4 Generics

Written by Marin Bencevic

Almost everyone using Swift – from complete beginners to seasoned veterans – has used generics, whether they know it or not. Generics power arrays and dictionaries, JSON decoding, Combine publishers and many other parts of Swift and iOS. Because you have already used many of these features, you know firsthand how powerful generics are. In this chapter, you'll learn how to harness that power to build generics-powered features.

You'll get intimately familiar with generics by continuing to work on the networking library you started in the previous chapter. This time, you'll modify it to use generics to create a nicer API. You'll learn how to write generic functions, classes and structs, how to use protocols with associated types, what type erasure is and how to put all that together to make a coherent API.

Before you do that, though, this chapter front-loads quite a bit of theory, giving you a reference of what generics have to offer. Don't worry — you'll get your hands dirty later in the chapter!

Getting started with generics

Although it's not required, you can follow along with this section by creating a new plain Swift Xcode playground. Begin by writing a generic function:

```
func replaceNilValues<T>(from array: [T?], with element: T) -> [T]
{
  array.compactMap {
    $0 == nil ? element : $0
  }
}
```

The one special bit of syntax that makes this function generic is `<T>` in the function's prototype. While parentheses (`()`) surround the function's parameters, angle brackets (`< >`) surround the function's **type parameters**. A generic function receives type parameters as part of a function call, just like it receives regular function parameters.

In this case, there is only one type parameter, called `T`. The name of this parameter isn't some magic, special constant — it's user-defined. In the example above, you could have used `Element` or anything else. Once you define the type parameter inside the angle brackets, you can use it in the rest of the function's declaration and even inside the function's body.

When you call this function, Swift replaces `T` with a concrete type that you're calling the function with.

```
let numbers: [Int?] = [32, 3, 24, nil, 4]
let filledNumbers = replaceNilValues(from: numbers, with: 0)
print(filledNumbers) // [32, 3, 24, 0, 4]
```

COPY

In the function's prototype, you defined that the function receives an array of optional `T`s as well as another `T` value. When you call the function with `numbers`, Swift knows that `numbers` has a type of `[Int?]` and can figure out that it needs to replace `T` with `Int`. Swift is smart like that.

This allows you to create a single function that works across all possible types, saving you from having to copy and paste functions. In a sense, generics are the opposite of protocols. Protocols allow you to call a function on multiple types where each type can specify its implementation of the function. Generics allow you to call a function on multiple types with the *same* implementation of that function.

> **Note**: When your function is *very* generic, and the type can be any type, it's fine to use single letter type parameter names like `T` and `U`. But more often than not, your type parameter will have some sort of semantic meaning. In those cases, it's best to use a more descriptive type name that hints at its meaning to the reader. For example, instead of using single letters, you might use `Element`, `Value`, `Output`, etc.

Of course, like regular parameters, you can have multiple comma-separated type parameters:

```
func replaceNils<K, V>(
  from dictionary: [K: V?],
  with element: V) -> [K: V] {
  dictionary.compactMapValues {
    $0 == nil ? element : $0
  }
}
```

COPY

However, sometimes instead of a function that works across *all* possible types, you want one that works across only some of them. Swift allows you to add constraints to your generic types:

```
func max<T: Comparable>(lhs: T, rhs: T) -> T {
  return lhs > rhs ? lhs : rhs
}
```

COPY

In the example above, you need the ability to compare two values with the `>` operator. Not every type in Swift can be compared (for example, is one `View` larger than another?). So you need to specify that `T` must conform to `Comparable`. Swift then knows there's a

valid implementation of `>` for `T`. You're using `Comparable` as a **generic constraint**: a way to tell Swift which types are accepted for the generic type parameter.

Generic types

Generic functions can take you only so far. At some point, you run into cases where you need a generic class or struct. You already use generic types all the time: Arrays, dictionaries and Combine publishers are all generic types.

Look at a generic struct:

```
struct Preference<T> {
  let key: String

  var value: T? {
    get {
      UserDefaults.standard.value(forKey: key) as? T
    } set {
      UserDefaults.standard.setValue(newValue, forKey: key)
    }
  }
}
```

Like generic functions, generic types also have type parameters, declared right next to the type name. The example above shows a generic struct that can store and retrieve any type from `UserDefaults`.

You can provide a concrete type to a generic type by writing the type inside angle brackets next to the type's name:

```
var volume = Preference<Float>(key: "audioVolume")
volume.value = 0.5
```

Here, Swift replaces `T` with `Float`. The process of replacing a type parameter with a concrete type value is called **specialization**. In this case, typing `<Float>` is necessary because Swift doesn't have a way to infer it. In other cases, when you use the type parameter in the initializer, Swift can figure out what your concrete type is without you having to write angle brackets.

Keep in mind, though, that `Preference` itself *is not a type*. If you try to use `Preference` as the type of a variable, you get a compiler error. Swift recognizes only specialized variants of the type, such as `Preference<String>`, as real types. Generic types by themselves are more like a blueprint: a type of scaffolding for you but not of much use for the compiler.

Protocols with associated types

Aside from generic structs, enums and classes, you can also have generic protocols. Except we don't call them that, we call them **protocols with associated types** or PATs for short. PATs are structured a little differently. Instead of the generic type being a parameter

of the protocol, it's one of the protocol's requirements, like protocol methods and properties.

```
protocol Request {
  associatedtype Model
  func fetch() -> AnyPublisher<Model, Error>
}
```

COPY

In the protocol above, `Model` is simply one of the protocol's requirements. To implement the protocol, you need to declare a concrete `Model` type by adding a `typealias` to your implementation:

```
struct TextRequest: Request {
  typealias Model = String

  func fetch() -> AnyPublisher<Model, Error> {
    Just("")
      .setFailureType(to: Error.self)
      .eraseToAnyPublisher()
  }
}
```

COPY

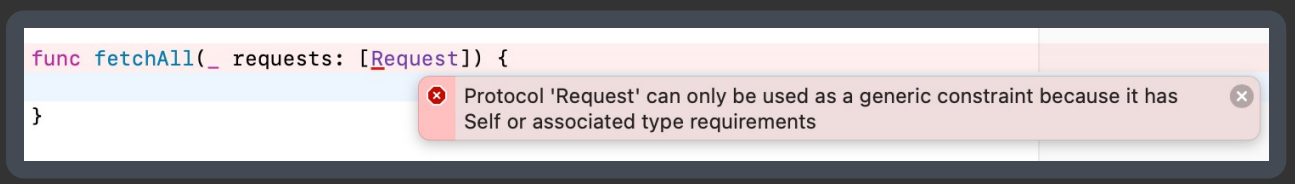
In most cases, Swift can figure out the associated type, so you don't need to add a `typealias` as long as you use the type when implementing one of the protocol's methods:

```
struct TextRequest: Request {
  func fetch() -> AnyPublisher<String, Error> {
    // ...
  }
}
```

COPY

In the example above, Swift sees that you use `String` in the place of `Model`, so it can infer that `String` is the associated type.

Like generic types, **PATs are not types**! If you don't believe me, try using a PAT as a type:



The error tells you that `Request` can be used only *as a generic constraint*, which is mentioned earlier in this section. Implied in that sentence is that it *cannot* be used as a type. The reason has to do with how Swift handles generic types. Swift needs to have a concrete type to work with at compile-time so it can save you from errors and undefined behavior while your program is running. A generic type without all of its type parameters is not a concrete type. Depending on the type parameters, method and property implementations can change, and the object itself can be laid out differently in memory. Because Swift always errs on the side of caution, it forces you to always use concrete, known types in your code.

It's good to be safe, but how the heck do you define an array of PATs, then? The answer is **type erasure**, and you'll see an example later in this chapter.

Extending generics

In the previous chapter, you've seen all the ways you can extend protocols and the types that implement them. Generics are no different! First of all, you can extend generics as any other type, with the added benefit that you get access to the type parameters inside the extension:

```
extension Preference {
  mutating func save(from untypedValue: Any) {
    if let value = untypedValue as? T {
      self.value = value
    }
  }
}
```

COPY

In the example above, you have access to `Preference`s type parameter `T`, which you use to cast the received value.

Like protocol extensions, you can also constrain extensions of generic types. For instance, you can constrain an extension to only generic types where the type parameter implements a protocol:

```
extension Preference where T: Decodable {
  mutating func save(from json: Data) throws {
    let decoder = JSONDecoder()
    self.value = try decoder.decode(T.self, from: json)
  }
}
```

COPY

In the code above, `save` will only exist on `Preference`s where the type parameter is `Decodable`.

You don't need to constrain the extension — you can also constrain a single method:

```
extension Preference {
  mutating func save(from json: Data) throws where T: Decodable {
    let decoder = JSONDecoder()
    self.value = try decoder.decode(T.self, from: json)
  }
}
```

COPY

This code does the same thing as the code block you just saw. But it constrains the method itself instead of the whole extension.

Extending PATs works the same way. You'll see an example of that when you get to the practical section of this chapter.

Self and meta-types

"What is the self?" is a philosophical question. More important for this chapter is explaining what `self` is as well as what `Self` and `T.self` are. These are much easier to answer than the philosophical question. Although this section might not relate directly to generics, it has a lot to do with the type system itself. And understanding the different *selves* and ways to use types in Swift will help you better understand generics.

As you already know, `self` is usually a reference to the object whose scope you're currently in. If you use `self` inside an instance method of a `User` struct, `self` will be that instance of that struct. So far, that's pretty straightforward. However, when you're in a class method of a class, `self` can't be a reference to an instance because *there is no instance*: You're in the class *itself*.

```
class Networker {
  class func whoAmI() {
    print(self)
  }
}

Networker.whoAmI() // "Networker"
```

In class and static methods, `self` has the value of the current type, not an instance. It makes sense when you think about it: Static and class methods exist on the type, not an instance.

However, all values in Swift need to have a type, including the `self` above. After all, you need to be able to store it in variables and return it from functions. What would be the type that holds `self` in class and static methods, that you now know holds *a type*? The answer is `Networker.Type`: a type encompassing all `Networker` subtypes! Just like `Int` holds all integer values, `Int.Type` holds all `Int` *type* values. These types that hold other types are called **meta-types**. It kind of makes your head spin, right?

```
class WebsocketNetworker: Networker {
  class func whoAmI() -> Networker.Type {
    return self
  }
}

let type: Networker.Type = WebsocketNetworker.whoAmI()
print(type)
```

In the example above, you declare a meta-type variable called `type`. The meta-type can hold not only the `Networker` type itself but also all of its subclasses, such as

`WebsocketNetworker`. In the case of protocols, a meta-type of a protocol (`YourProtocol.Type`) can hold the protocol type as well as all concrete types conforming to that protocol.

To use a type itself as a value, such as to pass it to a function or store it in a variable, you need to use `Type.self`:

```
let networkerType: Networker.Type = Networker.self
```

COPY

You have to do this for practical reasons. Usually, type names are used to declare the type of a variable or function parameter. When they're not used for declaring types, they're used implicitly as initializers. Using `.self` makes it clearer that you need the type as a value rather than as the type of something else and that you are not calling an initializer.

Finally, there is `Self` with a capital "S". Thankfully, this one is less convoluted than all this meta-talk. `Self` is always an alias to the *concrete* type of the scope it appears in. Concrete is emphasized because `Self` will always be a concrete type, even if it's used inside a protocol method.

```
extension Request {
  func whoAmI() {
    print(Self.self)
  }
}

TextRequest().whoAmI() // "TextRequest"
```

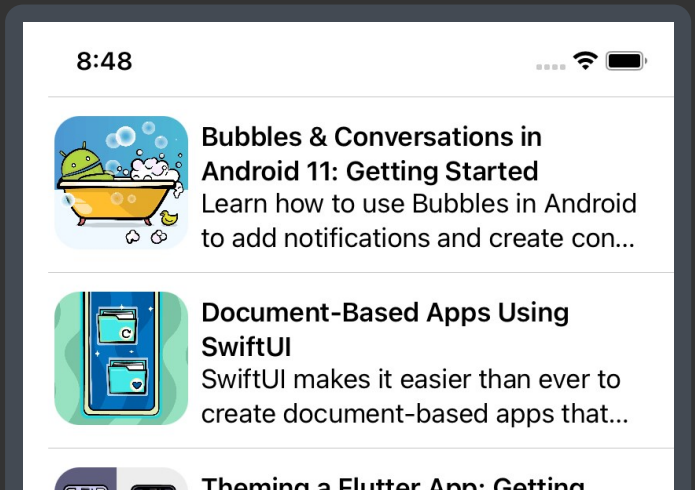
COPY

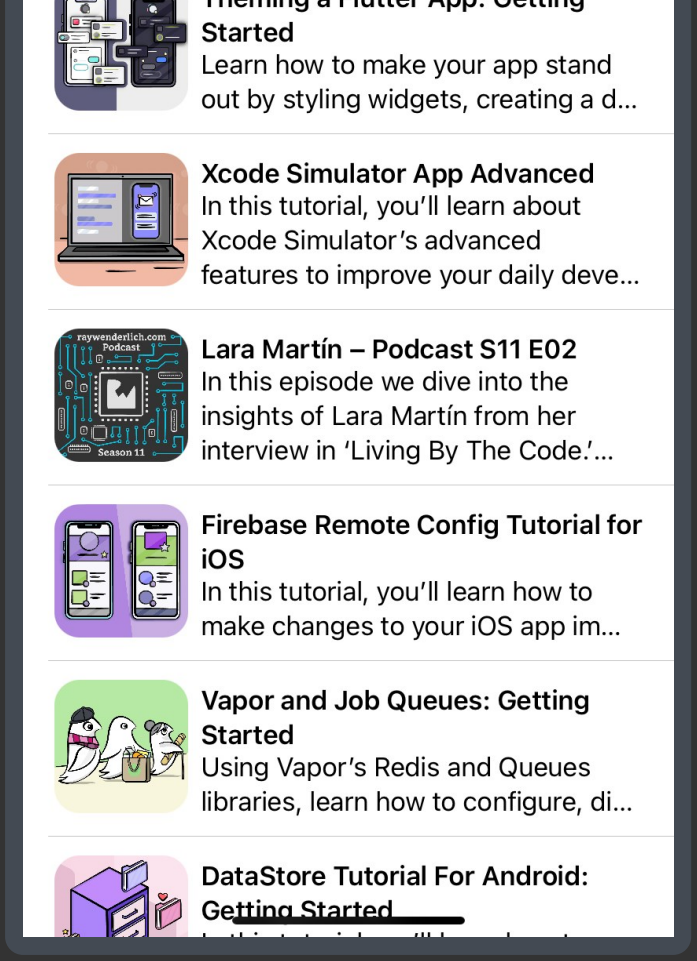
`Self` is useful when you want to return *the current concrete type* from a protocol method or use it as an initializer inside a static method when creating factory methods.

That's enough theory. It's time to fire up Xcode and get into using generics.

Creating a generic networking library

Open the starter project provided in this chapter's materials in Xcode. This project is almost the same one that you wrote in the previous chapter. If you haven't already, read the previous chapter to get familiar with the project. It's a tiny raywenderlich.com client app that uses your networking library powered by protocols.





In this chapter, you'll expand that library to use generics to provide an even nicer API for your users.

Making `Networker` generic

You'll start by adding a generic function in **Networker.swift** that can download a `Decodable` type and decode it. Add the following function prototype to `Networking`:

```
func fetch<T: Decodable>(url: URL) -> AnyPublisher<T, Error>
```

`fetch(url:)` is a generic function with one generic type parameter called `T`. You declare that `T` is a type that must conform to `Decodable`. Once you declare `T` as a type parameter, you can use it elsewhere in the type signature — like as the return value, for instance.

Next, implement the method inside `Networker`:

```
func fetch<T: Decodable>(url: URL) -> AnyPublisher<T, Error> {
  URLSession.shared.dataTaskPublisher(for: url)
    .map { $0.data }
    .decode(type: T.self, decoder: JSONDecoder())
    .eraseToAnyPublisher()
}
```

Because you have access to the generic type parameter `T`, you can use it in the function's body to decode the received data into `T`. Swift will replace `T` with whatever concrete type the function is called with. Because you declared that `T` conforms to `Decodable`,

the compiler is happy with your code. If someone tries to call this function with a non-`Decodable` type like `UIImage`, the compiler will throw an error.

Using PATs

Using the generic function is good when you have quick one-off requests. But it would help to be able to create reusable requests you can fire from anywhere in your code. To do this, you'll turn `Request` into a protocol with an associated type. Open **Request.swift** and add the following two lines to the protocol:

```
associatedtype Output
func decode(_ data: Data) throws -> Output
```

COPY

The `Output` type tells the user what this request is supposed to fetch. It can be an `Article`, `[Articles]`, `User`, etc. The `decode(_:)` function is responsible for converting the data received from `URLSession` into the output type.

Next, add a default implementation of `decode(_:)` to the bottom of the file:

```
extension Request where Output: Decodable {
  func decode(_ data: Data) throws -> Output {
    let decoder = JSONDecoder()
    return try decoder.decode(Output.self, from: data)
  }
}
```

COPY

When you create a `Request` implementation whose type conforms to `Decodable`, you'll get this implementation for free. It will try to use a JSON decoder to return the `Output` type.

Because the raywenderlich.com API provides a JSON response that doesn't necessarily match the models defined in your project, you won't be able to use this default implementation. You'll provide your own in **ArticleRequest.swift**.

Add a new method to the struct:

```
func decode(_ data: Data) throws -> [Article] {
  let decoder = JSONDecoder()
  let articlesCollection = try decoder.decode(Articles.self, from:
data)
  return articlesCollection.data.map { $0.article }
}
```

COPY

You first decode the received data into `Articles`, a helper struct that matches the API's response. You then convert that to an array of `Article` and return it. Notice you haven't specified that `Output` is `[Article]`. Because you used it as the return type of `decode(_:)`, Swift can infer the type of `Output` without you saying so.

Next, you'll also implement `decode(_:)` for images. Open **ImageRequest.swift** and add an enum *inside* the struct:

```
enum Error: Swift.Error {
  case invalidData
}
```

You create a custom enum to represent different kinds of errors that can occur while decoding the image. In this case, you'll use only one error, but you can expand this in your own code to be more descriptive. By conforming to Swift's `Error` type, you get the ability to use this enum as the error type of a Combine publisher or to use it with the `throw` keyword.

Finally, implement `decode(_:)` in the struct:

```
func decode(_ data: Data) throws -> UIImage {
  guard let image = UIImage(data: data) else {
    throw Error.invalidData
  }
  return image
}
```

You try to convert the data to `UIImage`. If it doesn't work, you throw the error you just declared.

Type constraints

Now that you've made changes to `Request`, it's time to use those changes in **Networker.swift**. You might have noticed a weird compiler error: "Protocol `Request` can be used only as a generic constraint because it has `Self` or associated type requirements".

Earlier, I mentioned that protocols with associated types are not types themselves, though they may act like types. Instead, they're type constraints.

Change the declaration of `fetch(_:)` in `Networking` to the following:

```
func fetch<R: Request>(_ request: R) -> AnyPublisher<R.Output,
Error>
```

You convert `fetch(_:)` to a generic function over the type `R`, representing any request. You know that it's a request since you declare that the type must conform to `Request`. You then return a publisher using the `Request`'s associated `Output` type. Here, you're no longer using `Request` as a concrete type. Instead, you're using it as a constraint on `R`, so the compiler error goes away.

Next, change `fetch(_:)` in `Networker` to match the new protocol requirement:

```
func fetch<R: Request>(_ request: R) -> AnyPublisher<R.Output,
Error> {
  var urlRequest = URLRequest(url: request.url)
```

```
  urlRequest.httpMethod = request.method.rawValue
  urlRequest.allHTTPHeaderFields = delegate?.headers(for: self)

  var publisher = URLSession.shared
    .dataTaskPublisher(for: urlRequest)
    .compactMap { $0.data }
    .eraseToAnyPublisher()

  if let delegate = delegate {
    publisher = delegate.networking(self, transformPublisher:
publisher)
  }

  return publisher.tryMap(request.decode).eraseToAnyPublisher()
}
```

The function stayed mostly the same except for a couple of key changes. First, you changed the declaration to make it a generic function. Second, you added a line at the end that tries to call the `Request`'s `decode(_:)` function to return the `Output` associated type.

Now that you've made all these changes, you can finally use the networker as intended. Open **ArticlesViewModel.swift**.

First, delete the `.tryMap([Article].init)` line from `fetchArticles`. Because `Request` does this for you, you no longer need the line. Further, because `ArticleRequest` declares `[Article]` as its `Output` type, Swift knows to publish `[Article]` values from the publisher, so the type system is happy.

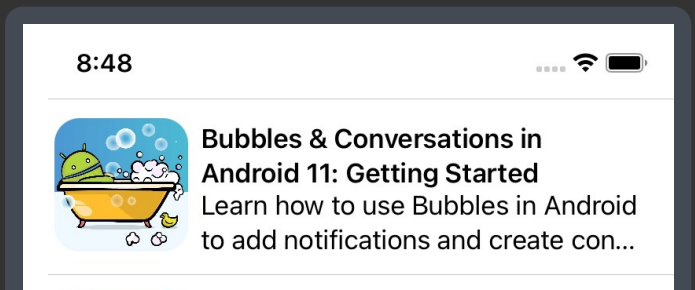
Next, inside `fetchImage`, replace the whole `fetch(_:)` call chain with the following:

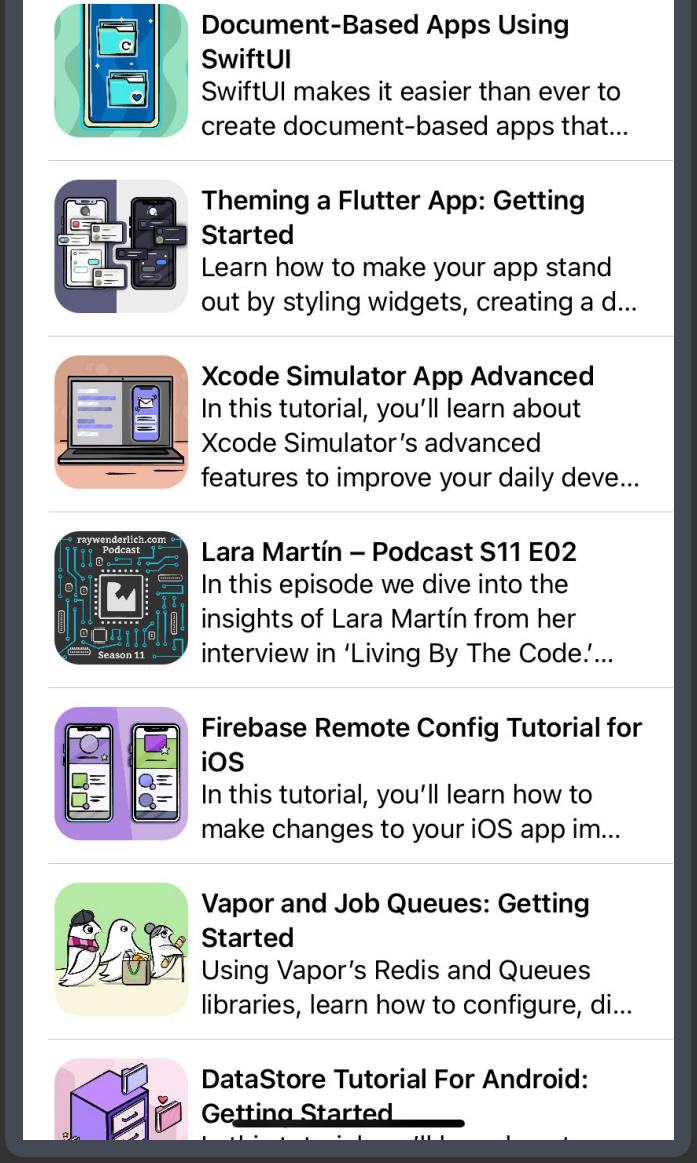
```
let request = ImageRequest(url: article.image)
networker.fetch(request)
  .sink(receiveCompletion: { completion in
    switch completion {
    case .failure(let error): print(error)
    default: break
    }
  }, receiveValue: { [weak self] image in
    self?.articles[articleIndex].downloadedImage = image
  })
  .store(in: &cancellables)
```

COPY

Again, there's no need to convert anything to `UIImage` because `ImageRequest` does that for you. Instead, you grab the image when it arrives or print out an error if it doesn't.

Build and run the project.





You should see a list of raywenderlich.com articles together with their images. Congratulations, you just made a working networking library using generics! Don't rest on your laurels, though. You can still improve the library further by adding caching using generics.

Adding caching with type erasure

In the previous chapter, you added a check in your view model that checked whether an image was already downloaded because the `fetchImage` function is called every time a row appears on the screen. This is an ad-hoc way to achieve caching. You can make this behavior more reusable by adding a generic cache class to your project.

Create a new Swift file called **RequestCache.swift** and add the following to the file:

```
class RequestCache<Value> {
  private var store: [Request: Value] = [:]
}
```

COPY

You create a new generic class that stores request responses in an in-memory cache inside a plain Swift dictionary. You'll store the responses keyed by the request that fetched them so you can always easily tie a request to its response.

You might notice a compiler error. The error is the same one you saw earlier: "Protocol

`Request` can only be used as a generic constraint because it has `Self` or associated type requirements". Earlier, you fixed this error by *not* using the `Request` type directly instead of using it as a constraint.

But in this case, that's not possible. You can't constrain the keys of a dictionary – they all need to be the same concrete type. In cases where you need to use a protocol with an associated type as a concrete type, you need to employ **type erasure**. You can think of PATs as generic protocols. And type erasure, as its name suggests, is a way to convert that generic protocol into a concrete type by *removing* type information.

Head to **Request.swift** and add a new struct to the bottom of the file:

```
struct AnyRequest: Hashable {
  let url: URL
  let method: HTTPMethod
}
```

COPY

This struct is what enables type erasure. You can convert any `Request`, regardless of its associated type, to an instance of `AnyRequest`. `AnyRequest` is just a plain Swift struct without any generics. So, naturally, you've lost type information along the way. You can't use `AnyRequest` to make your code more type-safe. But sometimes you can get away with discarding type information, allowing you to write your code more easily.

Now, you can go back to **RequestCache.swift** and use the type-erased struct instead of `Request` in `store`'s declaration:

```
private var store: [AnyRequest: Value] = [:]
```

COPY

You're not the only one using type erasure in this way. Plenty of Apple's APIs use the same pattern, such as Combine's `AnyCancellable` or `AnyPublisher`. `AnySequence`, `AnyIterator` and `AnyCollection` can help you create your own sequences and collections more easily. SwiftUI's `AnyView` allows you to store different types of views in the same data structure. These are just a few examples to show you that type erasure is a common pattern in Swift. Because the pattern is common, getting familiar with it will help you understand existing APIs as well as create new APIs in the future.

Fetching and saving a response

You can now continue writing your class. You'll add two methods to the class, one to fetch a stored response and another to save a response. Add a new method below the property you just declared:

```
func response<R: Request>(for request: R) -> Value?
  where R.Output == Value {
  let erasedRequest = AnyRequest(url: request.url, method:
request.method)
  return store[erasedRequest]
}
```

COPY

This function will get called when someone wants to retrieve an already stored response for a given request. The function's prototype might look a bit complicated, so breaking it down will help. First, you declare a generic parameter `R` that must conform to `Request`. You then receive a parameter of that same type and return an instance of `Value`, the generic type parameter of the `RequestCache` class. Finally, you specify that `R`'s associated `Output` type must be the same type as the class's type parameter.

The last bit of the function's prototype verifies that no one will accidentally call the function with the wrong request. If the cache stores images (`RequestCache<UIImage>`), only `Request`s that fetch images can be retrieved (`R.Output == UIImage`). If you try to call the method with a mismatched `Request`, you'll get a compiler error:

However, keep in mind that this is *still* a generic function, so multiple `Request` types can be retrieved as long as their output type matches the type of the stored values. For instance, you can use both `AvatarThumbnailRequest` and `ImageRequest` to call this method on a `RequestCache<UIImage>` instance.

Inside the method, you use type erasure by constructing an `AnyRequest` from the provided request, which you can use to retrieve a value from the dictionary.

Next, add a method to save a new response for a request:

```
func saveResponse<R: Request>(_ response: Value, for request: R)
  where R.Output == Value {
  let erasedRequest = AnyRequest(url: request.url, method:
request.method)
  store[erasedRequest] = response
}
```

Once again, you add a `where` clause to the function's signature to verify that the request type matches, disallowing accidental wrong entries. As you did in `response(for:)`, you use type erasure to store a new response in the dictionary.

Now that you have a generic cache, you can create a cache to store all your downloaded images. Open **Networking.swift** and add a new property to `Networker`:

```
private let imageCache = RequestCache<UIImage>()
```

You'll use this instance to store the image request responses.

Next, add a new (unfinished) method to the class:

```
func fetchWithCache<R: Request>(_ request: R)
  -> AnyPublisher<R.Output, Error> where R.Output == UIImage {
  if let response = imageCache.response(for: request) {
```

```
    return Just<R.Output>(response)
      .setFailureType(to: Error.self)
      .eraseToAnyPublisher()
  }
}
```

You'll finish up the method in a bit.

You create a new generic method that receives a request and returns a publisher just like `fetch(_:)`. But this method uses the cache to retrieve responses if they're already stored or store new responses if they aren't. The function's prototype declares that this method can be used only by requests whose `Output` type is `UIImage` because you currently only cache `UIImage` instances.

Inside the method, you first check if a response is already cached. If it is, you return a publisher that emits the cached value using `Just`.

Finish the method with the following code at the bottom:

```
return fetch(request)
  .handleEvents(receiveOutput: {
    self.imageCache.saveResponse($0, for: request)
  })
  .eraseToAnyPublisher()
```

If there is no cached response, you use `fetch(_:)` to return a new publisher. You also subscribe to the publisher's output event so you can store the response in the cache.

Next, add your new method to `Networking`:

```
func fetchWithCache<R: Request>(_ request: R)
  -> AnyPublisher<R.Output, Error> where R.Output == UIImage
```

Now that you have your new caching method in `Networker`, it's time to use it from the view model. Open **ArticlesViewModel.swift** and change `fetchImage` to the following:

```
func fetchImage(for article: Article) {
  guard let articleIndex = articles.firstIndex(
    where: { $0.id == article.id }) else {
    return
  }

  let request = ImageRequest(url: article.image)
  networker.fetchWithCache(request)
    .sink(receiveCompletion: { error in
      print(error)
    }, receiveValue: { image in
      self.articles[articleIndex].downloadedImage = image
    })
    .store(in: &cancellables)
}
```

You no longer need to perform any checks here. `Networker` takes care of all caching, letting the view model focus on preparing the view's data and not worry about storing requests.

Build and run the project. It should work just like before, except now you have a much nicer API.

Key points

- Methods, structs, enums and classes all can become generic by adding **type parameters** inside angle brackets!
- Protocols can be generic as well through use of **protocols with associated types**.
- `self` has the value of the current type in static methods and computed properties, and the type of `self` in those cases is a **meta-type**.
- `Self` always has the value of the current concrete type.
- You can use extensions with **generic constraints**, using the `where` keyword to extend generic types when their type parameters satisfy specific requirements.
- You can also specialize methods themselves by using the `where` keyword.
- Use **type erasure** to use generics and PATs as regular types.

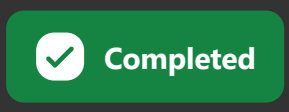
Where to go from here?

Now that you're more familiar with generics, you can explore the many generic types inside Swift itself. For instance:

- `Array` (https://bit.ly/3tzPr9V) is a generic struct.
- `Collection` (https://bit.ly/39WVNbJ) is a PAT.
- `AnyHashable` (https://bit.ly/3p4rwMw) is an example of how Swift uses type erasure.

If you want even more introduction to generics, look at the Swift Generics (Expanded) (https://apple.co/3cNKeW7) WWDC 2018 session.

If you want more behind the scenes information on how generics are implemented and laid out in memory, check out the Understanding Swift Performance (https://apple.co/2YTlQtT) WWDC 2016 session.



5. Numerics & Ranges →

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

Have feedback to share about the online reading experience? If you have feedback about the UI, UX, highlighting, or other features of our online readers, you can send them to the design team with the form below:

Feedback about the UI, UX, or other features of the online reader? Leave them here!

Send Feedback