<S:wift, G:enerics>

@elmkretzer

Nah,
why care?

id was good enough

```
@property (nonatomic, weak)
id<YeahDelegate> delegate;
```

Sorry, this is not a Generic

Ultimately, what is your goal?

- 📱 Build great products
- 😴 Write less code
- 🗒️ Read less code
- 🛀 Enjoy easy life

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Generics.html

Ok,
gimme

They’re Everywhere

```swift
let easy: String? = "i am a generic"
```

They’re Everywhere

```swift
let easy: Optional<String> = "Oh"
```

Bring Your Own

```
struct Something<T> {
  let with: T
}
```

Bring Your Own

```
let a = Something(with: 5)
print(a.dynamicType)

// Something<Int>
```

Bring Your Own

```
let b = Something(with: "🎁")
print(b.dynamicType)

// Something<String>
```

Bring Your Own

```
let c = Something(with: [1, 2, 3])
print(c.dynamicType)

// Something<Array<Int>>
```

Bring Your Own

```
let d = Something(
  with: { (i: Int) -> Int in i }
)
print(d.dynamicType)

// Something<Int -> Int>
```

Bring Your Own

```
let e: Something<Double> = Something(with: 5)
print(e.dynamicType)

// Something<Double>
```

A B T
easy as
1 2 3

Flexible

```
// T time, but you could
// write Coffee as well
func identity<T>(t: T) -> T {
  // ehm what?
  return t
}
```

Restricted

```
// when type signature is given,
// hard to implement in wrong way
func const<A, B>(a: A) -> B -> A {
  // wtf - does that even make sense?
  return { _ in a }
}
```

Flexible && Restricted

```
enum Decision<T> {
  case KeepAll
  case ChangeTo(T)
  var go: T -> T {
    switch self {
    case .KeepAll: return identity
    case .ChangeTo(let x): return const(x)
    }
  }
}

[1,2,3].map(Decision.KeepAll.go)
// 1, 2, 3
[“a”,"b","c"].map(Decision.ChangeTo("z").go)
// z, z, z
```

Bounded Parametric Polymorphism 🙄

Do more stuff by constraining

```
// PAT - Protocol delivers contract
// with a associated type for the return value
protocol Doable {
  typealias Stuff
  func doMore() -> Stuff
}

// the function is not even interested
// in the implementation
func doMore<D: Doable>(ds: [D]) -> [D.Stuff] {
  return ds.map { d in d.doMore() }
}
```

Mess with the stdlib

```
// retroactive modeling
// apply your logic on existing generics
extension Array where Element: Doable {

  // automagically access associated
  // type `Stuff` from the generic `Element`
  func doMore() -> [Element.Stuff] {
    return self.map { d in d.doMore() }
  }

}
```

RealWorldPlease ©

kinda Arrows

Generic Functions w/o PATs

```swift
// overload operator
func +<A, B, C>(lhs: A -> B, rhs: B -> C) -> A -> C {
  return { rhs(lhs($0)) }
}

// get some functions
let tupleWithSquare: Int -> (Int, Int) = { ($0, $0 * $0) }
let joinTuple: (Int, Int) -> (Int) = { $0.1 + $0.0 }

// mix them together
let addTheSquare = tupleWithSquare + joinTuple

// sprinkle some variables and you are ready
let t = addTheSquare(3)
// 12
```

Attributed String

Generic Types w/o PATs

```swift
// add computed property as extension
extension NSMutableAttributedString {
  // the type corresponds to the attribute
  public var fontAttribute: StringAttributer<UIFont> {
    get { return StringAttributer(string: self, attribute: NSFontAttributeName) }
    set { /* please the compiler */ }
  }
}

// define the generic attribute access
public struct StringAttributer<T> {
  let string: NSMutableAttributedString
  let attribute: String
  // subscript sugar
  public subscript(from start: Int, to end: Int) -> T? {
    get {
      return string.attributesAtIndex(start, effectiveRange: nil)[attribute] as? T
    }
    set {
      guard let value = newValue as? AnyObject else { return }
      string.addAttribute(attribute, value: value, range: NSMakeRange(start, end - start))
    }
  }

}

// usage
let string = "test"
let attrString = NSMutableAttributedString(string: string)
attrString.fontAttribute[from: 0, to: 1] = UIFont.boldSystemFontOfSize(12)
```

cells, with no strings attached

Generic Functions w/ PATs

```
// 1. basic protocol for type constaints
//    in class use: static let identifier = “xxx”
protocol DequeueableCell {
  static var identifier: String { get }
}

// 2. add constrained generic method on UITableView
extension UITableView {

  // generic function with constraint
  func dequeueReusableCellFor<T: DequeueableCell>(ip: NSIndexPath) -> T {
   return dequeueReusableCellWithIdentifier(T.identifier,
                                            forIndexPath: ip) as! T
  }

}

// 3. usage when MyCell conforms to DequeueableCell
let cell: MyCell = tableView.dequeueReusableCellFor(indexPath)
```

UserDefaultable

Generic Types w/ PATs

```swift
// just to shorten the code ;-)
typealias NSUD = NSUserDefaults

// 1. Protocol for a Type that provides get/set for a ValueType from/to NSUD
protocol UserDefaultable {
  typealias ValueType
  static func get(key: String, fromDefaults defaults: NSUD) -> ValueType?
  static func set(value: ValueType?, forKey key: String, inDefaults defaults: NSUD) -> Void
}

// 2. we are lazy we add the default behavior as extension
//    and treat all as object
extension UserDefaultable {
  // move getter from NSUserDefaults to Protocol
  static func get(key: String, fromDefaults defaults: NSUD) -> ValueType? {
    return defaults.objectForKey(key) as? ValueType
  }
  // move setter from NSUserDefaults to Protocol
  static func set(value: ValueType?, forKey key: String, inDefaults defaults: NSUD) -> Void {
    guard let value = value as? AnyObject else { return /* possibly delete */ }
    defaults.setObject(value, forKey: key)
  }
}
```

```swift
// wrap up the UserDefaultable Type in a generic struct
struct UserDefaultsEntry<T: UserDefaultable> {
  let key: String
  let defaults: NSUserDefaults
  // T knows quite a bit:
  // 1. Type of value
  // 2. get the value
  // 3. set the value
  var value: T.ValueType? {
    get { return T.get(key, fromDefaults: defaults) }
    set { T.set(newValue, forKey: key, inDefaults: defaults) }
  }
}
```

```
// 1. conform to Protocol
extension String: UserDefaultable {
  typealias ValueType = String
}

// 2. add wrapper
class UserDefaults {

  // hold defaults
  static var defaults: NSUD {
    return NSUD.standardUserDefaults()
  }

  // generic helper method
  static func value<T>(key: String) -> UserDefaultsEntry<T> {
    return UserDefaultsEntry(key:key, defaults:defaults)
  }

  // here is the good part
  // UserDefaults.userName.value = „easy to handle"
  static var userName: UserDefaultsEntry<String> {
    get { return value("userName") }
    set { /* please the compiler */ }
  }

}
```

Big Picture

🔭

```swift
// don't care about the details
static func getAllStoredOrFetch<
  S, T where
  S: Storable,
  T: OwnedStorable,
  T: RemoteFetchableEntity,
  T.OwnerType == S,
  T.RemoteOwnerType == S,
  T.FetchableType == [T]
  >(forOwner owner: T.OwnerType) -> Signal<[T]> {
    // 1. gets all stored domains of T for owner T.OwnerType
    // 2. if not available it will update from remote
    //    as T is RemoteFetchableEntity
    let signal = Signal([T]())
      .flatMap(SignalOperation.getAllStoredWithin(owner))
      .ensure(SignalOperation.updateFromRemote(T.fetch, forOwner: owner))
    return signal
}
```

Now?

Take Away

- Classes, Structs, Enums, Functions support Generics (+ declarations inside)
- T can be (mostly) anything
- Unleash full power when used in combination with PATs
- Retroactive modeling of stdlib generic types
- Think of all the stuff where basically the same happens with different types

Swift 2.x

Generics in Swift

https://github.com/apple/swift/blob/master/docs/Generics.rst

Type Checker

https://github.com/apple/swift/blob/master/docs/TypeChecker.rst

Swift 3.0

Complete generics: Generics are used pervasively in a number of Swift libraries, especially the standard library. However, there are a number of generics features the standard library requires to fully realize its vision, including recursive protocol constraints, the ability to make a constrained extension conform to a new protocol (i.e., an array of Equatable elements is Equatable), and so on. Swift 3.0 should provide those generics features needed by the standard library, because they affect the standard library's ABI.

https://github.com/apple/swift-evolution/blob/master/README.md

happy-go-swift

@elmkretzer