



@subdigital



NS *Screencast*

The background is a dark blue gradient with several overlapping, semi-transparent silhouettes of electric guitars. The text 'VIBRATO.FM' is prominently displayed in the center, rendered in a bold, white, sans-serif font with a yellow-to-orange gradient and a 3D effect. The text is slightly curved, following the general shape of the guitars behind it.

VIBRATO.FM

Objective-C

[]

2009





C#

```
BCSItem *item = [[BCSItem alloc] initWithIdentifier:@"id1"];  
  
[item favorite];
```

$(\curvearrowright \circ \square \circ) \curvearrowright \smile \perp \text{---} \perp$

```
NSFileManager *fm = [NSFileManager defaultManager];
NSError *error = nil;
BOOL result = [fm removeItemAtPath:path error:&error];

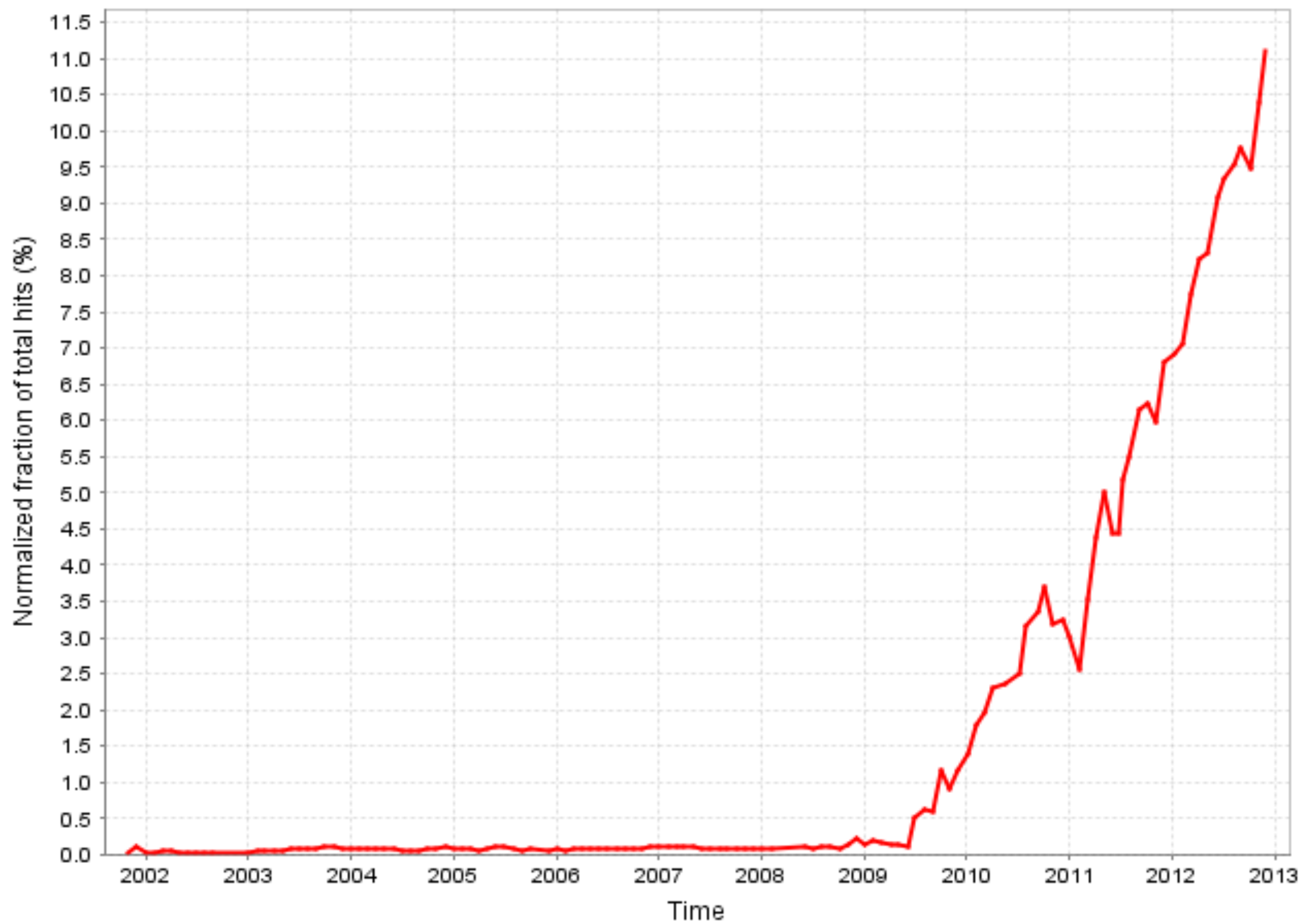
if (!result) {
    NSLog("The error was: %@", [error localizedDescription]);
}
```



Stockholm Syndrome?

Objective-C has some really great features.

TIOBE Programming Community Index Objective-C



2014



Swift is different.

Swift is

- clean & concise
- memory managed
- safe
- functional (?)

Why do we need a safe language?

remember goto fail?

no?

```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail; <-----
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

Hidden backdoor to root in OS X

<https://truesecdev.wordpress.com/2015/04/09/hidden-backdoor-api-to-root-privileges-in-apple-os-x/>



tom robinson

@tlobinson

+ Follow

Fun, OS X privilege escalation, in part due to Objective-C's ability to send messages to "nil"...

truesecdev.wordpress.com/2015/04/09/hid

<https://twitter.com/tlobinson/status/586202234582507520>

“Swift Sucks”

“What the hell were they thinking?”

"Let's implement our own open-source Swift"

A nighttime cityscape featuring a wide, empty road with white lane markings receding into the distance. In the background, several tall skyscrapers are visible, some with lights on. Two bright, out-of-focus light sources, possibly streetlights or distant buildings, are visible on the left side of the road. The sky is a deep blue. Overlaid on the center of the image is a white rectangular box containing the text "CHILL OUT" in a large, white, serif font.

CHILL OUT

Swift is unique

Idiomatic Swift

idiomatic | ,idēə'matik |

adjective

- 1 using, containing, or denoting expressions that are natural to a native speaker: *distinctive idiomatic dialogue.*
- 2 appropriate to the style of art or music associated with a particular period, individual, or group: *a short Bach piece containing lots of idiomatic motifs.*

We are still learning what idiomatic
Swift *is*.

A night sky filled with stars, with the silhouettes of trees visible at the bottom.

So let's take a look at some cases from a Swift
viewpoint.

Lazy Properties with Closures

```
func viewDidLoad() {  
    super.viewDidLoad()  
  
    let config = NSURLSessionConfiguration.defaultSessionConfiguration()  
  
    session = NSURLSession(  
        configuration: config,  
        delegate: self,  
        delegateQueue: nil)  
  
    navigationItem.leftBarButtonItem = UIBarButtonItem(  
        barButtonItemSystemItem: .Bookmarks,  
        target: self,  
        action: "bookmarks:")  
  
    navigationItem.rightBarButtonItem = UIBarButtonItem(  
        barButtonItemSystemItem: .Add,  
        target: self,  
        action: "add:")  
  
    ...  
}
```

```
@interface MyViewController ()

@property (nonatomic, strong) NSURLSessionConfiguration *configuration;

@end

- (NSURLSessionConfiguration *)configuration {
    if (_configuration == nil) {
        _configuration = [NSURLSessionConfiguration defaultSessionConfiguration];
    }
    return _configuration;
}
```



```
lazy var configuration = URLSessionConfiguration.defaultSessionConfiguration()

lazy var session: NSURLSession = {
    return NSURLSession(configuration: self.configuration,
        delegate: self,
        delegateQueue: nil)
}()
```

```
lazy var configuration = URLSessionConfiguration.defaultSessionConfiguration()
```

```
lazy var session: URLSession = {  
    return URLSession(configuration: self.configuration,  
        delegate: self,  
        delegateQueue: nil)  
}()
```

```
lazy var leftBarButtonItem: UIBarButtonItem = {  
    return UIBarButtonItem(  
        barButtonItemSystemItem: .Bookmarks,  
        target: self,  
        action: "bookmarks:")  
}()
```

```
lazy var rightBarButtonItem: UIBarButtonItem = {  
    return UIBarButtonItem(  
        barButtonItemSystemItem: .Add,  
        target: self,  
        action: "add:")  
}()
```

```
func viewDidLoad() {  
    super.viewDidLoad()  
  
    let config = NSURLSessionConfiguration.defaultSessionConfiguration()  
  
    session = NSURLSession(  
        configuration: config,  
        delegate: self,  
        delegateQueue: nil)  
  
    navigationItem.leftBarButtonItem = UIBarButtonItem(  
        barButtonItemSystemItem: .Bookmarks,  
        target: self,  
        action: "bookmarks:")  
  
    navigationItem.rightBarButtonItem = UIBarButtonItem(  
        barButtonItemSystemItem: .Add,  
        target: self,  
        action: "add:")  
  
    ...  
}
```

```
func viewDidLoad() {  
    super.viewDidLoad()  
  
    navigationItem.leftBarButtonItem = leftBarButtonItem  
  
    navigationItem.rightBarButtonItem = rightBarButtonItem  
}
```

Returning errors

```
var error: NSError? = nil
let object = NSJSONSerialization.JSONObjectWithData(data,
    options: nil,
    error: &error
)
```



```
func JSONObjectWithData(  
    data: NSData,  
    options: NSJSONReadingOptions = nil,  
) -> (AnyObject?, NSError?)
```

```
let result = JSONObjectWithData(data)
```

```
result.0 // object?
```

```
result.1 // error?
```

```
let (object, error) = JSONObjectWithData(data)

if let someObject = object {
    // can work with someObject
} else {
    // error
}
```

```
func JSONObjectWithData(  
    data: NSData,  
    options: NSJSONReadingOptions = nil,  
) -> (obj: AnyObject?, error: NSError?)
```

```
let result = JSONObjectWithData(data)
```

```
result.obj // object?
```

```
result.error // error?
```

Sometimes `if let` sucks

Swift Pattern Matching

```
switch(result) {  
    case (.Some(let x), nil):  
        println("x's type is AnyObject")  
    case (nil, .Some(let e)):  
        println("e's type is NSError")  
    default:  
}
```

A more general solution

```
enum Result<T> {  
    case Value(T)  
    case Error(NSError)  
}
```

```
func JSONObjectWithData(  
    data: NSData,  
    options: NSJSONReadingOptions = nil  
) -> Result<AnyObject>
```

```
func JSONObjectWithData(  
    data: NSData,  
    options: NSJSONReadingOptions = nil  
) -> Result<AnyObject> {  
  
    ...  
  
    if let error = parserError {  
        return .Error(error)  
    } else {  
        return .Value(parsedObject)  
    }  
}
```

```
let result = JSONObjectWithData(data)

switch(result) {
    case .Value(let obj):
        println("we parsed the object: \(obj)")

    case .Error(let error):
        println("error parsing json: \(error)")
}
```

(Side Note: Boxing currently required)

Hopefully Apple will fix this soon, but currently you need to box the T value for this to work:

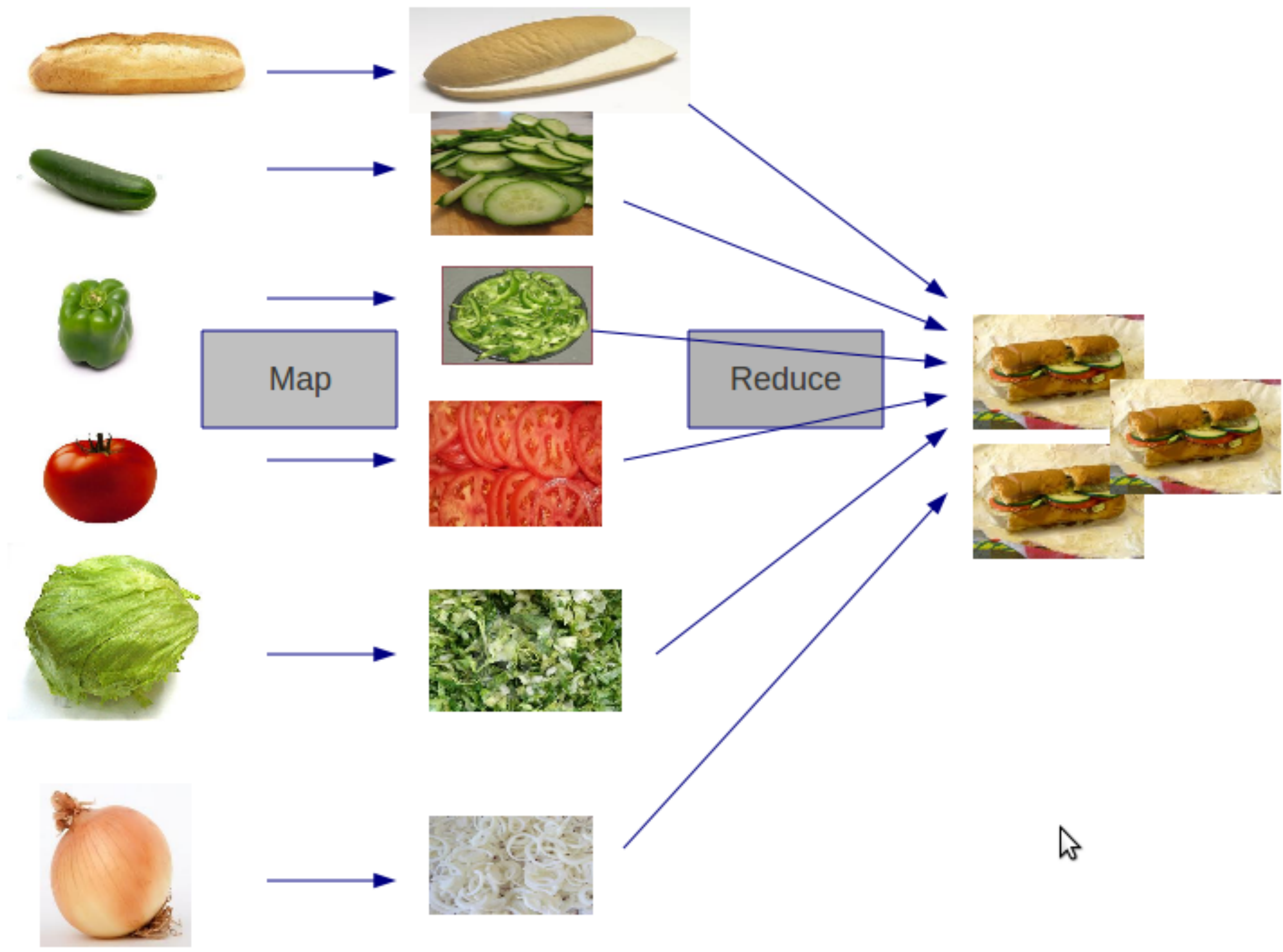
```
class Box<T> {  
    var unbox: T  
    init(_ value: T) {  
        unbox = value  
    }  
}
```

```
enum Result<T> {  
    case Value(Box<T>)  
    case Error(NSError)  
}
```

```
switch result {  
    case .Value(let box):  
        let val = box.unbox  
        ...  
}
```

More on Result<T> later...

Map / Reduce



```
struct Item {  
    let name: String  
    let price: Double  
    let taxable: Bool  
}
```

Given a Shopping cart full of items...

```
let shoppingCart = [  
  Item(name: "Apples",    price: 0.67, taxable: false),  
  Item(name: "Bananas",   price: 1.87, taxable: false),  
  Item(name: "Beer",      price: 6.99, taxable: true),  
  Item(name: "Candy",     price: 4.00, taxable: true),  
  Item(name: "Ice Cream", price: 6.00, taxable: true)  
]
```

Compute the amount of tax you
have to pay...

```
let TaxRate = 0.0825
```

```
let TaxRate = 0.0825  
var tax = 0.0
```

```
let TaxRate = 0.0825
var tax = 0.0

for item in shoppingCart {

}
println(tax)
```

```
let TaxRate = 0.0825
var tax = 0.0

for item in shoppingCart {
    if item.taxable {

    }
}
println(tax)
```

```
let TaxRate = 0.0825
var tax = 0.0

for item in shoppingCart {
    if item.taxable {
        tax += item.price * TaxRate
    }
}
println(tax)
```

How can we express this in terms of
map, filter, and reduce?

map

```
func map<C : CollectionType, T>(
    source: C,
    transform: (C.Generator.Element) -> T) -> [T]
```

```
func map<S : SequenceType, T>(
    source: S,
    transform: (S.Generator.Element) -> T) -> [T]
```

```
let items = [1, 2, 3]
```

```
map(items, { (item: Int) -> Int in  
    return item * 2  
})
```

```
let items = [1, 2, 3]

map(items, { (item) -> Int in
    return item * 2
})
```

```
let items = [1, 2, 3]
```

```
map(items, { item in  
    return item * 2  
})
```

```
map(items, {  
    return $0 * 2  
})
```

```
map(items, { $0 * 2 })
```

```
map(items) { $0 * 2 }
```

```
[1,2,3].map { $0 * 2 }
```

```
[1,2,3].map { $0 * 2 } // => [2, 4, 6]
```

filter

```
func filter<S : SequenceType>(
    source: S,
    includeElement: (S.Generator.Element) -> Bool)
    -> [S.Generator.Element]
```

```
[1,2,3].filter { $0 % 2 == 0 }
```

```
[1,2,3].filter { $0 % 2 == 0 } // => [2]
```

reduce

```
func reduce<S : SequenceType, U>(
    sequence: S,
    initial: U,
    combine: @noescape (U, S.Generator.Element) -> U) -> U
```

```
[1,2,3].reduce(0) {  
  (sum, n) in  
  return sum + n  
}
```

All set?

First get the taxable items

```
let taxable = shoppingCart.filter { $0.taxable }
```

Map items to their prices

```
let taxable = shoppingCart.filter { $0.taxable }  
let taxableAmounts = taxable.map { $0.price }
```

Reduce to a single value (the amount of tax)

```
let taxable = shoppingCart.filter { $0.taxable }  
let taxableAmounts = taxable.map { $0.price }  
  
let tax = taxableAmounts.reduce(0) {  
    (tax, itemPrice) in  
    return tax + (itemPrice * TaxRate)  
}
```

Can simplify the reduce block...

```
let taxable = shoppingCart.filter { $0.taxable }
let taxableAmounts = taxable.map { $0.price }
let itemTaxes = taxable.map { $0 * TaxRate }
let tax = itemTaxes.reduce(0) {
    (tax, itemTax) in
    return tax + (itemTax)
}
```

...into an existing function

```
func +(lhs: Int, rhs: Int) -> Int {  
    return lhs + rhs  
}
```

Look at the Shape of the function...

$$(T, T) \rightarrow U$$

Look at the Shape of the function...

😭 -> T

$(T, T) \rightarrow T$

$$(U, T) \rightarrow U$$

Can just use the + function...

```
let taxable = shoppingCart.filter { $0.taxable }  
let taxableAmounts = taxable.map { $0.price }  
let itemTaxes = taxable.map { $0 * TaxRate }  
let tax = itemTaxes.reduce(0, combine: +)
```

... and chain them together...

```
let tax = shoppingCart
    .filter      { $0.taxable }
    .map         { $0.price }
    .map         { $0 * TaxRate }
    .reduce(0, combine: +)

println(tax)    // 1.40
```

Functions

(no interim state, no loops, no conditionals)

```
func taxFunction(rate: Double)
```

```
func taxFunction(rate: Double) -> ([Item]) -> Double {
```

```
func taxFunction(rate: Double) -> ([Item]) -> Double {  
  
    return { (items) in  
        return items  
            .filter      { $0.taxable }  
            .map          { $0.price }  
            .map          { $0 * rate }  
            .reduce(0, combine: +)  
    }  
  
}
```

```
let texasTax = taxFunction(0.0825)  
let tax = texasTax(items)
```

Flat Map

We know what map is, what is flatten?

```
flatten([[5, 6], [4], [7, 8]]) -> [5, 6, 4, 7, 8]
```

it is often handy to flatten before
mapping

(flat, map)

```
struct Person {  
    var name: String  
    var emails: [String]  
}
```

"Give me a list of all email
addresses"

```
people.map { $0.emails } ???
```

There is no flatten function, but we could write one...

```
func flatten<T>(array: [[T]]) -> [T] {  
    var result = [T]()  
    for item in array {  
        for subItem in item {  
            result.append(subItem)  
        }  
    }  
  
    return result  
}
```

...ore more succinctly, with `reduce`:

```
func flatten<T>(array: [[T]]) -> [T] {  
    return array.reduce([], combine: +)  
}
```

Yay! Swift 1.2 has a flatMap
function :)

```
[[1, 2], [3], [4, 5]].flatMap { $0 }
```

```
// -> [1, 2, 3, 4, 5]
```

```
people.flatMap { $0.emails }
```

flatMap for Result<T> ?

```
enum Result<T> {  
    case Value(Box<T>)  
    case Error(NSError)  
  
    func flatMap<U>(next: T -> Result<U>) -> Result<U> {  
        switch self {  
        case .Value(let box):  
            let val = box.unbox  
            return next(val)  
        case .Error(let err):  
            return .Error(err)  
        }  
    }  
}
```

```
func getValue() -> Result<Int>  
func convertToFloat(i: Int) -> Result<Float>  
func squareRoot(x: Float) -> Result<Float>
```

```
getValue().flatMap(convertToFloat).flatMap(squareRoot)
```

Dealing with Index Paths

*Remember static table views before
Storyboards?*

```
func tableView(tableView: UITableView,
    cellForRowAtIndexPath: NSIndexPath) -> UITableViewCell {

    if indexPath.section == 0 {
        if indexPath.row == 0 {
            //
        } else {
            // ...
        }
    } else if indexPath.section ==
        numberOfSectionsInTableView(tableView) - 1 {
        // ...
    } else {

        // ...

    }

}
```

```
func tableView(tableView: UITableView,  
    cellForRowAtIndexPath: NSIndexPath) -> UITableViewCell {  
  
    switch (indexPath) {  
        case (0, 0): ...  
        case (0, let row): ...  
        case (let section, let row) where isLastSection(section):  
            ...  
        case (let section, let row):  
            ...  
    }  
}
```

Enums for API Endpoints

Consider an API with some paths:

- `/notifications`
- `/users/:username`
- `/categories/:category_id/products/:product_id`

```
enum ApiEndpoints {  
    case Notifications  
    case UserProfile(String)  
    case ProductDetail(Int, Int)  
}
```

```
enum ApiEndpoints {  
    case Notifications  
    case UserProfile(String)  
    case ProductDetail(Int, Int)  
  
    var path: String {  
        switch(self) {  
            case .Notifications: return "/notifications"  
            case .UserProfile(let username):  
                return "/users/\(username.stringByAddingPercentEscapes...)"  
            case .ProductDetail(let categoryId, let productId):  
                return "/categories/\(categoryId)/products/\(productId)"  
        }  
    }  
}
```

```
ApiClient.request(.UserProfile("subdigital")) {  
    ...  
}
```

Swift is still new



"When we learn a new programming language, we may need [to] discard what we thought it should be, and learn what it will be."

— *a Swift student*

thanks!

@subdigital • @nsscreencast