Swift Extensions

- Extend existing data structure, even if no access to source

```swift
extension CGPoint {
    func distance(to point: CGPoint) -> CGFloat {
        return hypot(point.x - x, point.y - y)
    }
}
```

```
let distance = pt.distance(to: other)
```

Limitations
- Only sees public members of original type
- Cannot override existing methods
- Can only add new computed properties

Extensions

- Primary Use: small, self-contained helper methods
- Use sparingly
- Can obfuscate code
- Don't use in place of good oo design
- When in doubt, don't!

Subtyping

- Sometimes “every B is an A”

- “B is a subtype of A” means:
  
  *Every object that satisfies the rules for a B also satisfies the rules for an A*

- Code written using A's specification operates correctly even if given a B object.
  - Plus: clarify design, share tests, (sometimes) share code
Substitutivity

- Subtypes are substitutable for supertypes
  - Instances of subtype won’t surprise client by:
    - failing to satisfy the supertype’s specification
    - having more expectations than the supertype’s specification

- B is a true subtype of A if B has a stronger specification than A
  - This is not the same as a Swift (C++/Java/...) subtype
  - Subtypes that are not true subtypes are confusing and dangerous

True Subtypes

For Classes

- If a B object is used in place of an A object, then the result should be consistent with having just used an A object.
- B can:
  - Add properties and methods (that preserve invariants)
  - Override a method with one having a stronger (or equal) spec
- B cannot:
  - Remove properties or method
  - Override a method with one having a weaker spec

Stronger Method Specifications

- Promise More: Stronger Post
  - Returns clause harder to satisfy
  - Fewer objects in modifies clause
  - Effects clause harder to satisfy

- Ask less of client: Weaker Pre
  - Requires clause easier to satisfy

Stronger Method Specifications

- Promise More: Stronger Post
  - Returns clause harder to satisfy
  - Fewer objects in modifies clause
  - Effects clause harder to satisfy

- Ask less of client: Weaker Pre
  - Requires clause easier to satisfy
**Stronger Method Specifications**

- Promise More: Stronger Post
  - Returns clause harder to satisfy
  - Fewer objects in modifies clause
  - Effects clause harder to satisfy

- Ask less of client: Weaker Pre
  - Requires clause easier to satisfy

```swift
class Point {
    /// -Effects:
    /// self.x != old(self.x)
    func move() {
    }
}

class StrongerPoint : Point {
    /// -Effects:
    /// self.x > old(self.x)
    func append(other: Array) {
    }
}
```

```swift
class Array {
    /// -Requires:
    /// self.items is sorted
    func index(of key: Int) {
    }
}

class StrongerArray : Array {
    /// -Requires:
    /// true
    func index(of key: Int) {
    }
}
```

**Swift Subtyping**

- Swift subtypes are declared:
  - class A : B { }
  - class A : P { }
  - class A : B, P1, P2 { }
  - protocol P : P2 { }

- But are these true subtypes? Why? Why Not?

**Swift Subtyping Guarantees**

- A variable’s run-time type is a Swift subtype of its static (declared or inferred) type
  - let a: A = B() // OK
class B : A
  - let b: B = A() // compile-time error
class A : P
  - var b = B()
    b = A() // compile-time error

- Corollaries:
  - Objects always have implementations of the methods specified by their static type
  - If all subtypes are true subtypes, then all objects meet the specification of their static type
Inheritance

class Product {
    private let name : String
    private let description : String
    private let unitPrice : Int

    public func price() -> Int {
        return unitPrice
    }
    ...
}

Inheritance

class SaleProduct {
    private let name : String
    private let description : String
    private let unitPrice : Int
    private let discount : Double

    public func price() -> Int {
        return Int(unitPrice * discount)
    }
    ...
}

Inheritance

class SaleProduct : Product {
    private let discount : Double
    override public func price() -> Int {
        return Int(super.price() * discount)
    }
    ...
}

Is Every Square a Rectangle?

class MutableRectangle {
    /// Effects: fits shape to given size:
    /// self.width = width,
    /// self.height = height
    func set(width : Int, height : Int) {
    }
    class MutableSquare : Rectangle {

    Are any of these good specs for `MutableSquare.set`?
    1. /// Requires: width == height
    /// Effects: fits shape to given size
    func set(width : Int, height : Int) {
    
    2. /// Effects: sets all edges to given size
    func set(width : Int, height : Int) {
    
    3. /// Effects: sets `self.width` and `self.height` to width
    func set(width : Int, height : Int) {
}
What's the Problem?
• MutableSquare is not a **true** subtype of MutableRectangle.
• MutableRectangle is not a **true** subtype of MutableSquare.
• Solutions:
  – Make them unrelated (or siblings)
  – Make them immutable (!)
  • Recovers mathematical intuition

MutableSets and Countable Sets

```swift
class CountingMutableSet: MutableSet {
    var addCount = 0       // should really be private...

    override func add(_ elem : Int) {
        addCount += 1
        super.add(elem)
    }

    override func addAll(_ elems : [Int]) {
        addCount += elems.count
        super.addAll(elems)
    }
}
```

Dependence on implementation
• What does this code print?
  ```swift
  let c = CountingMutableSet()
  c.add(1)
  c.add(2)
  c.addAll([3,4])
  print(c.addCount)
  ```
  • Depends on impl. of addAll in MutableSet
  – Different implementations may behave differently!
  – If MutableSet's addAll calls add, then double-counting

MutableSets

```swift
class MutableSet {
    private var elems : [Int] = []

    func add(_ elem : Int) {
        if !elems.contains(elem) {
            elems.append(elem)
        }
    }

    func addAll(_ elems : [Int]) {
        for elem in elems {
            add(elem)
        }
    }

    func contains(_ elem : Int) -> Bool {
        return elems.contains(elem)
    }

    var count : Int {
        return elems.count
    }
}
```
Composition / Delegation

class CountingMutableSet {
    let delegate = MutableSet()
    addCount = 0

    func add(_ elem : Int) {
        addCount += 1
        delegate.add(elem)
    }

    func addAll(_ elems : [Int]) {
        addCount += elems
        delegate.addAll(elems)
    }
    ...
}

The implementation no longer matters

DrawableGraph delegates to Graph...

Protocols

protocol Set {
    func add(_ elem : Int)
    func addAll(_ other : [Int])
    func contains(_ elem : Int) -> Bool
    var count : Int { get }
}

class MutableSet : Set { ... }

class CountingMutableSet : Set { ... }

Protocol Declaration

Contract

Code for all protocol members

Protocols

• Class/struct can implement many protocols.

• Range (ie, 0..<5):
  − Equatable —
  − Indexable — startIndex, endIndex, index(after:), subscripting (e.g. []), index(offsetBy:)
  − Sequence — makeIterator (thus supports for in)
  − Collection — basically Indexable & Sequence & Equatable & ...
  − ...

Protocols in Foundation Library

• Array also a Collection and Sequence
• Dictionary is also a Collection and Sequence
• Set is also a Collection and Sequence
• String is also a Collection and Sequence
• ... is also a Collection and Sequence

• Can write code that works on all of them!
protocol Sequence {
    associatedtype Element
    func makeIterator() -> Iterator
    func contains<Element>() -> Bool
    func contains(where: Element) -> Bool
    func first(where: Element) -> Element?
    func min() -> Element?
    func max() -> Element?
    func sorted() -> [Element]
    func reversed() -> [Element]
    func filter<Element>(where: Element) -> [Element]
    func prefix(Int) -> SubSequence
}

// Client
let q = Queue()
...
let max = q.max()
...
let pos = q.filter { $0 > 0 }
Delegation in UIKit

- Pattern for blind communication

Controller

- should ...
- will ...
- did ...

Delegate

Model

View

conforms to a delegate protocol

UIScrollView

- scrollView.contentSize: Size of view we are scrolling around in.
• `scrollView.contentOffset` : origin of content view in scroll view.

• `scrollView.contentOffset` : origin of content view in scroll view.

• `scrollView.zoomScale` : scale applied to the scrollview's content
UIScrollView Delegation


.protocol UIScrollViewDelegate {
    .func scrollViewDidScroll(scrollView: UIScrollView)
    .func viewForZooming(in scrollView: UIScrollView) -> UIView
}

class UIScrollView : UIView {
    .weak var delegate: UIScrollViewDelegate?
    ...
}

UIScrollView Delegation

class ImageViewController : UIViewController, UIScrollViewDelegate {

    .let imageView : UIImageView = ...

    @IBOutlet var scrollView : UIScrollView! {
        .didSet { scrollView.delegate = self }
    }

    .func viewForZooming(in scrollView: UIScrollView) {
        .return imageView
    }

UIScrollView Properties

- You Must Set These! Don't Forget! Really!
    - .var contentSize : CGRect
    - .var minimumZoomScale : CGFloat
    - .var maximumZoomScale : CGFloat

- These may look familiar
    - .var contentOffset : CGPoint
    - .var zoomScale : CGFloat

Delegation in UIKit

UIScrollView conforms to UIScrollViewDelegate

Controller

scrollViewDidScroll

scrollViewForZooming

Model

delegate
**UIScrollView Methods**

- `func scrollRectToVisible(CGRect, animated: Bool)`
- `func zoom(to: CGRect, animated: Bool)`
- `func setContentOffset(CGPoint, animated: Bool)`
- `func setZoomScale(CGFloat, animated: Bool)`

**UITextFields**

- Another delegation example!
- Allows you to handle keyboard entry in different ways