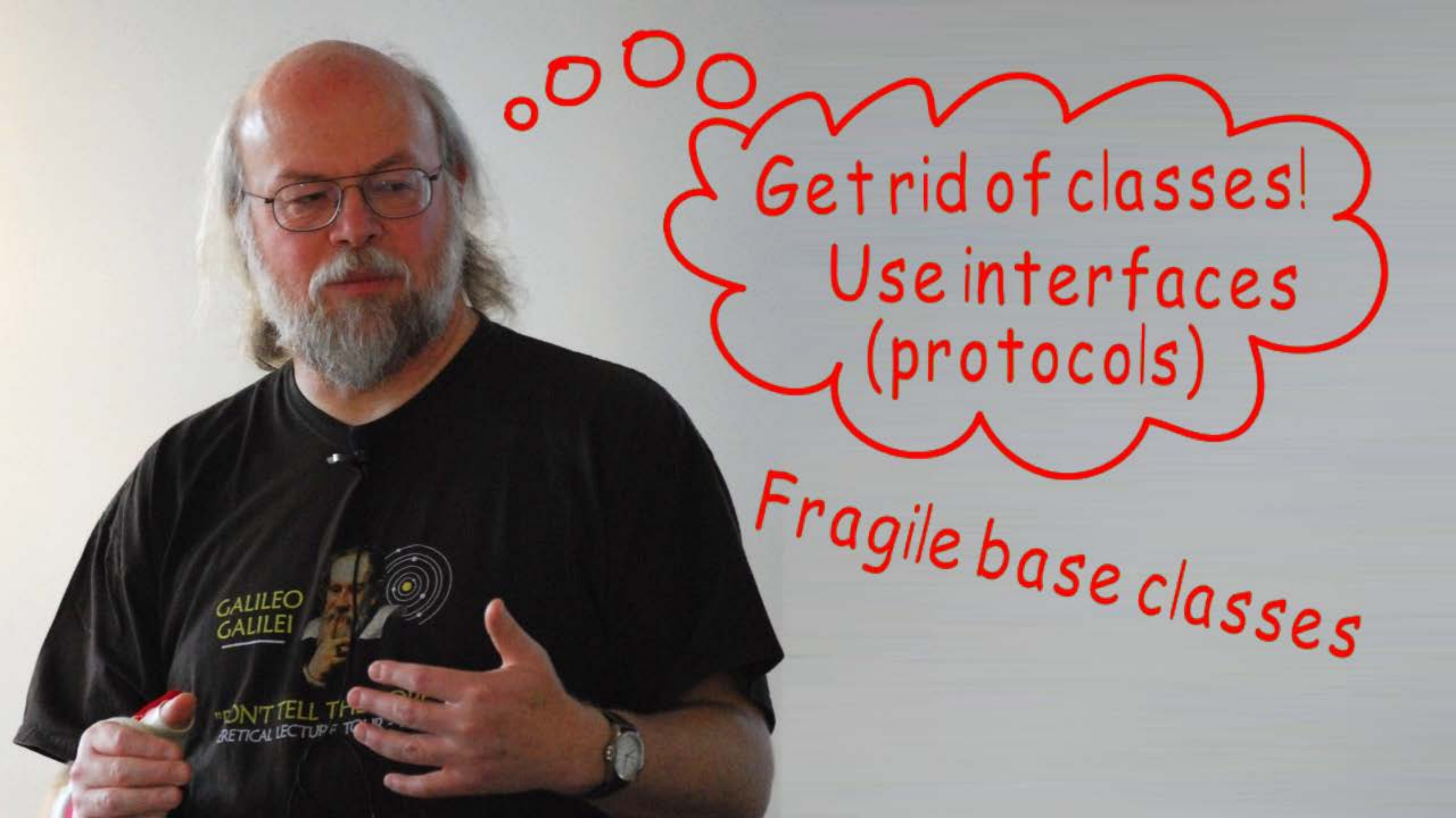
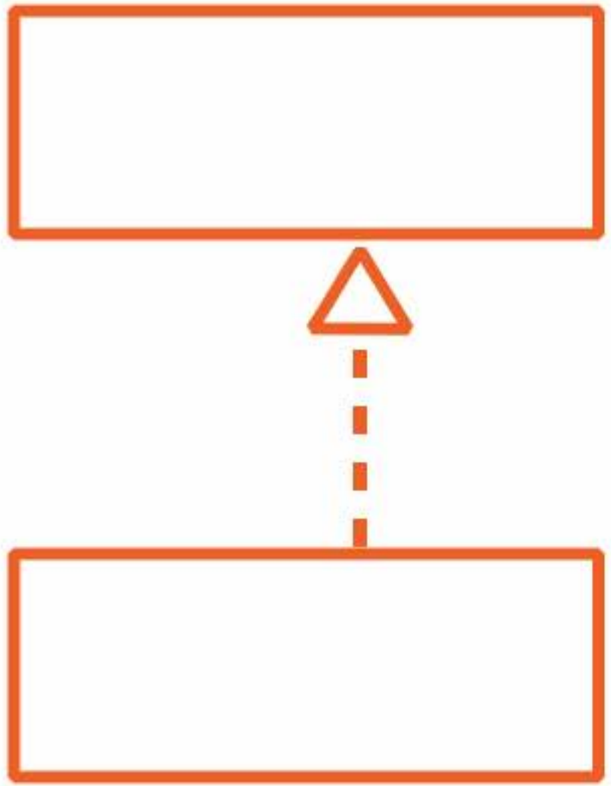
Protocols

Allen Holub

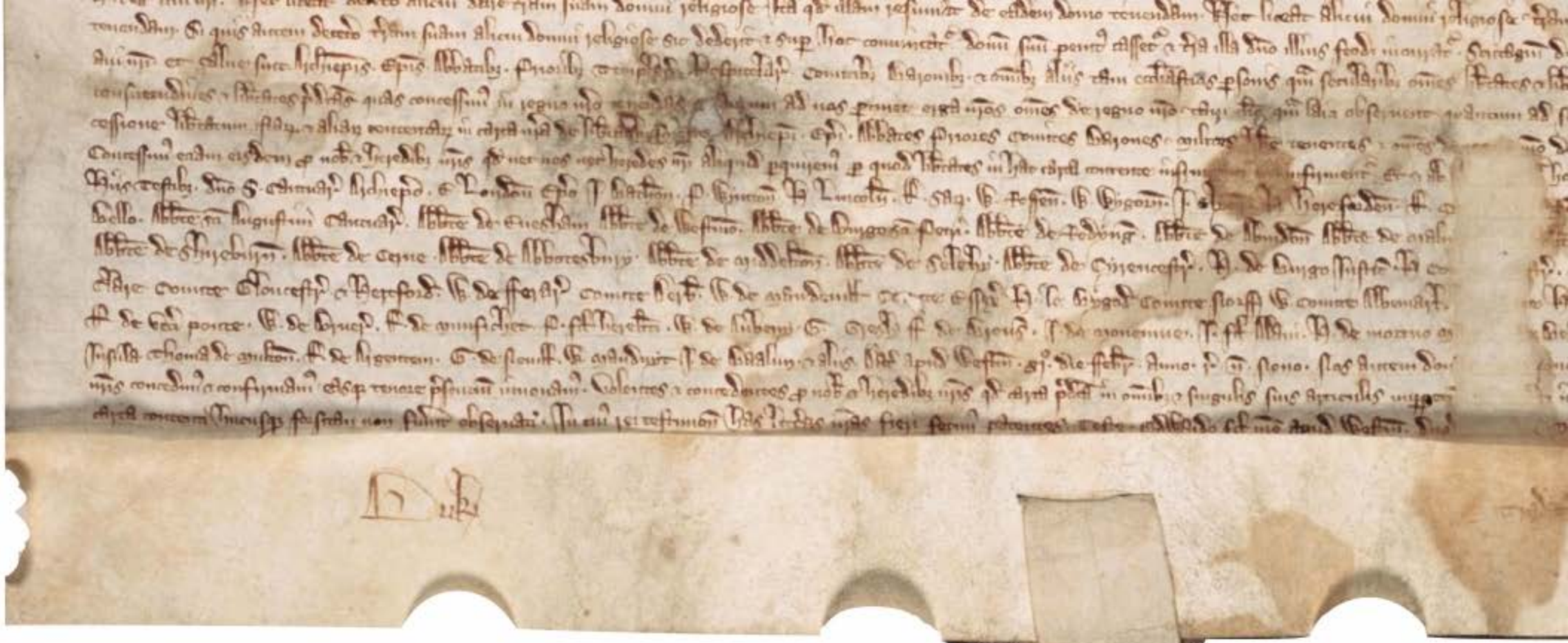
http://holub.com | Allen Holub | @allenholub


Get rid of classes!
Use interfaces
(protocols)
Fragile base classes
GALILEO
GALILEI



protocol == interface

adopt == implement



A protocol is a contract

```
protocol Representable {
   enum Format { case XML, JSON }
}
```

```
enum Format { case XML, JSON }
protocol Representable {
}
```

```
enum Format { case XML, JSON }
protocol Representable {
   func representation( asType: Format ) -> String
   init(asType: Format, contents: String )
}
class Employee  : Representable   {
   private var name: String = "Fred"
   func representation( asType: Format ) -> String {
     switch( asType ) {
     case .XML:  return "<employee name=\"\(name)\"/>"
     case .JSON: return "\"employee\":{\"name\":\(name)}"
     }
   }
   required init(asType: Format, contents: String) {
     /*…*/
   }
   init( name: String ){ self.name = name }
}
```

```
func doSomething( x: Representable ){
    print("\(x.representation(.XML))")
}

doSomething( Employee("Fred") )
```

```
protocol Cacheable {
    func flush()
}

class CacheableEmployee: Employee, Cacheable {
    func flush() -> String {
        /*...*/
    }
}

func doSomething( x: Representable ){
    print("\(x.representation(.XML))")
}


doSomething( Employee("Fred") )
```

```
protocol Cacheable   {
    func flush()
}

class CacheableEmployee: Employee, Cacheable {
    func flush() -> String {
        /*...*/
    }
}

func doSomething( x: protocol<Representable, Cacheable>){
    print("\(x.representation(.XML))")
    x.flush()
}

doSomething( CacheableEmployee("Fred") )
```

```
protocol Cacheable : Representable {
    static var versionID: Double { get set }
    var        objectID:  String { get }
    init()
    func           flush     () -> String
    mutating func load      (flushId: String )
    static func    setTargets(to:  NSOutputStream, from:NSInputStream)
}
```

```
protocol Cacheable : Representable {
    static var versionID: Double { get set }
    var        objectID:  String { get }
    init()
    func           flush     () -> String
    mutating func load      (flushId: String )
    static func    setTargets(to:  NSOutputStream, from:NSInputStream)
}
```

```
protocol Representable {
  enum Format { case XML, JSON }
  func representation asType: Format)->String
  init(asType: Format, contents: String)
}
```

```
class CacheableEmployee: Employee, Cacheable {
    static var versionID = 1.2
    static var idPool    = 0;
    let        myId      = ++idPool
    var objectID: String {return "CacheableEmployee"
                              "-\(CacheableEmployee.versionID)"+"-\(myId)" }
    required init(){
       super.init(asType:Format.JSON, contents:"")
    }
    required init(asType: Format, contents: String){
       super.init(asType:asType,contents:contents)
    }
    func           load      (flushId: String ) {/*…*/}
    func           flush     () -> String       {/*…*/ return objectID }
    static func setTargets(to: NSOutputStream,from: NSInputStream){}
}
```

```
class SomeClass<T: Cloneable> {
}


class SomeOtherClass<T where T:Cloneable,
                             T:Representable> {
}
```

```
protocol MyProtocol                        { func f()   }
class    AdoptingClass : MyProtocol { func f(){} }

let protocolRef :MyProtocol = AdoptingClass()
```

```
protocol MyProtocol                        { func f()   }
class    AdoptingClass : MyProtocol { func f(){} }

let protocolRef = AdoptingClass() as MyProtocol
```

```swift
import Foundation

protocol MyProtocol                    { func f()   }
class    AdoptingClass : MyProtocol { func f(){} }

let protocolRef = AdoptingClass() as MyProtocol


let anything:Any = AdoptingClass()

if( anything is MyProtocol ){
    /*…*/
}

if let implementsProtocol = anything as? MyProtocol {
    /*…*/
}

let myProtocolObject = anything as! MyProtocol
```

```
protocol SupportsReplace {
    func replaceWith(other: Self)
}

class ReplaceableEmployee : Employee, SupportsReplace {
    func replaceWith( other: ReplaceableEmployee ) {
        name = other.name
    }
}

func overwrite(original: SupportsReplace, with: SupportsReplace){
   original.replaceWith( with )
}

let fred   = ReplaceableEmployee(name:"Fred")
let barney = ReplaceableEmployee(name:"Barney")

overwrite( fred, with: barney )
```

```
protocol SupportsReplace {
    func replaceWith(other: Self)
}

class ReplaceableEmployee : Employee, SupportsReplace {
    func replaceWith( other: ReplaceableEmployee ) {
        name = other.name
    }
}

func overwrite<T:SupportsReplace>(original: T, with: T ){
   original.replaceWith( with )
}



let fred   = ReplaceableEmployee(name:"Fred")
let barney = ReplaceableEmployee(name:"Barney")

overwrite( fred, with: barney )
```

```
protocol Cloneable: class {
    func clone() -> Self
}


class CloneableEmployee: Employee, Cloneable {
    // No initializers specified, so we inherit all of them

    func clone() -> Self {
        return self
    }
 }
```

```
protocol Cloneable: class {
    func clone() -> Self
}


final class CloneableEmployee: Employee, Cloneable {
    // No initializers specified, so we inherit all of them

    func clone() -> CloneableEmployee  {
       return  CloneableEmployee("Fred")
    }
 }
```

```
protocol Cloneable: class {
    func clone() -> Cloneable
}


class CloneableEmployee: Employee, Cloneable {
    // No initializers specified, so we inherit all of them

    func clone() -> Cloneable {
        return CloneableEmployee("Fred")
    }
 }


let wilma         = CloneableEmployee(name:"Wilma")
let cloneOfWilma = wilma.clone() as! CloneableEmployee
```

```
protocol Cloneable: class {
    func clone() -> Any
}


class CloneableEmployee: Employee, Cloneable {
    // No initializers specified, so we inherit all of them

    func clone() -> Any     {
        return CloneableEmployee("Fred")
    }
 }


let wilma         = CloneableEmployee(name:"Wilma")
let cloneOfWilma = wilma.clone() as! CloneableEmployee
```

protocol Generic<>

```
protocol Container {
    typealias T
    mutating func append(item: T)
}

class Queue : Container {
    typealias T = String
    var data: [String] = []
    func append( item:String ){ data.append(item) }
}
```

```
protocol Container {
    typealias T
    mutating func append(item: T)
}

class Queue : Container {
    typealias T = String
    var data: [String] = []
    func append( item:String ){ data.append(item) }
}
```

```
protocol Container {
    typealias T
    mutating func append(item: T)
}

class Queue : Container {
    typealias T = String
    var data: [String] = []
    func append( item:String ){ data.append(item) }
}

extension Array: Container {}

func allItemsMatch
    <C1:Container, C2:Container where C1.T==C2.T, C1.T:Equatable>
    (first: C1, second: C2) -> Bool {
    /*…*/
}
```

@objc

```swift
import Foundation

enum        SwiftEnum     {/*…*/}
@objc enum ObjcEnum: Int {case X,Y}


protocol   SwiftProtocol {/*…*/}
@objc protocol ObjcProtocol {
    func f (x: Int)
}


class ObjcSubclass: ObjcProtol {
    @objc func f( x: Int ){}
}
```

```
import Foundation

enum        SwiftEnum      {/*…*/}
@objc enum ObjcEnum: Int {case X,Y}


protocol    SwiftProtocol {/*…*/}
@objc protocol ObjcProtocol {
    func f (x: Int)
}


class ObjcSubclass: NSObject,
                    ObjcProtol {
    func f( x: Int ){}
}
```

```swift
import Foundation

enum        SwiftEnum      {/*…*/}
@objc enum ObjcEnum: Int {case X,Y}


protocol    SwiftProtocol {/*…*/}
@objc protocol ObjcProtocol {
    func f (x: Int)
}


class ObjcSubclass: NSObject,
                    ObjcProtol {
    func f( x: Int ){}
}
```

```swift
@objc protocol ObjcProtocol {
  func f ( x: Int )   // String, etc.
  func f ( x: AnyObject )
  func f ( x: ObjcEnum )
  func f ( x: [Int] )
  func f ( x: [String:Int] )
  func f ( x: ObjcProtocol )
  func f ( x: ()->() )
  func f () -> (Int)->Int
  func >= (left:Int,right:Int)->Bool
}
protocol MyProtocol : ObjcProtocol {
  typealias T
  func f ( x: Any )
  func f ( x: SwiftEnum )
  func f ( genericT: T )
  func f ( optional: Int? )
  func f ( x: SwiftProtocol )
  func f ( inout x: Int )
  func f ( tuple:(Int,Int) )
  func f ( x: Self )
}
```

```
import Foundation

@objc protocol HasOptionalMembers {
    optional var  optVar:Int   {get}
    optional func optMethod()->Int
    func doSomething()
}
```

```
import Foundation

@objc protocol HasOptionalMembers {
    optional var  optVar:Int   {get}
    optional func optMethod()->Int
    func doSomething()
}

class MyClass: HasOptionalMembers {
    @objc var  optVar:Int         {return 0}
    @objc func optMethod()->Int   {return 0}
    @objc func doSomething
```

```
import Foundation

@objc protocol HasOptionalMembers {
    optional var  optVar:Int   {get}
    optional func optMethod()->Int
    func doSomething()
}

class MyClass: NSObject, HasOptionalMembers {
    var  optVar:Int          {return 0}
    func optMethod()->Int  {return 0}
    func doSomething()->() {}
}
```

```swift
import Foundation

@objc protocol HasOptionalMembers {
    optional var  optVar:Int   {get}
    optional func optMethod()->Int
    func doSomething()
}

class MyClass: NSObject, HasOptionalMembers {
    func doSomething()->() {}
}

let opt:HasOptionalMembers = MyClass()
if let result1  = opt.optMethod?() {/*…*/}
if let v1       = opt.optVar       {/*…*/}
let    v2: Int! = opt.optVar
```

```
protocol TextRepresentable {
    func asText() -> String
}

class Person {
  /*…*/
}

extension Person: TextRepresentable {
  func asText() -> String {return "Allen"}
}
```

```
protocol TextRepresentable {
    func asText() -> String
}

class Person {
   func asText() -> String {return "Allen"}
   /*…*/
}

extension Person: TextRepresentable {
}
```

```
public protocol MyCollection {
    typealias T
    init( _ args: T... )
    func elements() -> [T]
}
```

```
public protocol MyCollection {
    typealias T
    init( _ args: T... )
    func elements() -> [T]
}
public class IntCollection : MyCollection {
    private var contents: [Int] = []
    public required init( _ args: Int... ){
        for i in args{ contents.append(i) } }
    public func elements() -> [Int]{ return contents }
}
```

```
public protocol MyCollection {
    typealias T
    init( _ args: T... )
    func elements() -> [T]
}
public class IntCollection : MyCollection {
    private var contents: [Int] = []
    public required init( _ args: Int... ){
        for i in args{ contents.append(i) } }
    public func elements() -> [Int]{ return contents }
}

let obj = IntCollection(1,2,3)

public extension MyCollection where T: SignedIntegerType {
}
```

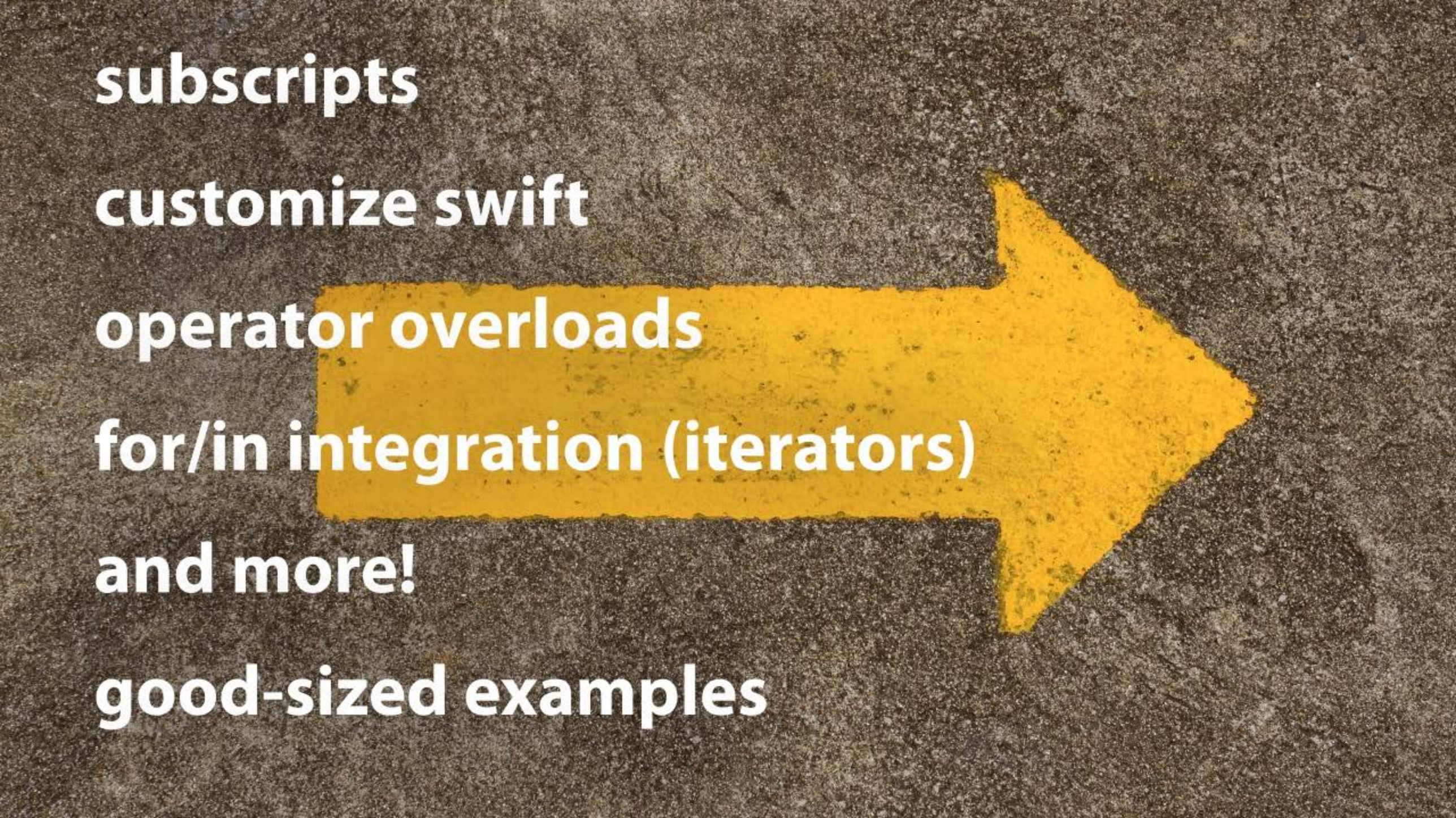
```swift
public protocol MyCollection {
    typealias T
    init( _ args: T... )
    func elements() -> [T]
}
public class IntCollection : MyCollection {
    private var contents: [Int] = []
    public required init( _ args: Int... ){
        for i in args{ contents.append(i) } }
    public func elements() -> [Int]{ return contents }
}

let obj = IntCollection(1,2,3)

public extension MyCollection where T: SignedIntegerType {
    public func sum() -> T {
        var total: T = 0
        for i in elements() { total += i }
        return total
    }
}
obj.sum()
```

Don't have to implement things I don't need!

Do exactly what's required, and nothing more!

Don't need fake stub implementations to satisfy the protocol

Build only what's required


subscripts
customize swift
operator overloads
for/in integration (iterators)
and more!
good-sized examples