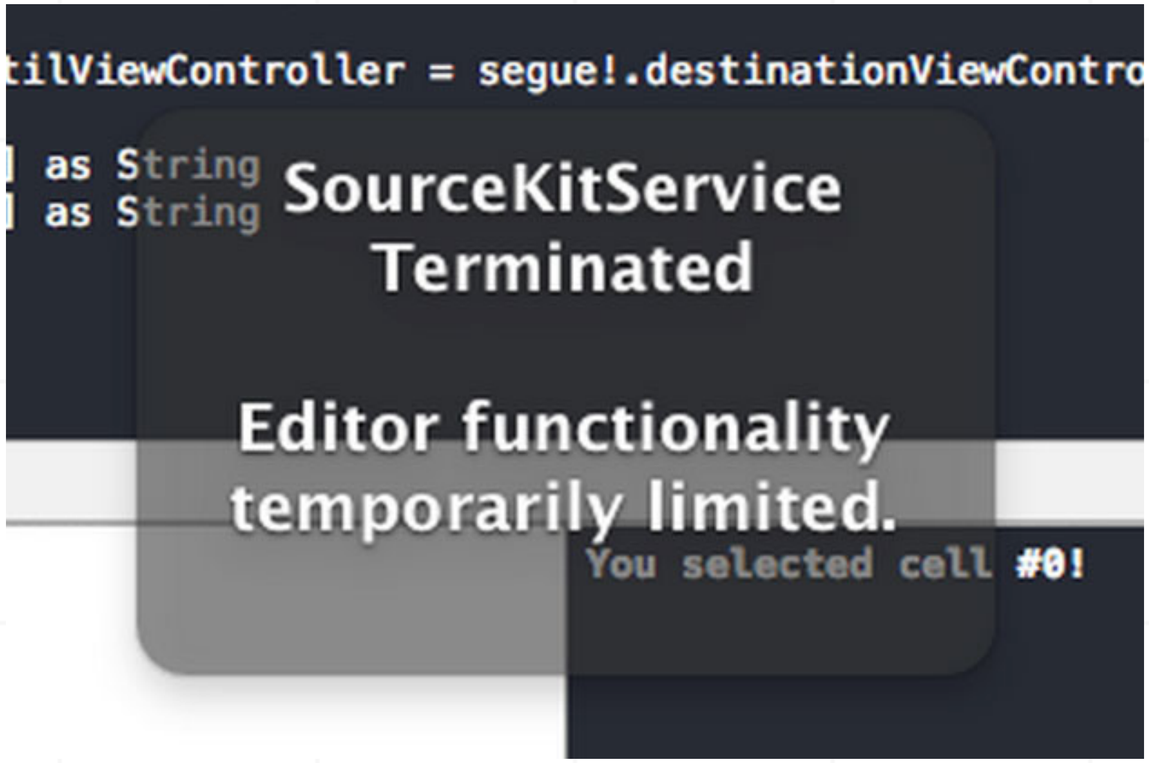
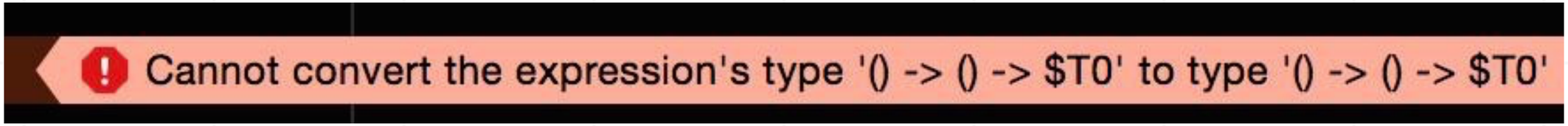
Why Swift?

A practical intro to thinking functionally


Store
Mac
iPhone
Watch
iPad
iPod
iTunes
Support
Swift.
What part of unsafeBitCast(Int(
userInfo[UIKeyboardAnimationCurveUserInfoKey]
as NSNumber), UIViewAnimationCurve.self) don't
you understand?


tilViewController = segue!.destinationViewContro
as String
as String
SourceKitService
Terminated
Editor functionality
temporarily limited.
You selected cell #0!


Cannot convert the expression's type '() -> () -> $T0' to type '() -> () -> $T0'

Why Swift? We ❤️ Objective C!

- Optionals
- `Result<T>`
- `Deferred<T>`

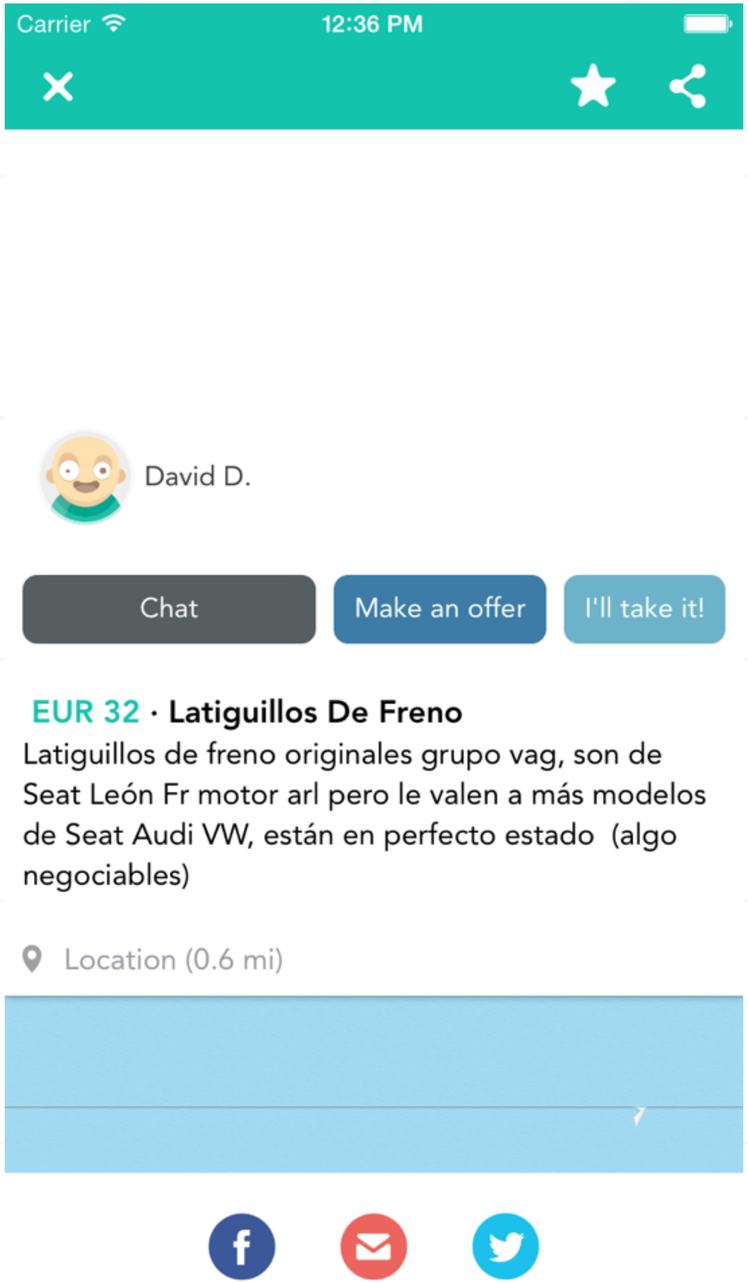
Optionals: they’re a PITA

```objc
- (void)storeLoginForURLString:(NSString *)URLString loginDetails:(NSDictionary *)loginDetailsDict
    passwordGenerationOptions:(NSDictionary *)passwordGenerationOptions forViewController:(UIViewController *)
    viewController sender:(id)sender completion:(void (^)(NSDictionary *, NSError *))completion;
{
    NSAssert(URLString != nil, @"URLString must not be nil");
    NSAssert(loginDetailsDict != nil, @"loginDetailsDict must not be nil");
    NSAssert(viewController != nil, @"viewController must not be nil");

    if (![self isSystemAppExtensionAPIAvailable]) {
        NSLog(@"Failed to storeLoginForURLString, system API is not available");
        if (completion) {
            completion(nil, [OnePasswordExtension systemAppExtensionAPINotAvailableError]);
        }

        return;
    }
```

```objc
@interface Image : NSObject

@property (nonatomic, copy) NSString *idImage;
@property (nonatomic, copy) NSString *idPicture;
@property (nonatomic, assign) NSInteger originalHeight;
@property (nonatomic, assign) NSInteger originalWidth;
@property (nonatomic, copy) NSString *xlargeUrl;
@property (nonatomic, copy) NSString *bigUrl;
@property (nonatomic, copy) NSString *mediumUrl;
@property (nonatomic, copy) NSString *smallUrl;
@property (nonatomic, copy) NSString *type;
@property (nonatomic, copy) NSString *averageHexColor;

@end
```

```objc
@interface User : NSObject

@property (nonatomic, copy) NSString *idUser;
@property (nonatomic, copy) NSString *uuid;
@property (nonatomic, copy) NSString *emailAddress;
@property (nonatomic, copy) NSString *microName;
@property (nonatomic, strong) SHImage *image;
@property (nonatomic, copy) NSString *idLanguage;
@property (nonatomic, copy) NSString *userName;
@property (nonatomic, copy) NSString *userLastName;
@property (nonatomic, strong) Location *location;
@property (nonatomic, strong) StatsUser *stats;
@property (nonatomic) BOOL emailSubscribed;
@property (nonatomic, copy) NSString *birthDay;
@property (nonatomic, copy) NSString *gender;
@property (nonatomic, copy) NSArray *favouriteCategories;

@end
```

```
2015-04-02 12:28:56.882 Wallapop[3066:91913] ***
Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '*** -[__NSArrayM
insertObject:atIndex:]: object cannot be nil'
```


Carrier
12:36 PM
David D.
Chat
Make an offer
I'll take it!
EUR 32 · Latiguillos De Freno
Latiguillos de freno originales grupo vag, son de Seat León Fr motor arl pero le valen a más modelos de Seat Audi VW, están en perfecto estado (algo negociables)
Location (0.6 mi)

```swift
public struct Image {
    public let pictureId : Int
    public let type : String
    public let originalWidth : Int
    public let originalHeight : Int
    public let smallURL : String
    public let mediumURL : String?
    public let bigURL : String?
    public let xlargeURL : String?
    public let averageHexColor : String?
}
```

```swift
public struct User {

    public enum Gender {
        case Male
        case Female
        case Unknown
    }

    public let userID : Int
    public let microname : String
    public let image : Image?
    public let gender : Gender
    public let location : UserLocation
    public let statsUser : UserStats
}
```

```swift
public struct Product {
    public let productID : Int
    public let title : String
    public let description : String
    public let categories : [ItemCategory]
    public let mainImage : Image
    public let images : [Image]
    public let sellerUser : User
    public let price : Int
    public let currency : Currency
    public let modifiedDate : NSDate
    public let publishDate : NSDate
    public let removed : Bool
    public let banned : Bool
    public let itemURL : String
    public let itemActionsAllowed : ActionsAllowed
    public let itemFlags : ItemFlags
}
```

```swift
public struct Currency {
    public let code : String
    public let symbol : String
    public let fractionDigits : Int
}

let currency = Currency(code: nil, symbol: "€", fractionDigits: 2)
```

Cannot invoke initializer for type 'Currency' with an argument list of type '(code: nil, symbol: String, fractionDigits: Int)'

As a physical manifestation of a logical system, computers are faced with the intractable problem of how to represent nothing with something

Mattt Thompson

Benefits of optionals

- Reduces the cognitive load when dealing with objects and APIs you did not write
  - No more maintenance code like Assertions when an API is called incorrectly or an object is initialised incorrectly

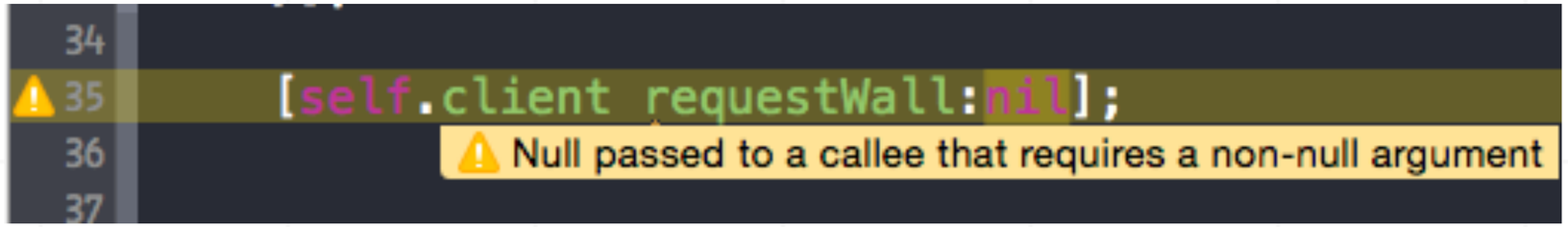
Drawbacks of optionals

- PITA to write sometimes: use Undefined
    - https://github.com/weissi/swift-undefined

- Interaction with Cocoa APIs is weird sometimes
    - Things are getting better with every release

Objective C improvements

- Apple introduced `__nullable` and `__nonnull`



Released with Xcode 6.3

`Result<T>`: monads for the rest of us

```
- (void)startRequest
{
    [self requestListProductsWithPage:self.currentPagination
        completionBlock:^(NSArray *array, NSError *error) {
        [self hideLoadingView];
        if (array) {
            if (self.initialPage) {
                [self.listProductsVC stopActivityOnPullToRefresh];
                [self.listProductsVC updateDataSource:array];
            } else {
                [self.listProductsVC addNewElementsToDataSou
                [self addProductRequestPageAnalyticsWit
            }
        }

        self.initialPage = NO;
        self.executingRequest = NO;
    }];
}
```

```
NSError *err = nil;
CGFloat result = [NMArithmetic divide:2.5 by:3.0
error:&err];

if (err) {
    NSLog(@"%@", err)
} else {
    [NMArithmetic doSomethingWithResult:result]
}
```

Result<T>

- Allows us to express with clarity what are the expected results from an function:

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

https://www.youtube.com/watch?v=LqwrUmuodyY

```
func divide (dividend : Float, divisor : Float)
    -> Result<Float>

let result = divide(dividend :2.5, divisor:3)

switch result {
case .Success(let quotient):
     doSomethingWithResult(quotient)
case .Failure(let error):
     print(error)
}
```

```
func testWallPage() {
    let exp = expectationWithDescription("")

    MMBarricade.selectResponseForRequest(WallSetName, withName: "Success")

    client.fetchWall().uponQueue(dispatch_get_main_queue()) { result in

        exp.fulfill()

        switch result {
        case .Success(let page):
            XCTAssert(page.contents.count > 0, "")
        case .Failure(let error):
        }
    }

    waitForExpectationsWithTimeout(5.0, handler: nil)
}
```

'case' label in a 'switch' should have at least one executable statement 2

Result<T>: Chaining

```
func map<U>(f: T -> U) -> Result<U> {
  switch self {
    case .Failure(let error): return .Failure(error)
    case .Success(let value): return .Success(f(value))
  }
}
```

Result<T>: Chaining

```
func fetchImageData(url : NSURL) -> Result<NSData>
func processImageData(data : NSData) -> UIImage

let imageResult = fetchImageData().map(processImageData)
```

Result<T>: Chaining

Result<T>: Chaining

```
func bind<U>(f: T -> Result<U>) -> Result<U> {
    switch self {
    case .Failure(let error): return .Failure(error)
    case .Success(let value): return f(value)
    }
}
```

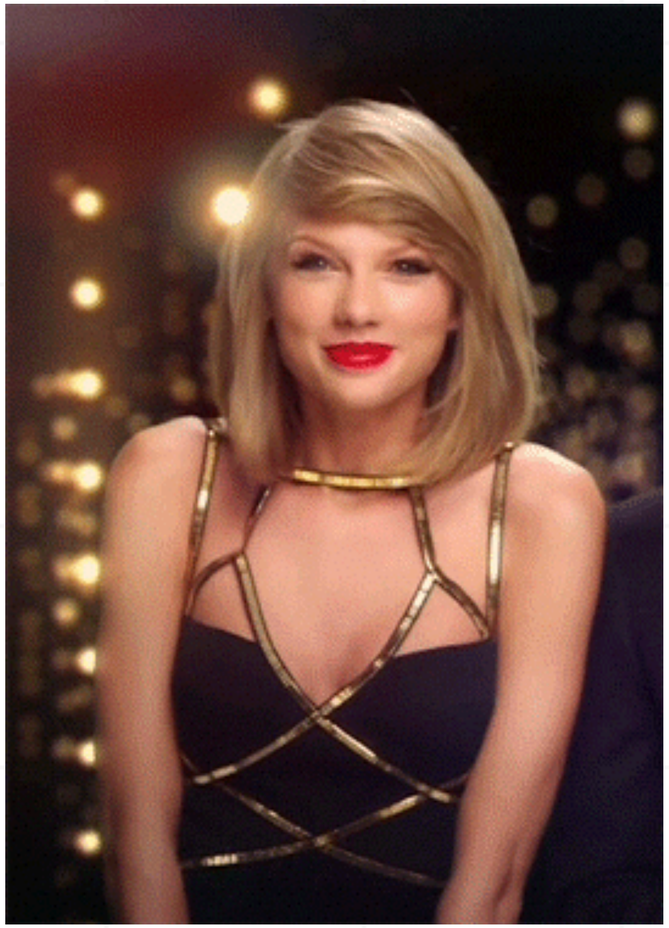
```
func fetchWallData () -> Result<NSData> {
    let mainBundle = NSBundle.mainBundle()
    guard let url = mainBundle.URLForResource("wall", withExtension: "json") else {
        return .Failure(NSError())
    }

    guard let data = NSData(contentsOfURL: url) else {
        return .Failure(NSError())
    }

    return .Success(data)
}
```

```
func parseWallData(data : NSData) -> Result<WallPage> {

    let jsonObject : AnyObject
    do {
        jsonObject = try NSJSONSerialization
            .JSONObjectWithData(data, options: .MutableContainers)
    } catch let error {
        return .Failure(error as NSError)
    }

    guard let jsonResult = jsonObject as? [String :  AnyObject] else {
            return .Failure(NSError())
    }

    guard let page = parseWallPage(jsonResult) else {
        return Result(NSError())
    }

    return .Success(page)
}
```

Result<T>: Chaining

```
func fetchWallData () -> Result<NSData>
func parseWallData(data : NSData) -> Result<WallPage>

let pageResult = fetchWallData().bind(parseWallData)
```

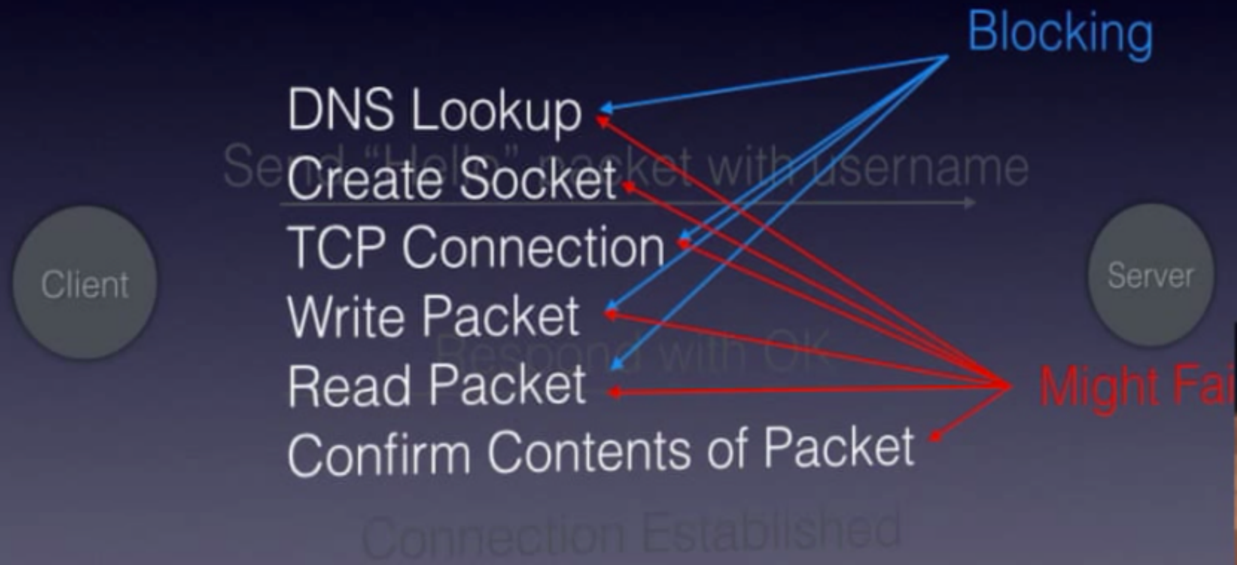
Happy Path Programming!!!



`Deferred<T>`: functional concurrency

Networking Example

"Just" a "Simple" Handshake



Deferred<T>

```
class Deferred<T> {
    init()
    init(value:T)

    var isFilled: Bool
    func fill (value : T)

    func peek () -> T
    func upon (block : T -> ())
}
```

Deferred<T>

```
extension Deferred {
    func map<U>(f: T -> U)
        -> Deferred<U>

    func bind<U>(f: T -> Deferred<U>)
        -> Deferred<U>
}
```

`Deferred<T>`

- Can be used trivially whenever a completionBlock was being used.
  - ✓ When calling **upon**, you can schedule a closure on any GCD queue, and, unlike completionBlocks in ObjC, multiple closures can be scheduled to be executed.

Deferred<T>

```
func both<U>(other: Deferred<U>) -> Deferred<(T,U)>
func all<T>(deferreds: [Deferred<T>]) -> Deferred<[T]>
func any<T>(deferreds: [Deferred<T>]) -> Deferred<Deferred<T>>
```

Deferred<T>

```
func fetchProductsInContext(context : NSManagedObjectContext)
    -> Deferred<[Product]> {

    let deferred = Deferred<[Product]>()

    dispatch_async(queue) {
        let products = fetchProductsInContextSync(context)
        deferred.fill(products)
    }

    return deferred
}
```

```
- (void)doSomethingAsyncWithFoo:(id)foo handler:(dispatch_block_t)handler
{
    [self.networkFetcher fetchBarWithFoo:foo handler:^(NSError *error, id bar){
        [self doSomethingElseWithBar:bar handler:handler];
    }]
}

- (void)doSomethingElseWithBar:(id)bar handler:(dispatch_block_t)handler
{
    [self.networkFetcher fetchBarWithFoo:foo handler:^(NSError *error){
        handler();
    }]
}
```

```objc
- (void)startCoreDataStack:(WPCStackSetupSuccessHandler)successHandler
                   failure:(WPCStackSetupErrorHandler)failureHandler;
{
    [self.system performBlock:^{

        //  1.- Migrate Stack if required
        NSError *migrationError = ({
            NSError *error;
            [self migrateIfNeeded:&error];
            error;
        });

        //  If migration failed, error out
        if (migrationError) {
            [self.system performDelegateBlock:^{
                failureHandler(WPCMigrationFailed);
            }];
            return;
        }

        //  2.- Set up persistent store
        NSError *persistentStoreError = ({
            NSError *error;
            [self createPersistentStoreCoordinator:&error];
            error;
        });

        //  If setting up store fails, error out
        if (persistentStoreError) {
            [self.system performDelegateBlock:^{
                failureHandler(WPCPersistentStoreCouldntBeCreated);
            }];
            return;
        }

        //  If we've made it this far, everything is working,
        //  switch back to main queue and go notify caller
        [self.system performDelegateBlock:^{

            [self managedObjectContext];
            [self backgroundContext];

            successHandler();
        }];
    }];
}
```

Result + Deferred: ≈>

```
infix operator ≈> { associativity left precedence 160 }

func ≈> <T, U>(lhs : Deferred<Result<T>>,
      rhs : T -> Deferred<Result<U>>) -> Deferred<Result<U>>
```

Deferred<Result<T>>

```
func fetchWallData() -> Deferred<Result<NSData>>
func parseWallPage(data:NSData) -> Deferred<Result<WallPage>>

func requestWallPage () -> Deferred<Result<WallPage>> {
    return fetchWallData() ≈> parseWallPage
}
```

Deferred<Result<T>>

```
let wallRequest = requestWallPage()
wallRequest.upon(dispatch_get_main_queue()) { result in
    switch result {
    case .Success (let box) :
        //Populate collectionView
    case .Failure (let error) :
        //Show failure message
    }
}
```


MAN... I'M PRETTY AWESOME.

Takeways: going forward

⌘*B* > ⌘*R*

Takeaways

- Thinking functionally makes things easier to test
  - More difficult to write, specially after years of OOP

Takeaways

- Swift compiler is like an abusive boss

Takeaways

- If your code compiles, chances are, it works
  - This is true for all Swift code, but even more if you use the types mentioned earlier

Takeaways

- Bridging to ObjC isn’t hard
  - It’s just boring

Takeaways

- It’s not as hard as it looks
  - Taking it one step at a time helps a lot

Takeaways

- OOP suits perfectly for UI implementations.
- OOP suits fine for modeling data
  - Functional is perfect for modeling the transformation of this data

Resources

- John Gallagher - Networking with Monads
- Nick Lockwood - Thoughts on Swift 2 Errors
- Andy Matuschak - Controlling Complexity in Swift
- Wallapop iOS Team - WallaFoundation