## 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
- Only computed properties can be added

## 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
  ```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)
}
```
- 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
  ```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

```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 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?

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

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

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)
    }
    ...
}

Inheritance

+ Avoids repeating code
+ Able to swap in new implementations as subclasses without breaking code
  – (if true subtypes)

- Unintuitive hierarchies
- Subtyping and inheritance are orthogonal concepts.
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.

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

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?
  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. (eg: UIViewController)
Design for Extensibility

1. Change Spec to indicate self calls
   - Less flexibility for implementers
2. Reimplement methods to never make self calls
   - Lots of code duplication
3. Extension via Composition/Delegation

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

```swift
class CountingMutableSet {
    let delegate = MutableSet()
    var addCount = 0
    func add(_ elem : Int) {
        addCount += 1
        delegate.add(elem)
    }
    func addAll(_ elems : [Int]) {
        addCount += elems.count
        delegate.addAll(elems)
    }
    func contains(_ elem : Int) -> Bool {
        return delegate.contains(elem)
    }
    var count : Int {
        return delegate.count
    }
}
```

Protocols

```swift
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

• 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 & ...
  – ...

protocol Equatable {
    static func == (lhs: Self, rhs: Self) -> Bool
}

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(elt: Element) -> Bool
    func first(where: (Element) -> Bool) -> Element?
    func min() -> Element?
    func max() -> Element?
    func sorted() -> [Element]
    func reversed() -> [Element]
    func map<T>(f: (Element) -> T) -> [T]
    func filter(isIncluded: (Element) -> Bool) -> [Element]
    func prefix(subsequenceLength: Int) -> Subsequence
}

class Queue: Sequence {
    typealias Element = Int
    private var elements: [Int]
    func makeIterator() -> Iterator {
        var iterator = self.elements.makeIterator()
        while let element = iterator.next() {
            if isIncluded(element) {
                result.append(element)
            }
        }
        return Array(result)
    }

    // Client
    let q = Queue()
    ...
    let max = q.max() // only uses Seq. protocol methods
    var result = [Element]()
    var iterator = self.makeIterator() while let element = iterator.next() {
        if isIncluded(element) {
            result.append(element)
        } ...
    }
    return Array(result)
}

class Queue: Sequence {
    typealias Element = Int
    private var elements: [Int]
    func makeIterator() -> Iterator {
        var iterator = self.elements.makeIterator()
        while let element = iterator.next() {
            if isIncluded(element) {
                result.append(element)
            } ...
    }
    ...
    }

    // Client
    let q = Queue()
    ...
    let max = q.max() // only uses Seq. protocol methods
    var result = [Element]()
    var iterator = self.makeIterator() while let element = iterator.next() {
        if isIncluded(element) {
            result.append(element)
        } ...
    }
    return Array(result)
}

// Client
let q = Queue()
...
let max = q.max() // only uses Seq. protocol methods
var result = [Element]()
var iterator = self.makeIterator() while let element = iterator.next() {
    if isIncluded(element) {
        result.append(element)
    } ...
} return Array(result)

...
interfaces and abstract classes

// root interface of collection hierarchy
interface Collection

// skeletal implementation of Collection
abstract class AbstractCollection implements Collection

// type of all ordered collections
interface List extends Collection

// skeletal implementation of List
abstract class AbstractList extends AbstractCollection implements List

// our old friend...
class Vector extends AbstractList

// our old friend...
class LinkedList extends AbstractList