Swift Extensions

- Extend existing data structure, even if no access to source
- Look at `pathForArrow` code

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

Extensions

- Limits
  - only sees public members of original type
  - cannot override existing methods
  - can only add new computed properties
- Use sparingly
  - obfuscates code
  - don’t use in place of good oo design
  - small, self-contained helper methods
  - 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

```swift
class Array {
    /// -Returns: index of
    /// key in items
    func index(of key: Int)
}
class StrongerArray : Array {
    /// -Returns: index of
    /// first occurrence of in
    /// key items
    func index(of key: Int)
}
class A {
...
}
class B : A {
...
}
```

```swift
class Array {
    /// -Modifies: self, other
    func append(other: Array)
}
class StrongerArray : Array {
    /// -Modifies: self
    func append(other: Array)
}```

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() {
    }
}
```

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

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

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

**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`
  - `let b: B = A() // compile-time error`
  - `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

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

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

Inheritance

```swift
class SaleProduct {
    private let name : String
    private let description : String
    private let unitPrice : Int

    private let discount : Double

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

Inheritance

```swift
class SaleProduct : Product {
    private let discount : Double

    override public func price() -> Int {
        return Double(super.price() * discount)
    }
    ...
}
```

Benefits of Inheritance

- Don’t repeat unchanged fields and methods
  - In implementation
    - Simpler maintenance: fix bugs once
  - In specification
    - Clients who understand the superclass specification need only study novel parts of the subclass
  - Modularity: can ignore private fields and methods of superclass (if properly defined)
  - Differences not buried under mass of similarities
- Able to substitute new implementations
  - No client code changes required to use new
**Downsides of Inheritance**

- Poor planning can lead to a muddled hierarchy
  - Relationships may not match intuition
- Poor design can produce subclasses that depend on many implementation details of superclasses
  - “fragile base class problem” – must evolve sub in tandem with super
  - encapsulation is broken (ex: self calls...)
- Subtyping and inheritance are orthogonal!
  - Subclassing gives you both. Sometimes you want just one

---

**Is Every Square a Rectangle?**

```swift
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

---

**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
  - Lesson: Subclassing often requires designing for extension
**Design For Extension**

- Change spec of MutableSet
  - Indicate all self-calls
  - Less flexibility for implementers of specification

- Avoid spec ambiguity by avoiding self-calls
  - “Re-implement” methods such as addAll
    - Requires re-implementing methods
  - Use composition
    - Delegate "real work" to a private object
    - No longer a subtype (unless an interface is handy)
    - Bad for callbacks, equality tests, etc.

**Composition / Delegation**

```swift
class CountingMutableSet {
    let delegate = MutableSet()
    var addCount = 0

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

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

**Composition and Subtyping**

- CountingMutableSet is no longer a MutableSet

- Class may be a true subtype of its delegate
  - But Swift doesn't know that
  - Swift requires declared relationships
  - Not enough just to meet specification

- Protocols to the rescue!
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 { ... }

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
• Can write code that works on all of them!

Protocols

• Class/struct can implement many protocols.

Protocol Extensions

protocol Sequence {
    associatedtype Element
    func makeIterator() -> Iterator
    func contains(Element) -> Bool
    func contains(where: (Element) -> Bool) -> Bool
    func min() -> Element?
    func max() -> Element?
    func sorted() -> [Element]
    func reversed() -> [Element]
    func map<T>((Element) -> T) -> [T]
    func filter((Element) -> Bool) -> [Element]
}

// Client
let q = Queue()
... let max = q.max() let pos = q.filter { $0 > 0 } if q.contains { abs($0) > 10 } for x in q { ... }
class Queue: Sequence {
    typealias Element = Int
    private var elems: [Int]
    func makeIterator() -> Iterator { .. }
    func add(_ elem: Int) { .. }
    func remove() -> Int { .. }
    var count: Int
}

protocol Sequence {
    associatedtype Element
    func makeIterator() -> Iterator
    func contains(Element) -> Bool
    func contains(where: (Element) -> Bool) -> Bool
    func first(where: (Element) -> Bool) -> Element?
    func min() -> Element?
    func max() -> Element?
    func sorted() -> [Element]
    func reversed() -> [Element]
    func map<T>((Element) -> T) -> [T]
    func filter((Element) -> Bool) -> [Element]
    func prefix(Int) -> SubSequence
}

// Client
let q = Queue()
...
let max = q.max()
let pos = q.filter { $0 > 0 }
if q.contains { abs($0) > 10 } for x in q {
    ...
}

Delegation in UIKit
- Pattern for blind communication

Controller
  should ...
  will ...
  did ...
delegate
Model
View

Delegation in UIKit
Controller
conforms to UIScrollViewDelegate
scrollViewDidScroll
viewForZooming
...
- `scrollView.contentSize`: Size of view we are scrolling around in.

- `scrollView.contentOffset`: Origin of content view in scroll view.
UIScrollView Delegation

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

class UIScrollView : UIView {
    weak var delegate: UIScrollViewDelegate?

    var imageView: UIImageView {
        weak var delegate: UIScrollViewDelegate?
    }
}

UIScrollView Delegation

class ImageViewController : UIViewController, UIScrollViewDelegate {

    let imageView : UIImage = ...

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

    func viewForZooming(in scrollView: UIScrollView) -> UIView {
        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

UIScrollView Methods

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

Extensions for Protocols

- Extensions can add default implementation to protocols.
- To conform to Sequence protocol, you write:
  - func makeIterator() -> IteratorProtocol{...}
- You get from Sequence extension for free:
  - contains(), forEach(), joined(separator:), min(), max(),
    filter() and map(), et. al.
  - This extension uses only Sequence protocol methods
    in its implementation.
- Functional programming: you never have to iterate over elements yourself...
Pluses and minuses of inheritance

- Inheritance is a powerful way to achieve code reuse
- Inheritance can break encapsulation
  - A subclass may need to depend on unspecified details of the implementation of its superclass
    - E.g., pattern of self-calls
  - Subclass may need to evolve in tandem with superclass
    - Okay within a package where implementation of both is under control of same programmer
- Authors of superclass should design and document self-use, to simplify extension
  - Otherwise, avoid implementation inheritance and use composition instead