

Swift Generics

It isn't supposed to hurt

Rob Napier – rob@neverwood.org
<https://github.com/rnapier/generics>

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, and certainly the most influential. "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,” but I worry that a lot of developers only really took away one phrase from the talk.

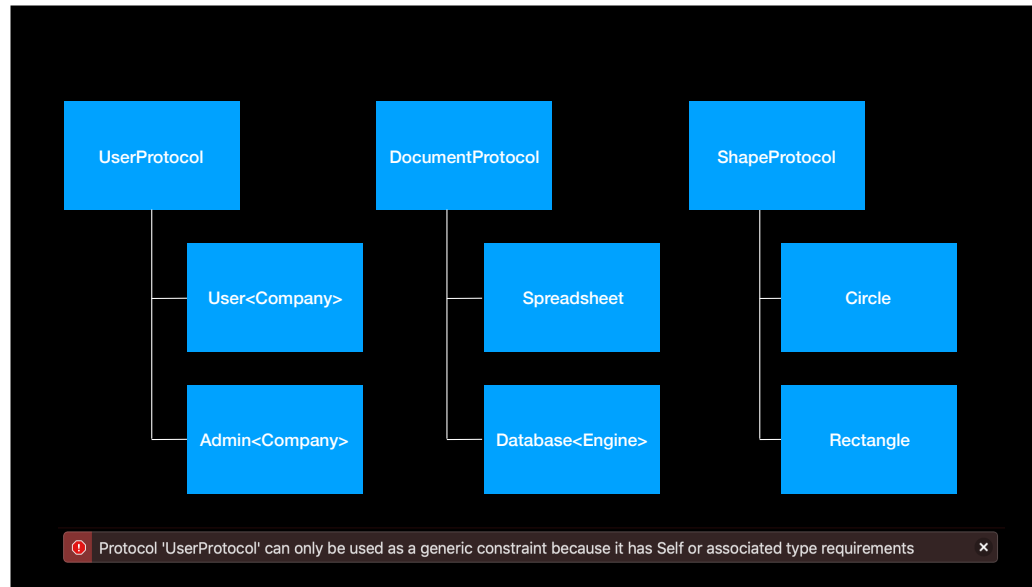


“Start with a protocol.”

—Dave Abrahams, WWDC 2015

“Start with a protocol.”

And so, dutifully, they start with a protocol.



They make a UserProtocol and a DocumentProtocol and a ShapeProtocol and on and on, <build>and then they start implementing all those protocols with generic subclasses and eventually they find themselves in a corner.<build> And that's where I usually come in, usually on Stack Overflow, and try to help people figure this stuff out.

“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. 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 see a lot of people go down a rat hole of making the 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  
}  
  
struct 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. It's a struct because I plan on it being stateless.

```

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()
}

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()
}

```

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 doe the code for fetchDocument look like?<build>

Wow, that's pretty similar.

```

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()
}

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.Type,  
                             id: Int,  
                             completion:  
                             @escaping (Result<Model, Error>) -> Void)  
{  
    ...  
}  
  
let value = try JSONDecoder().decode(Int.self, from: data) ✓  
  
let value: Int = try JSONDecoder().decode(data) ✗
```

First, I add a generic type parameter, `Model`. You'll notice that I pass the type of model as a parameter. It doesn't even need a name, because the value isn't used. It's just there to nail down the type. Doing it that way makes type inference a lot nicer. If you think about `decode.<build>` you pass the type as the first parameter in the same way. If you didn't do that, `<build>` then you'd have to add a type annotation on `value`, and even though that's a little shorter, it doesn't scale as well when things get more complicated. So make sure that every type parameter shows up in a function parameter, rather than just the type of the return value or something in a closure.


```

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()
}

```

The rest is basically the same. 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. Listen to what I said. It requires that the type be Decodable and also requires other things. I didn't say that Fetchable is Decodable. It isn't. I'm going to say that again. Model is not Decodable.

Protocols do not conform to themselves

Protocols do not conform to themselves. A type that conforms to Fetchable, must also conform to Decodable, but Fetchable is not Decodable. Why do I keep repeating this. Because you will forget, and it will bite you. What would it mean if Fetchable were Decodable?

```
func decode<T>(<_ type: T.Type, from data: Data) throws -> T
    where T : Decodable

try JSONDecoder().decode(Fetchable.self, from: data)
```

Well, remember that JSONDecoder's decode method requires a type that conforms to Decodable. If Fetchable conformed to Decodable, I could write this.<build>

And in fact, i see people try to write that all the time. But how could that possibly work? How can JSONDecoder know which of an unbounded number of possible types you want this JSON to be decoded into?

**Protocols do not conform to
themselves**

And so, I will remind you once more.<point>

```

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, back to our real problem. We have this Fetchable protocol, and so 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.

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.

<build>

OK, that last bit is actually slightly...a lie. It's legal Swift, and it's good Swift, and it absolutely should work. And this code does work. But if you use this exactly the way I do in this example, it triggers an incredibly obscure bug that crashes the compiler.

```
class NetworkTransport: Transport {
    static let shared = NetworkTransport()
    let session: URLSession
    init(session: URLSession = .shared) { self.session = session }

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

<https://bugs.swift.org/browse/SR-10481>

So, we can't really do that, and we do have to make a little NetworkTransport wrapper until the compiler bug is fixed. So you're going to see NetworkTransport a few places as we go forward, but remember, as soon as the bug is fixed, that could be URLSession instead.

```
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 = NetworkTransport.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.


```

struct 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: NetworkTransport.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
}
```

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.

```
protocol IDType: Codable, Hashable {  
    associatedtype Value  
    var value: Value { get }  
    init(value: Value)  
}  
  
let refreshIDs: [IDType] = [User.ID(4),  
                             Document.ID("budget")]
```

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

Now, IDType is a PAT, a protocol with an associated type. It can't be put in a variable. It can't be put in an Array. And as soon as you write this line of code, trying to put together a list of IDs to be refreshed or whatever, you're going to get this error. And then you're going to hunt around, and somewhere you're going to read the words type-eraser.

You Don't Need a Type-Eraser

And while there are definitely some cases where a type eraser is useful, they're kind of rare, and this isn't one of them. If you have an associated type, and you think you need a type-eraser to deal with it, you almost certainly need to redesign your types instead.

```
protocol IDType: Codable, Hashable {  
    associatedtype Value  
    var value: Value { get }  
    init(value: Value)  
}  
  
let refreshIDs: [IDType] = [User.ID(4),  
                             Document.ID("budget")]
```

What's the next line of code?

So you've written this line of code, to gather up the list of IDs to refresh, and it doesn't work. But let's pretend it did. Let's pretend that Generalized Existentials have come to our rescue and this works.<build>

What's the next line of code? I means seriously. Look at IDType and what it promises. What would you possibly do with "an ID type" where you don't know the type of Value. Where it could be literally anything at all. You don't even know what type IDType is the ID for. I mean seriously, think about it. What's the next line of code that does something useful? But if your goal is to put together a list of things to refresh, then just put together a list of things to refresh.

```
for id in refreshIDs {  
  guard let model = modelTypeForID[id] else { continue }  
  refresh(model, id: id)  
}
```

Let's say you had this in mind. What you really want to do is call this refresh function for each ID. And that's fine. I mean, this code won't actually work, and if you're waiting for generalized existentials to fix it for you, they won't fix that either. But the idea is fine.

```
struct RefreshRequests {
    let perform: () -> Void
}

extension RefreshRequests {
    init(userID: User.ID) {
        self.init(perform: { refresh(User.self, id: userID) })
    }

    init(documentID: Document.ID) {
        self.init(perform: { refresh(Document.self, id: documentID) })
    }
}

let refreshes = [
    RefreshRequests(userID: User.ID(4)),
    RefreshRequests(documentID: Document.ID("budget")),
]

for refresh in refreshes { refresh.perform() }
```

But then build what you want. You have some requests to refresh. Put those in an array, and perform them. There's no need for all the complexity of PATs. You don't even need generics. Just functions.

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

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

PATs are useful. They're just not for putting in arrays.

The most important part of IDType was the free stuff for User.ID and Document.ID.

They didn't have to restate that they conform to Codable and Hashable. They didn't have to create this init. That's what a PAT is for. Or it's to allow a specific type to be passed to some function as a generic constraint. But if you want to put it in an Array, you don't want a PAT.


```
protocol Fetchable: Decodable {
    static var apiBase: String { get }
    associatedtype ID: IDType
    var id: ID { get }
}

func fetch<Model: Fetchable>(
    _ model: Model.Type,
    id: Model.ID,
    with transport: Transport = NetworkTransport.shared,
    completion: @escaping (Result<Model, Error>) -> Void)
{
    let urlRequest = URLRequest(url: baseURL
        .appendingPathComponent(Model.apiBase)
        .appendingPathComponent("\(id.value)")
    )
    ...
}
```

?

And then we can add this to Fetchable and fetch. Now, IDType is a PAT, that means we're making Fetchable a PAT, too. So we need to ask the question, do we ever want Fetchable to be in an array?

I don't think so?<build> But this should make you a little nervous. We've just taken a non-PAT and made it a PAT. That's not necessary bad, but it's a big move. In this case, I think it's fine.

```

// 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())
        })
    }
}

// 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)
        }
    }
}

```

Back to fetch. 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.

```

// 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())
        })
    }
}

// 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)
        }
    }
}

```

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.



```
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
}

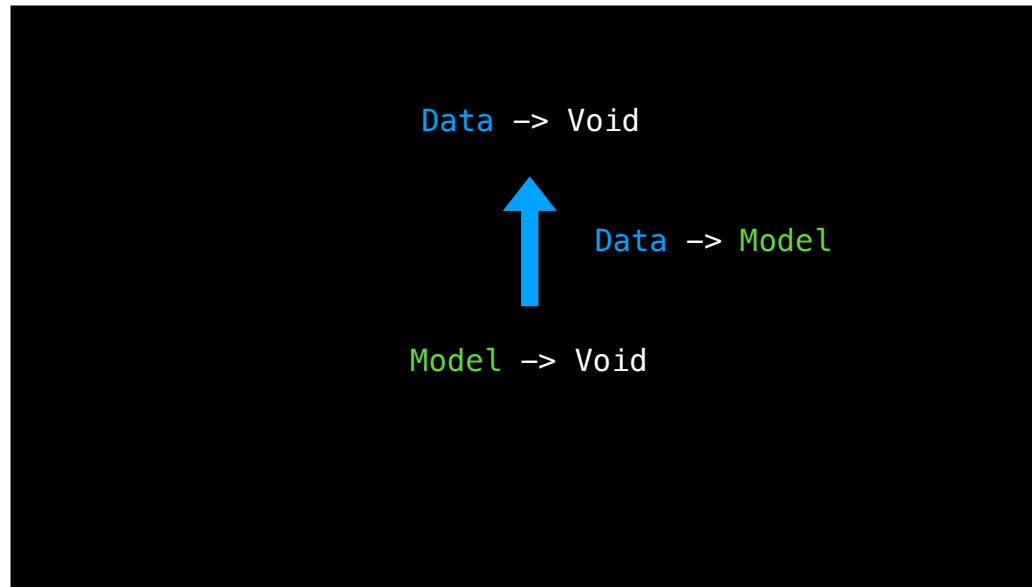
let request = Request.fetching(User.self,
                                id: User.ID(0),
                                completion: { user in ... })

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.



So to simplify a little. What do we need? 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.

The logo for Equatable is centered on a solid black rectangular background. The word "Equatable" is written in a white, serif font.

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. 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 has 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 $!(a == b)$

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
<https://github.com/rnapier/generics>