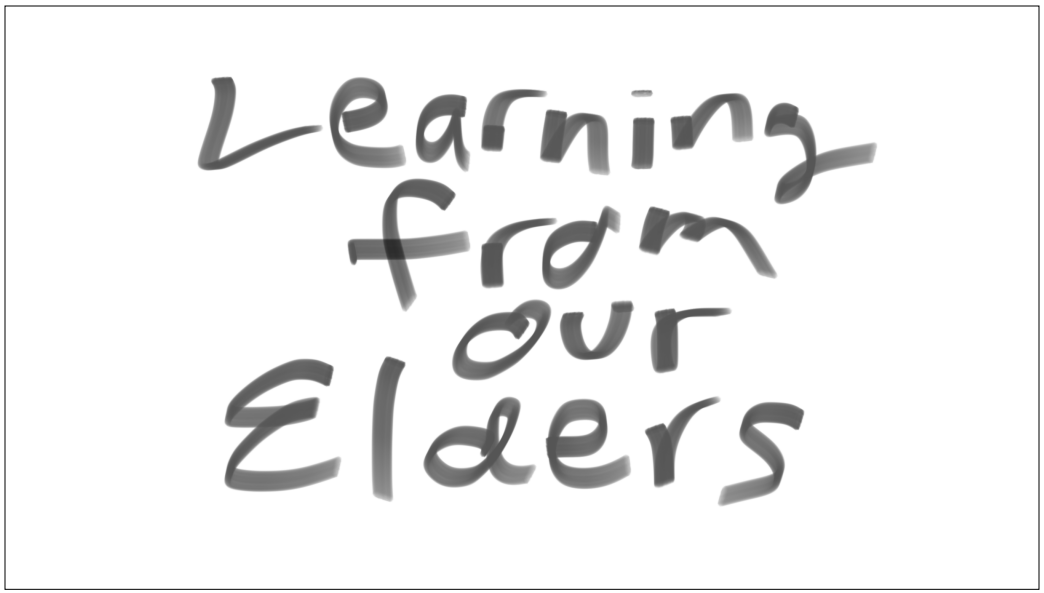
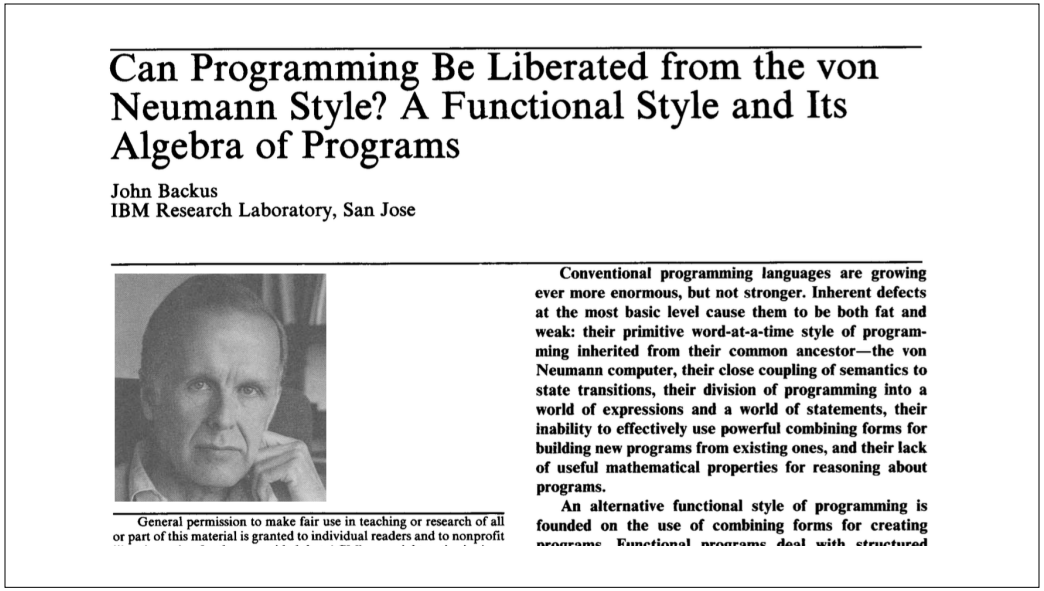

Learning
from
our
Elders



Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

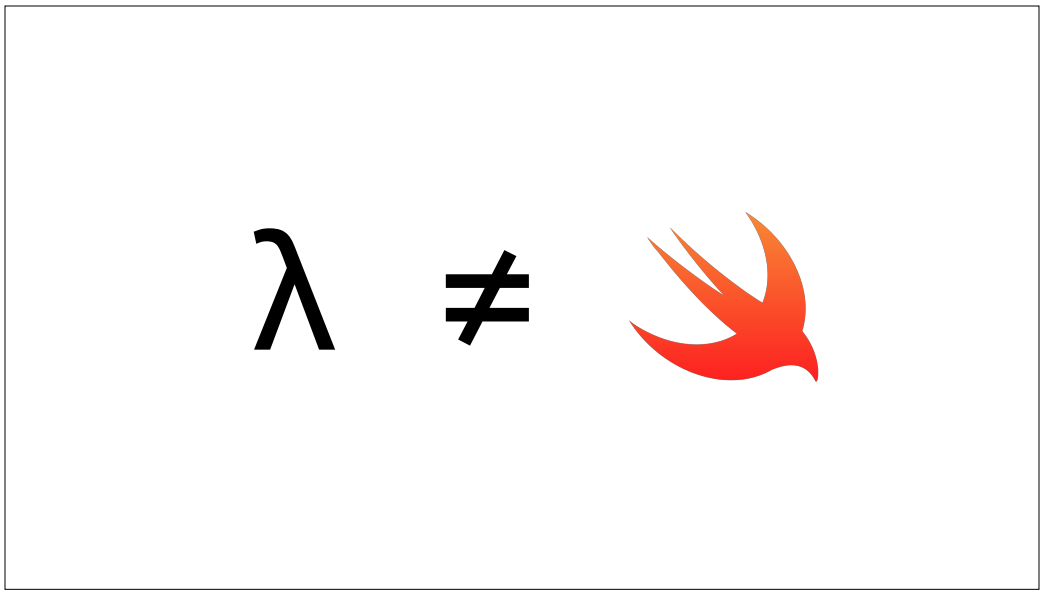
John Backus
IBM Research Laboratory, San Jose

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

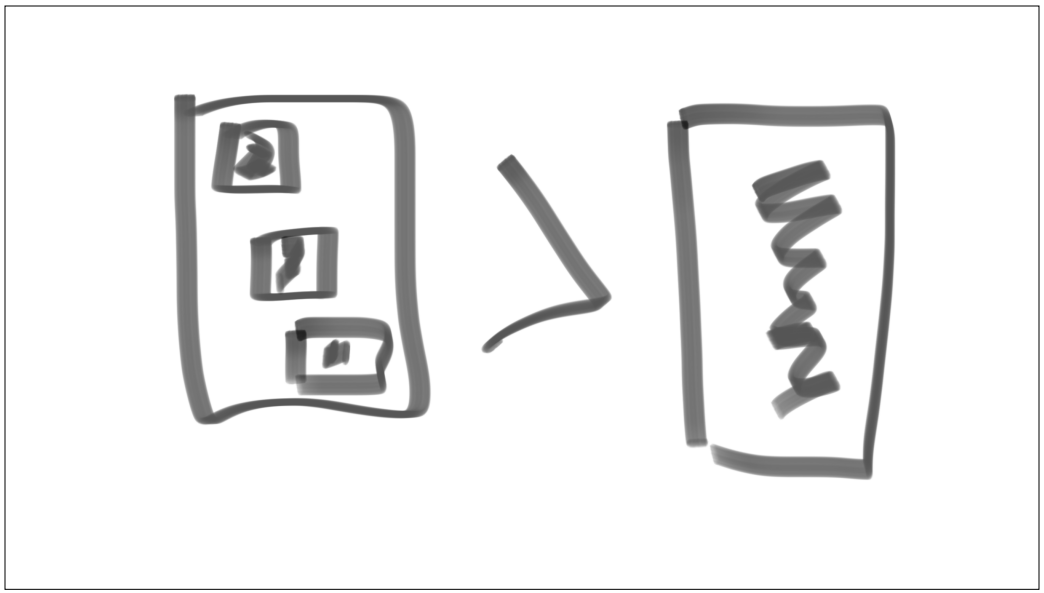
An alternative functional style of programming is founded on the use of combining forms for creating

In 1977, John Backus, who helped invent FORTRAN and ALGOL wins the Turing award. And when you win the Turing award, they let you give a lecture. This was his. “Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs.” I love that title. “Liberated.” It’s a great word for programming. By “von Neumann style” he basically means FORTRAN and ALGOL, which he’s basically apologizing for. Today he’d mean C. But generally, he means imperative programming. Step-by-step mutation of some “state” to get to a final goal.

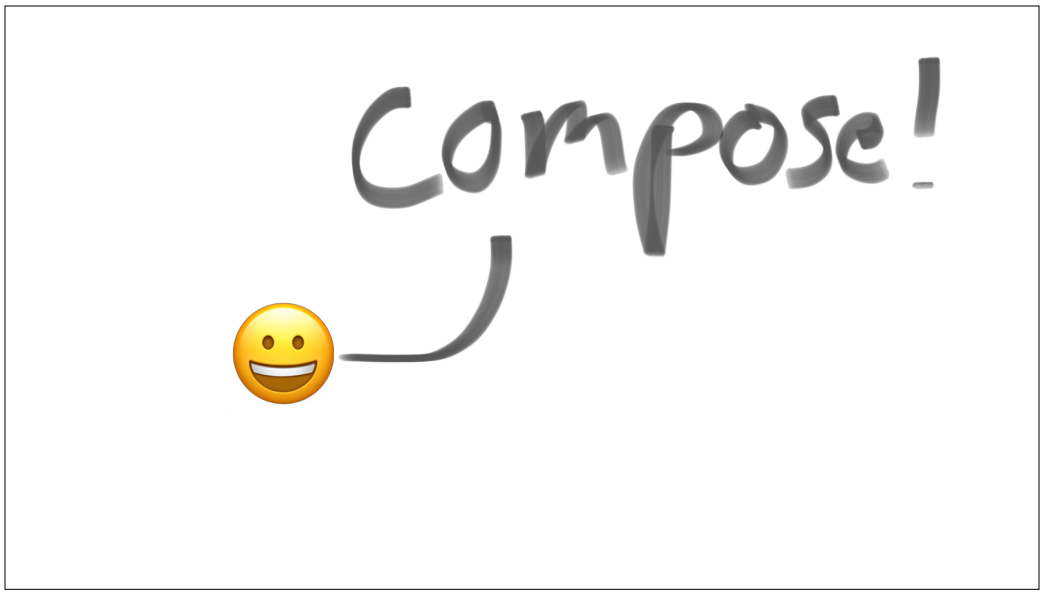
The most important ideas to me from this paper aren’t about functional programming. The important idea in this paper is that we can decompose complicated things into simple things, make those simple things generic and reusable, and we can recombine them using rules. An “algebra.” That’s what an algebra is. It’s a set of rules for tearing things apart, transforming them, and putting them together.



I want to be clear. Swift is not a functional programming language. Trying to make it one leads to all kinds headaches. But there are many lessons from decades of research in functional programming that apply to Swift. We're not going to talk about immutable data or recursion or monads. We're going to talk about algebra. An algebra of programs. We're going to talk about composition.



Building things out of existing and reusable pieces is better than building a new giant piece of custom code for every problem. There are the obvious benefits of reuse, but I think comprehension is just as important. It's possible to reason about small pieces of software in ways that can't be done on large, monolithic code bases. And I think most people today would agree with that. Even most people in 1977 agreed with that. So what's the point?



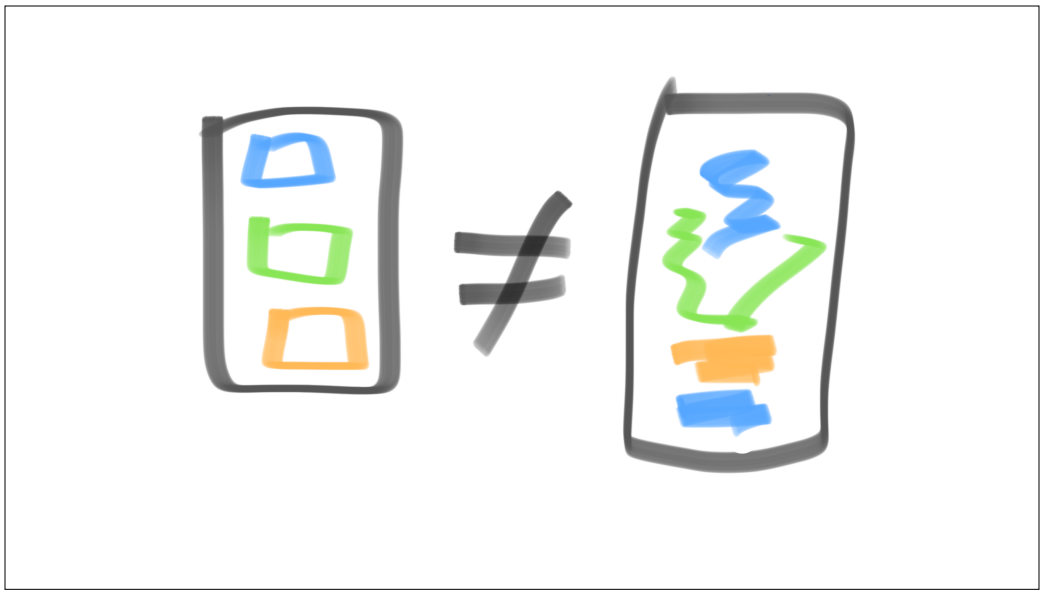
I mean we've all learned our lessons about that, right? We know composition is the way. We use delegates and protocols and we keep our binding loose and classes abstract. I mean, we don't build tightly-coupled, monolithic systems anymore like they did in the FORTRAN days.

Hmph.
I mean, our classes do just one thing, and they do it well.

```
class MyViewController: UIViewController,
                        UITableViewDataSource,
                        UITableViewDelegate,
                        UISearchResultsUpdating,
                        UISearchControllerDelegate,
                        UISearchBarDelegate,
                        MyViewModelDelegate,
                        MyDetailViewControllerDelegate {
```

Hmmph.

No Rob, don’t you see the protocols? That’s composition, right? It’s composed out of data sourcing and results updating and searching bar dele…gating…ness. I mean, we gotta put this stuff somewhere, right?



An object that conforms to a bunch of delegate protocols is not composition. It’s the opposite of composition. It’s taking things that were explicitly compartmentalized and gloms them together into one massive object.

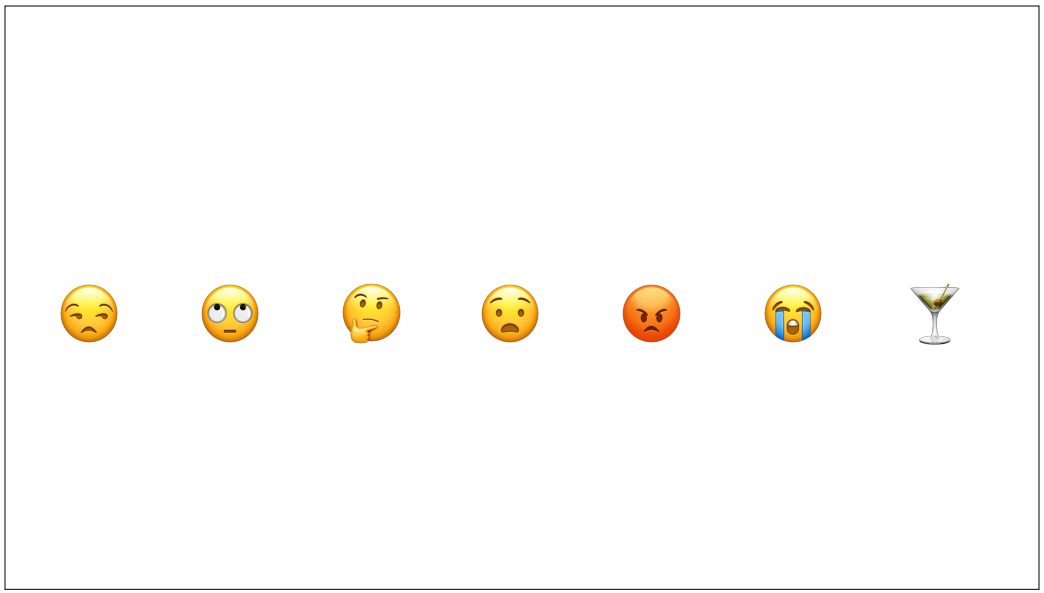
```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    if loggedIn {
        return values.count
    } else {
        return 1
    }
}

override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if loggedIn  {
        return normalCell(. . .)
    } else {
        return pleaseLoginCell(. . .)
    }
}
```

I run into this a lot. You've got a table view or collection view that has a couple of states, maybe if you're logged in you get all your instabook-tweets or whatever, but if you're not logged in you get this one "please login" cell or "loading" or whatever. So in each delegate method you have some if-else to do the right thing. And it's ok, and it doesn't bother too much. And then you add searching.

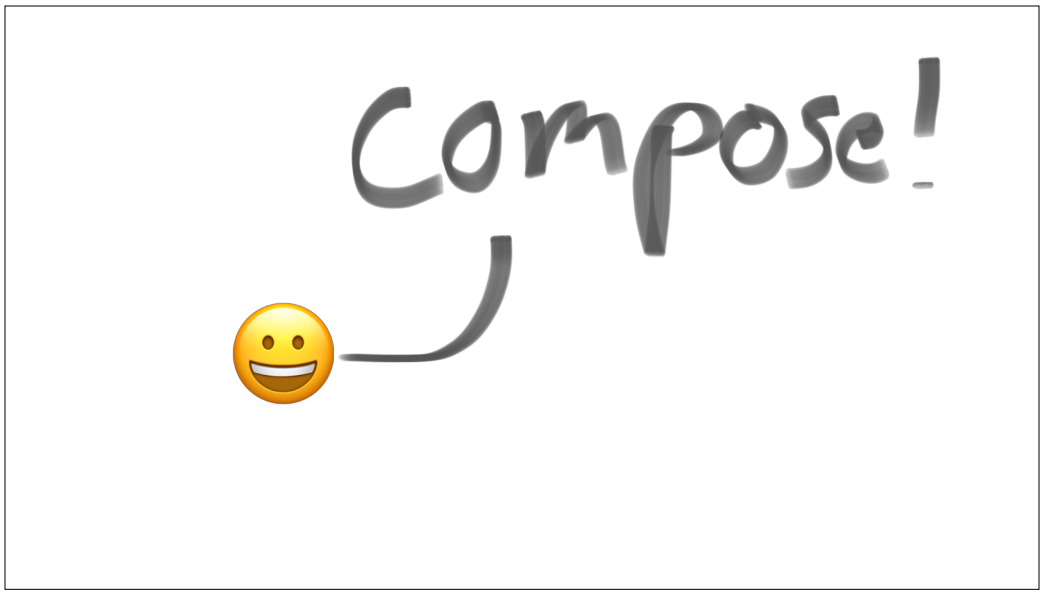
```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    if loggedIn {
        if searching {
            return searchResults.count
        } else {
            return values.count
        }
    } else {
        return 1
    }
}

override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if loggedIn {
        if searching {
            return normalCell(. . . but with the search results . . .)
        } else {
            return normalCell(. . .)
        }
    } else {
        return pleaseLoginCell(. . .)
    }
}
```

And that’s not *too* bad. I mean we can definitely work with this. Sure.

And then they call you up and say, yeah, we want people who have never logged in to get some default content to get them started, but if they had trouble logging in, then they should see an error, but if they refresh, then they should get cached content.



And then you go through the seven stages of implementing a complicated feature in this code base.
Sigh. There has to be a better way.
Glad you asked.



What if we used composition? I mean actual composition. What would that look like?

```
class BrowsingDataSource: NSObject, NSTableViewDataSource {

class SearchingDataSource: NSObject, NSTableViewDataSource {

class LoginDataSource: NSObject, NSTableViewDataSource {
```

There is no rule that says the data source has to be a view controller. There unfortunately is a rule that says it has to be an NSObject, but it doesn't have to be a view controller. I'm free to create a data source that is nothing but a data source. We can make one data source for our browsing state, and a separate data source for our searching state, and another data source for our not logged in state, and on and on. When you change states, you attach a different data sources to the table view. Each does one thing. It converts that input data into cells for the table view. Do one thing. Do it well.

Now this doesn’t save us from messy conditional logic. That’s your business logic that handles all the special unicorns of your magical rainbows app. It is messy. It deals with people, and people are messy. They want messy, special, inconsistent things. But, now that logic can all live in just one function and it isn’t spread out over every delegate method.

```
class BrowsingDataSource: NSObject, NSTableViewDataSource {

class SearchingDataSource: NSObject, NSTableViewDataSource {

class LoginDataSource: NSObject, NSTableViewDataSource {
```

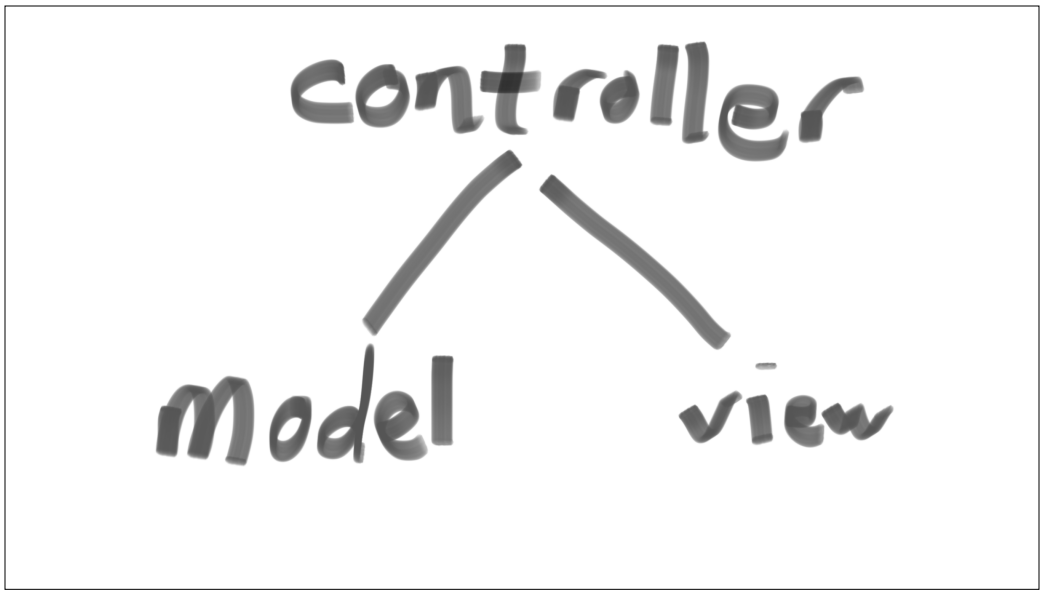
So breaking it up makes simpler code, but what else does it do? Let's look at these two. **<build>**
When you think about it, browsing and searching are really the same thing. One just has a filter. I mean really, both have a filter, just one of the filters says "show everything."

So maybe we could put those together. We could build a fetch request data source.
Yes, brilliant. Wait. No. That’s never going to work. I mean we could do it, yeah, but we definitely couldn’t reuse it. I mean everyone knows that this is how you configure cells.

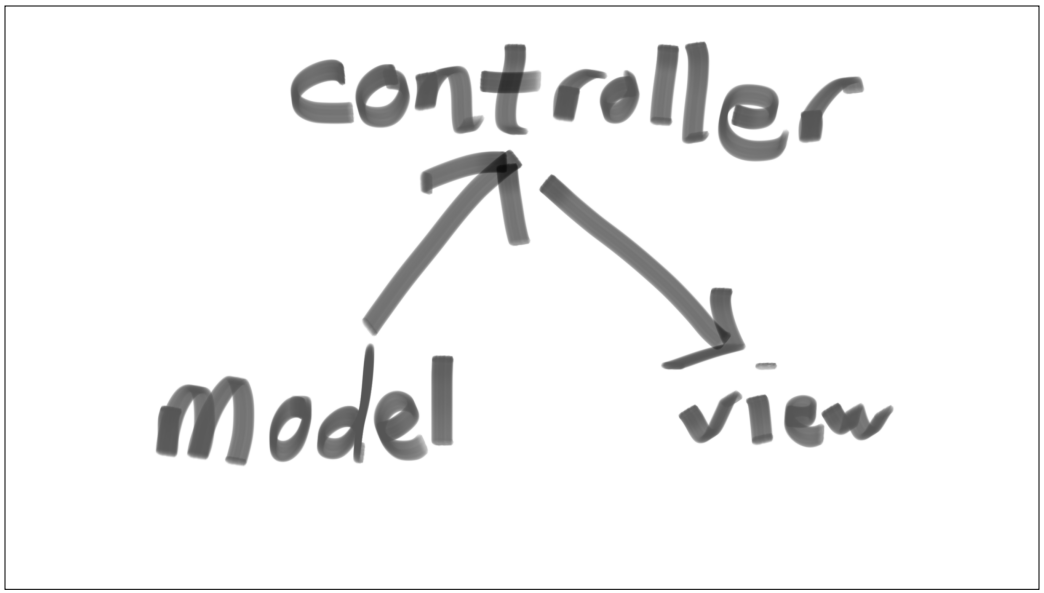
```
func tableView(_ tableView: UITableView,
                cellForRowAt indexPath: IndexPath) ->
                UITableViewCell {

    let cell = tableView
                    .dequeueReusableCell(withIdentifier: "Person")
                    as! PersonCell

    let person = persons[indexPath.row]

    cell.textLabel?.text = person.name
    cell.imageView?.image = person.thumbnail
    . . . and more and more . . .

    return cell
}
```

You get the cell. You get your model object, and then you plug each piece of data into the cell's subviews. And since every cell is different, there's no way to make this generic.

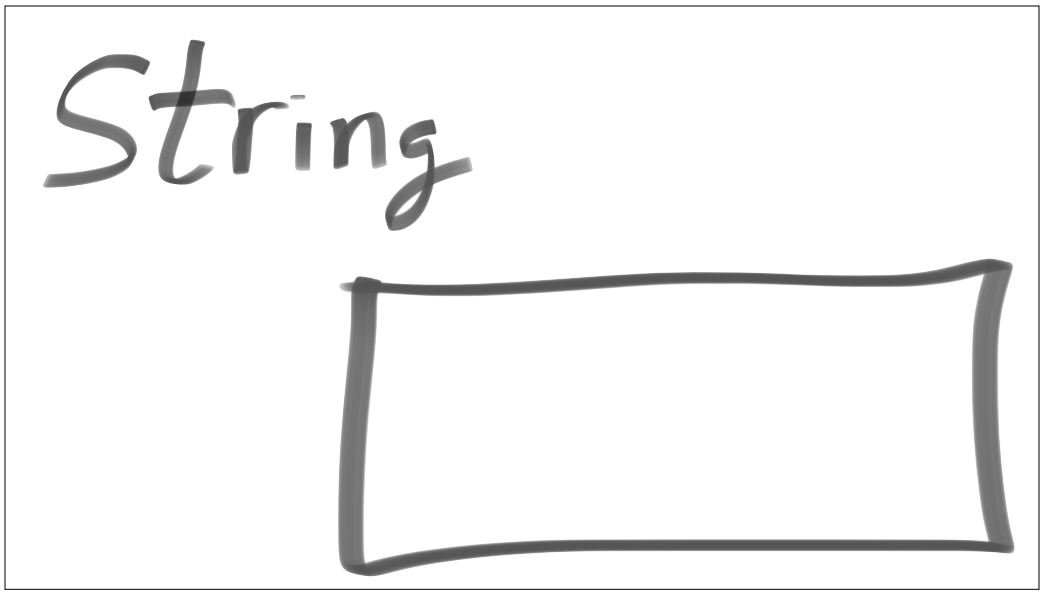
You're right. Because this code is wrong. And it's wrong because we don't understand the point of MVC. We don't understand correct composition in Cocoa.



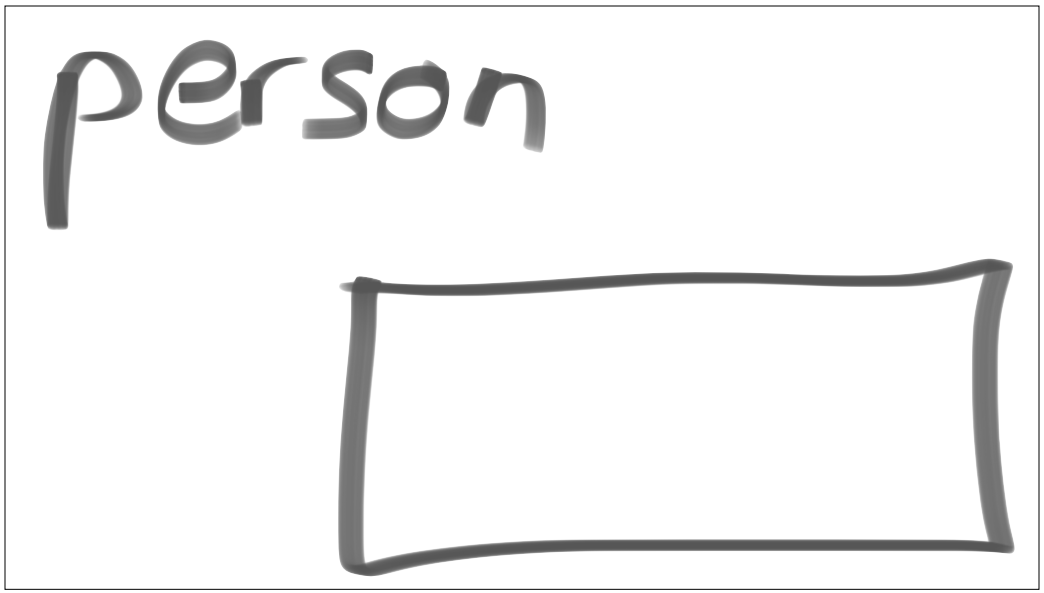
Most apps I’ve dealt with basically get this idea. Controllers glue together models and views. Great.



But I think a lot of apps miss is that controllers actually glue models and views together. That doesn't mean they tear about models and manipulate the inner workings of views. It means they coordinate getting the right model to the right view. They hand models to views for display. It's up to a view to display it. That includes understanding the model object they're handed.



I mean, you hand a string to a label. **<build>**That's the label's model object.
You don't tear the string down into characters and convert them to glyphs and then inject each glyph into the right place in the view. That's the job of the label. The label accepts it model, and displays it.



The same is true about higher level model objects. Your person cell should accept a whole person**<build>**
and should take care of all the display logic, including animations that happen inside the cell. The amount of complexity I've seen in view controllers trying to manage animations inside cells while they're scrolling is crazy. Each cell should take care of displaying itself. The cells should be coordinated by the controller. That's composition.

```
func tableView(_ tableView: UITableView,
                cellForRowAt indexPath: IndexPath) ->
                UITableViewCell {

    let cell = tableView
                    .dequeueReusableCell(withIdentifier: "Person")
                    as! PersonCell

    let person = persons[indexPath.row]

    cell.textLabel?.text = person.name
    cell.imageView?.image = person.thumbnail
    . . . and more and more . . .

    return cell
}
```

And when you do that, then this complicated logic, configuring the cell

```
func tableView(_ tableView: UITableView,
                cellForRowAt indexPath: IndexPath) ->
                UITableViewCell {

    let cell = tableView
                    .dequeueReusableCell(withIdentifier: "Person")
                    as! PersonCell

    cell.person = persons[indexPath.row]

    return cell
}
```

becomes this. One assignment of a model object. And you can make it generic.

```
func tableView(_ tableView: UITableView,
                cellForRowAt indexPath: IndexPath) ->
                UITableViewCell {

    let cell = tableView
                    .dequeueReusableCell(withIdentifier: identifier)
                    as! Cell

    cell.model = models[indexPath.row]

    return cell
}
```

I can make Cell and Model generic types. I can say that whatever cell you pass me has to have a settable model property using a protocol.
And now the only thing this data source has to worry about is how many rows there are and how to convert index paths into the correct model object. My data sources get simple. They get focused We've torn them down until they only solve one problem.

```
class FetchRequestDataSource: NSObject, NSTableViewDataSource {
```

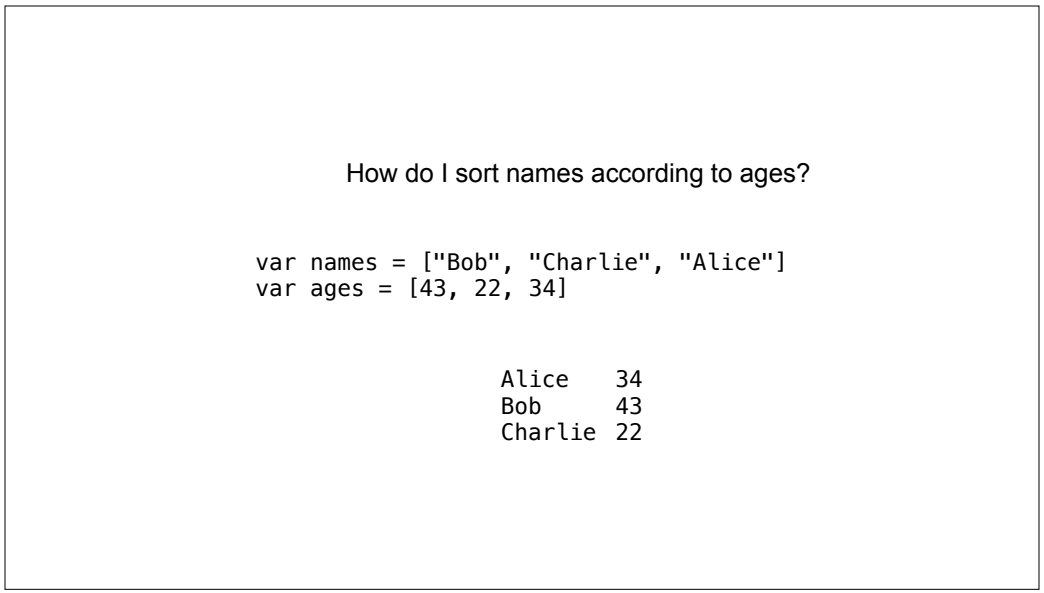
```
class LoginDataSource: NSObject, NSTableViewDataSource {
```

And I can do the same thing to my LoginDataSource.

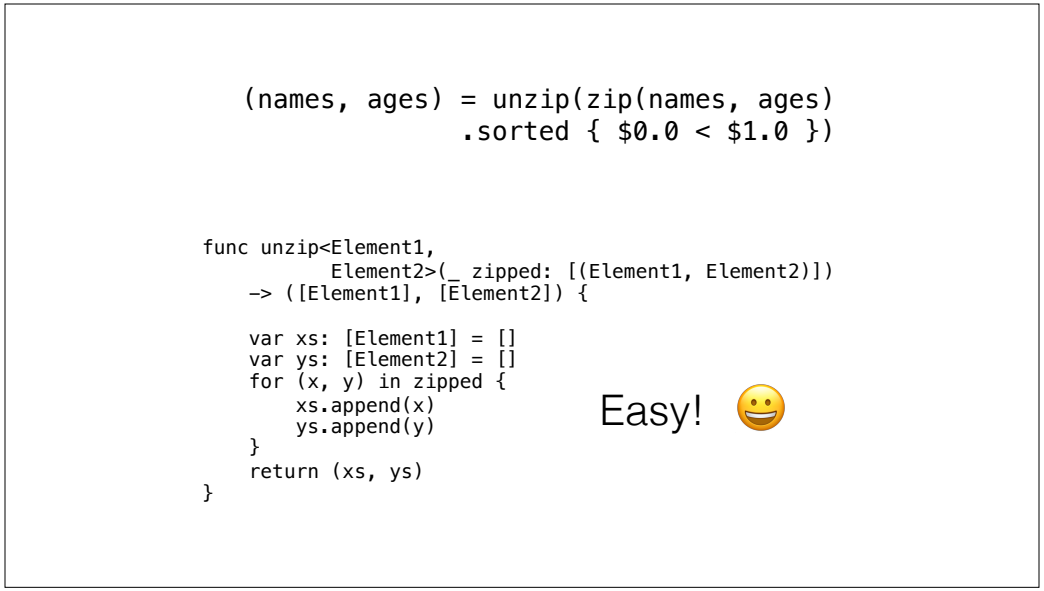
```
class FetchRequestDataSource: NSObject, NSTableViewDataSource {
```

```
class StaticCellDataSource: NSObject, NSTableViewDataSource {
```

It could be a StaticCellDataSource that hands back just one cell and you initialize with a storyboard and a cell identifier. And you can start to imagine Array data sources and network response data sources. And with closures and generics this gets very powerful and reusable.
And now that we've torn everything apart and made them all small and simple, you start to think about how you can put things together to solve problems that are big and complicated.



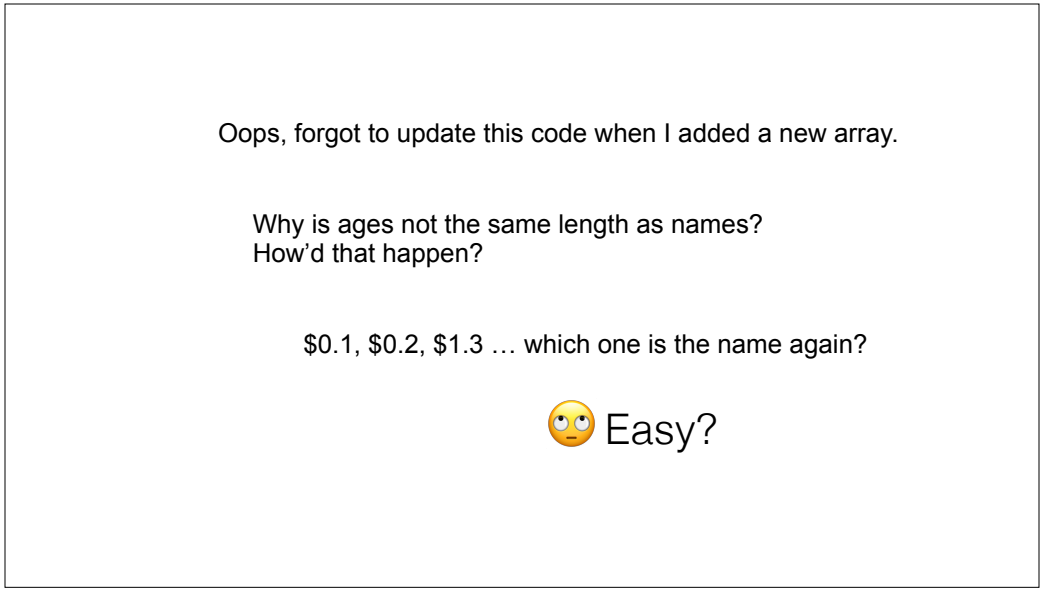
OK, new example. This is a question I see a lot. I have two arrays, names and ages, and I want to sort them “together,” one based on the other. So sort names and make ages match.



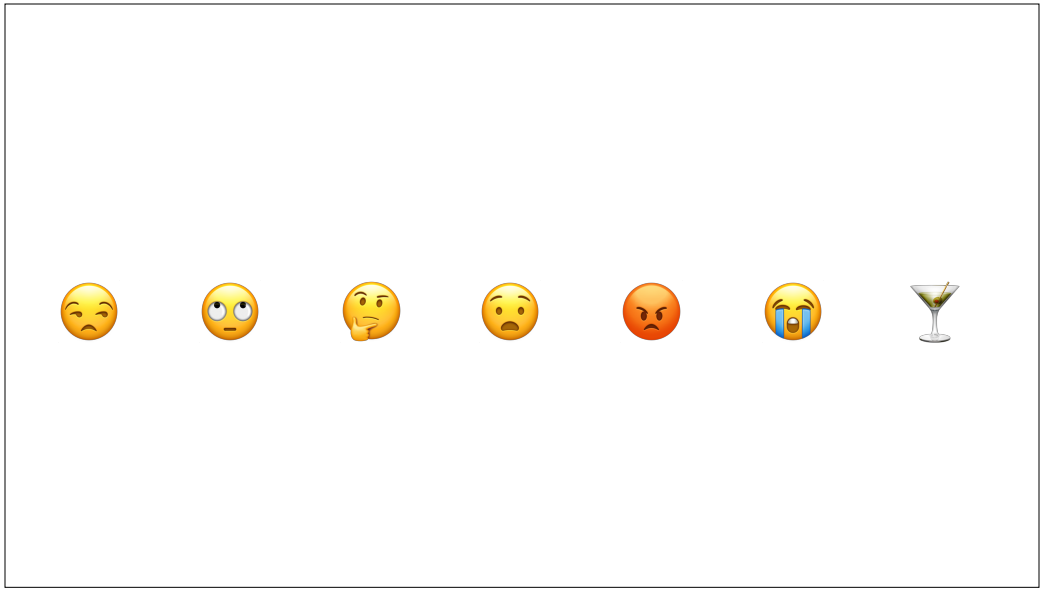
And a lot of times, someone answers along these lines. You zip all the arrays together into tuples, and then you sort the tuples, then you unzip them back into arrays or something like that.

<build> And maybe your build this little unzip helper function because for some bizarre reason Swift doesn't have it.

<build> But hey, there you go. Easy.



And then you add another array for userids and you need go back and remember to sort it, too. And every time you add a new field, you need to remember all the little places you were keeping your arrays in sync for adding or deleting or moving things. And you finally get it all working and put to bed for a few months and go on work on other things, and then your boss calls you up and says hey, could you sort by manager first, and then by name? And you look at your code, and remember all the little corner cases. And. Well.



Sigh.

Simple! 😀

```
struct Person {
    let name: String
    let age: Int
}

var persons = [Person(name: "Alice", age: 43),
               Person(name: "Bob", age: 34),
               Person(name: "Charlie", age: 22),
               ]

persons.sort { $0.age < $1.age }
```

Instead of having many arrays of one property each, we could have one array of a struct with many properties. We've turned it inside out. Instead of spreading implementation details through all the users of this data, we have one type, Person, that encapsulates everything, and now we can add new properties without messing with all the existing code.

An struct of arrays is the dual of an array of structs. A "dual" is when you turn a data structure inside out, or look at it from another direction, slice it another way, so it can do all the same things, but maybe in a simpler way. The same idea comes up over and over.

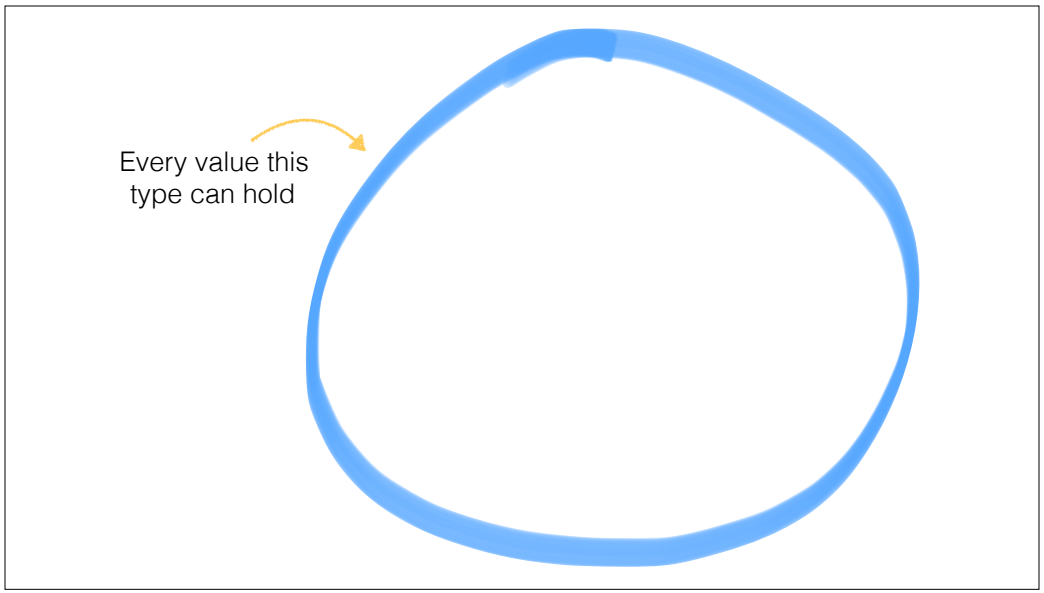
```
class PersonView: UIView {
    var name: String?
    var age: Int?
    var city: String?
}
```

For instance, I see this kind of thing all the time. Some class that has a whole bunch of optionals in it. But that probably isn't what you mean. Does it make sense in your program to have an age, but no name? If all of these things need to be set together, then turn it inside out. Lift them into a type.

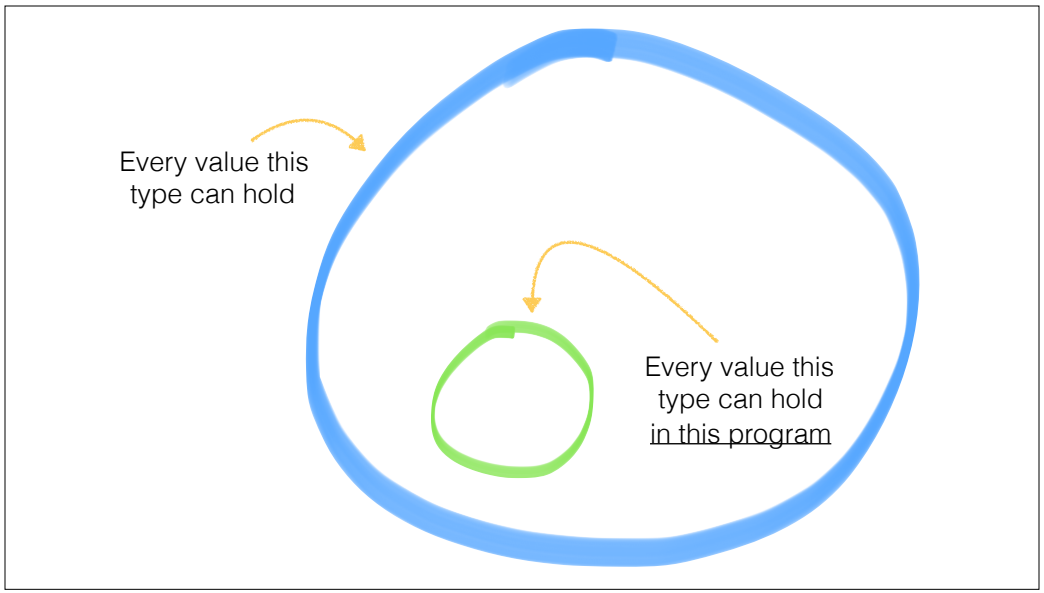
```
struct Person {
    var name: String
    var age: Int
    var city: String

}

class PersonView: UIView {
    var person: Person?
}
```

Now there's only one optional, and either all the data is there or none of it.

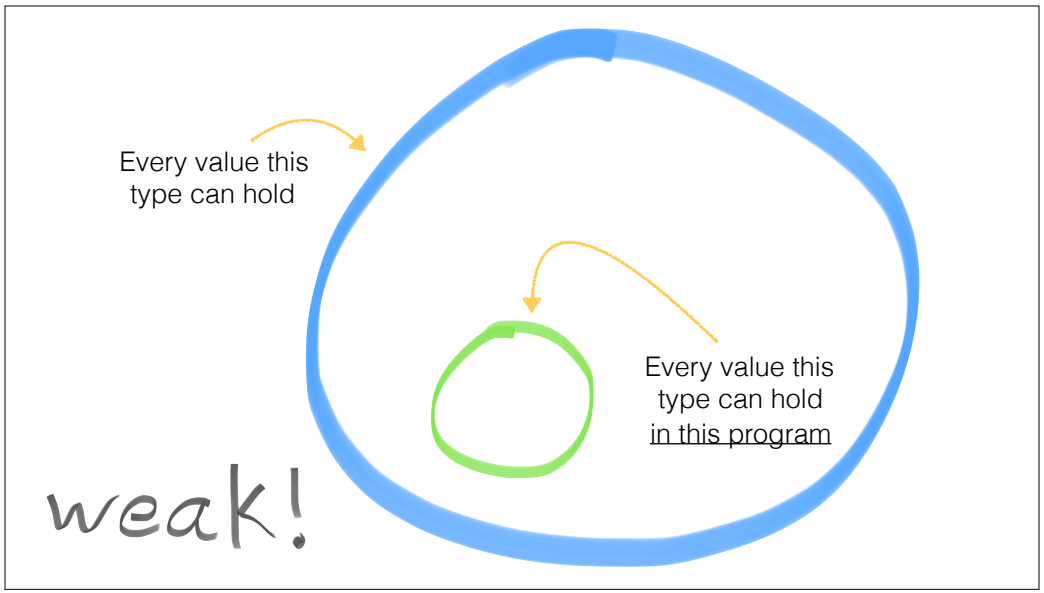
It's all about lifting raw data into types. What's a type?



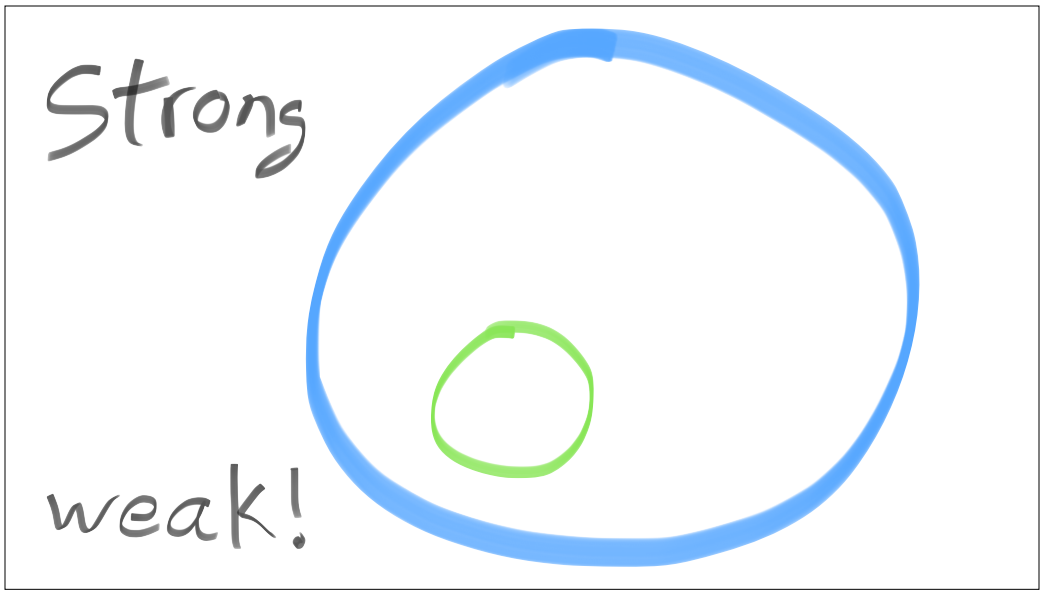
Let's think of a Venn diagram of every value a type could hold. So for an integer, that would be like 2^{64} values or whatever. Or for a dictionary, it would be every possible key-value pair that would be legal to put in the dictionary without creating a compiler error. this is really what "type" means. It's a named subset of values, out of all the possible values that could be expressed in your language.



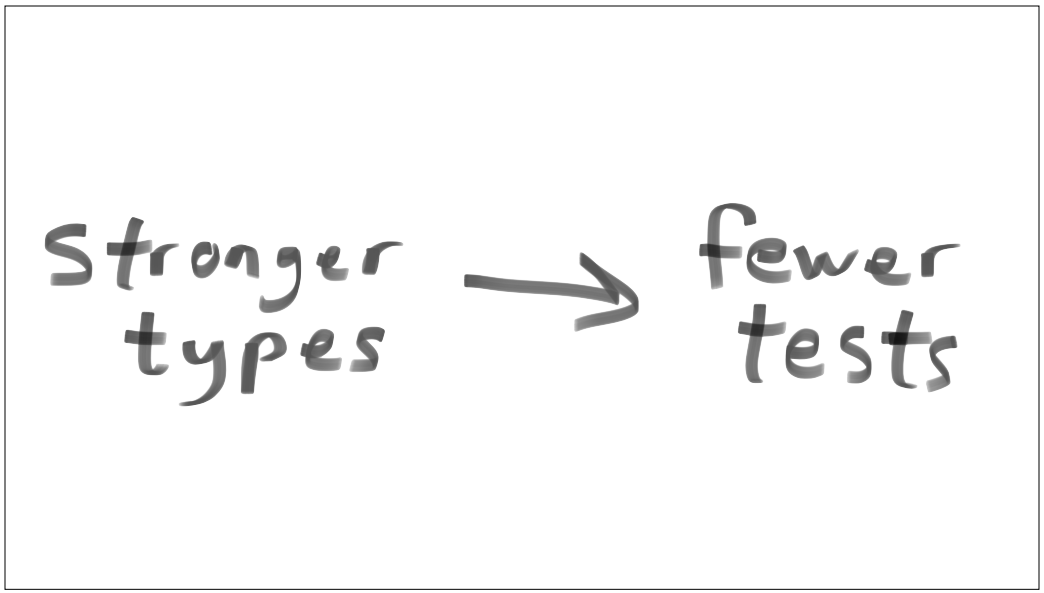
Now let's think about every value that would be legal to store in that type in this program. So if you've represented your current state, like loading, viewing, editing, as an integer, you're using a type that could hold something something quintillion values to hold one of 3 actual values. That's a pretty weak type.



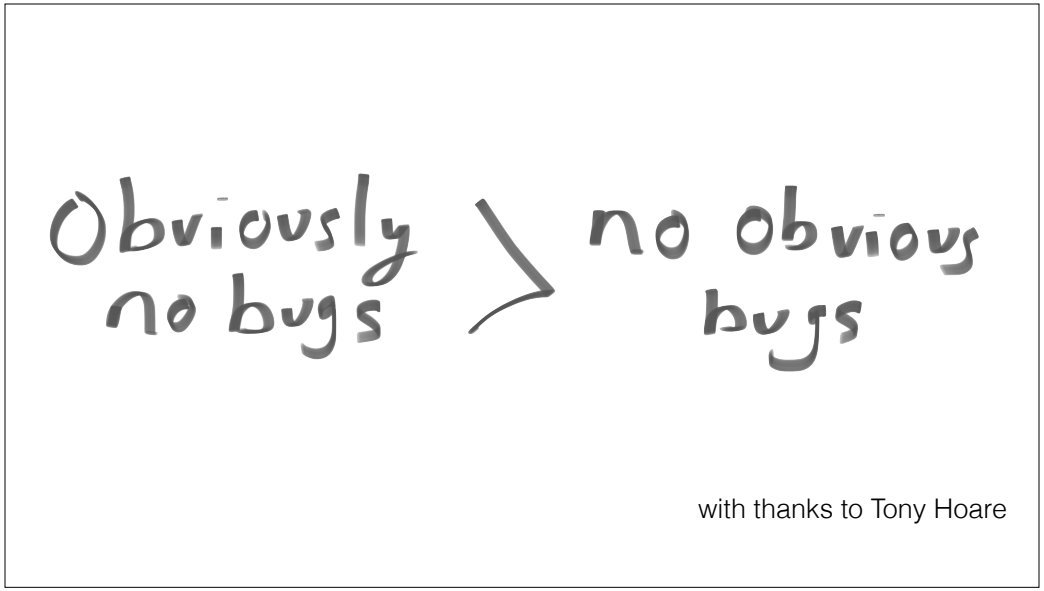
And if you're using a string that could hold any of, I-don't-even-think-we-have-names-for-those-numbers possible values, to hold one of say twenty possible strings, that's a really weak type.
Does that mean that strings are always weak? Not at all. If your data is "arbitrary text from the internet," then string is strong type. Its possible values match, at least vaguely, its legal values. But if you can reasonably count all the legal values of your string variable, you're probably using it in a weak way.



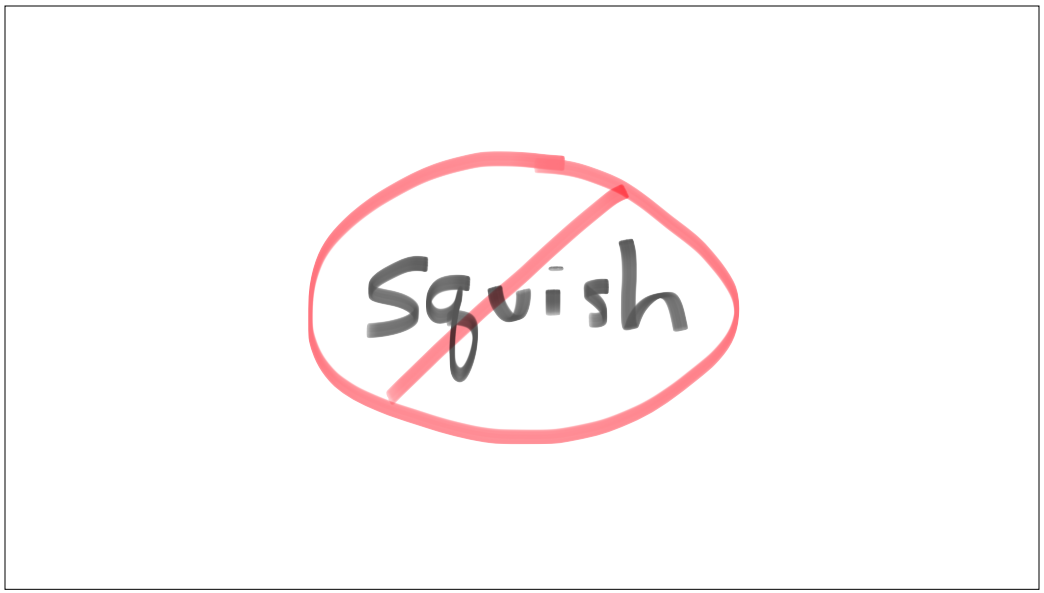
So here’s the maybe counter-intuitive part. When we want to make our types strong, we shrink them.<build>
We give them fewer values. Ideally, exactly the same as the number of legal values.



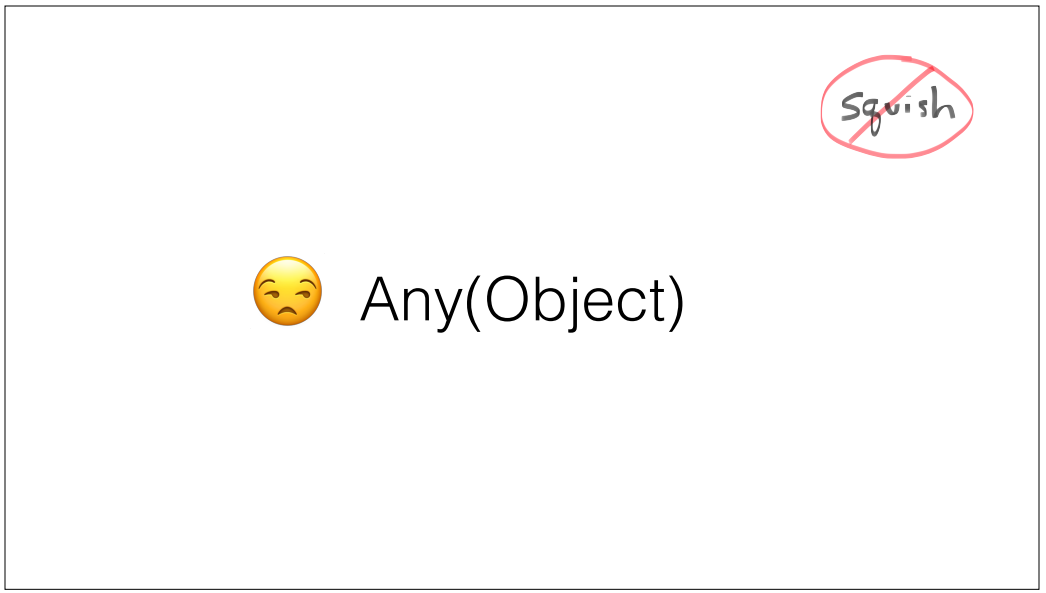
The more places our types can hold illegal values, the more time we spend writing preconditions and unit tests to make sure they don’t. The more constrained our types, the stronger our types, the more whole classes of bugs are impossible. Stronger types require fewer tests.



I'd rather code that obviously has no bugs, that is so simple and type constrained that its only possible behavior is correct behavior, than a program that I've tested enough that it has no bugs that I've been able to find.



So what does that mean in practice?**<build>** It means to look carefully and critically at your squishy, squishy types and make them stronger.



Any and AnyObject are almost the squishiest types available. Believe it or not, there are types in Swift that are even squishier than Any, but they all have Unsafe in their names and if you want to talk about why they're squishier, we can go talk afterwards, but for now, let's say Any is the squishiest type. It can hold anything, right? So it's as large as our entire universe of Swift values. It is very rare that you would want that, really. The print function comes to mind, and not much else.

Now because of Cocoa, Any and AnyObject pop up sometimes, especially in JSON parsing. (Do we talk about anything other than JSON parsing?) But once you've dealt with parsing, it should have a struct or a class or something specific. Unless you're intentionally tricking the complier, there is no reason to store Any in a property.

After Any, Dictionary is probably the most abused type in Swift. A dictionary is an arbitrary mapping of keys to values. Arbitrary. That means any legal key should be ok.

```
var dict: [String: String] = [:]
dict["name"] = "Me"
dict["title"] = "Champion of Types"
```

```
struct Person {
    var name: String
    var title: String
}
```

Do these look like “arbitrary keys” to you, like I could subscript by any string at all and it would make sense in this program?
Probably not. Here’s what I bet you meant. <build>
You meant a struct. Stronger type.

Or String. String is so commonly abused that it has a name: Stringly Typed. Do you have code like this?

```
switch kind {
case "start": print("Start")
case "end": print("Done")
default: fatalError("This shouldn't happen")
}
```

```
enum Kind {
    case start
    case end
}
```

where you have to compare your variable to a bunch of hard-coded strings?

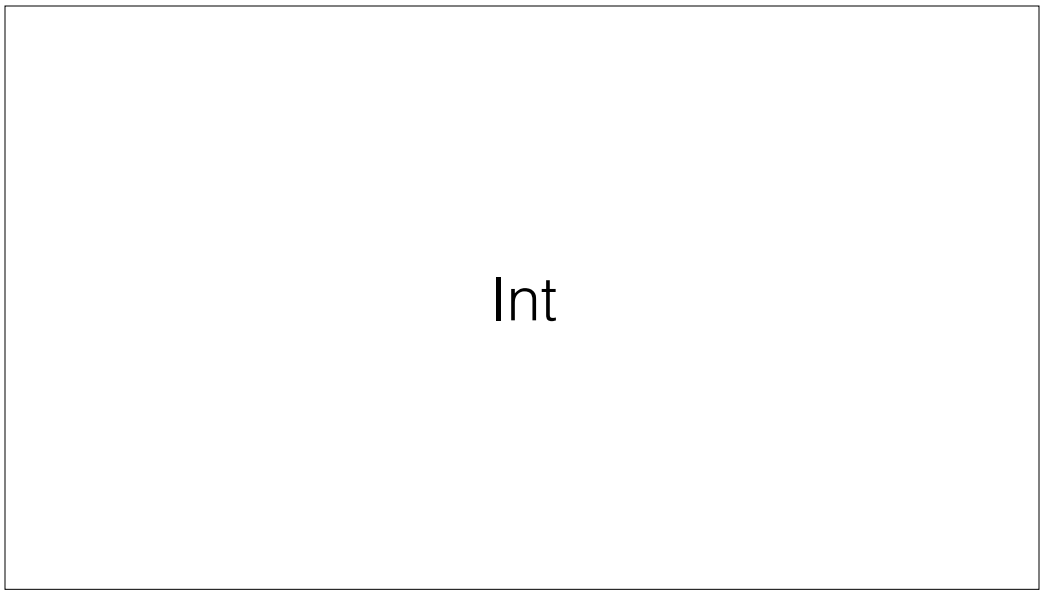
<build>

This is usually an enum. Then you don’t have to worry about forgetting a case or mistyping a string.

```
switch kind {
case "start": print("Start")
case "end": print("Done")
default: fatalError("This shouldn't happen")
}
```

```
enum Kind: String {
    case start
    case end
}
```

In Swift you can even get the best of both worlds and make it a String convertible enum. Now you can treat it as a string when you need to, while still getting the benefits of an enum.



Or how about the lowly Int. I mean of course it should often be an enum. But what about when it’s not one of some short list of values. What about when it’s an identifier, for instance?

```
struct Person {
    let id: Int
    let name: String
}
```

How about this? Is this ok? Well name makes sense. I mean, a name in principle could be any possible string. We don't want to artificially limit that. But is id really an integer? Does it make sense to add two IDs to each other? I can't imagine that making sense. Is there anything really "number-ish" about IDs. If we decided they should strings instead, would that be ok? Probably, but we'd have to change all the APIs. Should we have to do that, just to change the storage implementation? I don't think so.

```
struct Person {
    struct ID {
        let value: Int
    }
    let id: ID
    let name: String
}
```

What if we did this? Lift the integer into a struct. Now it has context. Now it’s more than an integer. It’s its own type. And if we change Int to Int64 or to String, we only have to fix it in a few places. That’s a strong type.

```
func login(username: String, password: String,
           completion: (String?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

This is a pretty common API that I've seen in lots of similar forms. It takes a username and password and a completion block. At some point it passes a token or an error to that block. This is a really common pattern, but we can do better.

```
func login(username: String, password: String,
           completion: (String?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

The first problem is this string in the completion block. What is that?

```
func login(username: String, password: String,
           completion: (_ token: String?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

I mean, I just told you it's a token, so we could stick a label on it. But that doesn't really help much. Labels aren't part of the type in Swift, at least not yet. And it's still a string. It's still easy to accidentally get non-tokens in there. Have you ever seen the string "Optional paren something or other" show up where you didn't expect it? That's because all strings are created equal. But we've already learned the answer to this. We can lift our String into a stronger type.

```
struct Token {
    let string: String
}
```

So we can make a Token that wraps a string.

```
func login(username: String, password: String,
           completion: (Token?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

And when we swap in our Token type and we don’t need labels anymore. The types tell us everything we need to know. Self-documenting code, right? But we still have problems here.

```
func login(username: String, password: String,
           completion: (Token?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

We’re passing a username and a password, and we really always pass them together and use them together. The password by itself isn’t really useful for anything. We want to keep it together with the username. A username AND a password. What we want is a way to combine types with AND.

```
struct Credential {
    var username: String
    var password: String
}
```

We have one of those. A struct. A credential is a username and a password. If you have a good struct, you should be able to say it’s the combination of all of its properties with their context.

```
func login(credential: Credential,
           completion: (Token?, Error?) -> Void)
```

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

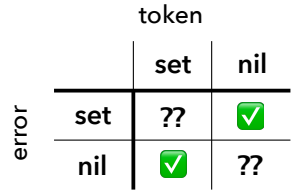
So creating a Credential struct type collapses the username and password parameters into one parameter. But it also opens up a lot of nice possibilities. We could pass other kinds of credentials. Access tokens or one-time-passwords, Facebook, Google. We could use computed getters to interface with Keychain. We get a lot more flexibility when we stop describing all the bits and pieces, and instead describe the composition.

```
func login(credential: Credential,
           completion: (Token?, Error?) -> Void)
```

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

But this still has problems. We pass this tuple of maybe a token and maybe an error. You hear the “and” in there? Maybe a token and maybe an error? Tuples are AND types. They’re just anonymous structs. But do we really mean “maybe a token and maybe an error?”

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```



Only two of the four cases are actually legal. Setting both or setting neither isn't defined. So now we have to think about what to do in those cases. Should we ignore it, or fatalError, or what?
The problem is we don't mean "maybe" anything. We mean a token OR an error. This shouldn't be an AND type, and it shouldn't have maybe in it. This should be an OR type. Can we compose types with OR?

```
enum Result<Value> {
    case success(Value)
    case failure(Error)
}
```

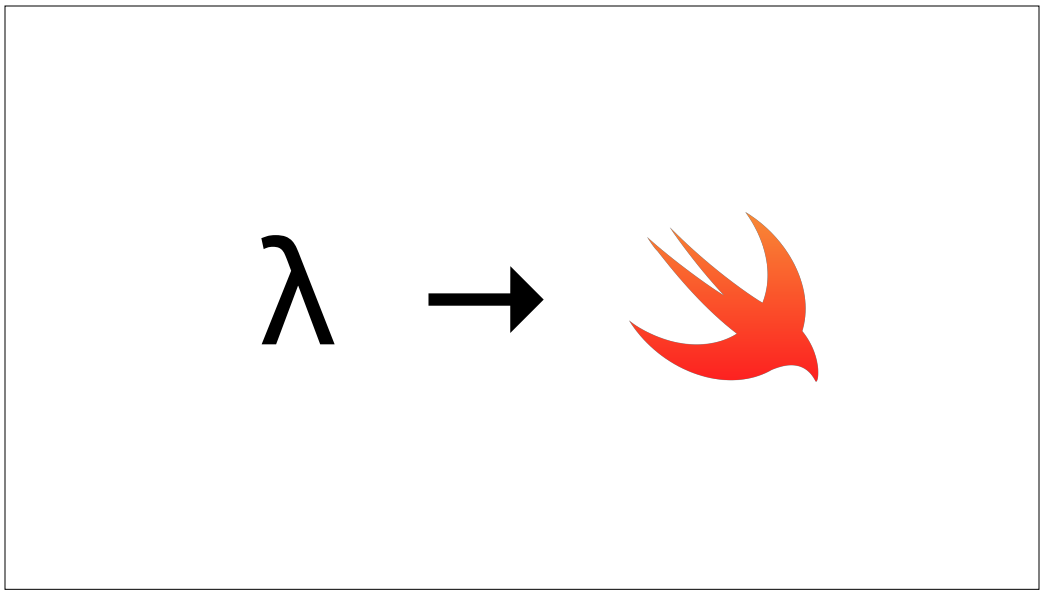
Of course we can. I wouldn't have asked otherwise. We can use an enum. An enum is an OR type. So, we're going to build this Result enum for our purposes. It bothers me that Swift doesn't have this built it, and I'm hoping it will some day. But for now, we can build it ourselves. It's either a successful value, or a failing error. See how we lift our Value type and our Error type into a context here? Our Value becomes more than it was. Now it's a successful value. That's context.

```
func login(credential: Credential,
           completion: (Result<Token>) -> Void)
```

```
login(credential: credential) { result in
    switch result {
    case .success(let token): // success
    case .failure(let error): // failure
    }
}
```

So, now that we've created Result, we can put our resulting Token in our API, we don't have impossible combinations anymore. We don't need assertions or tests. It's impossible to get it wrong. That's what we like. I'd rather make bugs impossible than write test cases. And now I think we have a nice API that says exactly what it means, what it is, rather than all its implementation details. You login with a credential, and it will complete with a Token if successful. That's good.

And we could do all this because we know how to read the types now. We know we we say "maybe a token and maybe an error" isn't right, it isn't what me mean. And we know how to say what do mean. And that let's us build something that isn't just more beautiful, but something that is easier to maintain, is more reusable, has fewer corner cases and fewer bugs. And maybe that'll let you get rich if that's your goal.

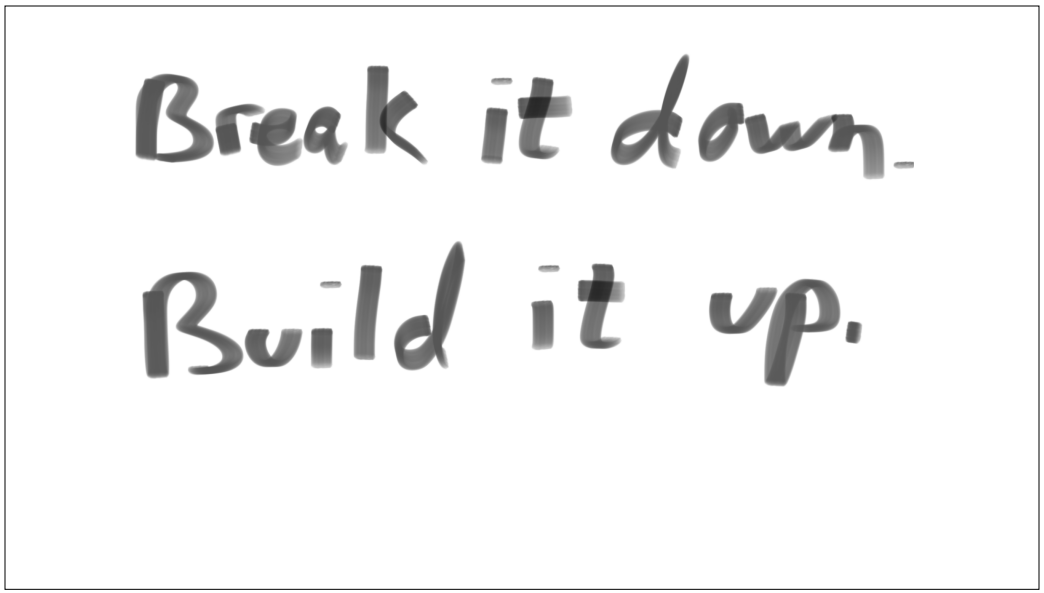


That’s the lesson of functional programming that we should bring to Swift.

The Lessons

- Break apart complicated things into simpler things
- Look for generic patterns in the simple things
- Lift and compose simple things to make complex ones

Complicated things can be broken down into smaller, simpler things. We can find generic solutions to those simple things. And we can put those simple solutions back together using consistent rules that let us reason about our programs and lets the compiler make a lot of bugs simply impossible.



And that’s something I think John Backus in the 70s and Crusty today, would completely agree on. Break down your overcomplicated types, and then build them back up with simple rules. Learn from our elders.