

Swift Generics

It isn't supposed to hurt

Rob Napier – rob@neverwood.org

github.com/rnapier/generics

@cocoaphony

robnapier.net/start-with-a-protocol

Developer Tools

#WWDC15

Protocol-Oriented Programming in Swift

Session 408

Dave Abrahams Professor of Blowing-Your-Mind

© 2015 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.

In 2015, at WWDC, Dave Abrahams gave what I believe is still the greatest Swift talk ever given, "Protocol-Oriented Programming in Swift," or as it is more affectionately known,


Meet Crusty

Don't call him "Jerome"



“The Crusty Talk.”

This is the talk that introduces the phrase “protocol oriented programming.” This was a great talk. I've watched it a few times. But in the beginning, I really only took away one line.

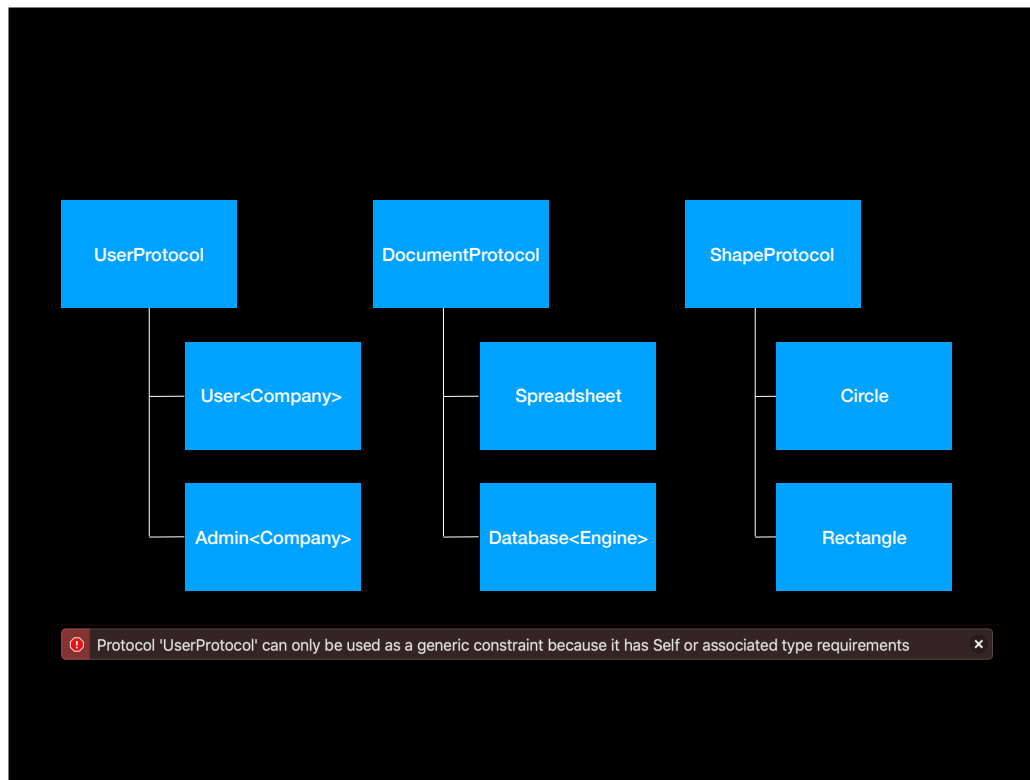


“Start with a protocol.”

–Dave Abrahams, WWDC 2015

“Start with a protocol.”

And so, dutifully, I started with a protocol.



And I made a UserProtocol and a DocumentProtocol and a ShapeProtocol and on and on,
<build>and then I started implementing all those protocols with generic subclasses and eventually I found myself in a corner.
<build>And a lot of other developers have found themselves in the same corner. I know, because I wind up talking to a lot of them on Stack Overflow.

“For example, if you want to write a generalized sort or binary search...Don't start with a class. Start with a protocol.”

–Dave Abrahams, WWDC 2015

This is what Dave actually said in the Crusty talk. “If you want to write a generalized [algorithm], don’t start with a class, start with a protocol.” The point was, if you are reaching for class inheritance, try a value type and a protocol first. It wasn’t always start with a protocol for everything. It was to use protocols to make things more general when that’s useful.

So today I’m going to walk through various examples, to show you how to design generic code a better way, with protocols, generics, functions, and most of all, composition. After all, a lot of the Crusty talk is really about inheritance, and the problems with inheritance. If you’re just trying to recreate class hierarchies in protocols, and still thinking in terms like a Cat is a kind of Animal, you’re not getting the real benefits that Swift promises.

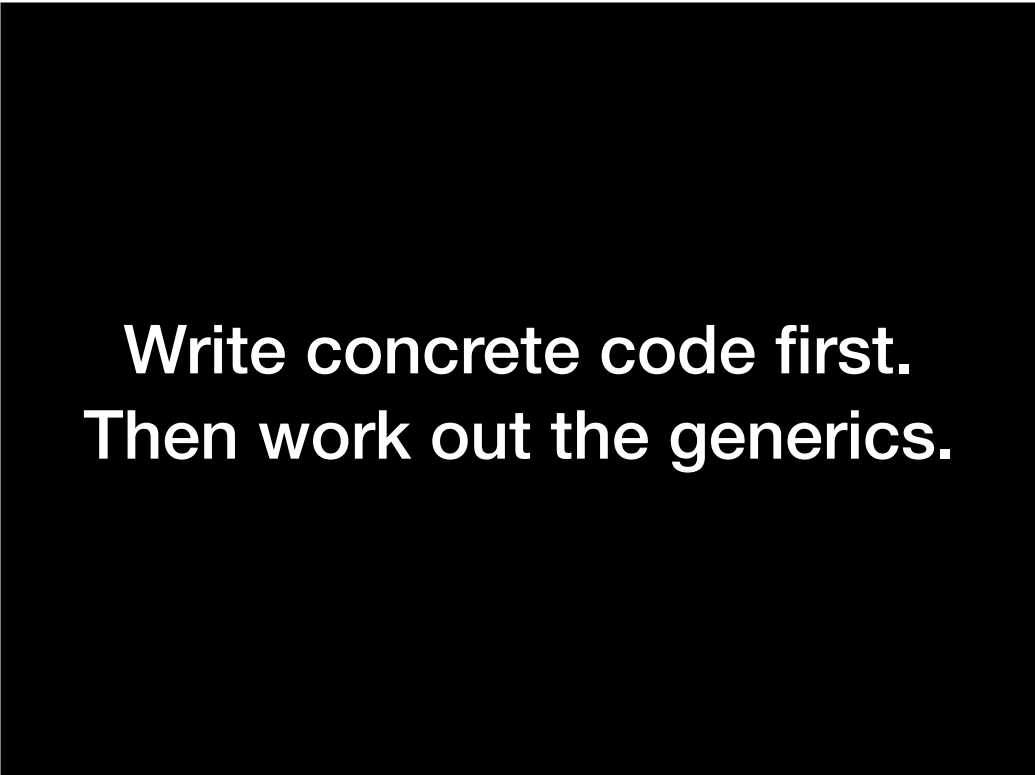
It's all about extensions

Instead of thinking first about inheritance, we think first about extensions and functions. What generic, reusable algorithms does this protocol allow me to write. And when you think in terms of the algorithms that you want to write, the ways you want to use this protocol, it unlocks a lot of power. We'll see some of that today. How protocols can help your use cases.

**“As generic as
possible”**

That doesn't even mean anything

I know I'm probably headed down a rat hole the moment I start trying to making my code “as generic as possible,” as if that really means anything. When you choose to make a system flexible along one axis, you almost always are going to make it harder to adapt along some other axis.



**Write concrete code first.
Then work out the generics.**

Maybe my whole generics talk should just be this one slide.

"Write concrete code first. Then work out the generics."

Start with concrete types, and clear use cases, and find the places that duplication really happens. Then find abstractions to fix those problems.

The power of Protocol Oriented Programming is that you don't have to decide when you make a type exactly how that type will be used. When you work with inheritance, you have to design your class hierarchy from the start. But with protocols, you can wait until later.



```
protocol Shape {  
    var center: CGPoint { get }  
}  
  
protocol Circle: Shape {  
    var radius: CGFloat { get }  
}  
  
protocol Rectangle: Shape {  
    var size: CGSize { get }  
}  
  
struct CircleImpl: Circle {  
    var center: CGPoint  
    var radius: CGFloat  
}  
  
struct RectangleImpl: Rectangle {  
    var center: CGPoint  
    var size: CGSize  
}
```

I see this kind of mistake a lot. You want a program that can draw shapes, so you make this shape protocol, that has a center. And then you inherit a Circle protocol off of that and a Rectangle protocol. And then you make circle and rectangle implementations.<build>

Seem familiar? This isn't actually giving us much flexibility. The protocols and their implementations are almost certainly going to evolve in lock-step.

This is really just reinventing class inheritance, which is the thing Crusty was warning against.

And protocols are going to fight you because they're not designed to support class-like inheritance. If you find you only have one implementation of a protocol in your shipping product, and the natural thing to name it is something something Impl, you're probably on the wrong road.



```
struct Circle {  
    var center: CGPoint  
    var radius: CGFloat  
}  
  
struct Rectangle {  
    var origin: CGPoint  
    var size: CGSize  
}
```

Instead, just let the types be the types, and work out the protocols when you need them.

As you get more experienced, you'll get better at guessing when you'll need a protocol earlier in the process. But if you have any doubt, add it later.

Real-life protocol design

OK, so let's try some real-world protocol oriented programming, taking it one step at a time and see how you do this in practice. I want to build a general purpose networking stack. Something that can connect to a server and parse the responses. But I also want it to be testable.

```
struct User: Codable, Hashable {  
    let id: Int  
    let name: String  
}  
  
struct Document: Codable, Hashable {  
    let id: Int  
    let title: String  
}  
  
final class Client {  
    ...  
}
```

So assume I have a bunch of types that I want to fetch from some API. I like to have at least a couple of concrete examples, so we're going to think about User and Document. They're just simple data. And that means we get Codable and Hashable for free. And we have this Client with nothing in it yet. Right now it's just going to be a namespace for our various fetching methods.

```

func fetchUser(id: Int,
               completion:
               @escaping (Result<User, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("user")
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(User.self,
                                   from: data)
            })
        }
    }.resume()
}

```

Here's the first Client method, that fetches a User. I'm sure you've all written code kind of like this a hundred times. Construct a URLRequest. Fetch it. Parse it. Pass it to the completion handler. Now, what does the code for fetchDocument look like?

```
func fetchDocument(id: Int,
                  completion:
    @escaping (Result<Document, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("document")
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Document.self,
                                   from: data)
            })
        }
    }.resume()
}
```

Wow, that's pretty similar. <go-back>

```

func fetchDocument(id: Int,
                  completion:
    @escaping (Result<Document, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("document")
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Document.self,
                                   from: data)
            })
        }
    }.resume()
}

```

What changes? Well, just these handful of pieces. So clearly we can extract something here.


```

func fetch<Model: Decodable>(_ model: Model.Type,
                             id: Int,
                             completion:
@escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("???user | document???")
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Model.self,
                                   from: data)
            })
        }
    }.resume()
}

```

I create a Model type, and use that everywhere that User or Document used to be. But what about this string that's either user or document? That's something that changes that isn't part of Decodable. So Decodable isn't powerful enough. We need a new protocol.

```
protocol Fetchable: Decodable {  
    static var apiBase: String { get }  
}
```

We need a protocol that requires first, that the type be Decodable, and also requires that it provide this extra string.

```

func fetch<Model: Fetchable>(_ model: Model.Type,
                             id: Int,
                             completion:
@escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Model.self,
                                   from: data)
            })
        }
    }.resume()
}

```

So with this Fetchable protocol, we can finish writing fetch. Now to use it, we need to make User and Document conform to Fetchable.

```
struct User: Codable, Hashable {
    let id: Int
    let name: String
}

struct Document: Codable, Hashable {
    let id: Int
    let name: String
}

extension User: Fetchable {
    static var apiBase: String { return "user" }
}

extension Document: Fetchable {
    static var apiBase: String { return "document" }
}
```

You might be tempted to add Fetchable here on the main definition. I generally wouldn't. I'd add them as an extension.<build> That's mostly style, but it raises an important way of thinking. One the most important aspects of protocol oriented programming is retroactive modeling. The fact that you can take a type like User, that wasn't designed to be Fetchable, and make it Fetchable in an extension. And that extension doesn't even have to be in the same module. You can take any type you want and conform it to your own protocols to let it be used in new and more powerful ways. There's no need to tie User to this one use case and this one API.

Of course you have to be a little careful about how you name your protocol properties, or you could have collisions, but this is a great example of how Swift lets you adapt types to use cases as you need them, rather than having to decide everything in the definition.

```

func fetch<Model: Fetchable>(_ model: Model.Type,
                             id: Int,
                             completion:
@escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Model.self,
                                   from: data)
            })
        }
    }.resume()
}

```

OK, coming back to fetch, now it's generic over the type of the model. But it's tied very tightly to URLSession. Now yes, that makes it hard to unit test. But I want you all to stop thinking about protocols as mocks.

Mocks are a terrible way to design protocols

Mocks are a terrible way to design protocols. The basic premise of a mock is to build a test object that mimics some other object you want to replace. That makes your protocols highly tied to that specific implementation, and means your protocols aren't giving you any power in your shipping code. They exist just so you can unit test. But protocols can give us so much more.

```

func fetch<Model: Fetchable>(_ model: Model.Type,
                             id: Int,
                             completion:
@escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Model.self,
                                   from: data)
            })
        }
    }.resume()
}

```

Our goal here isn't to mock URLSession. It's to abstract how we transform URLRequest into data. We don't care that dataTask returns a task that has to be resumed. We also don't care that dataTask passes an URLResponse. We don't use it, and including it ties us to HTTP in ways that may not be convenient.

```
protocol Transport {
    func fetch(request: URLRequest,
               completion: @escaping (Result<Data, Error>) -> Void)
}

extension URLSession: Transport {
    func fetch(request: URLRequest,
               completion: @escaping (Result<Data, Error>) -> Void)
    {
        self.dataTask(with: request) { (data, _, error) in
            if let error = error { completion(.failure(error)) }
            else if let data = data { completion(.success(data)) }
        }.resume()
    }
}
```

We just want to transform `URLRequests` into data asynchronously. We don't care if that connects to a network, or a database, or flat files, or an internal cache, or even, yes, a unit test mock that returns canned data. All fine. Don't care. Because we pass a full `URLRequest`, the transport could decide what to do based on the URL scheme, or it could ignore the URL scheme. I'll show you later just how powerful this is. But the point is, this is not a mock of `URLSession`. This is a protocol for converting `URLRequests` into data.

<build> And I can conform `URLSession` to `Transport`. Notice I don't need any wrappers around `URLSession`. This is retroactive modeling again. `URLSession` is now a `Transport`, even though it's an Apple class that I don't have any access to.


```

func fetch<Model: Fetchable>(
    _ model: Model.Type,
    id: Int,
    completion: @escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    let session = URLSession.shared

    session.dataTask(with: urlRequest) {
        (data, _, error) in
        if let error = error {
            completion(.failure(error))
        }
        else if let data = data {
            let decoder = JSONDecoder()
            completion(Result {
                try decoder.decode(Model.self,
                                   from: data)
            })
        }
    }.resume()
}

```

So coming back to our method.

```

func fetch<Model: Fetchable>(
    _ model: Model.Type,
    id: Int,
    completion: @escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    transport.fetch(request: urlRequest) { data in
        completion(Result {
            let decoder = JSONDecoder()
            return try decoder.decode(Model.self,
                                     from: data.get())
        })
    }
}

```

I can extract a transport. But now you have to know about transports when you create a client, and that's a little annoying. It's almost always going to be the network transport. So we can clean that up with a default.

```

struct Client {
    let transport: Transport

    init(transport: Transport) {
        self.transport = transport
    }

    func fetch<Model: Fetchable>(
        model: Model.Type,
        id: Int,
        completion: @escaping (Result<Model, Error>) -> Void)
    {
        let urlRequest = URLRequest(url: baseUrl
            .appendingPathComponent(Model.apiBase)
            .appendingPathComponent("\(id)")
        )

        transport.fetch(request: urlRequest) { data in
            completion(Result {
                let decoder = JSONDecoder()
                return try decoder.decode(Model.self,
                    from: data.get())
            })
        }
    }
}

```

I can extract a transport. But now you have to know about transports when you create a client, and that's a little annoying. It's almost always going to be the network transport. So we can clean that up with a default.

```

struct Client {
  let transport: Transport

  init(transport: Transport = URLSession.shared) {
    self.transport = transport
  }

  func fetch<Model: Fetchable>(
    _ model: Model.Type,
    _ id: Int,
    completion: @escaping (Result<Model, Error>) -> Void)
  {
    let urlRequest = URLRequest(url: baseURL
      .appendingPathComponent(Model.apiBase)
      .appendingPathComponent("\(id)")
    )

    transport.fetch(request: urlRequest) { data in
      completion(Result {
        let decoder = JSONDecoder()
        return try decoder.decode(Model.self,
          from: data.get())
      })
    }
  }
}

```

Now the transport is abstract. All it has to do is convert an URLRequest into data. I'm hoping you can immediately imagine how to make a mock transport for unit testing. I want to focus on a more interesting transport. I want a transport that adds extra headers to any other transport.

```
class AddHeaders: Transport
{
    func fetch(request: URLRequest,
               completion: @escaping (Result<Data, Error>) -> Void)
    {
        var newRequest = request
        for (key, value) in headers {
            newRequest.addValue(value, forHTTPHeaderField: key)
        }
        base.fetch(request: newRequest, completion: completion)
    }

    let base: Transport
    var headers: [String: String]
}

let transport = AddHeaders(base: URLSession.shared,
                           headers: ["Authorization": "..."])
```

And this is all it takes. Now I can make a transport that extends other transports, without having to know anything about those other transports. That means when I'm doing unit testing, I just have to swap in the lowest level piece, which is tiny, and everything else remains fully testable. But it's more than unit tests. It means I can extend existing systems in a really flexible way. I can add encryption or logging or caching or priority queues or automatic retries or whatever without intermingling that with the network layer. So yes, I get mocks, but I get so much more.

And I still haven't needed an associated type or a Self requirement. Transport works fine without that.

But let's go deeper.

```
let base: Transport
```

Side-bar: Existentials

Little side-bar. The transport variable here is not “some type conforming to Transport.” It’s type is really the Transport existential. An existential is a little box that’s put around another type by the compiler. It has a different memory layout than the underlying type, and introduces things like dynamic dispatch. So, for instance, that may mean copying data when you make this assignment, or allocating heap memory. My point isn’t about performance, it’s just understanding what’s going on.

```
func process<T: Transport>(transport: T) {}  
func process(transport: Transport) {}
```

These two functions are different. The first one requires some concrete type that conforms to `Transport`, and the entire function is potentially rewritten to handle that specific type at compile-time. That's generic specialization.

The second requires a `Transport` existential. Which is a different thing. Remembering back to when I said that protocols do not conform to themselves, that means you can't pass an existential, a variable of type `Transport`, when a concrete type is required like in the first case, but you can pass it in the second case.

Generally speaking, I recommend the second form when you can get away with it. There are performance arguments for the first form, but it's not certain to be faster, and it's more restrictive in what it can accept, so it can bite you. But the real reason that I recommend the second form is because it's usually what you mean, and you should say what you mean. You usually mean "this function accepts a `Transport`," not "this function accepts a concrete type that conforms to `Transport`." If you need the later, then use that.

Generalized Existentials

I say most of this to explain this term you may see thrown around, called Generalized Existentials. It means an existential that can handle a PAT. This would be a big and important feature for Swift, but a lot of people treat it like the magical solution to all their protocol problems, and it isn't. In fact, I suspect whenever we get generalized existentials, developers will get themselves even more twisted up because the compiler will let them go a lot further down the wrong road. Most protocol problems I see people run into aren't language problems, they're design problems, and improving your design makes the need for generalized existentials go away.

Even so there are some really good use cases for them, and I do hope Swift gets them. I just think that a lot of comments that start, "once Swift has Generalized Existentials, then..." are wrong.


```
struct User: Codable, Equatable {  
    let id: Int  
    let name: String  
}  
  
struct Document: Codable, Equatable {  
    let id: Int  
    let title: String  
}
```

Let's go back to the model. These IDs are Ints. I don't like that. Identifiers are not Ints. You can't add or subtract them. You can't divide identifiers. What would that even mean? You don't want to mix up user ids and document ids, either. And in any case, what happens if one of these types change. Say documents switch to using strings instead of Ints. I've had that happen twice to me in the last year. So how can we improve this?

```
struct User: Codable, Equatable {  
    struct ID: Codable, Hashable {  
        let value: Int  
    }  
    let id: ID  
    let name: String  
}  
  
let user = User(id: User.ID(value: 1), name: "alice")
```

Just small struct. But I don't love that value tag in the initializer. Let's fix it.

```
struct User: Codable, Equatable {  
    struct ID: Codable, Hashable {  
        let value: Int  
        init(_ value: Int) { self.value = value }  
    }  
    let id: ID  
    let name: String  
}
```

```
let user = User(id: User.ID(1), name: "alice")
```

OK, that's better.

```
struct User: Codable, Equatable {
    struct ID: Codable, Hashable {
        let value: Int
        init(_ value: Int) { self.value = value }
    }
    let id: ID
    let name: String
}

struct Document: Codable, Equatable {
    struct ID: Codable, Hashable {
        let value: String
        init(_ value: String) { self.value = value }
    }
    let id: ID
    let title: String
}
```

And the same for Document. OK, clearly there's some opportunity for code sharing here. I know it's only 4 lines of code, but it's going to be repeated for every type. What can we do about that?

```
protocol IDType: Codable, Hashable {
    associatedtype Value
    var value: Value { get }
    init(value: Value)
}

extension IDType {
    init(_ value: Value) { self.init(value: value) }
}

struct User: Codable, Equatable {
    struct ID: IDType { let value: Int }
    let id: ID
    let name: String
}

struct Document: Codable, Equatable {
    struct ID: IDType { let value: String }
    let id: ID
    let title: String
}
```

We could make a protocol. IDType. In order to conform to IDType, a type needs to be Codable and Hashable, and it needs an init that takes a value. Those are all things that Swift will automatically synthesize for us, so they're free.

```
protocol IDType: Codable, Hashable {
    associatedtype Value
    var value: Value { get }
    init(value: Value)
}

extension IDType {
    init(_ value: Value) { self.init(value: value) }
}

struct User: Codable, Equatable {
    struct ID: IDType { let value: Int }
    let id: ID
    let name: String
}

struct Document: Codable, Equatable {
    struct ID: IDType { let value: String }
    let id: ID
    let title: String
}
```

And then, types that conform to IDType get this no-label initializer.

```
func fetch<Model>(_ model: Model.Type, id: Int,  
                 completion: @escaping (Result<Model, Error>) -> Void)  
where Model: Fetchable
```

So then I'd update fetch, to accept an ID rather than an Int.

```
func fetch<Model>(_ model: Model.Type, id: Model.ID,
                 completion: @escaping (Result<Model, Error>) -> Void)
where Model: Fetchable

client.fetch(User.self, id: User.ID(1), completion: { print($0)} )
```

This creates awkward repetition in the API.<build>

You can't just pass User.ID, because IDs don't know what model they apply to.

I could add an extra “Model” associated type to the Identifier protocol, but it gets a bit messy. Some of it is just Swift syntax (where clauses and the like), but it really comes down to Identifier not being a very good protocol. Look at the implementations:


```
struct User.ID : Identifier { let value: Int }  
struct Document.ID: Identifier { let value: String }
```

If you think about any other implementations, they're going to be almost identical: a struct with a single property called value. It's hard to imagine any other way you'd want to implement this protocol. If every instance of a protocol conforms in exactly the same way, it should probably be a generic struct.

```

struct Identifier<Model, Value>
  where Value: Codable & Hashable
{
  let value: Value
  init(_ value: Value) { self.value = value }
}

extension Identifier: Codable, Hashable {
  init(from decoder: Decoder) throws {
    self.init(try decoder.singleValueContainer()
              .decode(Value.self))
  }

  func encode(to encoder: Encoder) throws {
    var container = encoder.singleValueContainer()
    try container.encode(value)
  }
}

```

Identifier has two type parameters. The Model is the type this identifier applies to. The Value is the kind of identifier it requires (Int, UInt64, String, etc). The Model isn't actually used anywhere, but it means that Identifier<User, Int> and Identifier<Document, Int> are completely different types and can't be mixed up. **<build>** And it needs to be Codable, which just encodes it as its raw value.

```
struct User: Codable, Hashable {  
    let id: Identifier<User, Int>  
    let name: String  
}
```

So User becomes this.

That's ok, but it'd be nicer to typealias it so I can refer to User.ID as a type:

```
struct User: Codable, Hashable {  
    typealias ID = Identifier<User, Int>  
    let id: ID  
    let name: String  
}
```

And now is the time to extract a protocol, because this is going to happen for every one of these model types.

```
protocol Identified: Codable {
    associatedtype IDType: Codable & Hashable
    typealias ID = Identifier<Self, IDType>
    var id: ID { get }
}

// User model object
struct User: Identified {
    typealias IDType = Int
    let id: ID
    let name: String
}

protocol Fetchable: Identified {
    static var apiBase: String { get }
}
```

And Identified is a PAT. So I need to ask the question again, should this ever be in an array? No. If you think they should, you're probably thinking of this protocol as "Model." Wouldn't you want model objects in an array? Sure. But that's not what this protocol means. It means Identified. Something with an identifier. What would an array of "things with identifiers" be good for? What interesting algorithms could you write based just on "it has an identifier." I'll talk about that more later.<build>

But for now, User can be Identified, and each line provides unique information.

<build>

And Fetchable just needs to require Identified

```
func fetch<Model>(_ model: Model.Type, id: Model.ID,  
                 completion: @escaping (Result<Model, Error>) -> Void)  
where Model: Fetchable  
  
client.fetch(User.self, id: User.ID(1), completion: { print($0)} )
```

And finally, fetch doesn't need any type parameters.

```
func fetch<Model>(id: Model.ID,  
                 completion: @escaping (Result<Model, Error>) -> Void)  
where Model: Fetchable  
  
client.fetch(id: User.ID(1), completion: { print($0)} )
```

The only thing that could be fetched with a User.ID is a User.

```
// GET /<model>/<id> -> Model

func fetch<Model: Fetchable>(
    id: Model.ID,
    completion:
        @escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    transport.fetch(request: urlRequest) {
        data in
        completion(Result {
            let decoder = JSONDecoder()
            return try decoder.decode(
                Model.self,
                from: data.get())
        })
    }
}
```

This is great for getting a model by ID, but I have other things I want to do, like POST to /keepalive and return if there was an error. And they're really similar, but kind of different.


```
// POST /keepalive -> Error?

func keepAlive(
    completion: @escaping (Error?) -> Void)
{
    var urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("keepalive")
    )
    urlRequest.httpMethod = "POST"

    transport.fetch(request: urlRequest) {
        switch $0 {
        case .success: completion(nil)
        case .failure(let error):
            completion(error)
        }
    }
}
```

```

// GET /<model>/<id> -> Model

func fetch<Model: Fetchable>(
    id: Model.ID,
    completion:
    @escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id)")
    )

    transport.fetch(request: urlRequest) {
        data in
        completion(Result {
            let decoder = JSONDecoder()
            return try decoder.decode(
                Model.self,
                from: data.get())
        })
    }
}

```

Both basically follow this pattern of build an URL request, pass it to transport, and then deal with the result. I know it's just one line that exactly duplicates, but the structure is still really similar, and it feels we could pull this apart. That fetch is doing too much.

```
// POST /keepalive -> Error?

func keepAlive(
    completion: @escaping (Error?) -> Void)
{
    var urlRequest = URLRequest(url: baseUrl
        .appendingPathComponent("keepalive")
    )
    urlRequest.httpMethod = "POST"

    transport.fetch(request: urlRequest) {
        switch $0 {
        case .success: completion(nil)
        case .failure(let error):
            completion(error)
        }
    }
}
```



```
func fetch(_ request: Request) {  
    transport.fetch(request: request.urlRequest,  
                    completion: request.completion)  
}  
  
protocol Request {  
    var urlRequest: URLRequest { get }  
    associatedtype Response  
    var completion: (Result<Response, Error>) -> Void { get }  
}
```

So maybe we pull out the part that changes and call it Request. But what should Request be? So often, I see people jump to this. **<build>** A PAT. So what's the question we ask whenever we make a PAT? Would I ever want an array of these? I think we would definitely want an array of requests. A list of pending requests. Chaining requests together. Requests that should be retried. We definitely want an array of requests. This is a great example where someone might come along as say, if only we had generalized existentials, then we could do this. No. That wouldn't fix anything. Transport.fetch create data, not Response. There's no language feature that would make this work. It's a design problem. So what do we do?

**Write concrete code first.
Then work out the generics.**

<point>

So what's a concrete Request?

```
func fetch(_ request: Request) {  
    transport.fetch(request: request.urlRequest,  
                    completion: request.completion)  
}  
  
struct Request {  
    let urlRequest: URLRequest  
    let completion: (Result<Data, Error>) -> Void  
}
```

A struct. Just a struct.

But, um... where's the JSON parser? Yeah, we're going to have to do something about that. First, start with the calling code. What do I wish would work?

```

struct Request {
    let urlRequest: URLRequest
    let completion: (Result<Data, Error>) -> Void
}

let request = Request.fetching(User.self,
                                id: User.ID(0),
                                completion: { user in ... })

static func fetching<Model: Fetchable>(
    _: Model.Type,
    id: Model.ID,
    completion: @escaping (Result<Model, Error>) -> Void)
    -> Request
{
    ???
}

```

I want to create a request that will fetch a model. Which means a function signature that looks like this. Right? **<build>** Now you might ask, why a static method rather than an initializer? Because it scales better for this kind of problem. Some kinds of requests might be different, but take the same parameters, and that gets awkward. Static methods fix that. But it's just a style detail.

```

struct Request {
    let urlRequest: URLRequest
    let completion: (Result<Data, Error>) -> Void
}

let request = Request.fetching(User.self,
                                id: User.ID(0),
                                completion: { user in ... })

static func fetching<Model: Fetchable>(
    _: Model.Type,
    id: Model.ID,
    completion: @escaping (Result<Model, Error>) -> Void)
    -> Request
{
    ???
}

```

So we've got these two closures with different types. We have one that takes a Model and return Void. And we need one that takes Data and returns Void.

`Data -> Void`



`Data -> Model`

`Model -> Void`

So to simplify a little. What do we need? **<build>** And the answer is we need a method that will convert Data to Model. How does that work?

`Data -> Void`

`Data -> Model -> Void`

When we combine them, we get Data to Model to Void. And that's the same thing as

```
Data -> Void
```

```
Data -> Void
```

When we combine them, we get Data to Model to Void. And that's the same thing as Data to Void. What did we just do? Type erasure. We combined functions and erased an intermediate type. When you think about getting rid of an extra type, I want you to think about functions, not "type erasers." They're usually the thing you actually want.

```

extension Client {

    // GET /<model>/<id> -> Model
    func fetch<Model: Fetchable>(
        _ model: Model.Type,
        id: Int,
        completion:
            @escaping (Result<Model, Error>) -> Void)
    {
        let urlRequest = URLRequest(url: baseURL
            .appendingPathComponent(Model.apiBase)
            .appendingPathComponent("\(id)")
        )

        transport.fetch(request: urlRequest) {
            data in
            completion(Result {
                let decoder = JSONDecoder()
                return try decoder.decode(
                    Model.self,
                    from: data.get())
            })
        }
    }
}

```

Fine, **algebra**. **Functional composition.** Yes, yes, how do we build it? We take our fetch method.

```

extension Request {
    // GET /<model>/<id> -> Model
    static func fetching<Model: Fetchable>(
        _: Model.Type,
        id: Model.ID,
        completion: @escaping (Result<Model, Error>) -> Void)
        -> Request
    {
        let urlRequest = URLRequest(url: baseUrl
            .appendingPathComponent(Model.apiBase)
            .appendingPathComponent("\(id)")
        )

        return self.init(urlRequest: urlRequest) {
            data in
            completion(Result {
                let decoder = JSONDecoder()
                return try decoder.decode(
                    Model.self,
                    from: data.get())
            })
        }
    }
}

```

And slide it into a Request method. We have all the flexibility of a protocol, without any of the associated type headaches, just by using structs.

```

extension Client {
    // POST /keepalive -> Error?
    func keepAlive(
        completion: @escaping (Error?) -> Void)
    {
        var urlRequest = URLRequest(url: baseURL
            .appendingPathComponent("keepalive")
        )
        urlRequest.httpMethod = "POST"

        transport.fetch(request: urlRequest) {
            switch $0 {
            case .success: completion(nil)
            case .failure(let error):
                completion(error)
            }
        }
    }
}

```

And exactly the same pattern applies to keep alive requests that take an Error to Void completion handler.

```

extension Request {
    // POST /keepalive -> Error?
    static func keepAlive(
        completion: @escaping (Error?) -> Void) -> Request
    {
        var urlRequest = URLRequest(url: baseURL
            .appendingPathComponent("keepalive")
        )
        urlRequest.httpMethod = "POST"

        return self.init(urlRequest: urlRequest) {
            switch $0 {
            case .success: completion(nil)
            case .failure(let error):
                completion(error)
            }
        }
    }
}

```

Generic code does not mean lots of generics, or protocols, or associated types. Generic code comes from taking concrete code, and separating the parts that change from the parts that don't. Sometimes that's a protocol. But sometimes it's just a function that transforms another function.

Type-Erasers

Any thing but that...

If you spend much time working with protocols, you'll eventually bump into type erasers.

04 AUG 2015

A Little Respect for AnySequence

Once upon a time, when Swift was young, there were a couple of types called `SequenceOf` and `GeneratorOf`, and they could type erase stuff. "Type erase?" you may ask. "I thought we *loved* types." We do. Don't worry. Our types aren't going anywhere. But sometimes we want them to be a little less...precise.

In Swift 2, our little type erasers got a rename and some friends. Now they're all named "Any"-something. So `SequenceOf` became `AnySequence` and `GeneratorOf` became `AnyGenerator` and there are a gaggle of indexes and collections from `AnyForwardIndex` to `AnyRandomAccessCollection`.

So what are these type erasers? Let's start with how to use one and we'll work backwards to why.

```
let seq = AnySequence([1,2,3])
```

This creates an `AnySequence<Int>`. It's just a sequence of Ints that we can iterate over. Isn't `[1,2,3]` also a sequence of Ints we can iterate over? Well, yeah. But it's also explicitly an `Array`. And sometimes you don't want to have to deal with that kind of implementation detail.

Who Needs Types Like That?

Let's consider a little more complicated case:

You may even come across one of my own blog posts on the subject. But I'm here to tell you, most of the time I don't recommend them. They add a lot of headaches that you can often avoid by rethinking your design, or just passing functions or concrete structs. Most of this talk so far has been about avoiding type erasers. They're really a last resort and they can add a lot of complexity. If you find yourself using them a lot, you're probably overusing protocols, and again, that's what this talk has been about. But, sometimes you need a type eraser, so let's talk about them.

```
public struct AnySequence<Element> : Sequence { ... }
```

```
let s = AnySequence([1,2,3])
```

First, what is it? It's an "any" type. AnySequence doesn't care what specific kind of sequence you have, as long as it acts like to Sequence. It's literally "any Sequence-conforming type." The "erasure" part is that you make a little box to hide the underlying type. That "little box" should sound familiar.

Type-erasers are just explicit existentials

A type-eraser is just an existential you build by hand. Which is why you usually see them when working with associated types. Swift automatically makes a type-eraser, makes an existential, for simple protocols. Some day it'll make type erasers automatically for PATs, the "generalized existential." But like I said about generalized existentials, they're not usually the solution to your problem, and neither are type erasers.

But, enough warnings. Sometimes they're exactly the right tool, so let's build some.

Functional Type-Erasers

There are two main ways of implementing type erasers in Swift. The first I call the functional type eraser, because it converts a value into a collection of functions. Specifically, we make a struct of closures.

```
protocol Frobulating {  
    associatedtype Input  
    associatedtype Output  
    func frobulate(from input: Input) -> Output  
}
```

```
struct Frobulator<Input, Output> {  
    let frobulate: (Input) -> Output  
}
```

```
(Input) -> Output
```

I'm going to start with this protocol. It's only one function, so you should strongly consider just replacing it with a generic struct, like we did with Request. Or, just use function. You might not need a struct, let alone a protocol. But for this example, I'm going to assume there are many more methods, or there's additional context, or something. But this idea, that you could just replace the protocol with a generic struct or even just a function is the whole point of a functional type eraser.

```

protocol Frobulating {
    associatedtype Input
    associatedtype Output
    func frobulate(from input: Input) -> Output
}

struct AnyFrobulator<Input, Output>: Frobulating {
    private let _frobulate: (Input) -> Output

    func frobulate(from input: Input) -> Output {
        _frobulate(input)
    }

    init<F: Frobulating>(_ base: F)
        where F.Input == Input, F.Output == Output {
        self._frobulate = base.frobulate
    }
}

```

And this is how it works. The associated types become generic parameters. And the functions become closures. Unfortunately Swift doesn't let a closure property be used to conform to a protocol, so we use this underscore property, but that's it. The `init` takes anything that conforms to `Frobulating`, and captures each of the methods as closure properties, in this line `self._frobulate = base.frobulate`. That captures `base` inside the closure.

And that's it. That's the whole type-eraser. It's tedious. But it's not hard.

You'll notice, though, that you can't get `base` back out of this type eraser. It's gone. It exists inside the closures, and there's no way to get it back out. And that leads us to the second kind of type-eraser.

Boxing Type Erasers

I call these boxing type erasers because they include a private box. Let's see why. The obvious way to get the original value back is by storing it in a property.

```

struct AnyFrobulator<Input, Output>: Frobulating {
  private let _frobulate: (Input) -> Output

  func frobulate(from input: Input) -> Output {
    _frobulate(input)
  }

  init<F: Frobulating>(_ base: F)
    where F.Input == Input, F.Output == Output {
    self._frobulate = base.frobulate
    self.base = base
  }

  let base: Any
}

let f = AnyFrobulator(SomeFrobulator())
let s = f.base as? SomeFrobulator

```

And that actually works fine for getting the original type back out if you need it. It's a little weird because now base is captured as a property and in all the closures. You might be tempted to forward the calls to base, but you can't do that, because base is Any. Any is the ultimate type eraser. It erases everything. So we'd have to know the underlying type in order to forward the calls. You can't as-cast to a variable. That's kind of annoying, but not really that big a deal usually.

But there's another problem. This whole technique relies on capturing the same value into various closures. But that only works if the protocol methods are non-mutating. If the protocol methods are marked mutating, and you're passed a value type, each closure has its own copy, so mutation won't work. So for both of these issues, we need a way to capture the value just once.


```

private class _Box<F: Frobulating>
where F.Input == Input, F.Output == Output {

    let _base: F
    init(_ base: F) { self._base = base }

    var base: Any { _base }

    func frobulate(from input: Input) -> Output {
        _base.frobulate(from: input)
    }
}

struct AnyFrobulator<Input, Output>: Frobulating {
    ...
    private var _box: _Box<????>
    ...
}

```

And we can do that. We could make a box that's generic over the original type. And then it could hold onto the original object as a property in that box, and forward method calls to it, even the mutating one. But now we have a new problem. **<build>**

I need a property to hold the box, but then I need to keep track of the original type. And that's the whole problem we're trying to solve. I want to hide that type. I need a way to talk about the box, without actually mentioning the type it holds. And that leads us to a very, very, ugly trick.

```

private class _BoxBase<Input, Output>
    : Frobulating
{
    var base: Any { fatalError() }
    func frobulate(from: Input) -> Output {
        fatalError()
    }
}

private class _Box<F: Frobulating>
    : _BoxBase<Input, Output> {
    ...
}

private var _box: _BoxBase<Input, Output>

self._box = _Box(base)

```

I'm going to walk through this twice because it's tricky, but it's also a very general technique for hiding types that you later need to get back.

We make an abstract class that's generic only over public information, the input and the output, not the concrete type. Now, Swift doesn't have abstract classes, so this is really ugly, and you wind up calling `fatalError` all over the place. **<build>**

Then we're going to make a subclass that is generic over the type we're erasing. **<build>**

And we'll make a property that's of the abstract class's type. **<build>**

But we'll assign an instance of the subclass's type.

I'm just going to stop for a second, and then I'm going to show it again in context.

```

struct AnyFrobulator<Input, Output>: Frobulating {
    // A base class that erases the concrete Frobulating type
    private class _BoxBase<Input, Output>: Frobulating {
        var base: Any { fatalError() }
        func frobulate(from: Input) -> Output { fatalError() }
    }

    ...
}

```

We have an abstract base class called BoxBase that is only generic over the public information, Input and Output. It conforms to the protocol. And it provides access to the base object as Any. And a reminder, it's Any because there's no more specific type we could describe it as.

```

struct AnyFrobulator<Input, Output>: Frobulating {
    // A base class that erases the concrete Frobulating type
    private class _BoxBase<Input, Output>: Frobulating {
        var base: Any { fatalError() }
        func frobulate(from: Input) -> Output { fatalError() }
    }

    // A subclass that knows the concrete Frobulating type
    private class _Box<F: Frobulating>: _BoxBase<Input, Output>
    where F.Input == Input, F.Output == Output {

        init(_ base: F) { self._base = base }

        override var base: Any { _base }
        override func frobulate(from input: Input) -> Output {
            _base.frobulate(from: input)
        }

        let _base: F
    }
    ...
}

```

Then, there's a concrete subclass called Box, that is generic over the specific concrete type. It captures the original object as a property, and forwards everything to it.

```

struct AnyFrobulator<Input, Output>: Frobulating {
    ...

    // And the actual box
    private var _box: _BoxBase<Input, Output>

    var base: Any { _box.base }

    init<F: Frobulating>(_ base: F)
        where F.Input == Input, F.Output == Output {
        self._box = _Box(base)
    }

    func frobulate(from input: Input) -> Output {
        _box.frobulate(from: input)
    }
}

```

Then, there's a property using the abstract base class, and in init, we set it using the concrete subclass. And that's it.

It's really ugly, particularly because it relies on an abstract superclass, which Swift doesn't have. And it requires a ridiculous amount of boilerplate code. But, this is pretty close to how type erasers in stdlib, like AnySequence, are implemented. Each one is a little different in the details, AnyHashable actually does this with a protocol and struct rather than two classes, because AnyHashable isn't itself generic, but this basic double-boxing approach is how "real" type-erasers are typically built.

It's not something to reach for lightly.

A large black square containing the word "Equatable" in white serif font, with the tagline "It isn't what you think." in a smaller white sans-serif font below it.

Equatable

It isn't what you think.

Just a few odds and ends before we finish up. Probably the most common cause of PATs sneaking in when you didn't mean them to is due to Equatable.

```
protocol Document {
  var path: String { get set }
}

struct TextDocument: Document, Equatable {
  var path: String
  var contents: String
}

struct Spreadsheet: Document, Equatable {
  var path: String
  var cells: [String: String]
}

let passwd = TextDocument(path: "/etc/passwd",
                          contents: "...")

let budget = Spreadsheet(path: "~/Documents/budget",
                         cells: ["A1": "-$46.00"])

let docs: [Document] = [passwd, budget]
```

Say you have bunch of documents with some path on the filesystem. TextDocuments. Spreadsheets. Whatever. **<build>** And you have some actual documents in a list. No problems here. Document is a simple protocol. We can have a list of them. But now you want to answer the question, do I already have some document in my list. But while TextDocuments and Spreadsheets are equatable. Documents, the protocol, are not.

```
protocol Document: Equatable {  
    var path: String { get set }  
}  
  
let docs: [Document] = [passwd, budget]
```

Protocol 'Document' can only be used as a generic constraint because it has Self or associated type requirements

No problem you say. I'll just add Equatable to Document. **<build>** And sure enough, blam, only used as a generic constraint. So what happened. Equatable is a PAT. It has a Self requirement, which is a special kind of associated type. So, remember the rule: PATs do not go in arrays. But so how do we say one document equals another document?


```
public protocol Equatable {  
    static func == (lhs: Self, rhs: Self) -> Bool  
}
```

Let's look at what Equatable means. In order to conform to Equatable, you need to implement this method. It determines whether two values of the same type are equal. Remember, protocols do not conform to things, they require things. So Document itself cannot be Equatable. Adding an Equatable requirement just requires conforming types to implement this method.

If Equatable means "comparable to its own concrete type," what do we really mean. Well, we mean something else. We don't mean Equatable.

```

protocol Document {
    var path: String { get set }
    func isEqual(to: Document) -> Bool
}

extension TextDocument: Document {
    func isEqual(to other: Document) -> Bool {
        guard let other = other as? TextDocument else {
            return false
        }
        return self == other
    }
}

extension Spreadsheet: Document {
    func isEqual(to other: Document) -> Bool {
        guard let other = other as? Spreadsheet else {
            return false
        }
        return self == other
    }
}

```

There is a very standard recipe for getting out of this hole, and it's is-equal-to.

And here's a common way to implement it for types that are already Equatable. You check if the values are the same type, and then check if they're equal. Now, as you see, almost everything here is duplicated, so we can simplify it with an extension.

```
protocol Document {
    var path: String { get set }
    func isEqual(to: Document) -> Bool
}

extension Document where Self: Equatable {
    func isEqual(to other: Document) -> Bool {
        guard let other = other as? Self else {
            return false
        }
        return self == other
    }
}
```

And this little snippet is what you're going to add to protocols that you want to treat in equally ways. When you do this, all your document types that are equatable get `isEqual(to:)` for free.

Now you're probably thinking, isn't this little block of code going to be repeated verbatim for a lot of different protocols. Yes. And here, we're at the limit of Swift's type system. There is no way to automatically add extensions to protocols, because that would require that a protocol conform to a protocols, which once again, they don't. So yes, you're going to paste these six lines of code into a lot of places. But in my experience, not nearly as many places as people fear. So just do it. If it's a real problem for you, then you're going to have to use a code generator like Sourcery to write it for you.

```
let passwd = TextDocument(path: "/etc/passwd",
                           contents: "...")

let budget = Spreadsheet(path: "~/Documents/budget",
                          cells: ["A1": "-$46.00"])

let docs: [Document] = [passwd, budget]

docs.contains(where: { $0.isEqual(to: passwd) })
```

Also remember, this isn't Equatable, so you don't get Equatable methods for free. You can't use the simple contains method; you have to use contains(where:). Now, of course if you want the simpler contains syntax, you can get it.

```
let passwd = TextDocument(path: "/etc/passwd",
                           contents: "...")

let budget = Spreadsheet(path: "~/Documents/budget",
                          cells: ["A1": "-$46.00"])

let docs: [Document] = [passwd, budget]

extension Sequence where Element == Document {
  func contains(_ element: Element) -> Bool {
    return contains(where: { $0.isEqual(to: element) })
  }
}

docs.contains(passwd)
```

Just add an extension on Sequence. Notice that this is Element equals Document, not Element conforms to Document. We only want this for Sequences of the Document existential.

Why add this isEqual(to:) method rather than the == operator? Because it's a little confusing. == suggests that this type conforms to Equatable, and that Equatable things will work. But that's not what's happening here. So, I don't suggest using ==. It's just too famous an operator to use for a slightly different meaning.

Protocols are not bags of syntax

Which brings us to my final point. Protocols are not bags of syntax. Equatable does not just mean that a type as an == operator. Equatable has very precise semantics.

Equatable Rules

- $a == a$ is always true (Reflexivity)
- $a == b$ implies $b == a$ (Symmetry)
- $a == b$ and $b == c$ implies $a == c$ (Transitivity)
- $a != b$ implies $!(a == b)$

Yes, and...

These are the rules people generally think of. They say that basic algebra has to apply. **<build>**
And yes, that's true, but it's not enough.

“Equality implies substitutability—any two instances that compare equally can be used interchangeably in any code that depends on their values.”

This is the more important requirement. Equality implies substitutability. If you say two things are equal, then you shouldn't care which one you have. If you care, then don't call that equal. The most common mistake is to call things equal that have the same ID, when other properties might be the different. Be really careful with that, because algorithms are allow to rely on the fact that you said they were equal.


```
protocol DefaultConstructible {  
    init()  
}
```

Protocols (a.k.a. concepts) are not just bags of syntax; unless you can attach semantics to the operations, you can't write useful generic algorithms against them. So we shouldn't have `DefaultConstructible` for the same reason we shouldn't have "Plusable" to represent something that lets you write $x + x$.

—Dave Abrahams

This idea, that protocols express meaning, not just syntax, was best explained during a discussion of whether `DefaultConstructible` should be in the standard library. Dave Abrahams, from the Crusty talk, explained no, "for the same reason we shouldn't have 'Plusable' to represent something that lets you write $x + x$." A protocol that requires an empty `init`, and nothing else, isn't enough to write interesting algorithms with, especially if you don't know anything about what the default value means. Is it a valid value? An invalid value? Empty? Some "end state?"

```
protocol RangeReplaceableCollection: Collection {  
    init()  
    mutating func replaceSubrange<C>(_ subrange: Range<Self.Index>,  
                                     with newElements: C)  
    where C : Collection, Self.Element == C.Element  
}
```

Compare it to a protocol like `RangeReplaceableCollection`. These are the only two methods that are required, and you get like two dozen other methods for free. But this `init` has semantics. It requires that it generate an empty collection. And with that, and a way to replace any arbitrary subrange, we can write about two dozen other algorithms just in terms of those two primitive operations.

Protocols are for algorithms

And that's the point of all of this. We make protocols to allow us to extract useful, reusable algorithms.

**Write concrete code first.
Then work out the generics.**

Start with the concrete. Find your protocols.

Swift Generics

It isn't supposed to hurt

Rob Napier – rob@neverwood.org

github.com/rnapier/generics

@cocoaphony

robnapier.net/start-with-a-protocol