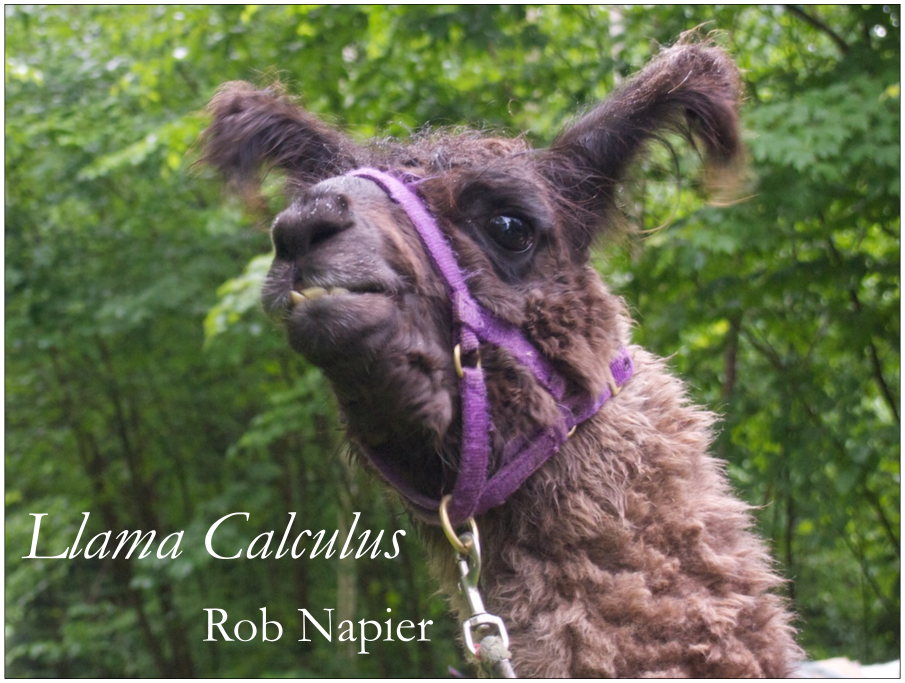
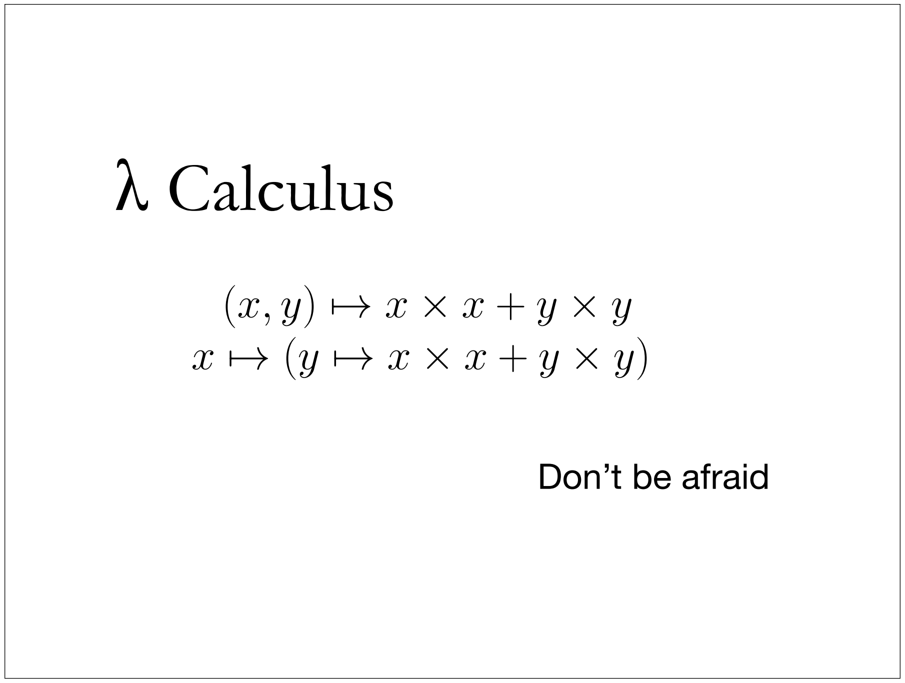


I'm Rob Napier, and today we're going to talk a little bit about functional programming in Swift. I went hiking with llamas recently, and I now know that it's impossible to take a picture of llamas that isn't funny. So here you go.



λ Calculus

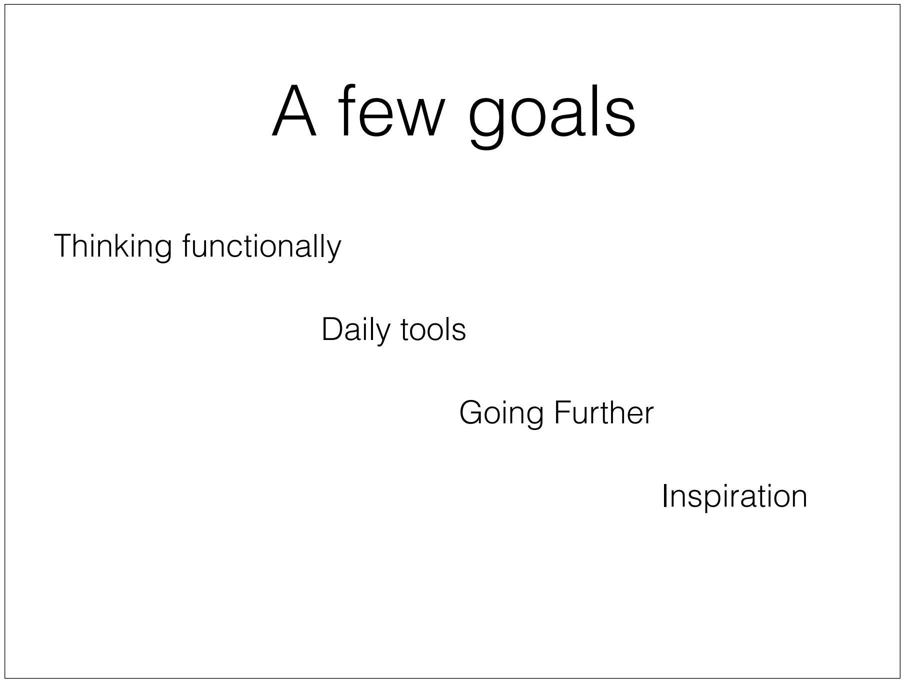
$$(x, y) \mapsto x \times x + y \times y$$
$$x \mapsto (y \mapsto x \times x + y \times y)$$

Don’t be afraid

But we’re not really going to talk about llamas today. It’s a joke folks. I can make it sound like there’s a story, but it’s just a joke. We’re going to talk about lambda calculus. And the truth is we’re not going to talk very much about lambda calculus today either. To keep this under an hour, I wound up having to drop the section of the talk that even touched on this stuff.

We’re really going to talk about how to apply the decades of lessons we’ve learned from functional programming languages to Swift.

By the time we’re done, I hope you see functional programming the same way I see llamas. A bit funny looking, but useful.



A few goals

Thinking functionally

Daily tools

Going Further

Inspiration

Today we're going to cover a mix of the practical and the theoretical. If I do my job well, you'll leave today with a few things you can start using immediately and you'll completely get it right away. Then there will be a few things you can keep in the back of your mind as "hey, I remember there was a tool for that" and you'll study it again later. And finally there are a few things that are here for inspiration, that you may never use, but show what's possible and hopefully get you interested in digging deeper.

Truth in Advertising

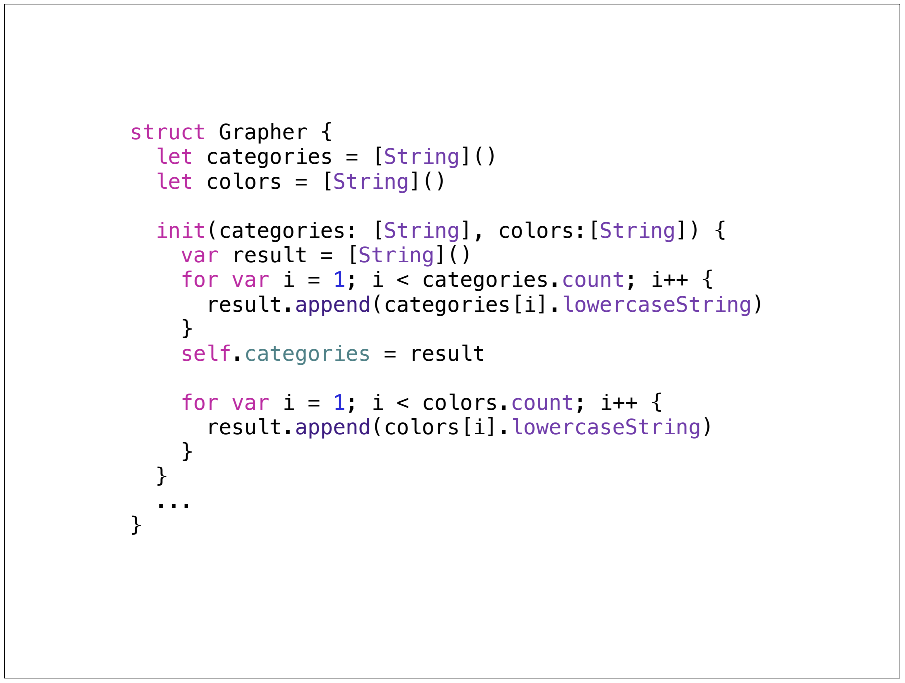
First a slight apology for some false advertising. When I got started on this talk, I planned to do deep into the lambda. Monads, applicative functors, futures. And those can be pretty awesome. But as I dug into it I realized that they're not what most Swift devs really need.

Swift isn't really a functional programming language. It may be someday, but it isn't yet. So to teach all the things I wanted to talk about, I had to sidestep the Swift standard library and basically build a bunch of fancy data structures, with new operators and patterns, and frankly it got rather out of control, and not as useful as I wanted it to be. And it would probably would have taken days to teach it all, and we have an hour.

I started tearing it down to the basics, to what Cocoa devs really need, and what Swift gives us today. And so forgive me, I'm not actually going to talk about some of the things I said I would. I'm not going to talk about asynchronous network handling. And I'm not going to talk about operators. I'm not going to use many fancy words at all.

The Motivating Example

Before solutions, we should talk about problems. Every good marketing pitch, I mean technical talk, starts with a motivating example. So let’s start with some code and see how we might improve it.

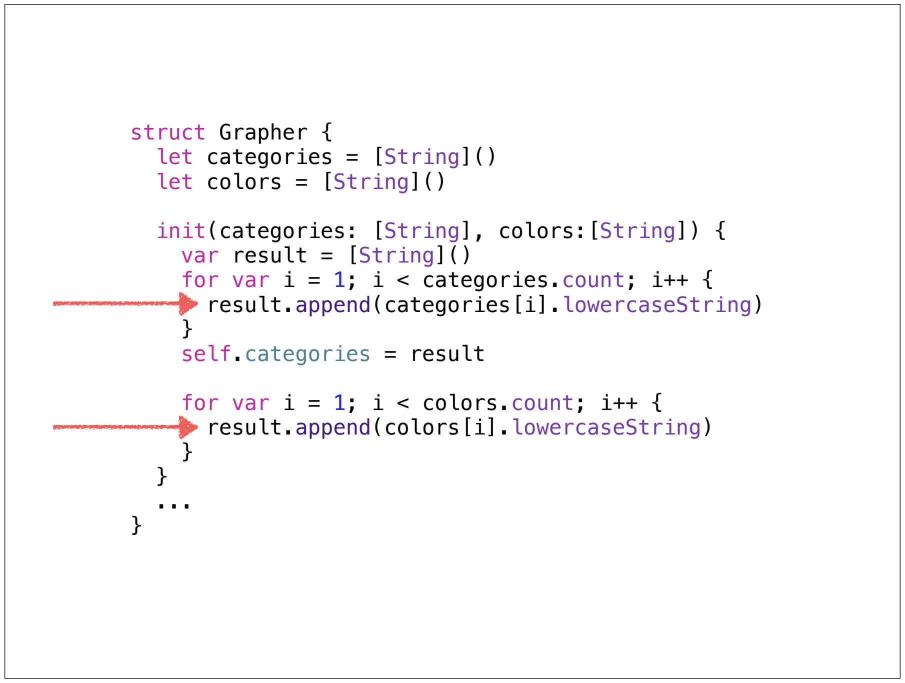


```
struct Grapher {
  let categories = [String]()
  let colors = [String]()

  init(categories: [String], colors:[String]) {
    var result = [String]()
    for var i = 1; i < categories.count; i++ {
      result.append(categories[i].lowercaseString)
    }
    self.categories = result

    for var i = 1; i < colors.count; i++ {
      result.append(colors[i].lowercaseString)
    }
  }
  ...
}
```

Let’s pretend we have this graphing class that takes a bunch of categories and colors and lays them out in a pretty way. Maybe makes sure that adjacent categories have contrasting colors or whatever. We’re just focusing on the init here.

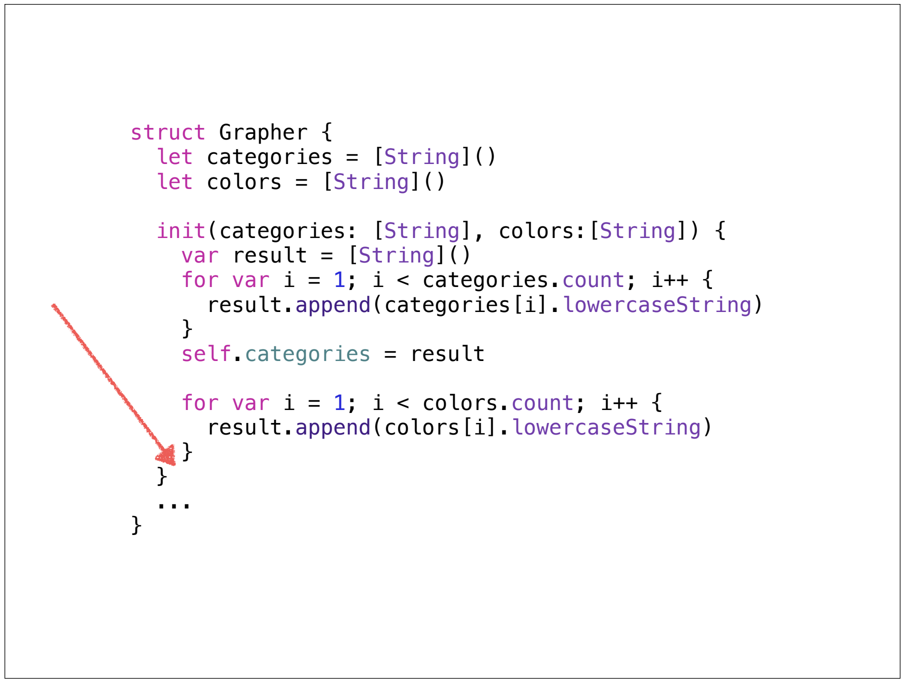
So we can do string comparisons more easily, we want to store everything internally in lowercase. So maybe we write it like this, which is pretty close to code I’ve seen posted to StackOverflow. We iterate over each of our parameters, convert to lowercase, and assign to our property.

Yeah, there are a few bugs here. Let’s take it step by step.



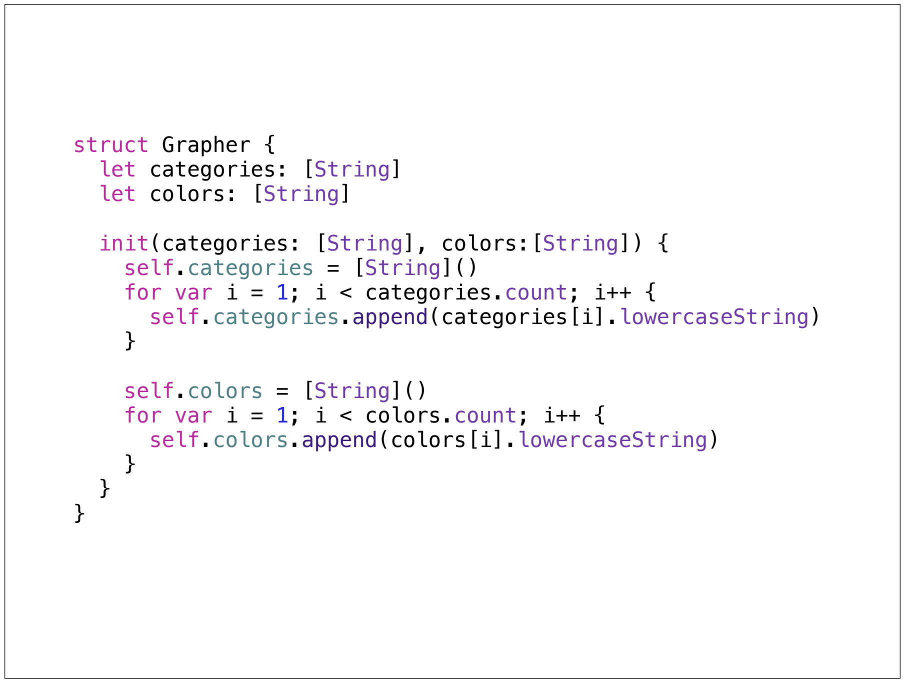
```
struct Grapher {
  let categories = [String]()
  let colors = [String]()

  init(categories: [String], colors:[String]) {
    var result = [String]()
    for var i = 1; i < categories.count; i++ {
      result.append(categories[i].lowercaseString)
    }
    self.categories = result

    for var i = 1; i < colors.count; i++ {
      result.append(colors[i].lowercaseString)
    }
  }
  ...
}
```

First bug. I reused the same temporary array twice without resetting it. So colors is going to start with categories. I’ve seen that bug more times than I like to admit.



```
struct Grapher {
  let categories = [String]()
  let colors = [String]()

  init(categories: [String], colors:[String]) {
    var result = [String]()
    for var i = 1; i < categories.count; i++ {
      result.append(categories[i].lowercaseString)
    }
    self.categories = result

    for var i = 1; i < colors.count; i++ {
      result.append(colors[i].lowercaseString)
    }
  }
  ...
}
```

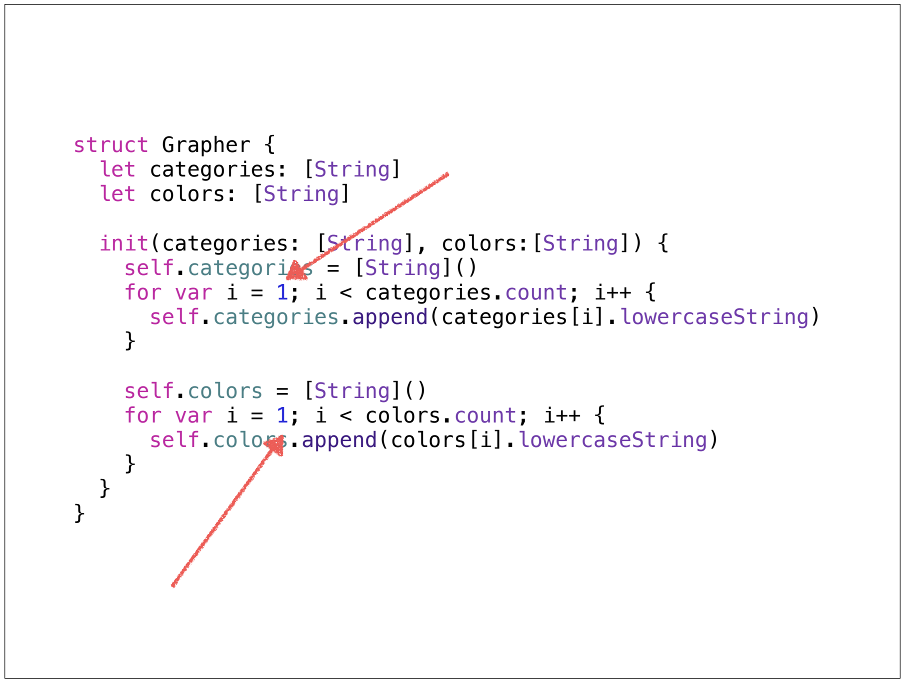
Second bug. I forgot to assign to the colors property at the end. Seen that bug pretty often, too, especially in init. Ten line function, and I'm already two bugs in.



```
struct Grapher {
  let categories: [String]
  let colors: [String]

  init(categories: [String], colors:[String]) {
    self.categories = [String]()
    for var i = 1; i < categories.count; i++ {
      self.categories.append(categories[i].lowercaseString)
    }

    self.colors = [String]()
    for var i = 1; i < colors.count; i++ {
      self.colors.append(colors[i].lowercaseString)
    }
  }
}
```

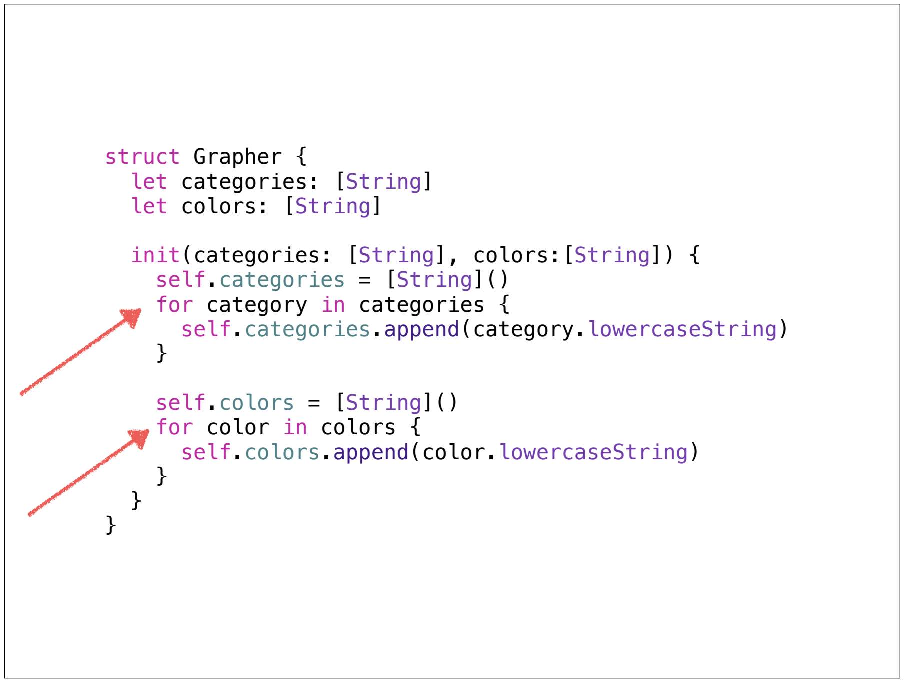
I can fix both pretty easily by getting rid of the temporary variables. Or I could just make sure to use the temporary variables correctly. But they show how it’s easy to make small mistakes. And I need to be careful every time I rewrite this boilerplate code. Functional programing is going to get rid of a lot of this boilerplate.

And there’s one more bug hiding.



```
struct Grapher {
  let categories: [String]
  let colors: [String]

  init(categories: [String], colors:[String]) {
    self.categories = [String]()
    for var i = 1; i < categories.count; i++ {
      self.categories.append(categories[i].lowercaseString)
    }

    self.colors = [String]()
    for var i = 1; i < colors.count; i++ {
      self.colors.append(colors[i].lowercaseString)
    }
  }
}
```

The *for* loop indexed from 1 rather than 0. Had the ending test been wrong, this bug would be really obvious since we'd crash. But this will kind of work. It's even possible the author meant to drop the first element? Probably not. But who konws. You just have to kind of figure it out from the context. And I've seen this bug in live code, too.

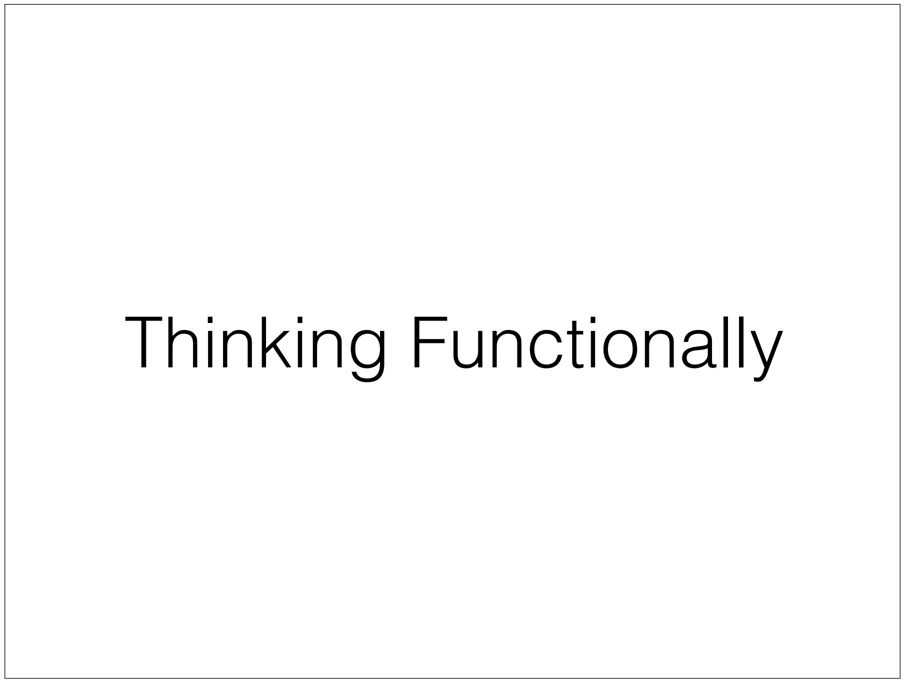


```
struct Grapher {
  let categories: [String]
  let colors: [String]

  init(categories: [String], colors:[String]) {
    self.categories = [String]()
    for category in categories {
      self.categories.append(category.lowercaseString)
    }

    self.colors = [String]()
    for color in colors {
      self.colors.append(color.lowercaseString)
    }
  }
}
```

That bug is exactly why ObjC and Swift have this *for-in* enumeration syntax.

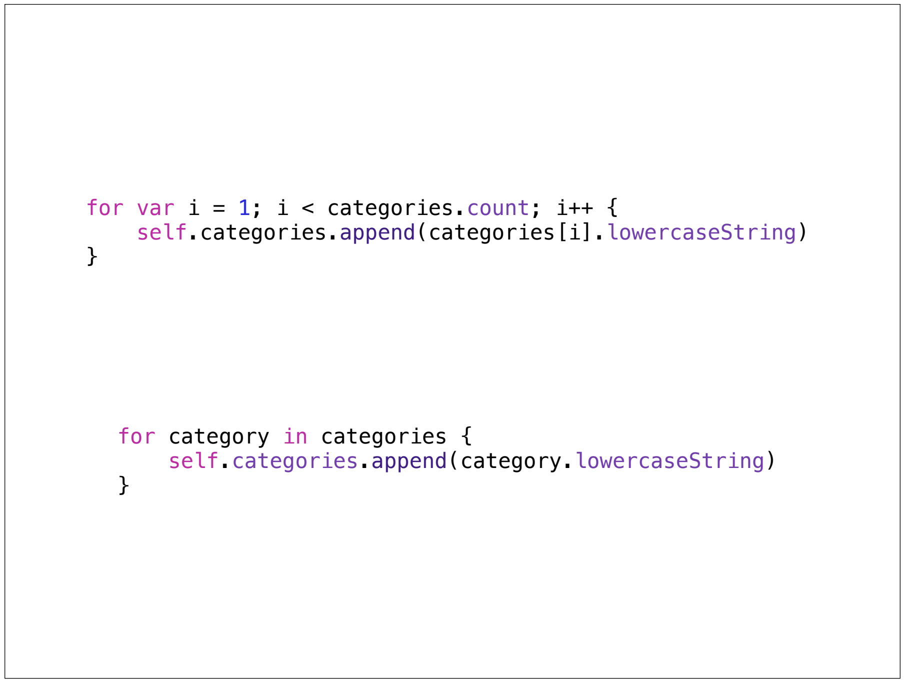
A *for* loop is imperative. A *for-in* loop is our first step towards functional thinking.

What's that mean?



Thinking Functionally

In functional thinking, we don’t assign. We don’t do. We just say what things are and leave the computation details to the system.



```
for var i = 1; i < categories.count; i++ {
    self.categories.append(categories[i].lowercaseString)
}
```

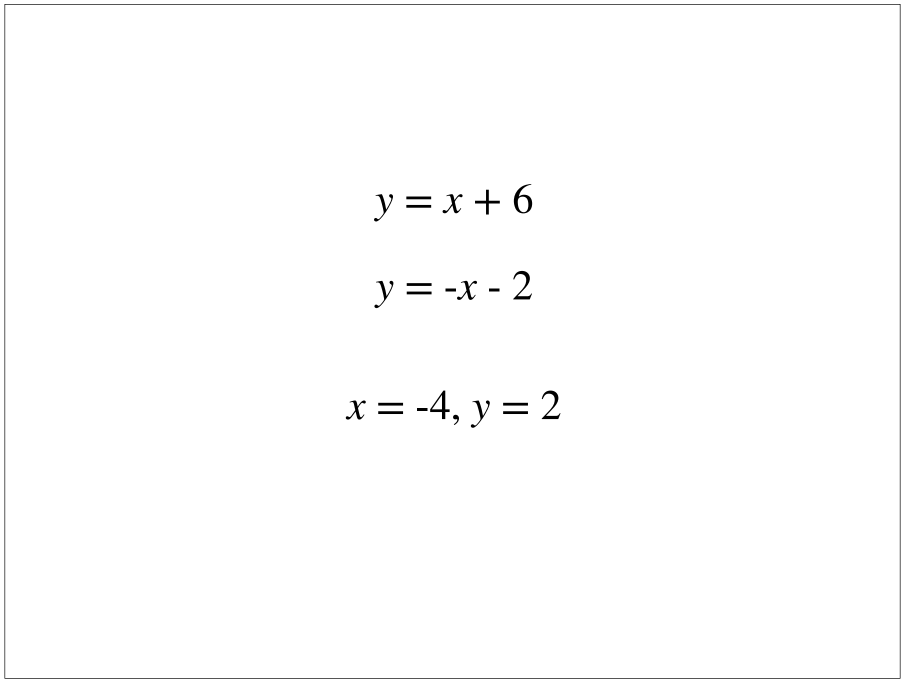
```
for category in categories {
    self.categories.append(category.lowercaseString)
}
```

In the *for* example, we explicitly say how to iterate over the array. And that means that we have to know it's an array and it can be subscripted and that calculating the length is cheap and it's indexed by integers.

<build> In the *for-in* example, we just say "*category* is bound to each of the elements in turn" and ignore the details of how. So it works for Dictionaries, Ranges, Arrays, any Sequence. We've factored out the mechanics of iterating.

If I had to pick one thing that encapsulates functional programming and functional thinking, it's that. The manual *for* loop duplicates a lot of boilerplate iteration logic. Initializing the iteration variable. Testing. Incrementing. Subscripting. The *for-in* loop abstracts away all those details about what it means to "iterate." We have the idea of iterating, and we have the idea of lowercasing, and we glue them together into this bigger idea. That's functional.

The second big idea here is immutability. In the *for* loop, the index *i* is modified in each loop. There is just one *i*



$y = x + 6$

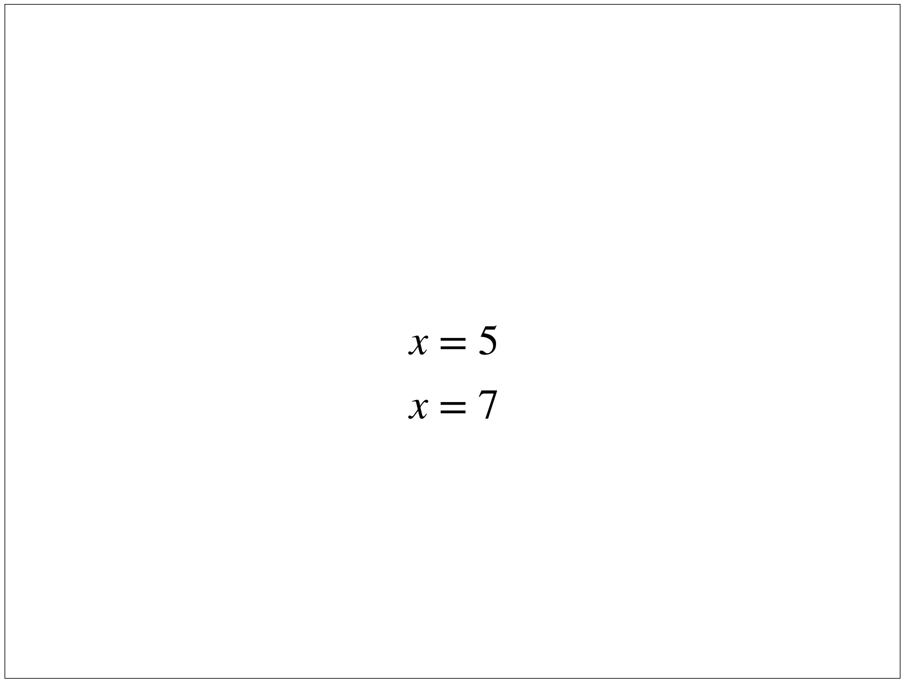
$y = -x - 2$

$x = -4, y = 2$

Let’s go back to Algebra I for a moment. These equations make perfect sense, and they have a unique solution.

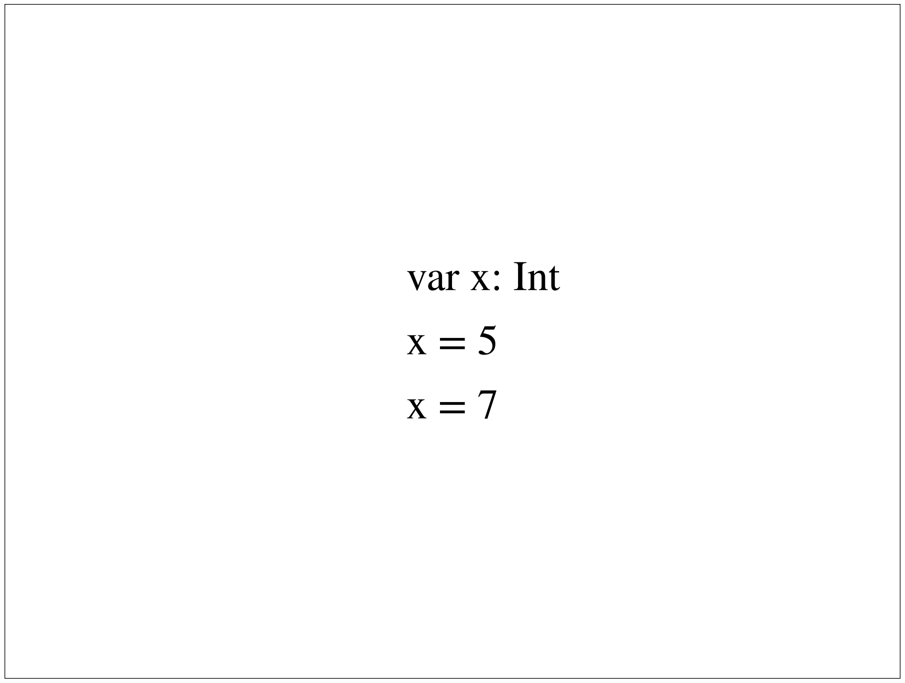
<build>

So that’s fine.



$$x = 5$$
$$x = 7$$

Here are two more equations. They kind of make sense, but the most you can say about them is that there is no such *x*.



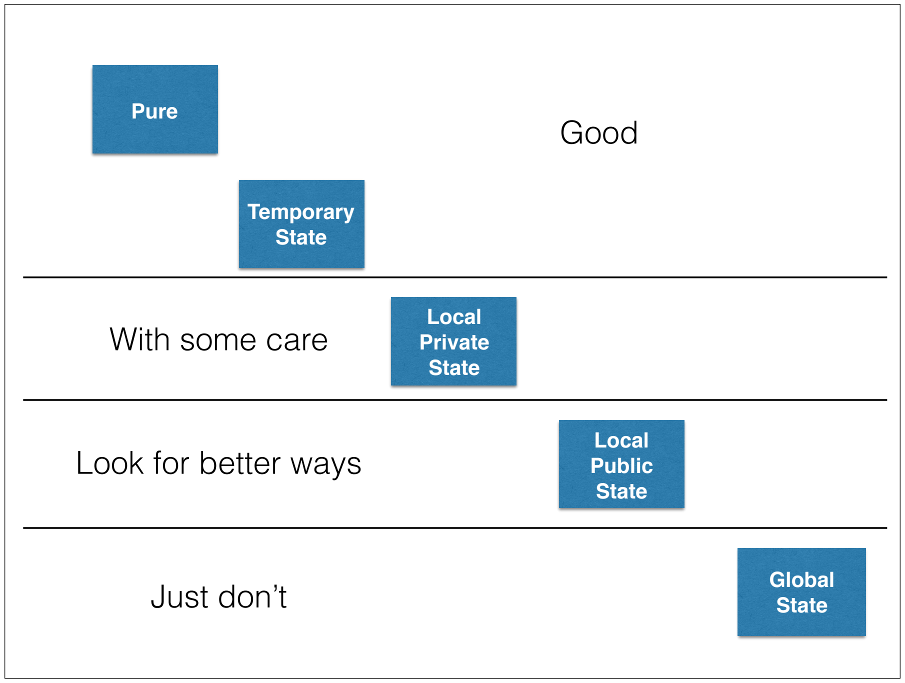
```
var x: Int
x = 5
x = 7
```

So why do we think this makes sense? This is actually a bit crazy, but it’s because most of the time when we say equals, we don’t mean equals.

In imperative programming, “x equals five” doesn’t mean “x equals five.” It means “store 5 in the memory location referenced by x.”

In math, and in functional programming, “x equals 5” means “x equals 5.” x is another name for 5, anywhere you see x you can put 5, and anywhere you see 5 you can put x.

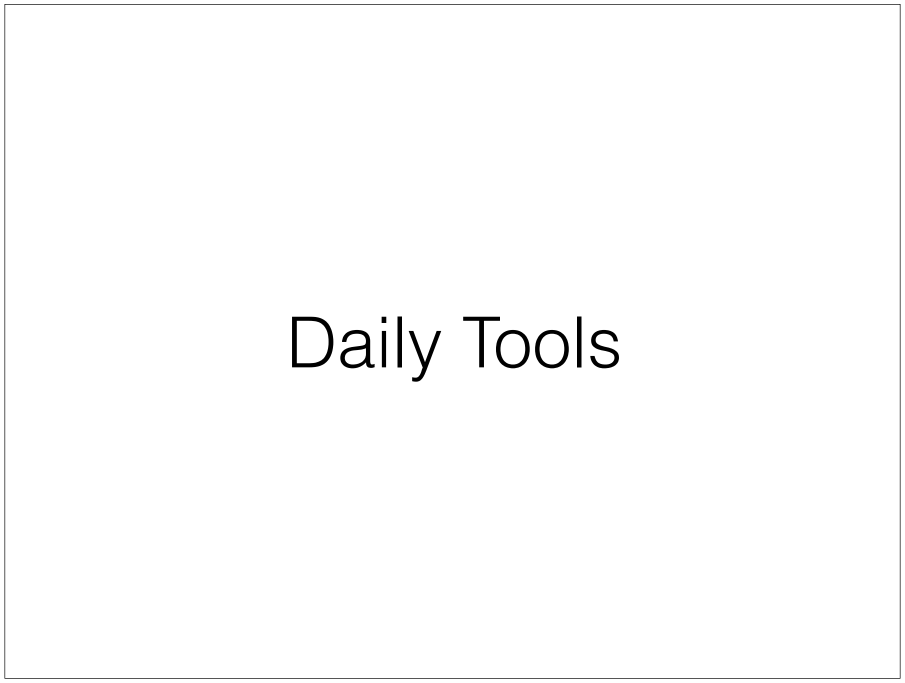
That’s “referential transparency.” It’s what lets the compiler simplify our code automatically. When we start mutating our variables, it’s harder to do that. When code depends on external state, then it matters how we got here. When code causes side-effects, we can’t just combine duplicate expressions or automatically cache results. We make it hard to understand the code. Hard to refactor the code. The more state, the more complicated the system.



Now, not all state is created equal. There's a hierarchy. At the top, there is totally pure code. No assignments, no side-effects. Just compute and return an answer.

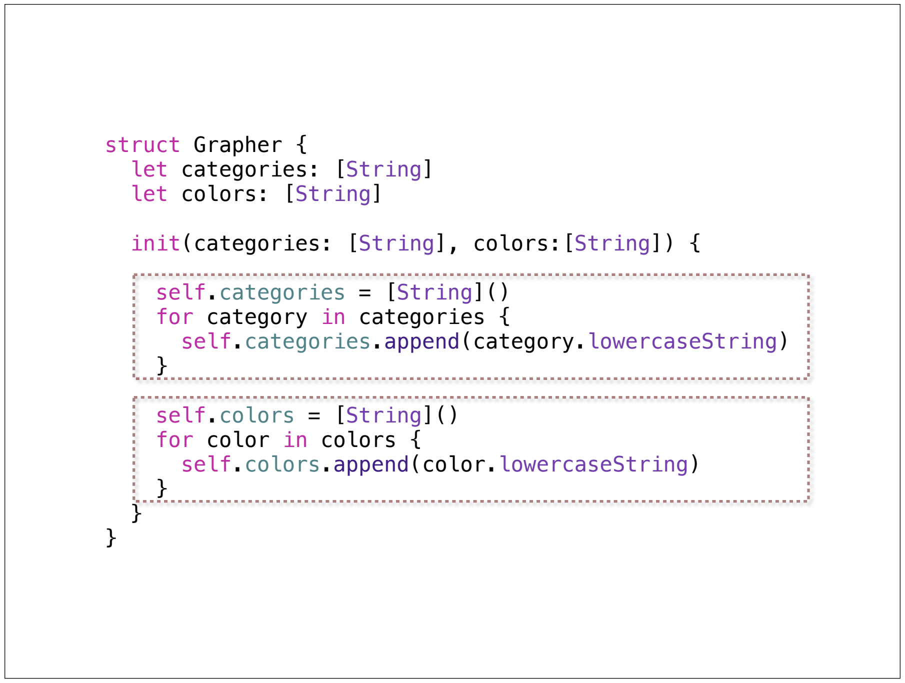
Then there are patterns where you have some temporary state, but it doesn't escape the function. A temporary array to build a result and immediately return it is like that. It still opens some opportunity for mistakes, but it's very localized and doesn't impact the rest of the system, so that's not such a big deal. The function still seems pure to the rest of the world. For the same parameters, the function is guaranteed to return the same result every time.

Beyond that is local private state. Lazy variables in an immutable object is like that. To the outside world, the object seems to have no mutable state, but it really has some hidden inside. Internal caches can create surprising bugs. Calling a function with the same parameters is supposed to return the same result, but it might not if there are caching issues. Multi-threading is harder. Higher up on the chart, you could always safely call a function on any thread, or on multiple threads at the same time, and it'd be safe. But with private state,

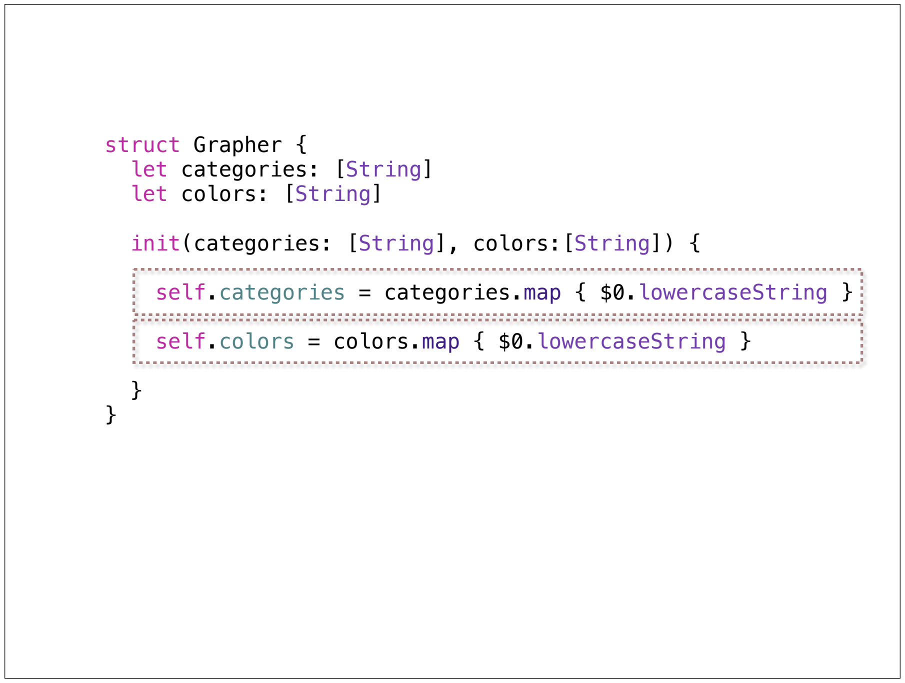


Daily Tools

So that's the motivation. Now we'll move into some day-to-day tools you can use to attack this stuff.


```
struct Grapher {
  let categories: [String]
  let colors: [String]

  init(categories: [String], colors:[String]) {
    self.categories = [String]()
    for category in categories {
      self.categories.append(category.lowercaseString)
    }

    self.colors = [String]()
    for color in colors {
      self.colors.append(color.lowercaseString)
    }
  }
}
```

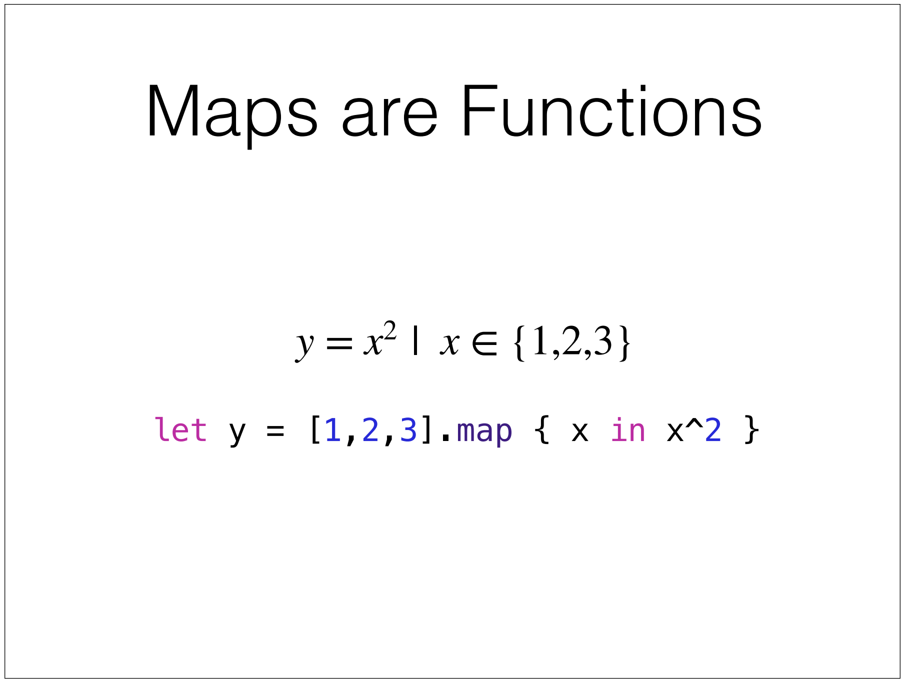
Let’s go back to our cleaned-up example. There’s a lot of almost-the-same-but-just-a-little-different code here. We initialize an array, then we loop over elements, and append something to the array. That seems like something we could make reusable. And Swift gives us that. It’s called map.



```swift
struct Grapher {
  let categories: [String]
  let colors: [String]

  init(categories: [String], colors:[String]) {
    self.categories = categories.map { $0.lowercaseString }
    self.colors = colors.map { $0.lowercaseString }
  }
}
```

map does what we want. It applies some function to each element and creates a new array from the results. We get rid of all the boilerplate, which means there are fewer places the bugs can hide. We made the code shorter and more focused on what we actually care about. And we've abstracted away the question of exactly how to iterate over the arrays. We could rip out this map and replace it with a parallel version or a lazy version. We could replace the arrays with sequences. We can change the implementation any way we want.

And we've defined our properties in terms of what they are rather than how to create them. The property is the lowercase version of the parameter. We leave to the system to make that be true.



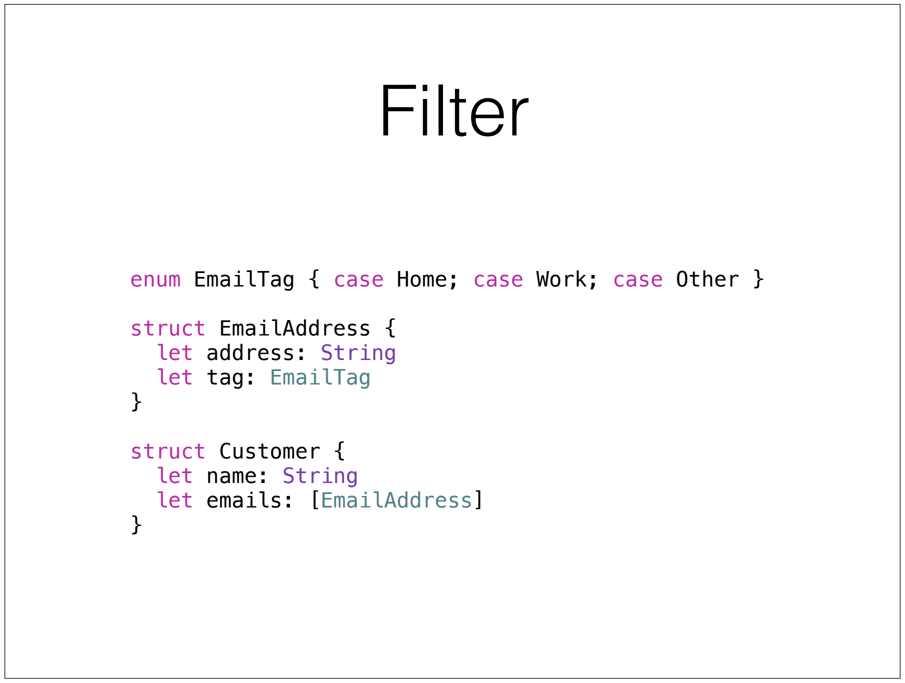
Maps are Functions

$$y = x^2 \mid x \in \{1,2,3\}$$

```
let y = [1,2,3].map { x in x^2 }
```

In a lot of ways, map expresses what it means to be a function. Given a series of inputs, it generates a series of outputs. This is exactly what you learned in Algebra.

It’s true that the *map* implementation is also a lot shorter than the *for-in* way, but sometimes the functional approach doesn’t save any code. The point is that it focuses on combining functions together, like map and lowercase, so that each function encapsulates just one piece of the problem, and we compose our program by clicking them together.

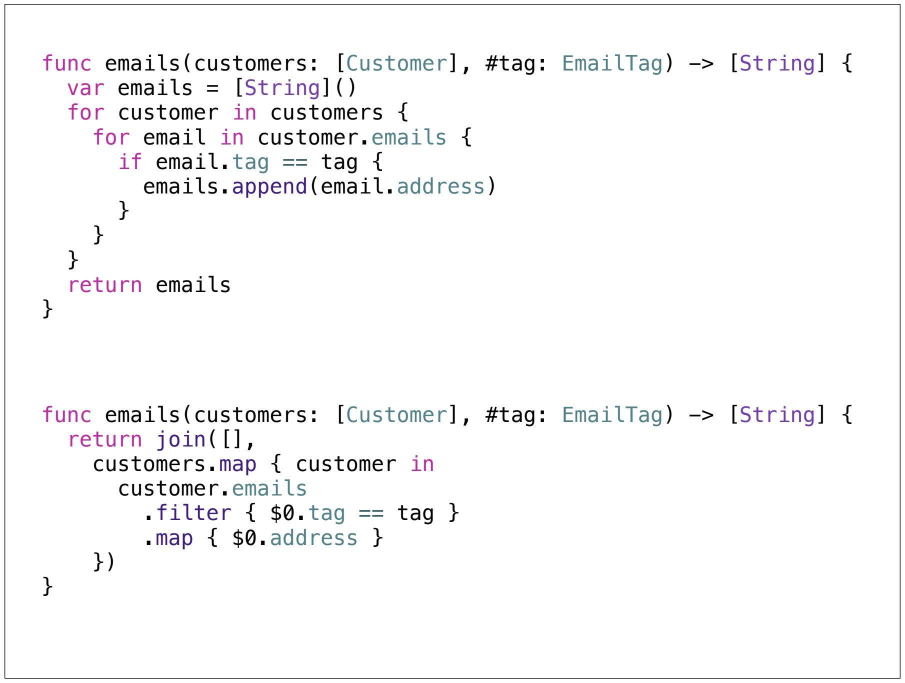


Filter

```
enum EmailTag { case Home; case Work; case Other }

struct EmailAddress {
  let address: String
  let tag: EmailTag
}

struct Customer {
  let name: String
  let emails: [EmailAddress]
}
```

Let’s move onto a little more complicated example. We’ll start with a data structure to hold customers and their email addresses.

Let’s say you’re building a chat application and you want to autocomplete based on just home email addresses. How would we pull out that list?



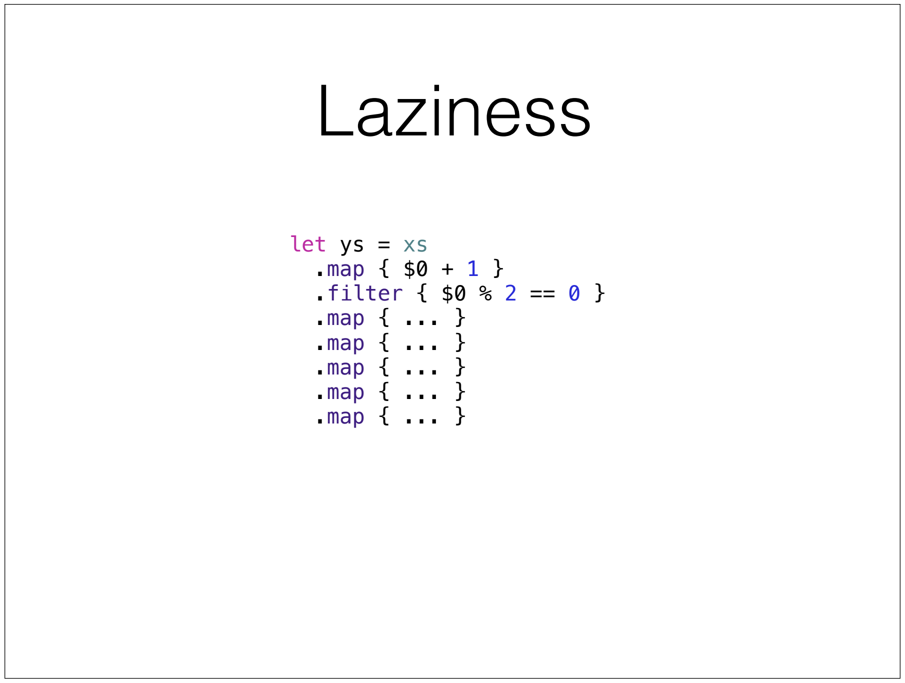
```swift
func emails(customers: [Customer], #tag: EmailTag) -> [String] {
  var emails = [String]()
  for customer in customers {
    for email in customer.emails {
      if email.tag == tag {
        emails.append(email.address)
      }
    }
  }
  return emails
}


func emails(customers: [Customer], #tag: EmailTag) -> [String] {
  return join([],
    customers.map { customer in
      customer.emails
        .filter { $0.tag == tag }
        .map { $0.address }
    })
}
```

This loops over the customers, then loops over the email addresses, and if the tag matches, add the string version of it to the result. Here's a more functional approach.

<build>

Instead of directly iterating and modifying a shared result value, this approach is more like a SQL query. It describes what we want rather than how to implement it. We map from customers to emails, we filter the emails to the ones we want, and we map those to strings. At the end, we join everything together into one list.

The code isn't really any shorter, but it's easier to keep adding transforms, filters and maps, certain that there are no side-effects we need to worry about.

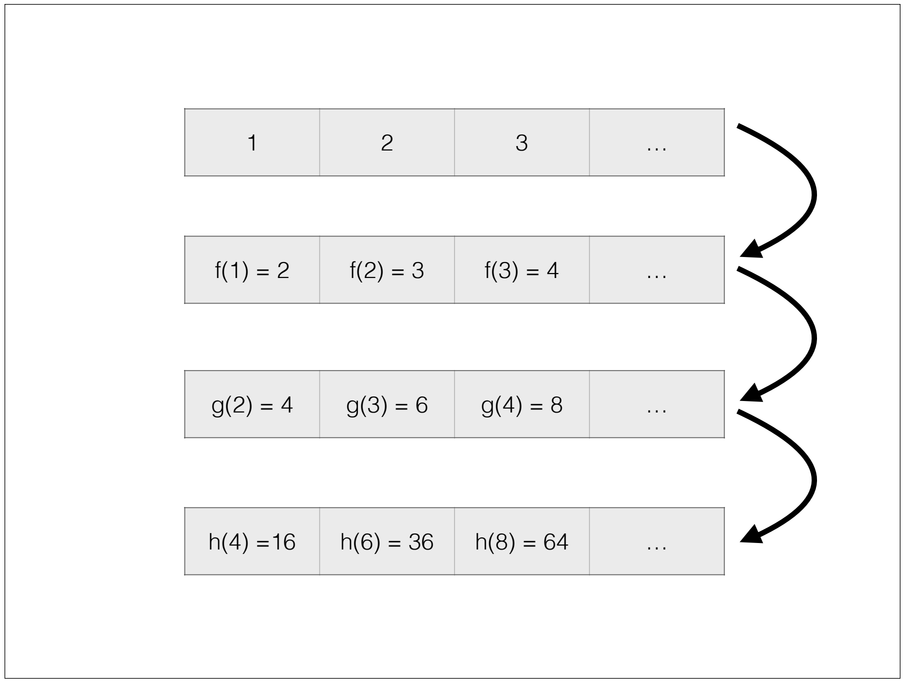


Laziness

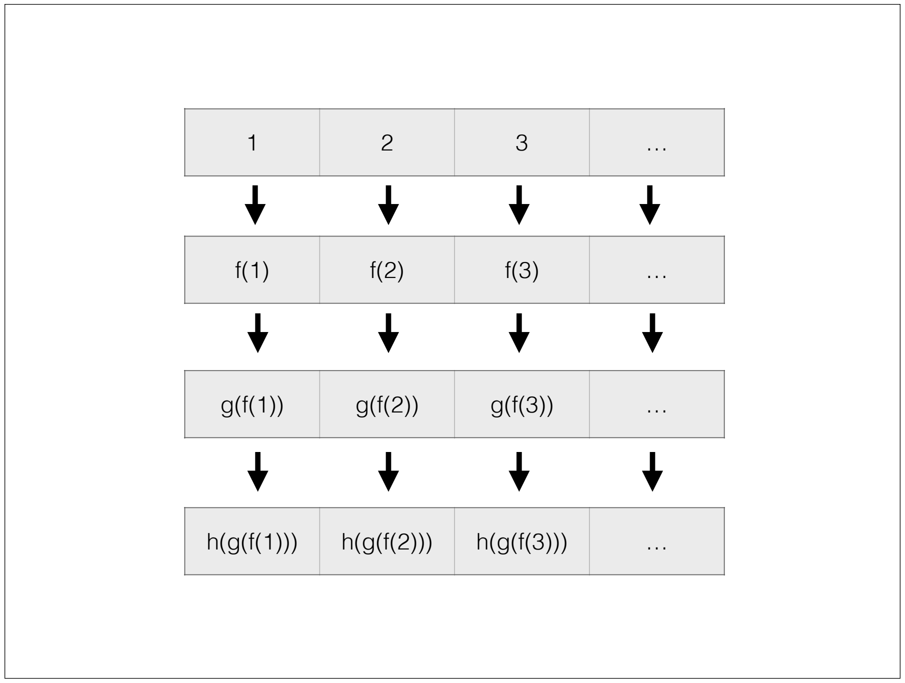
```
let ys = xs
  .map { $0 + 1 }
  .filter { $0 % 2 == 0 }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }
```

Usually chaining together map and filter is great. It can be a lot easier to read, and easier to break out pieces of it into other functions. But there is one problem with doing this in Swift.

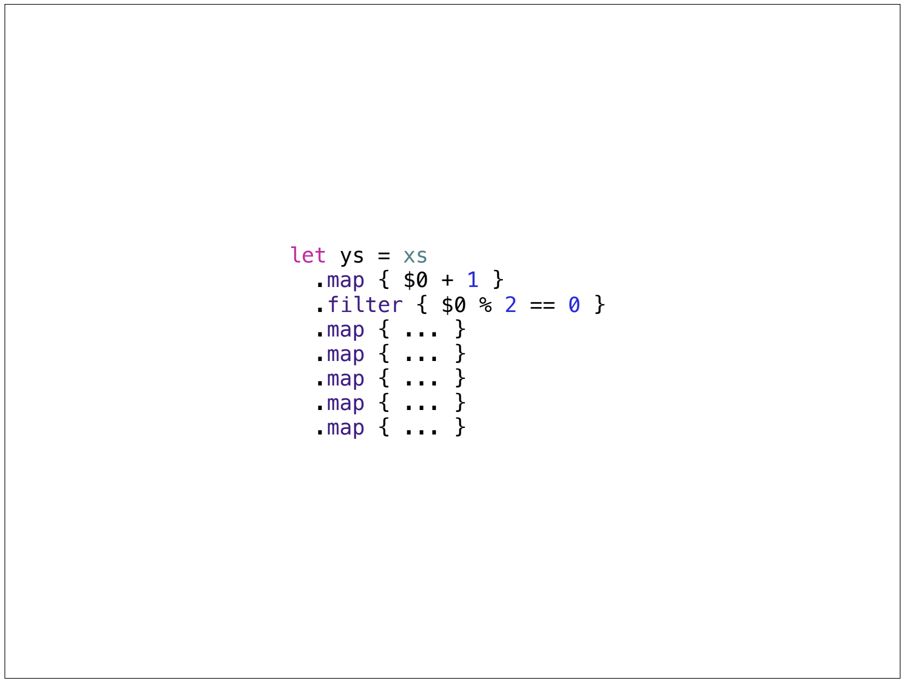
In Swift, these functions return arrays. That means that at each step along this chain, a complete new array is allocated. That's no big deal in most cases, but if *xs* were really big, that would be a problem. You'd see a significant memory spike and a performance hit.


1 | 2 | 3 | …
f(1) = 2 | f(2) = 3 | f(3) = 4 | …
g(2) = 4 | g(3) = 6 | g(4) = 8 | …
h(4) =16 | h(6) = 36 | h(8) = 64 | …

At each step we compute everything. But we really don't care about the intermediate results. We just want the last row.

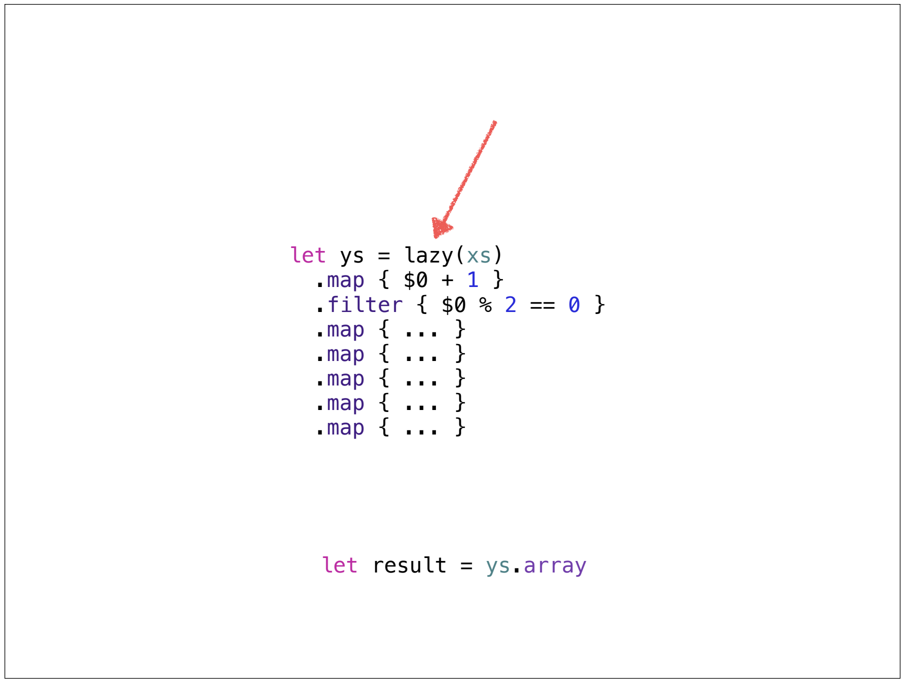


What we really want is this. Just accumulate all the operations we need to do, and do them when we need them. No need to allocate an array at each step to keep track of a temporary value. What's great in Swift is that this is trivial.



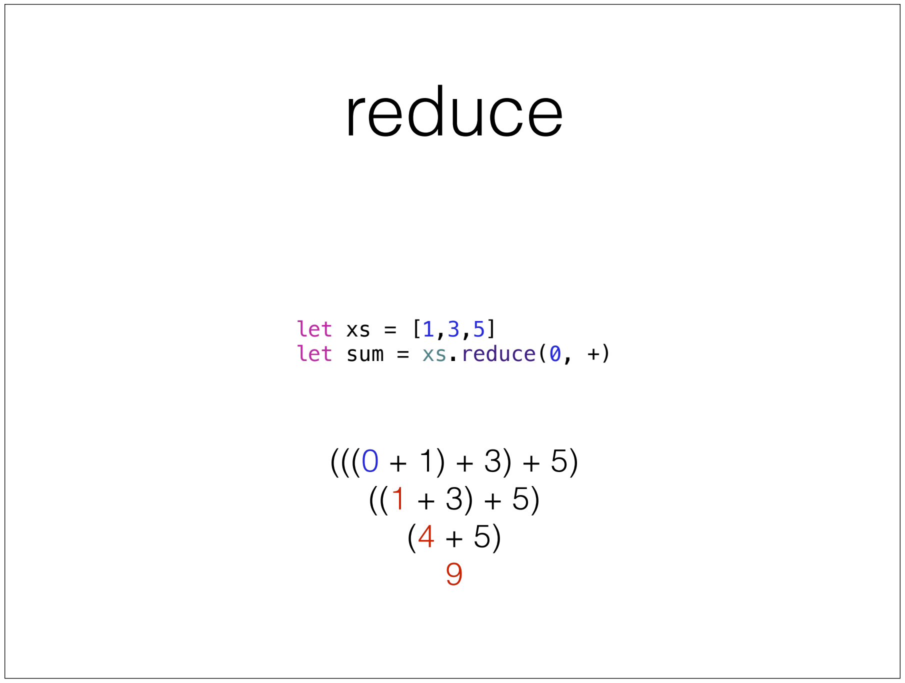
```
let ys = xs
  .map { $0 + 1 }
  .filter { $0 % 2 == 0 }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }
```

We take this code.


```
let ys = lazy(xs)
  .map { $0 + 1 }
  .filter { $0 % 2 == 0 }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }
  .map { ... }



   let result = ys.array
```

And we make this change. We add lazy to the initial list. And seriously, that's usually about it. Now why not do that all the time? The very first release of Swift did. All maps were lazy. But laziness can sometimes be a little confusing. If you maps have side effects, they generally shouldn't, but if they do, they won't actually happen until the the map is computed. If the operations are time consuming, they might wind up being computed on the main thread rather than a background thread. Lazy sequences don't print well in the debugger, so that can be a little annoying. And lazy collections have to hold onto their source, so if you filtered most of the collection, you can't free up as much memory as you'd like.

But for the most part, lazy collections are very useful in cases where you have multiple steps and are worried about performance. And the problems aren't major. If you need to make sure that the sequence is fully computed, you can convert it to an array at any time like this.



reduce

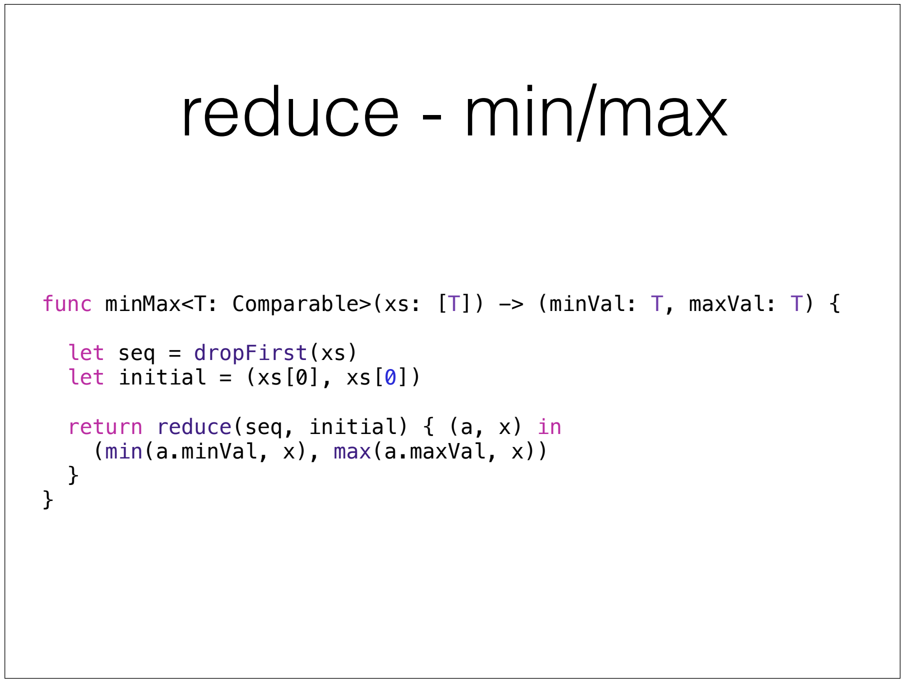
```
let xs = [1,3,5]
let sum = xs.reduce(0, +)
```

(((0 + 1) + 3) + 5)
((1 + 3) + 5)
(4 + 5)
9

Reduce is the swiss-army knife of functions. It consolidates an array to a single value. The canonical example of reduce is the summing function. It takes a starting value and a combining function, and combines the list. For instance, the above code translates into this.

<build>

The parentheses may seem unnecessary, but I want to make it clear exactly what's happening. The plus function…remember, plus is just a function that takes its left and right parameters…is passed zero, the starting value, and one. The result of that is one. The plus function is then called with one and the next value, three. The result of that is four. The plus function is then called with four and five, returning nine, which is the final result.



reduce - min/max

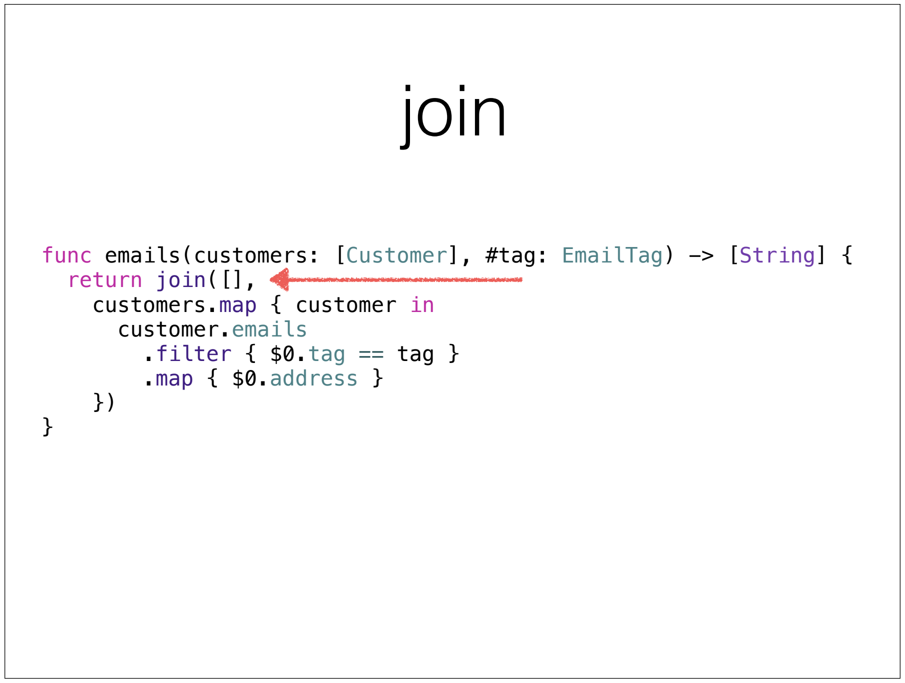
```swift
func minMax<T: Comparable>(xs: [T]) -> (minVal: T, maxVal: T) {

  let seq = dropFirst(xs)
  let initial = (xs[0], xs[0])

  return reduce(seq, initial) { (a, x) in
    (min(a.minVal, x), max(a.maxVal, x))
  }
}
```

We can use reduce in lots of other ways. Let's reduce a list to both its min and max at the same time.

This one shows a few important points. First, notice our function is passed an accumulator, *a*, and a current value, *x*. We just need to return the next value. That value can be anything. We just need to make sure the initial value has the same type.

I say "anything is fine," but I will warn you that you generally shouldn't reduce an array to another array. In other languages that's fine, but in Swift it can be very expensive because Swift doesn't let related arrays share memory the way most functional languages do. But if you're reducing an array to an array, you usually should use map and filter instead in Swift.

Notice the dropFirst. That returns a sequence starting at the second element. Reduce requires an initial value. Sometimes that's easy, but sometimes there's no obvious first element. That's true for min and max, particularly if you write it as a generic function like this one. I mean, what's the ultimate maximum value for



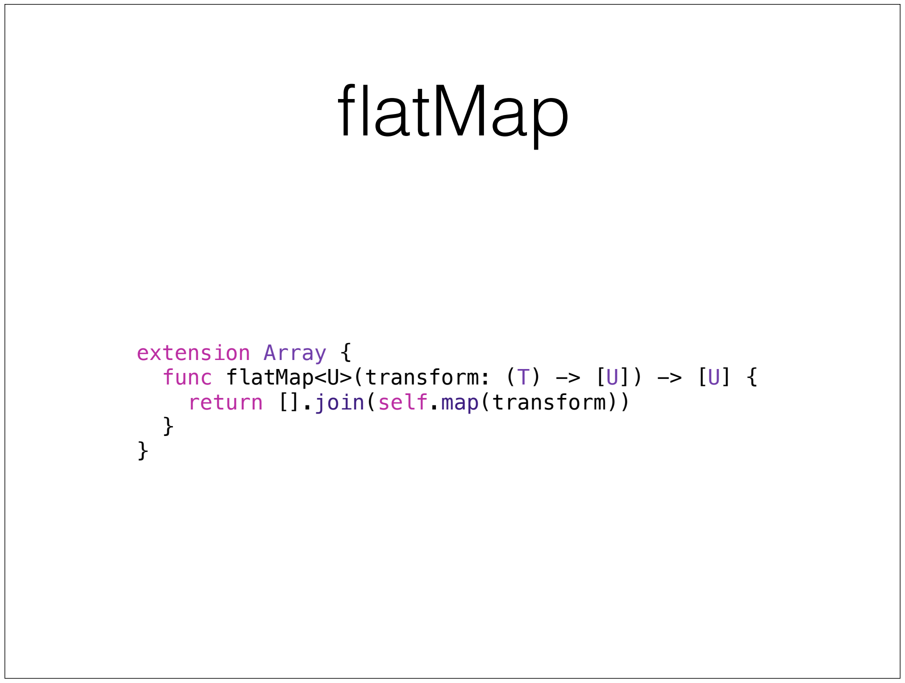
join

```
func emails(customers: [Customer], #tag: EmailTag) -> [String] {
  return join([],
    customers.map { customer in
      customer.emails
        .filter { $0.tag == tag }
        .map { $0.address }
    })
}
```

Just a few more odds and ends tools, and then we'll get into some more functional concepts.

Sometimes when you're mapping, you find yourself with a list of lists when you just wanted a flat list. That happened in our email example earlier.

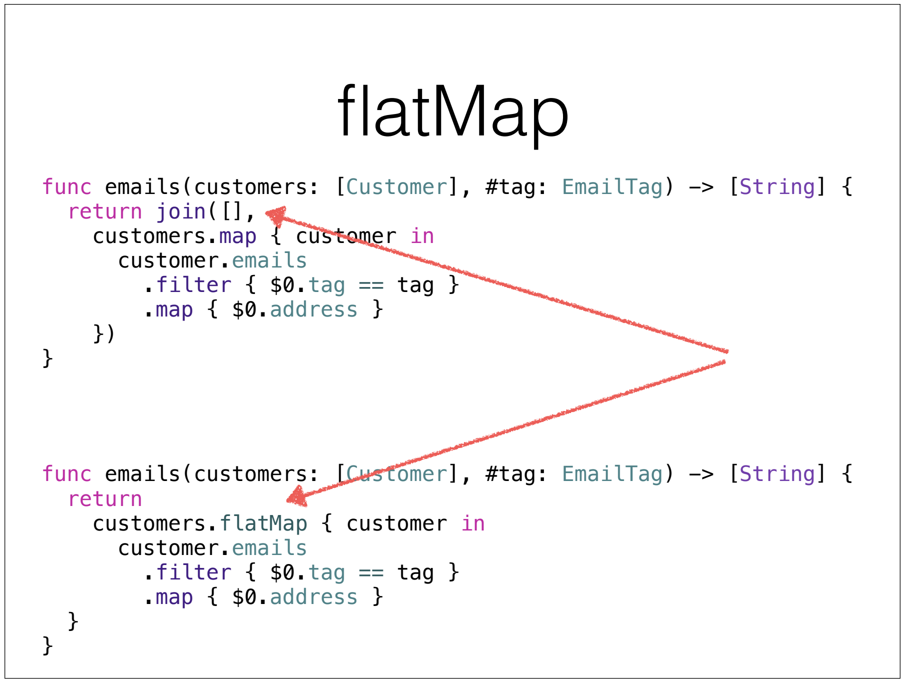
When we map a list of customers to the customers' email addresses, we wind up with a list of lists of email addresses. You'll want to flatten that out, and the easiest way to do that in Swift is with *join*, as I show here. That just says to join all the lists together, separated by nothing.



flatMap

```
extension Array {
  func flatMap<U>(transform: (T) -> [U]) -> [U] {
    return [].join(self.map(transform))
  }
}
```

If you combine join and map, you get another, very interesting function. Swift doesn't have a specific name for it, but it's called flatMap in Scala and concatMap in Haskell. I'm mostly focused on built-in functionality in this talk, but flatMap is so incredibly simple and useful that I'm making an exception. As you see, it's really simple, just join and map.



flatMap

```swift
func emails(customers: [Customer], #tag: EmailTag) -> [String] {
  return join([],
    customers.map { customer in
      customer.emails
        .filter { $0.tag == tag }
        .map { $0.address }
    })
}
```

```swift
func emails(customers: [Customer], #tag: EmailTag) -> [String] {
  return
    customers.flatMap { customer in
      customer.emails
        .filter { $0.tag == tag }
        .map { $0.address }
  }
}
```

It's not that this saves a little code. It's that it chains. You can read it from top to bottom and that's the order it's computed. With join, you read down to the bottom, and then jump back to the top. Imagine you had a series of mappings with joins scattered through. It gets confusing really quickly. flatMap makes it simple.

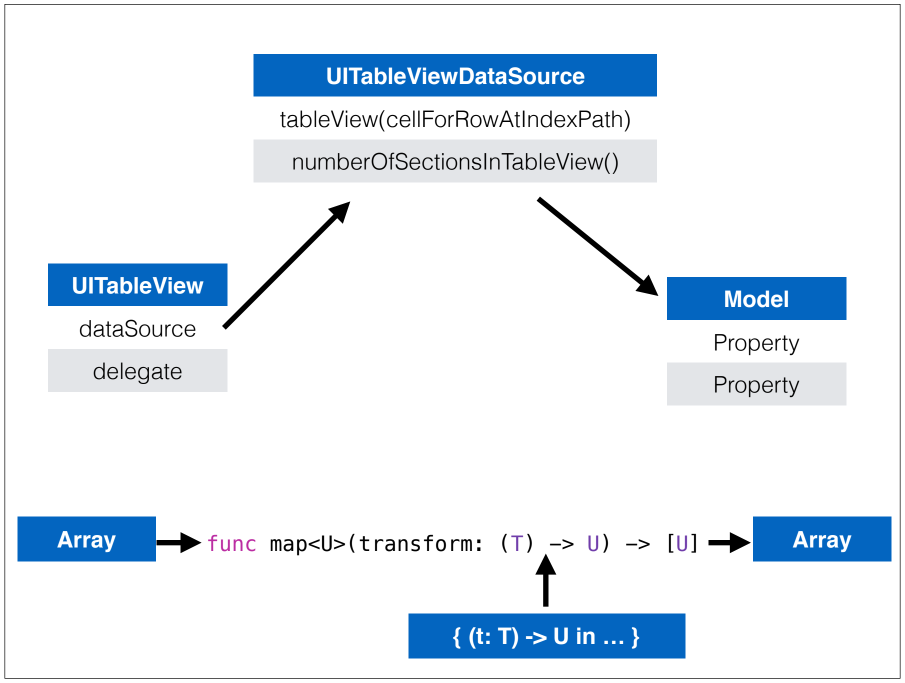
Humble as it is, flatMap turns out to be a really important idea that pops up in the strangest places. We'll be seeing it again later.

That's the end of the everyday stuff I'm going to talk about today. Before I go on to some more advanced ideas, any questions?

Going Further

A key differentiator between object oriented programming and functional programming is how they put things together. How they “compose.” Not surprisingly, the key composition element in OOP is the object. Good object oriented design often involves a lot of HAS-A relationships with objects cooperating to get the job done.

The key Functional element of composition is, of course, the function. Good functional design involves tying together small, generic functions.

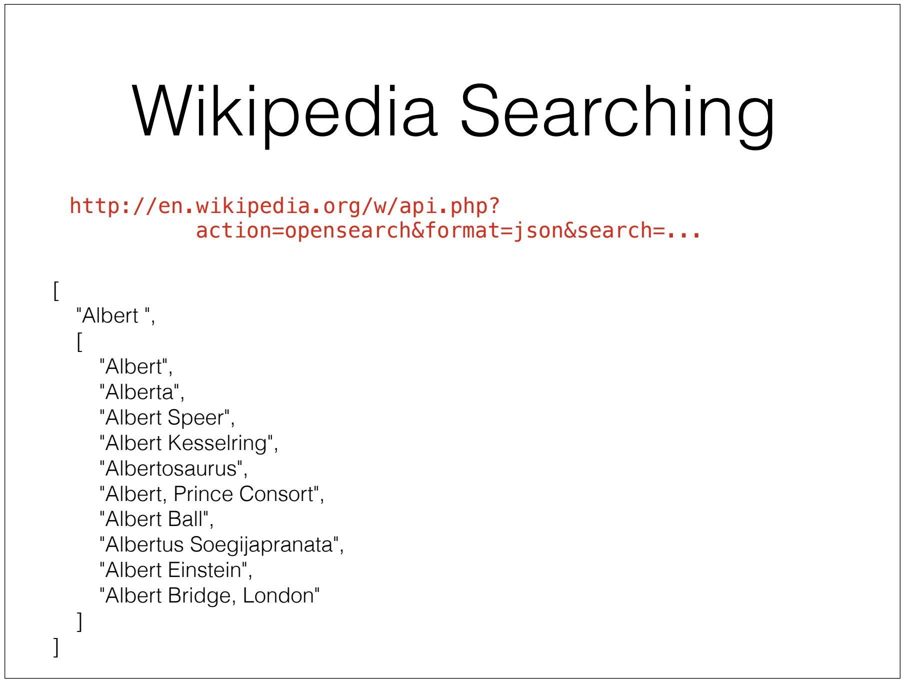


So in OOP, you spend a lot of time implementing objects that conform to some protocol. And when it lines up with the object you want to work with, they can click together. Sometimes you need to create an adapter to bridge two unrelated interfaces. That's what a UITableViewDataSource usually is. In good MVC design, the datasource is an adapter between some model-specific interface and the specific protocol that UITableView requires.

<build>

In a functional paradigm, the interface you're usually working to is a single function signature. So the map function can click together any sequence of values and any function that takes a value and returns a value.

Just like you might build a complex object by clicking together smaller, simpler objects, you can build a complex function by clicking together smaller, simpler functions.

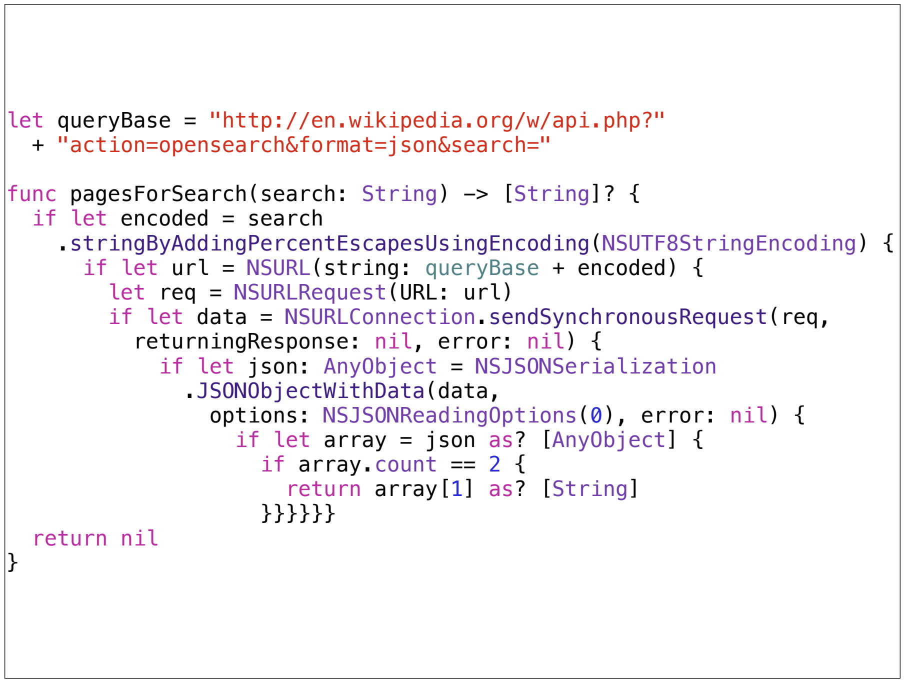


Wikipedia Searching

```
http://en.wikipedia.org/w/api.php?
         action=opensearch&format=json&search=...
```

```
[
  "Albert ",
  [
    "Albert",
    "Alberta",
    "Albert Speer",
    "Albert Kesselring",
    "Albertosaurus",
    "Albert, Prince Consort",
    "Albert Ball",
    "Albertus Soegijapranata",
    "Albert Einstein",
    "Albert Bridge, London"
  ]
]
```

Let’s consider a very common iOS problem and discuss some solutions by decomposing the problem into functions. My favorite sample problem is the Wikipedia API. It looks like a lot of real world problems while not requiring any fancy setup or API keys. We’ll try one of the simplest things you can do with Wikipedia: perform a basic search. You post to a simple REST endpoint, and you get some JSON back. It’s a two-element array, and the first element is your search term, and the second element is an array of the results.

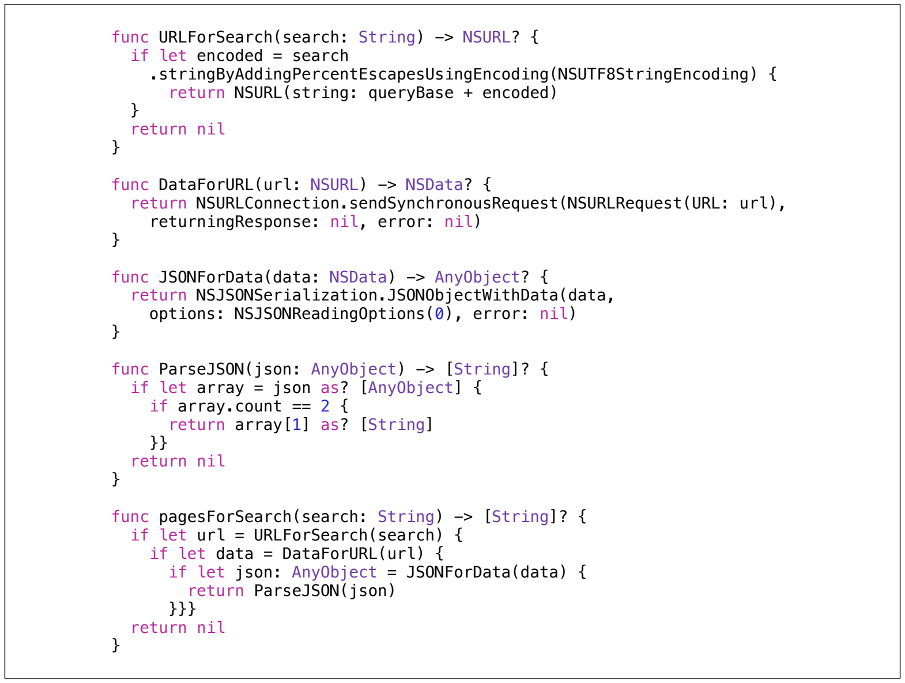


```
let queryBase = "http://en.wikipedia.org/w/api.php?"
  + "action=opensearch&format=json&search="

func pagesForSearch(search: String) -> [String]? {
  if let encoded = search
    .stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding) {
      if let url = NSURL(string: queryBase + encoded) {
        let req = NSURLRequest(URL: url)
        if let data = NSURLConnection.sendSynchronousRequest(req,
          returningResponse: nil, error: nil) {
            if let json: AnyObject = NSJSONSerialization
              .JSONObjectWithData(data,
                options: NSJSONReadingOptions(0), error: nil) {
                  if let array = json as? [AnyObject] {
                    if array.count == 2 {
                      return array[1] as? [String]
                    }}}}}}
  return nil
}
```

So here's a function that implements that. It takes a search term, synchronously fetches it from Wikipedia, and parses the JSON. Getting it onto a slide took a little wrapping, but this is pretty close to what it looks like in Xcode. It's a complicated problem. Dealing with AnyObject in Swift always is. But frankly this is a pretty unfortunate stuff. It's barely comprehensible.

The talk I'm giving now is about the fifth version I wrote. In earlier drafts, at this point, I immediately jumped to a fancy solution with custom operators and monads. But here's the thing. It was only a little easier to read, and broke the compiler every time I touched it, which means it took me over an hour to get it working. I almost gave up on the whole talk, swore off functional programming, and considered a new career as a cabinet maker.

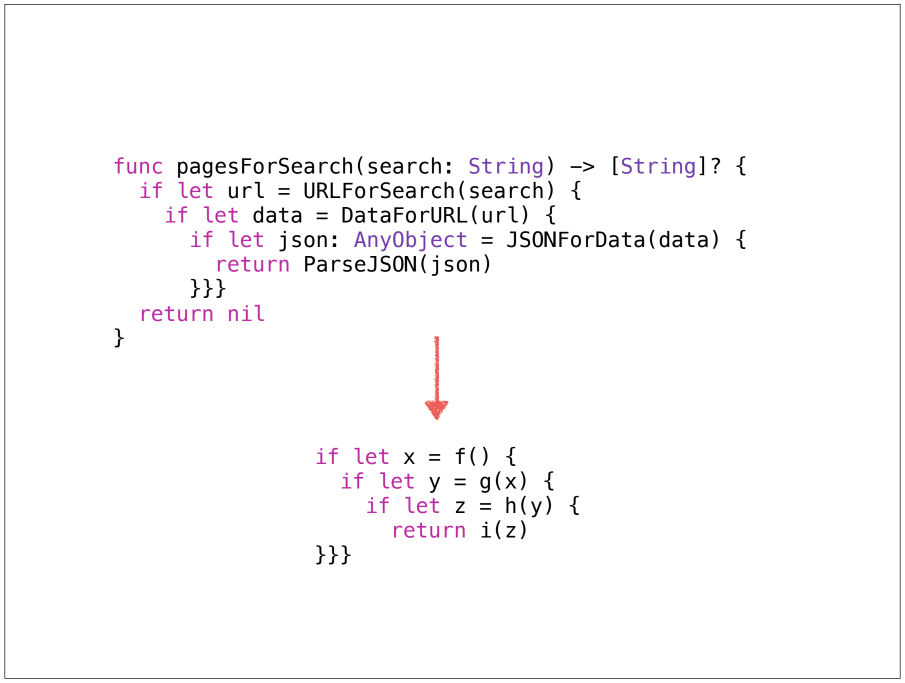
I walked away for a while and finally remembered the goal. The goal isn't monads or functors or whatnot. The goal is clear, robust code. I went back to the basics. If this were C, what would I do? I'd break it up into smaller functions. I'd decompose the problem into functions.



```swift
func URLForSearch(search: String) -> NSURL? {
  if let encoded = search
    .stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding) {
      return NSURL(string: queryBase + encoded)
  }
  return nil
}

func DataForURL(url: NSURL) -> NSData? {
  return NSURLConnection.sendSynchronousRequest(NSURLRequest(URL: url),
    returningResponse: nil, error: nil)
}

func JSONForData(data: NSData) -> AnyObject? {
  return NSJSONSerialization.JSONObjectWithData(data,
    options: NSJSONReadingOptions(0), error: nil)
}

func ParseJSON(json: AnyObject) -> [String]? {
  if let array = json as? [AnyObject] {
    if array.count == 2 {
      return array[1] as? [String]
    }}
  return nil
}

func pagesForSearch(search: String) -> [String]? {
  if let url = URLForSearch(search) {
    if let data = DataForURL(url) {
      if let json: AnyObject = JSONForData(data) {
        return ParseJSON(json)
      }}}
  return nil
}
```

The fact that you can easily chain together map, filter, reduce, flatMap, and all the rest sometimes creates the temptation to turn your whole program into one giant chained expression. But you have to resist that urge. Break down your expressions into simple functions. They'll be easier to write, easier to read, easier to test, and they won't make the swift compiler explode so much.

OK, so that leaves us with this. Still kind of long. Nothing fancy here. I just broke up the big crazy chain of if-let statements into smaller functions. And they're really not that bad. We really could just leave it here and call it a day. But we can do better.



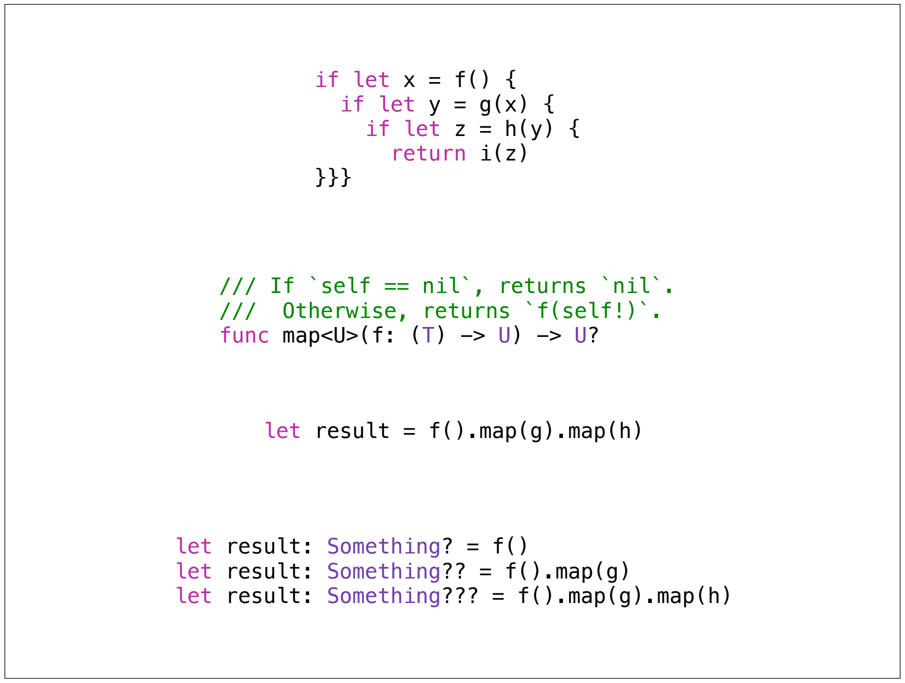
```swift
func pagesForSearch(search: String) -> [String]? {
  if let url = URLForSearch(search) {
    if let data = DataForURL(url) {
      if let json: AnyObject = JSONForData(data) {
        return ParseJSON(json)
      }}}
  return nil
}
```

```swift
if let x = f() {
  if let y = g(x) {
    if let z = h(y) {
      return i(z)
}}}
```

In several places, we have this basic form.

if-let something, then use that result to get another thing, and then use that to get another thing. And so on. That's actually kind of similar to our array mapping earlier. We're using a series of functions to transform *x* into *y* and then *y* into *z*. We're just mapping over optionals rather than over arrays.

Swift has this built-in. There's a map method on Optional.



```swift
if let x = f() {
  if let y = g(x) {
    if let z = h(y) {
      return i(z)
}}}
```

```swift
/// If `self == nil`, returns `nil`.
///  Otherwise, returns `f(self!)`.
func map<U>(f: (T) -> U) -> U?
```

```swift
let result = f().map(g).map(h)
```

```swift
let result: Something? = f()
let result: Something?? = f().map(g)
let result: Something??? = f().map(g).map(h)
```

And there it is. Take a mapping from T to U and, given an optional T, return an optional U. So how do we use that? We could try this.

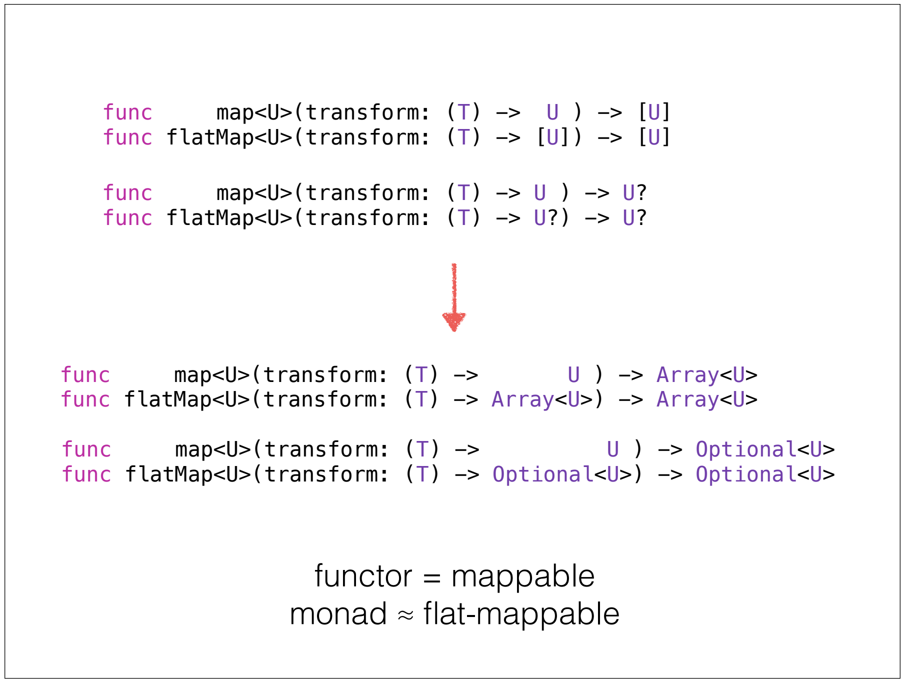
<build>

And that seems a good idea, but it doesn't quite work in this particular case. *f*, *g*, and *h* all return optionals.

<build>

So our "U" is Optional, and each stage of *map* will nest it one level deeper. *result* is going to wind up being and optional of an optional of an optional. That's no good.

But doesn't this sound like a familiar problem?



```
func     map<U>(transform: (T) ->  U ) -> [U]
func flatMap<U>(transform: (T) -> [U]) -> [U]

func     map<U>(transform: (T) -> U ) -> U?
func flatMap<U>(transform: (T) -> U?) -> U?
```

$\downarrow$

```
func     map<U>(transform: (T) ->        U ) -> Array<U>
func flatMap<U>(transform: (T) -> Array<U>) -> Array<U>

func     map<U>(transform: (T) ->           U ) -> Optional<U>
func flatMap<U>(transform: (T) -> Optional<U>) -> Optional<U>
```

functor = mappable
monad ≈ flat-mappable

Remember how we had that nesting problem for Array, so we created flatMap?

<build>

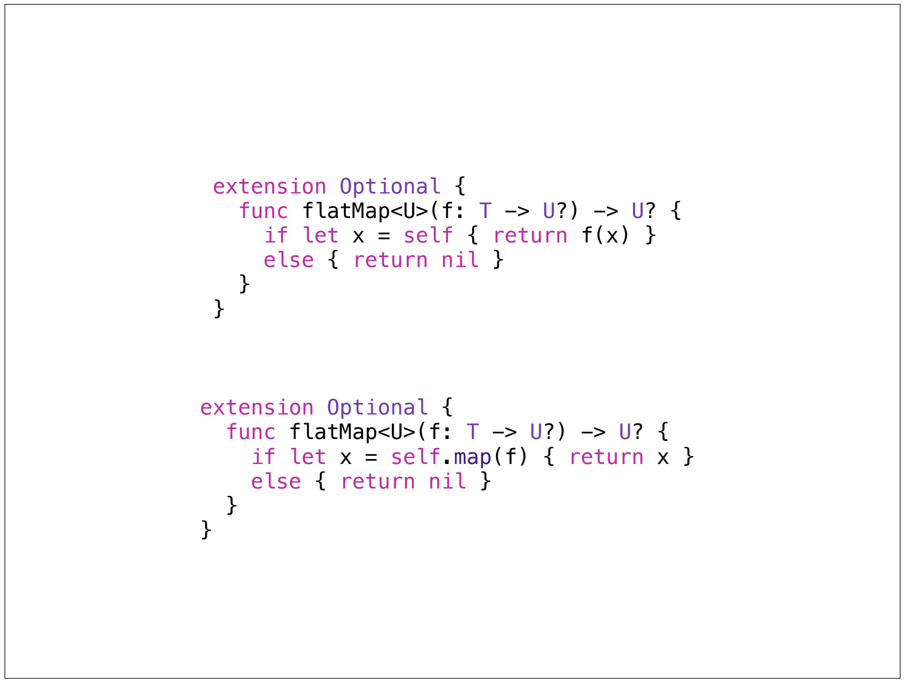
We can do the same thing for Optional. This maybe is a little clearer if we write it without the syntactic sugar.

<build>

Mapping is a kind of pattern that can applied to lots of things, not just lists. And many things that can be mapped can also be flatMapped.

Quick side vocabulary lesson. Things that can be sensibly mapped are called Functors. And things that can be sensibly flatMapped are more or less called monads. The definition of monads is slightly more technical, but “flatMappable” is pretty close.

<build>



```
extension Optional {
  func flatMap<U>(f: T -> U?) -> U? {
    if let x = self { return f(x) }
    else { return nil }
  }
}
```

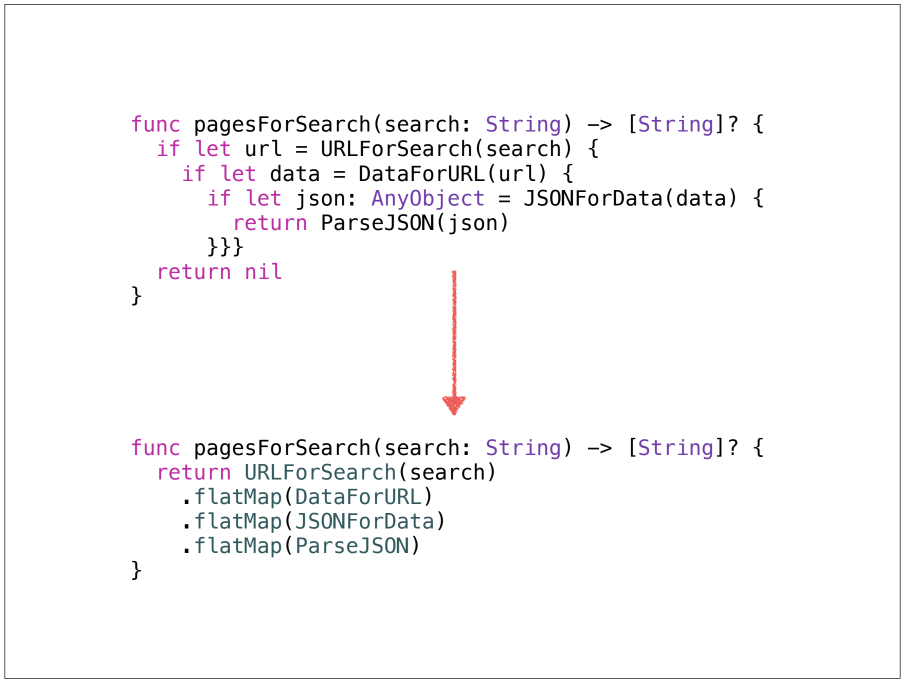
```
extension Optional {
  func flatMap<U>(f: T -> U?) -> U? {
    if let x = self.map(f) { return x }
    else { return nil }
  }
}
```

So let’s write a flatMap for Optional. Seems pretty straightforward. If the optional has a value, pass it to f and return the result. Otherwise return nil. We’re basically flattening first, then mapping. By flattening I mean taking an optional of an optional and returning an optional. But we could also write it the other way

<build>

and map first, then flatten. These two approaches are exactly the same. I just find the first way is a little easier to read, and it doesn’t need an extra function call to map. But they’re the same thing. Which gets us back to the power of functional programming. As long as these functions return the same values for all inputs, the outside world cannot care which one we use. Since there’s no state, the outside world cannot distinguish between these implementations. Changing the implementation should never require retesting other code.

Because Swift allows any type to automatically promote to its optional, it turns out that you can almost always use a flatMap instead of map on optionals. Swift will just fix up your function’s return type behind the scenes.



```swift
func pagesForSearch(search: String) -> [String]? {
  if let url = URLForSearch(search) {
    if let data = DataForURL(url) {
      if let json: AnyObject = JSONForData(data) {
        return ParseJSON(json)
      }}}
  return nil
}
```

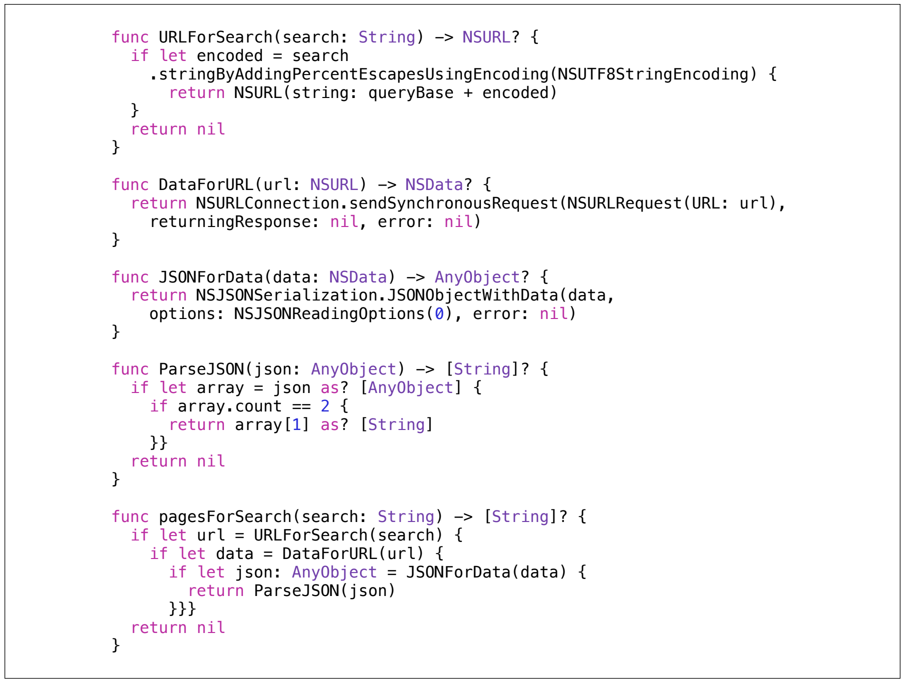
```swift
func pagesForSearch(search: String) -> [String]? {
  return URLForSearch(search)
    .flatMap(DataForURL)
    .flatMap(JSONForData)
    .flatMap(ParseJSON)
}
```

OK, and now rewritten with flatMaps.

<build>

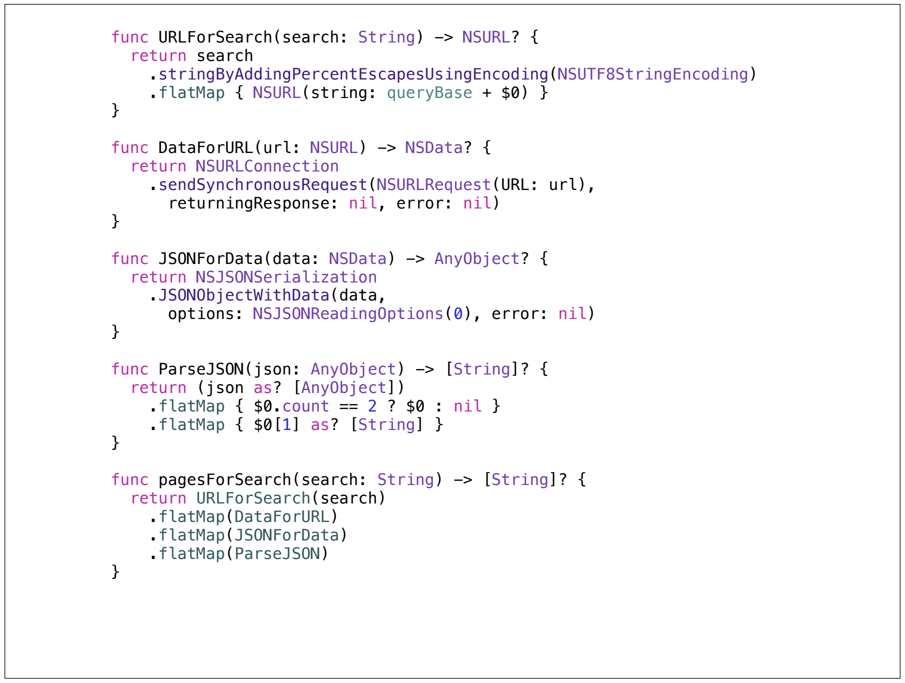
It's a little shorter, but more importantly, I think it's easier to see what's going on here by getting rid of some of the if-let noise. And all by adding a two-line extension.

I'm going to pause for a second here. It should be pretty clear how these two blocks of code are the same. In each case we unpack our optional, compute something on its contents, if any, and then repack the result into a new optional. It's that unpack, compute, repack sequence that makes this a monad. And that sequence turns out to be basically equivalent to map plus flatten.



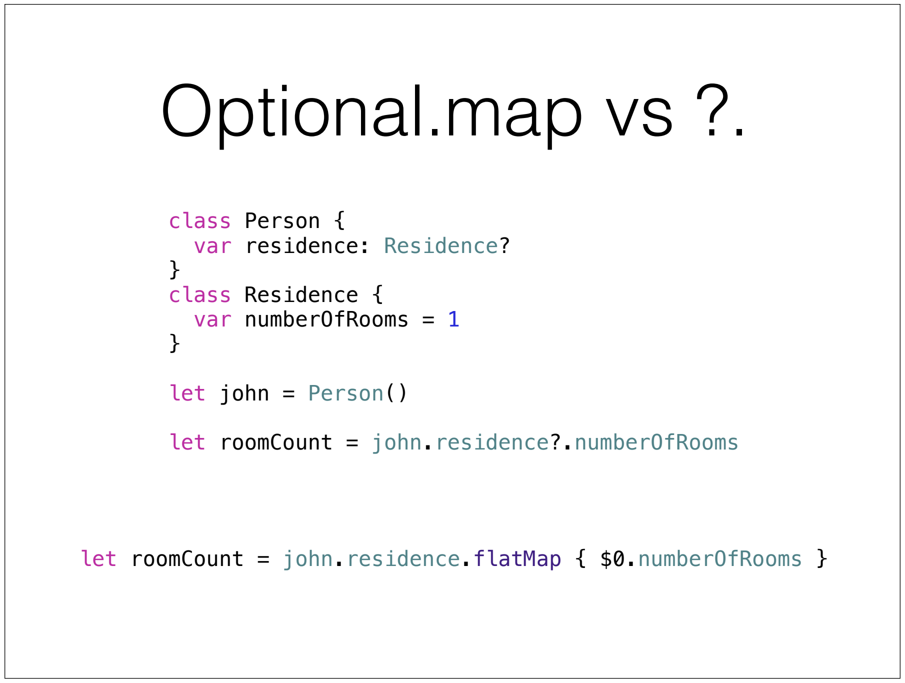
```
func URLForSearch(search: String) -> NSURL? {
  if let encoded = search
    .stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding) {
      return NSURL(string: queryBase + encoded)
  }
  return nil
}

func DataForURL(url: NSURL) -> NSData? {
  return NSURLConnection.sendSynchronousRequest(NSURLRequest(URL: url),
    returningResponse: nil, error: nil)
}

func JSONForData(data: NSData) -> AnyObject? {
  return NSJSONSerialization.JSONObjectWithData(data,
    options: NSJSONReadingOptions(0), error: nil)
}

func ParseJSON(json: AnyObject) -> [String]? {
  if let array = json as? [AnyObject] {
    if array.count == 2 {
      return array[1] as? [String]
    }}
  return nil
}

func pagesForSearch(search: String) -> [String]? {
  if let url = URLForSearch(search) {
    if let data = DataForURL(url) {
      if let json: AnyObject = JSONForData(data) {
        return ParseJSON(json)
      }}}
  return nil
}
```

So, this was our code broken down in to traditional functions. We can replace the if-lets with flatMap in several of these, and this is what we get.



```
func URLForSearch(search: String) -> NSURL? {
  return search
    .stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding)
    .flatMap { NSURL(string: queryBase + $0) }
}

func DataForURL(url: NSURL) -> NSData? {
  return NSURLConnection
    .sendSynchronousRequest(NSURLRequest(URL: url),
      returningResponse: nil, error: nil)
}

func JSONForData(data: NSData) -> AnyObject? {
  return NSJSONSerialization
    .JSONObjectWithData(data,
      options: NSJSONReadingOptions(0), error: nil)
}

func ParseJSON(json: AnyObject) -> [String]? {
  return (json as? [AnyObject])
    .flatMap { $0.count == 2 ? $0 : nil }
    .flatMap { $0[1] as? [String] }
}

func pagesForSearch(search: String) -> [String]? {
  return URLForSearch(search)
    .flatMap(DataForURL)
    .flatMap(JSONForData)
    .flatMap(ParseJSON)
}
```

It gets rid of a lot of the stair stepping, and I think is a pretty nice style, particularly in the last function. The key is keeping these expressions short. You might be tempted to combine them all into a long pagesForSearch, but if you make any mistakes at all, the compiler will go crazy. The type inference gets very confusing, and it’s easy to lose track of what you’re doing. The first step is good decomposition into small functions. Then you can worry about fancier features like flatMap. Even so, I kind of like this code, even if it still fills up the whole slide.



Optional.map vs ?.

```
class Person {
  var residence: Residence?
}
class Residence {
  var numberOfRooms = 1
}

let john = Person()

let roomCount = john.residence?.numberOfRooms
```

```
let roomCount = john.residence.flatMap { $0.numberOfRooms }
```

You may notice that flatMapping Optionals is very similar to optional chaining. It's actually just a more flexible form.

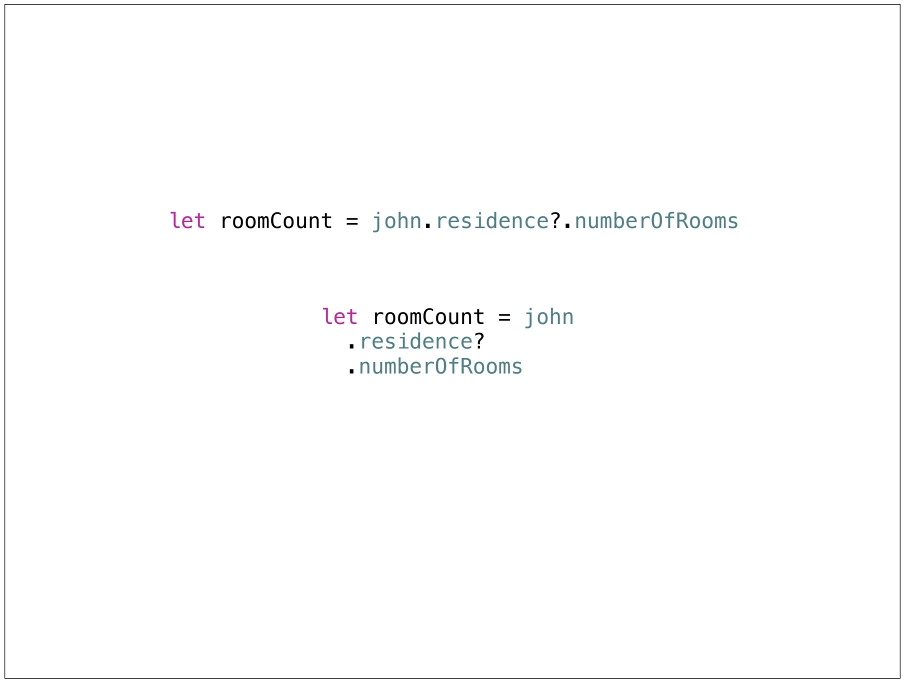
You may be familiar with this example from the Swift programming guide.

This use of optional chaining works the same as this flatMap.

<build>

If you can use optional chaining, you should. It's definitely the more Swifty approach, and it works great if every step is just a method call on the previous step. But when optional chaining isn't enough, like in our JSON parsing example, then flatMap is more flexible.

The thing to look for is flatMaps that just call a method on dollar-zero. That's when you know that optional chaining was probably an easier way.



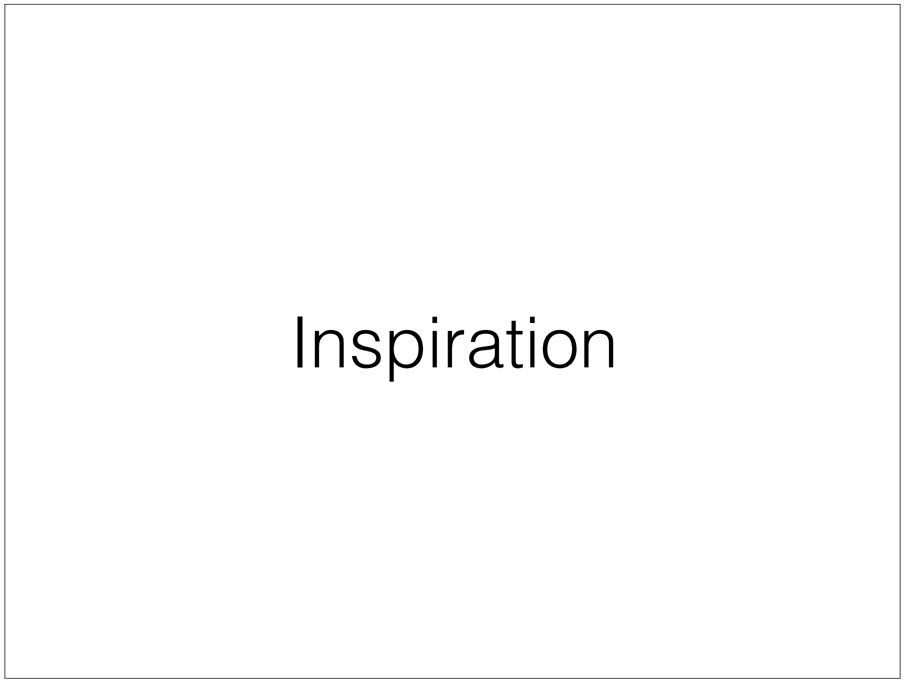
```
let roomCount = john.residence?.numberOfRooms
```

```
let roomCount = john
  .residence?
  .numberOfRooms
```

And in the “hey, did you know you could do that?” category. This expression can be written this way.

<build>

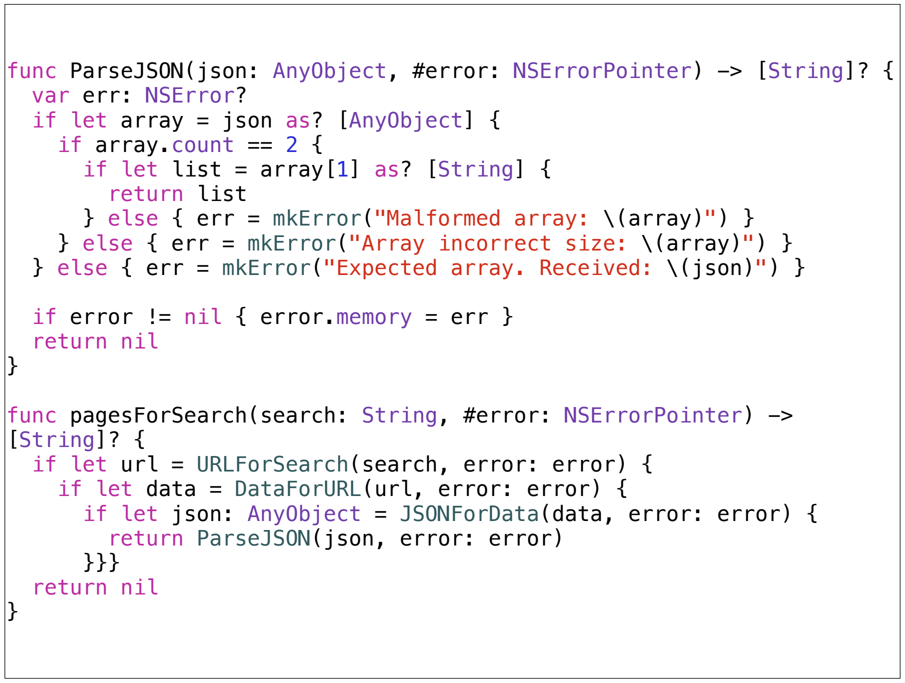
For a short expression like this, it’s silly. But I like how you can split the question mark and dot this way and break it up very nicely. Just in case you run into that situation.



Inspiration

I have a couple of thoughts here that I'm not going to dive into deeply, but I want to throw out there for ideas.

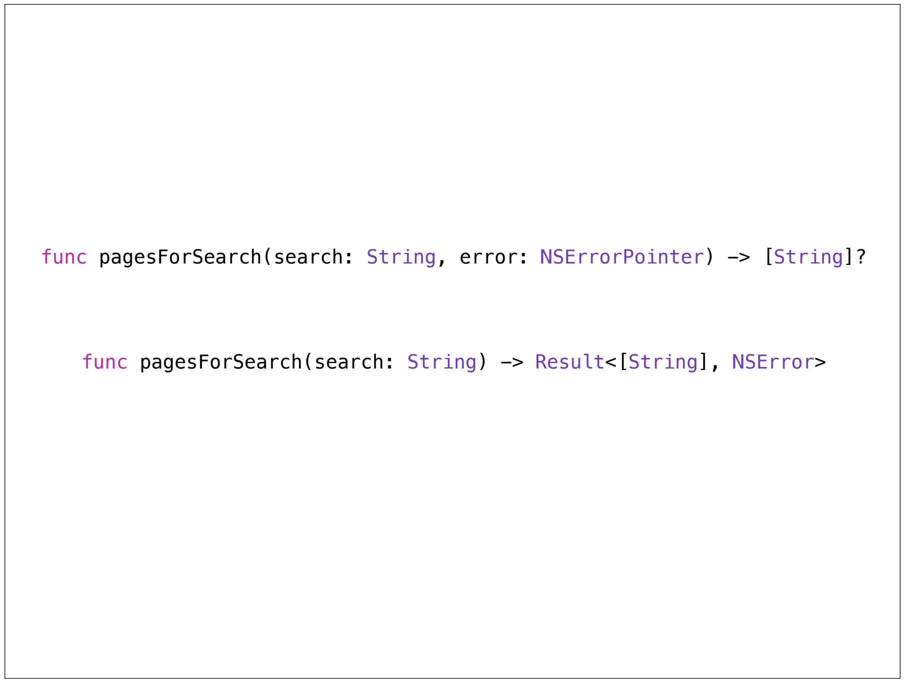
The function we've been discussing just returns an optional. If it fails, we don't get any hint about what the problem was. That's not great. The usual Cocoa approach is an error pointer. Here are a couple of examples of how that would work in this code. When we're calling Cocoa functions, this works ok.



```swift
func ParseJSON(json: AnyObject, #error: NSErrorPointer) -> [String]? {
  var err: NSError?
  if let array = json as? [AnyObject] {
    if array.count == 2 {
      if let list = array[1] as? [String] {
        return list
      } else { err = mkError("Malformed array: \(array)") }
    } else { err = mkError("Array incorrect size: \(array)") }
  } else { err = mkError("Expected array. Received: \(json)") }

  if error != nil { error.memory = err }
  return nil
}

func pagesForSearch(search: String, #error: NSErrorPointer) ->
[String]? {
  if let url = URLForSearch(search, error: error) {
    if let data = DataForURL(url, error: error) {
      if let json: AnyObject = JSONForData(data, error: error) {
        return ParseJSON(json, error: error)
      }}}
  return nil
}
```

But the pattern is kind of ugly in pure Swift code. We wind up with errors in else legs, a long way from the actual cause of the error. That makes it hard to refactor this code. And we have to use this little piece of error ! = nil boilerplate to check if we can set the error and then set the error. It's not terrible, but it could be better.

flatMap doesn't directly help us here, because flatMap can't provide an "else" leg. We could create helper functions to solve this, and I've tried that, but there's another way we can go.

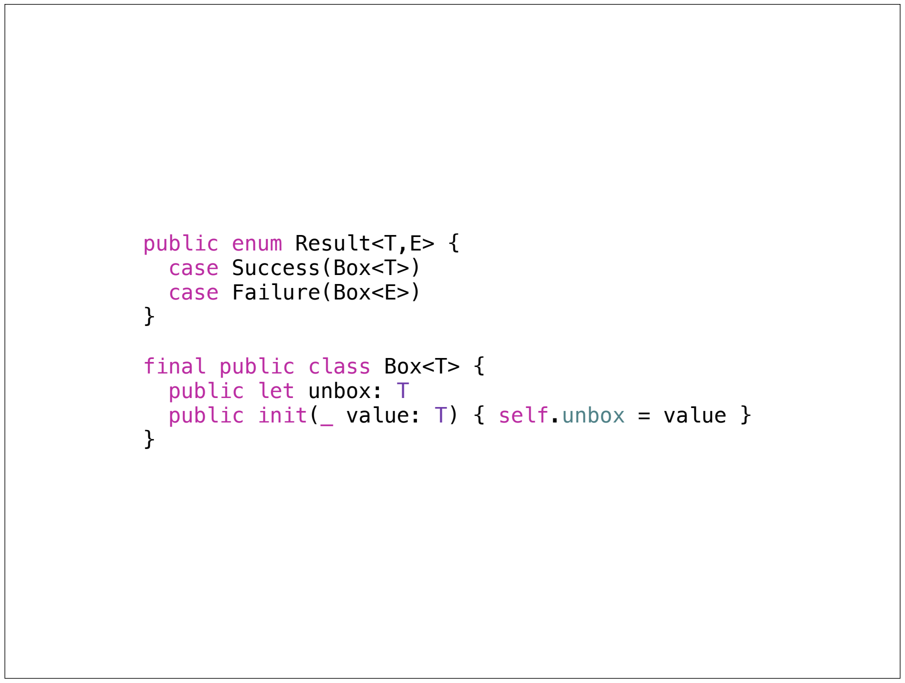


```swift
func pagesForSearch(search: String, error: NSErrorPointer) -> [String]?

func pagesForSearch(search: String) -> Result<[String], NSError>
```

What if we could change this. Into this.

<build>

So what’s a Result? Well, it’s either a value or an error. In some languages we’d call that an Either, but here we’ll call it a Result to make it clear that one thing it could be is a value, and the other thing it could be is an error.

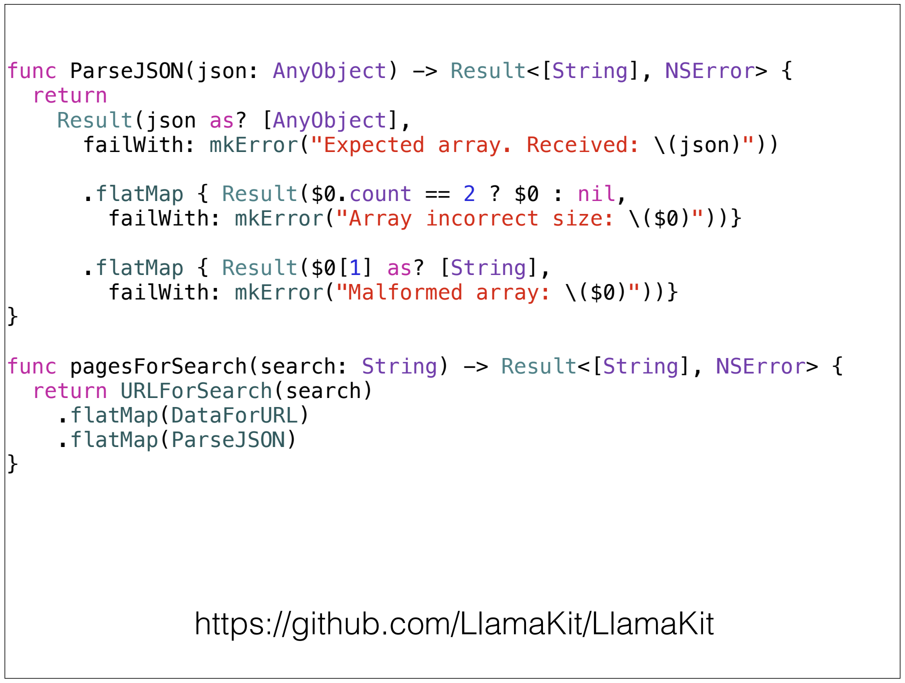


```
public enum Result<T,E> {
  case Success(Box<T>)
  case Failure(Box<E>)
}

final public class Box<T> {
  public let unbox: T
  public init(_ value: T) { self.unbox = value }
}
```

So how's a Result work? Well, it's just an enum. Now Swift has some unfortunate limitations around putting generic things in enums. So we have to put our values in something with a known size. We call that thing a box. This is just because of a missing feature in Swift, and there's a decent chance it'll be fixed soon and we won't need boxes anymore. But today you do. To use a Result enum, you need a Box class.

```swift
func ParseJSON(json: AnyObject, #error: NSErrorPointer) -> [String]? {
  var err: NSError?
  if let array = json as? [AnyObject] {
    if array.count == 2 {
      if let list = array[1] as? [String] {
        return list
      } else { err = mkError("Malformed array: \(array)") }
    } else { err = mkError("Array incorrect size: \(array)") }
  } else { err = mkError("Expected array. Received: \(json)") }

  if error != nil { error.memory = err }
  return nil
}

func pagesForSearch(search: String, #error: NSErrorPointer) ->
[String]? {
  if let url = URLForSearch(search, error: error) {
    if let data = DataForURL(url, error: error) {
      if let json: AnyObject = JSONForData(data, error: error) {
        return ParseJSON(json, error: error)
      }}}
  return nil
}
```

So, what does the same code look like with Result? Here was the original code.



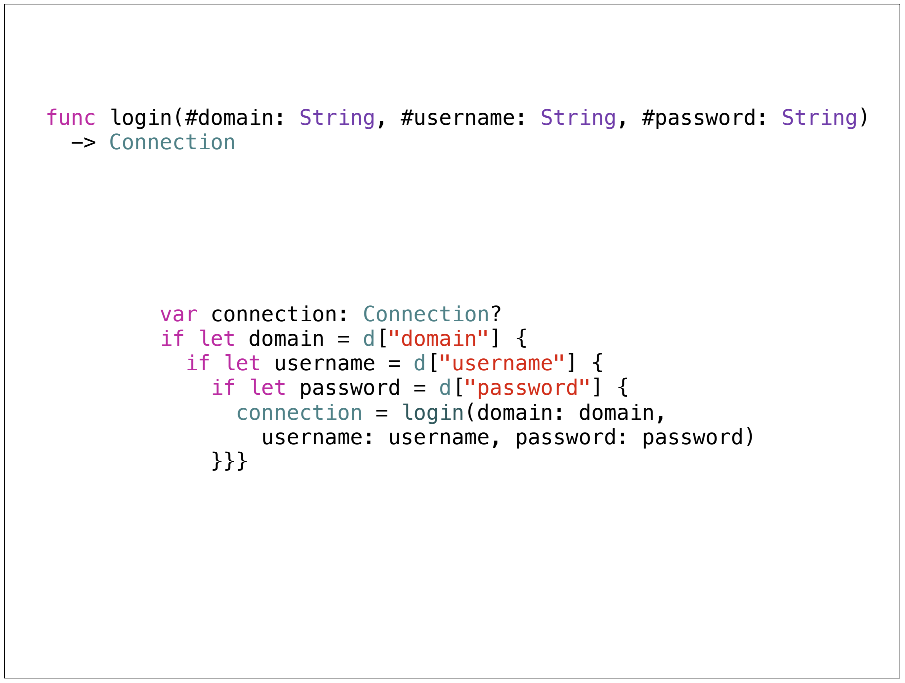
```
func ParseJSON(json: AnyObject) -> Result<[String], NSError> {
  return
    Result(json as? [AnyObject],
      failWith: mkError("Expected array. Received: \(json)"))

      .flatMap { Result($0.count == 2 ? $0 : nil,
        failWith: mkError("Array incorrect size: \($0)"))}

      .flatMap { Result($0[1] as? [String],
        failWith: mkError("Malformed array: \($0)"))}
}

func pagesForSearch(search: String) -> Result<[String], NSError> {
  return URLForSearch(search)
    .flatMap(DataForURL)
    .flatMap(ParseJSON)
}
```

https://github.com/LlamaKit/LlamaKit

And here it is with Result. Now the error generation is close to the code that generated it. That makes it easier to track down errors and to reorganize the code. And overall, the code I think it just less cluttered and easier to follow. It's still complicated because the problem is complicated. But more of the code here is focused on the actual problem, and less on the infrastructure.

We don't have time to go through all of the details of how Result works and exactly how to use it, but I wanted to give a hint about something you might dig into later. If you're interested in this idea, take a look at the LlamaKit framework. This is still in the experimental stage, but it's already fairly useful for these kinds of problems.

Just like optionals, you see we can flatMap results, so if the result is successful, we keep going, otherwise we return the first error we encountered. So there's that flatMap pattern again, and we can keep reusing these small ideas and building more complex things on top of them.

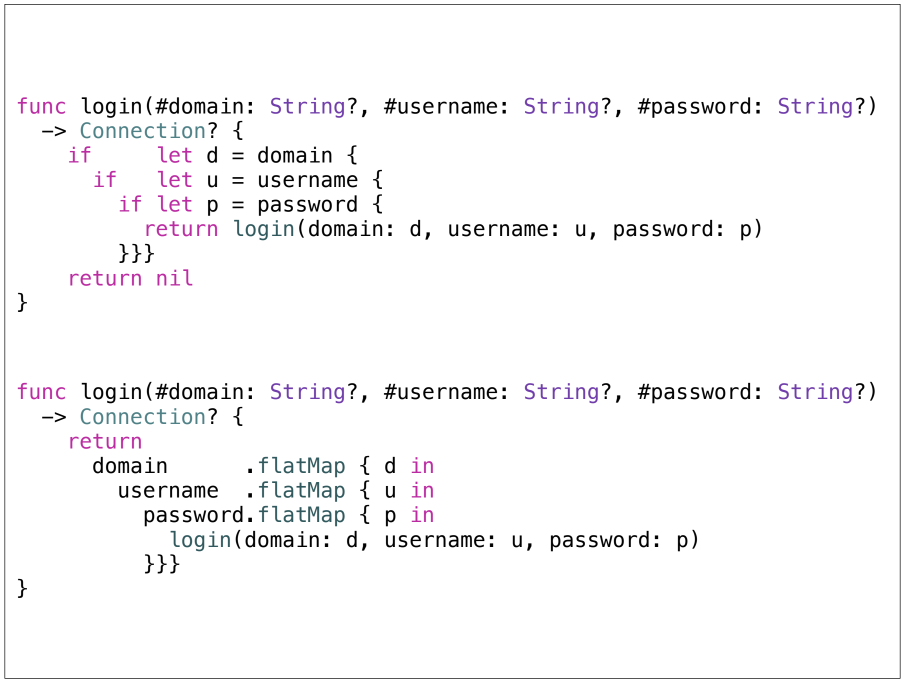


```
func login(#domain: String, #username: String, #password: String)
  -> Connection




         var connection: Connection?
         if let domain = d["domain"] {
           if let username = d["username"] {
             if let password = d["password"] {
               connection = login(domain: domain,
                 username: username, password: password)
             }}}
```

Another flatMap point. All the examples I've given here rely on the fact that each step needs exactly the input from the previous step, and nothing else. That's not always the case. Say you have some function that needs several values, all of which you might fail to have. Maybe you have them all in a dictionary. So you wind up with code like this.

A lot of blogging pixels have been spilt on trying to make this nicer. I'm convinced that Swift will, in the not too distant future, create a specialized syntax to handle this case. A lot of other languages have one. Scala calls it a for-comprehension and Haskell calls it do-notation. In any case, there are language improvements that can fix this. But what do we do in the meantime?

Frankly, 90% of the time, I think the answer is that you just wrap it.

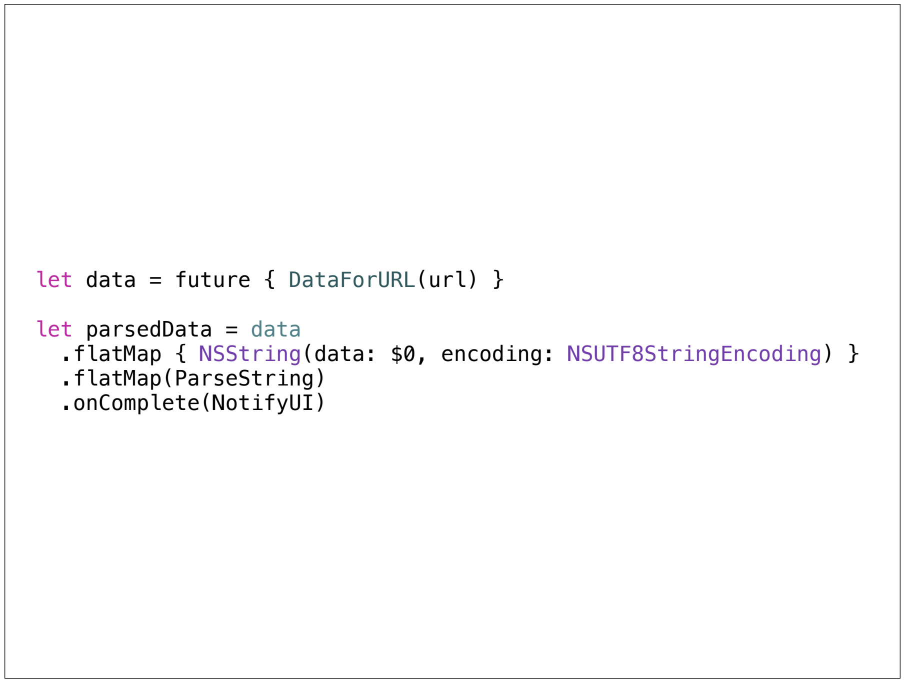


```
func login(#domain: String?, #username: String?, #password: String?)
  -> Connection? {
    if     let d = domain {
      if   let u = username {
        if let p = password {
          return login(domain: d, username: u, password: p)
        }}}
    return nil
}


func login(#domain: String?, #username: String?, #password: String?)
  -> Connection? {
    return
      domain      .flatMap { d in
        username  .flatMap { u in
          password.flatMap { p in
            login(domain: d, username: u, password: p)
          }}}
}
```

I just overload the function with a version that takes optionals and passes it along to the version that takes non-optionals. Or you could just do the work right here, since a function that takes optionals will also take non-optionals.

Now, you can of course use flatMap here, and I think it's slightly better, but not as much as you'd hope.

The problem is that you need to nest the flatMaps in order to keep all the values around. Frankly I think it's a toss up which way to go. We just have to wait for Swift to catch up with a comprehension syntax.



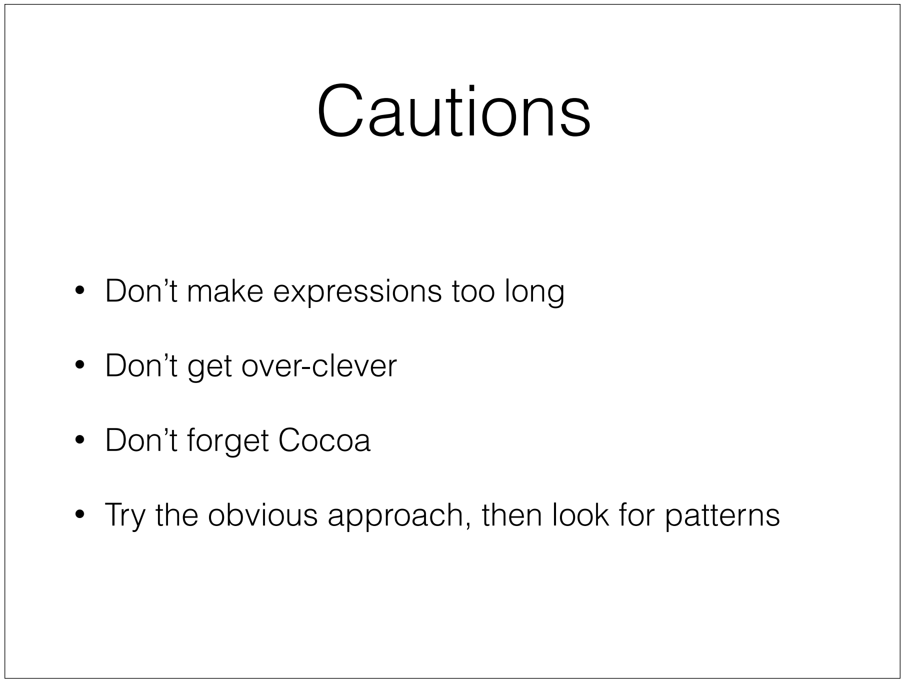
```
let data = future { DataForURL(url) }

let parsedData = data
  .flatMap { NSString(data: $0, encoding: NSUTF8StringEncoding) }
  .flatMap(ParseString)
  .onComplete(NotifyUI)
```

And just one more thing to inspire and think about. What if you could have a result, but it doesn't have a value quite yet? You just know that it'll probably have a value in the future, or an error in the future. Whatever. But you still want to think about it now, even though you don't know what it's going to be. We call that a Future, and it can look a lot like a Result or an Optional.

So what if you could do this, and let the system worry about GCD queues and all that for you. Whenever everything is downloaded, then data is going to hold either the parsed result, or an error. And you can pass this object along to other parts of the system to deal with. And when it's done processing, it can even send a notification or update the UI or whatever you want.

This idea of chaining together functions based on what they mean, like "the result of all this when it's done, or an error" without having to worry about the details of how that's handled, is the whole point of functional programming.

Final Thoughts



Cautions

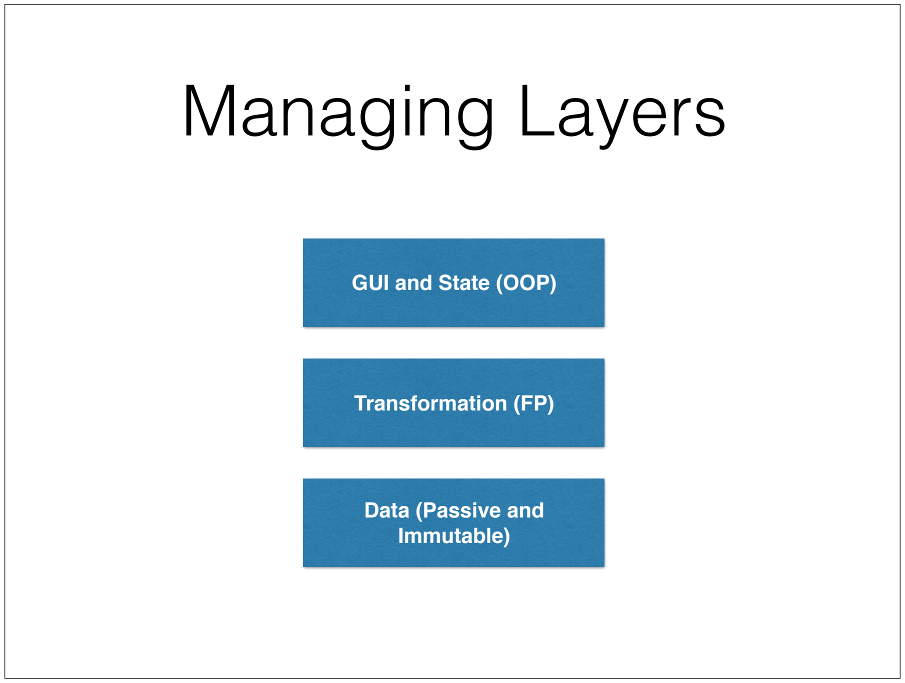
- Don’t make expressions too long
- Don’t get over-clever
- Don’t forget Cocoa
- Try the obvious approach, then look for patterns

To wrap it up, I want to give a few cautions.

<build>Because functional programming is all about chaining together expressions, it’s easy to overdo it and try to do too much in a single function. As we discussed before, break them up. Keep your functions small.

<build>On a similar note, be careful about getting over-clever. There are a lot of very fancy things you can do that at a minimum will tend to throw the compiler into fits. I’ve avoided creating customer operators here. That’s on purpose. Build things up simply. You don’t have to abstract out every repeated pattern. I’ve see so much over-complicated code from people obsessed with DRY. Sometimes it’s ok to repeat yourself a little bit in order to be clear.

<build>Don’t forget Cocoa. Swift is tied at the hip to Cocoa. It’s easy to come up with really elegant ways to do things that turn out to be wildly complicated when you have to bridge to Cocoa objects. Don’t blindly port things from other languages to Swift without considering that.

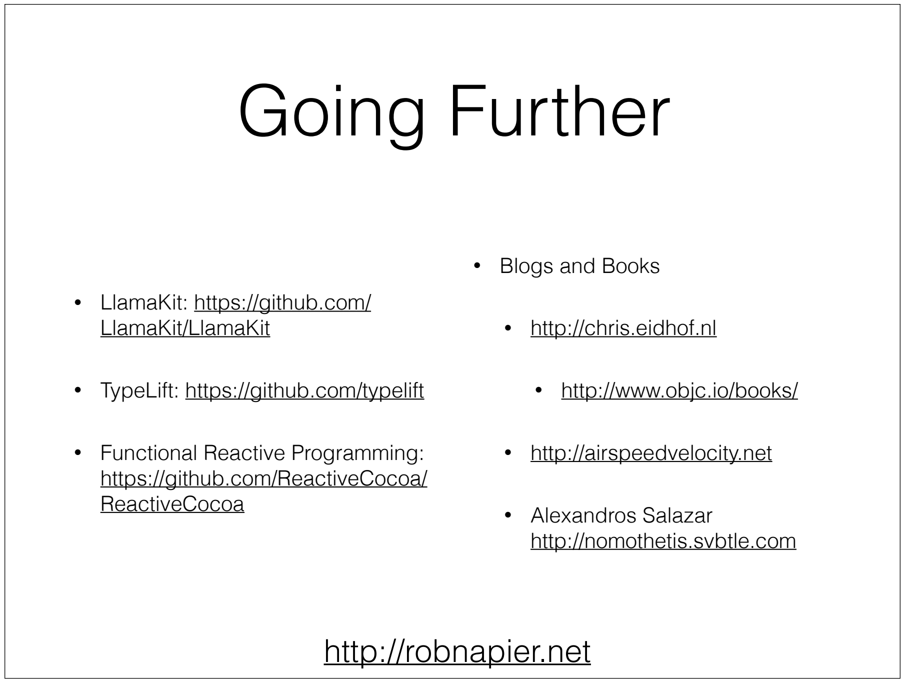


Managing Layers

Many functional ideas work well throughout every program, but at the very top, where you're taking input from the user and creating pixels on the screen, keeping everything state-free can be a real challenge. And it may not be worth that challenge.

We have ReactiveCocoa, which brings functional programming ideas to the UI. And I really want ReactiveCocoa to be the right answer, but it's definitely best engaged when you already have a fairly good grounding in functional programming. For most of you I recommend instead layering your program to manage the state rather than trying to get rid of all state.

At the bottom, you have your data objects, your model. Make those structs, and make them immutable as much as you can. Don't put too many methods on them. Keep them simple. At the top, keep track of the state of your program and deal directly with the outside world. In the middle, have transformational logic that converts data into exactly the things you need to display. That's the part to make functional, and you should try to make it as much of your program as possible.



Going Further

- LlamaKit: https://github.com/LlamaKit/LlamaKit
- TypeLift: https://github.com/typelift
- Functional Reactive Programming: https://github.com/ReactiveCocoa/ReactiveCocoa

- Blogs and Books
  - http://chris.eidhof.nl
    - http://www.objc.io/books/
  - http://airspeedvelocity.net
  - Alexandros Salazar http://nomothetis.svbtle.com

http://robnapier.net

And finally, here are links that I'll put up on my site. I'm collecting a lot of the ideas I discussed today into a very small framework I call LlamaKit. It's as much a discussion about how to do this well as a real framework. It's designed mostly to let frameworks interoperate with a shared Result, and maybe someday a shared Future and the like. But it has a mailing list, and it's a good place to discuss how this might work best in Swift.

If you want to go even deeper, there's a project called TypeLift that replaces most of stdlib and Cocoa with a system that is much more Haskell-like. So if you feel hemmed in by Cocoa, that's definitely a project to look into.

And if you're interested in pushing functional programming all the way into the GUI, you should definitely look at Reactive Cocoa. I don't know if it's the right approach today, but it's worth keeping an eye on.

And finally there are the blogs. Chris Eidhof has some great discussion of functional programming in Swift, and he cowritten a whole book on the subject. His is the place for deep FP, with fancy operators and no holds