Hidden Gems in Swift

@akashivskyy

iOS

Agenda

- Literal Convertibles
- String Interpolation
- Pattern Matching
- Reflection
- Objective-C Bridging

Literal Convertibles

Literal convertibles


```
struct RegularExpression {
   let pattern: String
   init(pattern: String)
}

let emailRegex = RegularExpression(
   pattern: "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}$"
)

// would be nice

let emailRegex = "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}$"
```

String literal convertible

1.0

```
extension RegularExpression: StringLiteralConvertible {

   typealias StringLiteralType = String

   init(stringLiteral value: StringLiteralType) {
      self.pattern = value
   }

}

extension RegularExpression: ExtendedGraphemeClusterLiteralConvertible
extension RegularExpression: UnicodeScalarLiteralConvertible
```

All kinds of literals

- Array – Array, ArraySlice, Set
- Boolean – Bool, ObjCBool
- Dictionary – Dictionary, DictionaryLiteral
- Float – Float, Double
- Nil – Optional, Selector, Pointer
- Integer – Int, UInt, Float, Double
- String – String, Character, Selector

String Interpolation

String interpolation


```
enum Byte: UInt8 {
   case Zero = 0
   case One = 1
}

let string = "\(Byte.Zero)" // "Byte.Zero"

// would be nice

let string = "\(Byte.Zero)" // "0"
```

Interpolation convertible  1.0

```
extension String /* : StringInterpolationConvertible */ {

   init(stringInterpolationSegment byte: Byte) {
      self = "\(byte.rawValue)"
   }

}

let string = "\(Byte.Zero)" // "0"
```

Pattern Matching

What are patterns?

- Enumeration cases
- Single equatable values
- Ranges and intervals
- Value bindings
- Type casts
- func ~= <T, U> (lhs: T, rhs: U) -> Bool
- Tuples of anything above

What are patterns?

- Enumeration cases
- Single equatable values
- Ranges and intervals
- Value bindings
- Type casts
- func ~= <T, U> (lhs: T, rhs: U) -> Bool
- Tuples of anything above

Where do we use them?

- Switch statements
- If-let bindings
- For-in loops
- Catch statements

case

1.0

```
let point = (1, 2)

switch point {

   case (0, 0):
      println("origin")
   default:
      println("arbitrary point")

}
```

case let where

1.0

```
let point = (3, 4)

switch point {

   case let (x, y) where x == y:
      println("point on x = y line")
   default:
      println("arbitrary point")

}
```

if let where

1.2

```
let point: (Int, Int)? = maybePoint()

if let (_, y) = point where y > 0 {
   println("point above x axis")
}
```

for in where

1.2

```
let points = [
   (1, 2),
   (-3, 4),
   (5, -6),
   (-7, -8),
   (9, 10)
]

for (x, y) in points where x > 0 && y > 0 {
   println("point in 1st quadrant: (\(x), \(y))")
}
```

if case


```
let point = (5, 6)

let (width, height) = (
   Int(UIScreen.mainScreen().bounds.width),
   Int(UIScreen.mainScreen().bounds.height)
)

if case (0 ... width, 0 ... height) = point {
   print("point on screen")
}
```

if case let where

2.0

```
let point = (7, 8)

if case let (x, 1 ..< Int.max) = point where x < 0 {
   print("point in 2nd quadrant")
}
```

if case let where

2.0

```
switch subject {
   case pattern where condition:

// becomes

if case pattern = subject where condition {

// multiple cases not yet supported

if case pattern1, pattern2 = subject { // compiler error
```

for case let in where

2.0

```
let points: [(Int, Int)?] = maybePoints()

for case .Some(let (x, y)) in points where x < 0 && y < 0 {
   print("point in 3rd quadrant: (\(x), \(y))")
}
```

for case let in where

2.0

```
for element in subject {
   if case pattern = element where condition {

// becomes

for case pattern in subject where condition {

// multiple cases not yet supported

for case pattern1, pattern2 in subject { // compiler error
```

Reflection

Default behavior


```
struct Vector {

    typealias Point = (x: Double, y: Double)

    let start: Point
    let end: Point

    var length: Double {
        return sqrt(pow(end.x - start.x, 2) + pow(end.y - start.y, 2))
    }

}

let unitVector = Vector(start: (0, 0), end: (1, 1))
```

Default behavior

2.0

```
(.0 0, .1 0)
(.0 1, .1 1)
```

Reflection methods

- Custom description
- Custom children tree
- Custom Quick Look preview

Custom description

2.0

```
extension Vector: CustomStringConvertible {

   var description: String {
      return "(\(start.x) × \(start.y)) → (\(end.x) × \(end.y))"
   }

}
```

Custom description

2.0

```
"(0.0 × 0.0) → (1.0 × 1.0)"
```

Custom mirror


```
extension Vector: CustomReflectable {

   func customMirror() -> Mirror {
      return Mirror(self, children: [
         "start": "\(start.x) x \(start.y)",
         "end": "\(end.x) x \(end.y)",
         "length": length
      ])
   }

}
```

Custom mirror

2.0

start "0.0 × 0.0"
end "1.0 × 1.0"
length 1.414213562373095

Custom preview

2.0

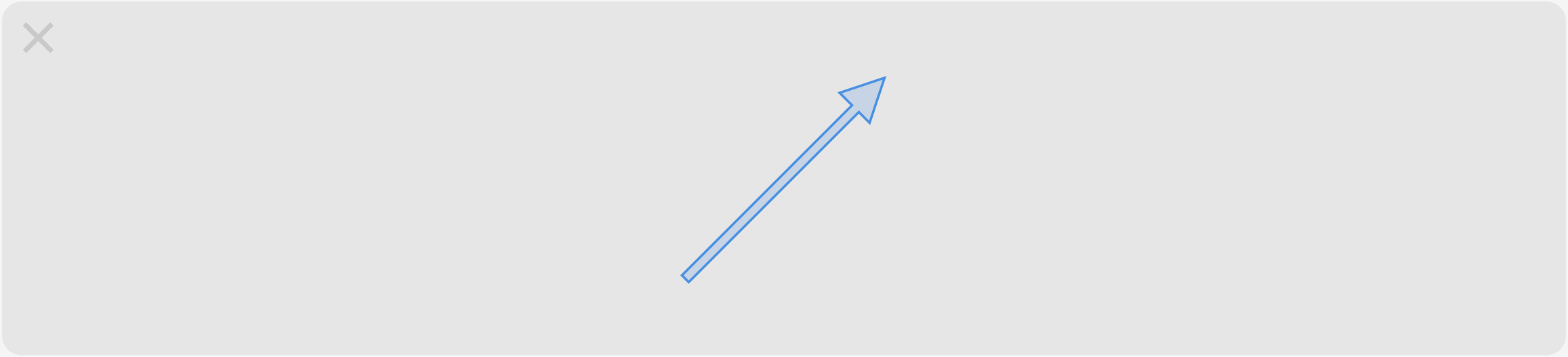
```
extension Vector: CustomPlaygroundQuickLookable {

    func customPlaygroundQuickLook() -> PlaygroundQuickLook {
        var bezierPath = UIBezierPath()
        // draw the path
        return .BezierPath(bezierPath)
    }

}
```

Custom preview

2.0



Reflection principles

- Overrides default type descriptors
- Provides rich visualization
- Read-only

Objective-C Bridging

Available bridging methods

- Inherit from Objective-C classes
- @objc attribute
- Bridging headers
- …and that’s basically it

Or is it?


```
@interface NSArray<Element> : NSObject // objective-c class

@end

struct Array<Element> { // generic swift struct

}

let swiftArray: [Int]
let objcArray = swiftArray as NSArray // no problem
```

Or is it?

2.0

```
@interface NSArray : NSObject

@end

struct Array<Element>: _ObjectiveCBridgeable {

}

let swiftArray: [Int]
let objcArray = swiftArray as NSArray
```

Bridgeable


```
protocol _ObjectiveCBridgeable {

    typealias _ObjectiveCType

    static func _isBridgedToObjectiveC() -> Bool
    static func _getObjectiveCType() -> Any.Type

    func _bridgeToObjectiveC() -> _ObjectiveCType

    static func _forceBridgeFromObjectiveC(...)
    static func _conditionallyBridgeFromObjectiveC(...)

}
```

Bridgeable

2.0

```
@interface XYZPoint : NSObject

- (instancetype)initWithX:(double)x y:(double)y;

@property double x;
@property double y;

@end
```

```
struct Point {
   let x: Double
   let y: Double
}
```


```
extension Point: _ObjectiveCBridgeable {

    typealias _ObjectiveCType = XYZPoint

    static func _isBridgedToObjectiveC() -> Bool {
        return true
    }

    static func _getObjectiveCType() -> Any.Type {
        return _ObjectiveCType.self
    }

    func _bridgeToObjectiveC() -> _ObjectiveCType {
        return XYZPoint(x: x, y: y)
    }

    static func _forceBridgeFromObjectiveC(source: _ObjectiveCType, inout result: Point?) {
        result = Point(x: source.x, y: source.y)
    }

    static func _conditionallyBridgeFromObjectiveC(source: _ObjectiveCType, inout result: Point?) -> Bool {
        _forceBridgeFromObjectiveC(source, result: &result)
        return true
    }

}
```

Bridgeable


```
let objcPoint = XYZPoint(x: 1, y: 2)
if let swiftPoint = objcPoint as? Point {
   // that's right
}

let objcPoint = XYZPoint(x: 3, y: 4)
let swiftPoint = objcPoint as Point // yeah

let swiftPoint = Point(x: 5, y: 6)
let objcPoint = swiftPoint as XYZPoint // hell yeah

let point: XYZPoint = Point(x: 7, y: 8) // mind: blown
```

Recap

- Literal Convertibles
- String Interpolation
- Pattern Matching
- Reflection
- Objective-C Bridging

How to learn the gems

- Carefully read Xcode release notes
- Follow right people on Twitter
- Study Swift module interface
- Use LLDB type lookup
- Experiment in playgrounds

Questions?

@akashivskyy
github.com/akashivskyy/talks

Thanks! 

@akashivskyy
github.com/akashivskyy/talks