToPoint(path, &transform,
  topLeft.x, topLeft.y)
  ```
  to
  ```
  path.addLine(transform: &transform, x:
  topLeft.x, y: topLeft.y)
  ```

SE-0072 – FULLY ELIMINATE IMPLICIT BRIDGING CONVERSIONS FROM SWIFT

Swift 2

```
let str: String = "hello"

func foo(arg: NSString) { }

foo(str)
```

Swift 3

```
let str: String = "hello"

func foo(arg: NSString) { }

foo(str as NSString)
```

Rationale

Better ObjC import eliminates most bridging

Implicit conversions subtle and hard to grasp

SE-0005 – BETTER TRANSLATION OF OBJECTIVE-C APIS INTO SWIFT

Swift 2

```
let content = listItemView.text
  .stringByTrimmingCharactersInSet(
    NSCharacterSet.whitespaceAnd
    NewlineCharacterSet())
```

Swift 3

```
let content = listItemView.text
  .trimming(.whitespaceAndNewlines)
```

SE-0069 – MUTABILITY AND FOUNDATION VALUE TYPES

- Provide Swift structs without NS for the following Foundation types and there mutable counterparts
  - NSDate, NSURL, NSData, NSNotification
  - NSIndexPath, NSIndexSet, NSCharacterSet
  - NSAffineTransform, NSDateComponents, NSPersonNameComponents, NSURLComponents, NSURLQueryItem, NSUUID
- Obj-C APIs using these types will be automatically bridged to use the structs in Swift

YOUR EVERYDAY TOOLBOX – IMPROVED

STANDARD LIBRARY

Sequence: first(where:)

Enhanced Floating Point Protocols

New Model for Collections and Indices

Code unit initializers on String

Convert pointers to Int

Collection: prefix(while:) and drop(while:)

Renamed Set APIs

Failable Numeric Conversion Initializers

IteratorType post-nil guarantee

A New Model for Collections and Indices

Decoupling Floating Point Strides from Generic Implementations

THE DEVELOPMENT GOES ON
– WWDC ISN'T THERE YET

IN REVIEW / AWAITING

SE-0079 – ALLOW USING OPTIONAL BINDING TO UPGRADE SELF FROM A WEAK TO STRONG REFERENCE

Swift 2

```
doSomething { [weak self] in
  guard let strongSelf = self
    else { return }
  ...
}
```

Swift 3

```
doSomething { [weak self] in
  guard let self = self
    else { return }
  ...
}
```

SE-0088 – MODERNIZE LIBDISPATCH FOR SWIFT 3 NAMING CONVENTIONS

Swift 2

```
let queue = dispatch_queue_create(
              "com.test.myqueue",
              nil)


dispatch_async(queue) {
  ...
}
```

Swift 3

```
let queue = DispatchQueue(label:
              "com.test.myqueue")


queue.asynchronously {
  ...
}
```

SE-0075 – ADDING A BUILD CONFIGURATION IMPORT TEST

Swift 3

```
#if canImport(UIKit)
  // UIKit-based code
#elseif canImport(Cocoa)
  // OSX code
#elseif
  // Workaround/text, whatever
#endif
```

Rationale

Checking for OS or device type is brittle and doesn't mirror the original intention

SE-0083 – REMOVE BRIDGING CONVERSION BEHAVIOR FROM DYNAMIC CASTS

- `as!`, `as?` and `is` can perform bridging conversions (e.g. `String` to `NSString`)
- Conversion operators otherwise only used for type checks
- Provide initialisers for conversion, like for native Swift types

SE-0090 – REMOVE .SELF AND FREELY ALLOW TYPE REFERENCES IN EXPRESSIONS

Swift 2

```
// to reference the metatype Int

let intType: Int.Type = Int.self
```

Swift 3

```
// to reference the metatype Int

let intType: Int.Type = Int
```

Rationale

`.self` needed for disambiguation but the issue can be handled now

SE-0081 – MOVE WHERE CLAUSE TO END OF DECLARATION

Swift 3

```
func anyCommonElements<T : SequenceType, U : SequenceType where
  T.Generator.Element: Equatable,
  T.Generator.Element == U.Generator.Element>(lhs: T, _ rhs: U) -> Bool {
    ...
}
// Swift 3
func anyCommonElements<T : SequenceType, U : SequenceType>(lhs: T, _ rhs: U)
  -> Bool where T.Generator.Element: Equatable,
                T.Generator.Element == U.Generator.Element {
    ...
}
```

Rationale

Easier to find argument list

SE-0087 – RENAME LAZY TO @LAZY

Swift 2

```
struct ResourceManager {
  lazy var resource: NSData =
             loadResource()
}
```

Swift 3

```
struct ResourceManager {
  @lazy var resource: NSData =
              loadResource()
}
```

Rationale

`@lazy` will probably be rewritten as a property behavior in Swift 4. Do breaking change now

SE-0077 – IMPROVED OPERATOR DECLARATIONS

- Currently operator precedenc is handled using an integer precedence
- Fragile: No longer possible to add intermediate precedence, e.g. < (130) and ?? (131)
- New: Specify partial order between precedence groups
- Add operators to precedence groups

SE-0084 – ALLOW TRAILING COMMAS IN PARAMETER LISTS AND TUPLES

Swift 3

```
func foo(_ arg: Int...) {

}

foo(1,
    2,
    3,
    )
```

Rationale

Supported by Array, Dictionary already

Easeier editing when commenting out varargs, default args

COCOAPODS BEWARE

PACKAGE MANAGER

THE WHISHLIST

DEFERRED

DEFERRED

- Abstract classes and methods
- Property Behaviors
- Allow Swift types to provide custom Objective-C representations

THANK YOU

Alex Hoppen

@alex_hoppen