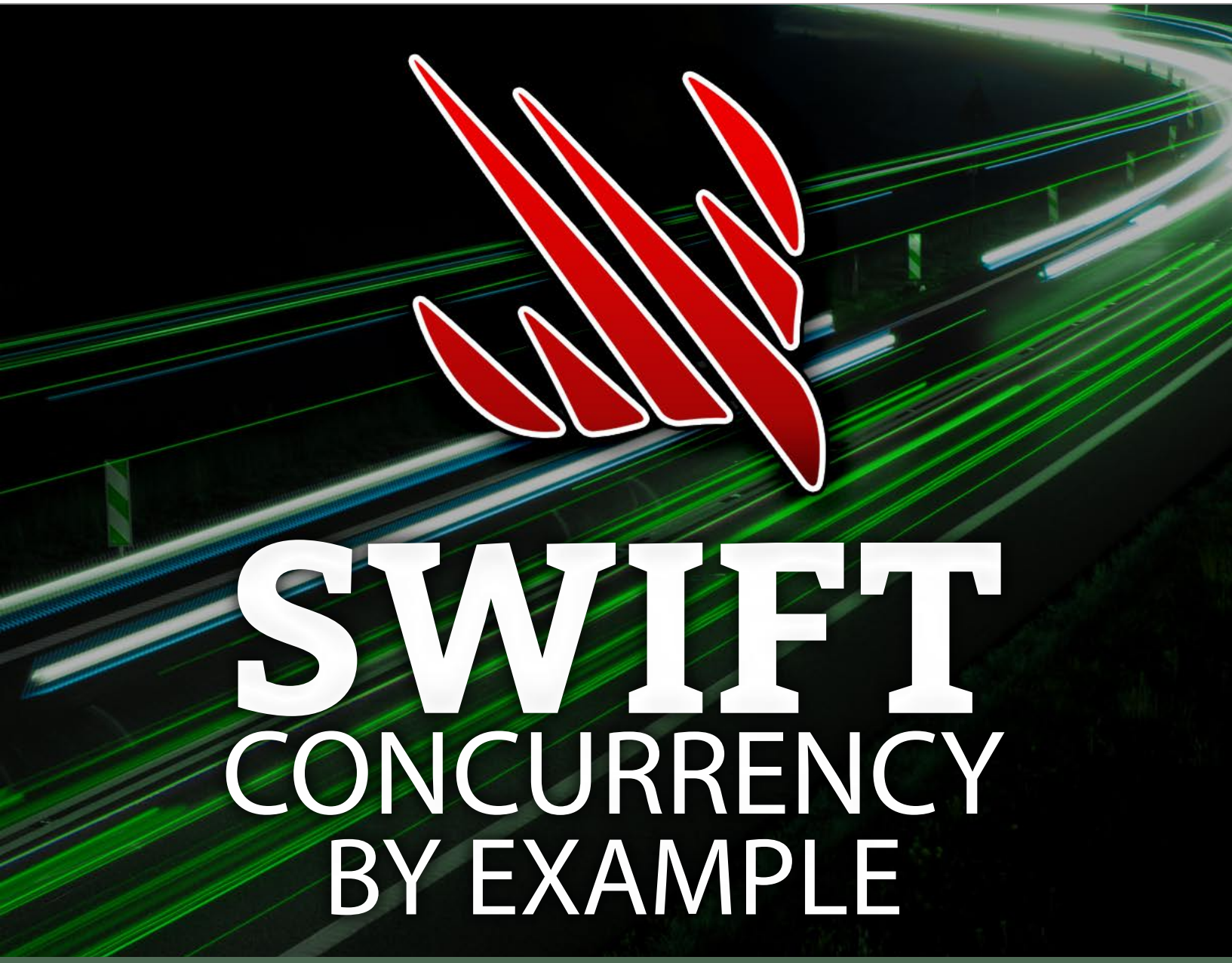
HACKING WITH SWIFT



SWIFT CONCURRENCY BY EXAMPLE

COMPLETE REFERENCE GUIDE

What it does, how it works, and best practices for your projects

Paul Hudson

Swift Concurrency by Example

Paul Hudson

Version: 2022-05-30

Contents

Introduction 5

This stuff is hard
How to follow this guide
Concurrency vs parallelism
Understanding threads and queues
Main thread and main queue: what's the difference?
Where is Swift concurrency supported?
Dedication

Async/await 16

What is a synchronous function?
What is an asynchronous function?
How to create and call an async function
How to call async throwing functions
What calls the first async function?
What's the performance cost of calling an async function?
How to create and use async properties
How to call an async function using async let
What's the difference between await and async let?
Why can't we call async functions using async var?
How to use continuations to convert completion handlers into async functions
How to create continuations that can throw errors
How to store continuations to be resumed later
How to fix the error "async call in a function that does not support concurrency"

Sequences and streams 59

What's the difference between Sequence, AsyncSequence, and AsyncStream?
How to loop over an AsyncSequence using for await
How to manipulate an AsyncSequence using map(), filter(), and more
How to create a custom AsyncSequence
How to convert an AsyncSequence into a Sequence

Tasks and task groups 80

What are tasks and task groups?
How to create and run a task
What's the difference between a task and a detached task?

How to get a Result from a task
How to control the priority of a task
Understanding how priority escalation works
How to cancel a Task
How to make a task sleep
How to voluntarily suspend a task
How to create a task group and add tasks to it
How to cancel a task group
How to handle different result types in a task group
What's the difference between async let, tasks, and task groups?
How to make async command-line tools and scripts
How to create and use task local values
How to run tasks using SwiftUI's task() modifier
Is it efficient to create many tasks?

Actors **151**

What is an actor and why does Swift have them?
How to create and use an actor in Swift
How to make function parameters isolated
How to make parts of an actor nonisolated
How to use @MainActor to run code on the main queue
Understanding how global actor inference works
What is actor hopping and how can it cause problems?
What's the difference between actors, classes, and structs?
Important: Do not use an actor for your SwiftUI data models

Solutions **184**

How to download JSON from the internet and decode it into any Codable type

Chapter 1
Introduction

This stuff is hard

You might think this is a strange way to start a book, but: concurrency is a hard topic, and you might sometimes find yourself feeling a bit lost or a bit confused while you're learning about it.

I'm not saying that to put you off, or to make you feel somehow scared by all the topics we're going to cover. In fact, my goal here is quite the opposite: I want you to know that *everyone* finds this stuff hard. If you ever feel like you're struggling, it's okay – it's not you being stupid, it's just plain difficult stuff.

It's my goal to try to make this all as straightforward, understandable, and most importantly *applicable* as I can, so that anyone who is an intermediate to advanced Swift developer can apply these concepts to their projects immediately. If you're a beginner developer you're welcome to try following along, but honestly I would suggest you come back later – lots of theory isn't likely to stay in your head unless you have the chance to actually apply it.

Our brains are inherently not built to work concurrently, and as a result it's often hard to think about. So, one last time: this stuff is hard, and if you're finding it hard it's just a sign your brain is working correctly.

How to follow this guide

This guide is called Swift Concurrency by Example because it focuses on providing as many examples as possible. My goal is to stay laser-focused on real-world problems and real-world solutions, and to give clear, pragmatic advice on how to use specific techniques and approaches.

A lot of the time I've tried to make the chapters in this book answer specific questions such as "what are…" or "how to…", so that you can see exactly what problem is being solved and get straight there. That also means I've tried to get to the point as fast as possible and stay there so that you can get answers and build understanding quickly.

You can read this in a linear order if you want, or just dive in to a particular chapter that interests you – either one works.

Concurrency vs parallelism

When working through concurrency in Swift, there are two words we use a lot: *concurrency* and *parallelism*. We give them very specific meanings that might not quite match how you use them in English, so it's worth clarifying up front what they mean in this specific context.

Imagine a single-core CPU running a desktop operating system: the user can run lots of programs, use the internet, maybe play some music, and from the user's perspective all those things are happening at the same time. However, we know that isn't possible: they have a single-core CPU, which means it can literally only do *one* thing at a time.

What you're seeing here is called *concurrency*: our user's apps have been written in such a way that one CPU core can juggle them so they appear to be running at the same time. One starts, makes a tiny bit of progress on its work, then pauses so that another one can start, make a bit of progress on *its* work, then pause, and so on. Because the CPU flips between programs so quickly, they appear to be running all at the same time, when really they aren't.

As soon as you add a *second* CPU core to a computer, then things can run in *parallel*. This is when two or more programs run at the same instant: one on the first core, and another on the second. As you might imagine, this means work can happen up to twice as fast because two programs are able to run at the same time.

The really interesting stuff lies in our ability to split up work in a single program. Yes, having two CPU cores allows a computer to run two separate programs at the same time, but we can also split up our work into smaller parts called threads, and *those* can be run in parallel too.

This splitting up of functionality requires us to do work – Swift can't decide by itself to run some parts of our code in parallel with other parts, because that would introduce a lot of surprising bugs. Instead, we have to tell Swift ahead of time which parts of our code can be split up if needed, and also tell it what we should do when those tasks complete.

When you boil it right down, that's the topic of this whole book: teaching Swift how it can split up the work in our programs so it runs as efficiently as possible. And it *is* about efficiency, because some Apple devices have many CPU cores – if your app is running full

screen on that device and you're only ever using one of those cores, you're only getting a tiny fraction of the device's possible performance.

More importantly, you're also helping to make sure your app remains responsive the entire time. Imagine if your user interface froze up every time you were waiting for the response to a network request – it would be a pretty horrible experience, right?

There's a famous computer scientist called Rob Pike, and I think he explained the difference between concurrency and parallelism beautifully. Here's what he said:

"Concurrency is about dealing with many things at once, parallelism is about doing many things at once. Concurrency is a way to structure things so you can maybe use parallelism to do a better job."

Understanding threads and queues

Every program launches with at least one thread where its work takes place, called the *main thread*. Super simple command-line apps for macOS might only ever have that one thread, iOS apps will have many more to do all sorts of other jobs, but either way that initial thread – the one the app is first launched with – always exists for the lifetime of the app, and it's always called the main thread.

This is important, because all your user interface work must take place on that main thread. Not some work some of the time, but *all* work *all* the time – if you try to update your UI from any other thread in your program you might find nothing happens, you might find your app crashes, or pretty much anywhere in between.

This rule exists for all apps that run on iOS, macOS, tvOS, and watchOS, and even though it's simple you will – *will* – forget it at some point in the future. It's like an initiation rite, except it happens more often than I'd like to admit even after years of programming.

Although Swift lets us create threads whenever we want, this is uncommon because it creates a lot of complexity. Each thread you create needs to run *somewhere*, and if you accidentally end up creating 40 threads when you have only 4 CPU cores, the system will need to spend a lot of time just swapping them.

Swapping threads is known as a *context switch*, and it has a performance cost: the system must stash away all the data the thread was using and remember how far it had progressed in its work, before giving another thread the chance to run. When this happens a lot – when you create many more threads compared to the number of available CPU cores – the cost of context switching grows high, and so it has a suitably disastrous-sounding name: *thread explosion*.

And so, apart from that main thread of work that starts our whole program and manages the user interface, we normally prefer to think of our work in terms of *queues*.

Queues work like they do in real life, where you might line up to buy something at a grocery

store: you join the queue at the back, then move forward again and again until you're at the front, at which point you can check out. Some bigger stores might have lots of queues leading up to lots of checkouts, and small stores might just have one queue with one checkout. You might occasionally see stores trying to avoid the problem of one queue moving faster than another by having a single shared queue feed into multiple checkouts – there are all sorts of possible combinations.

Swift's queues work exactly the same way: we create a queue and add work to it, and the system will remove and execute work from there in the order it was added. Sometimes the queues are *serial*, which means they remove one piece of work from the front of the queue and complete it before going onto the next piece of work; and sometimes they are *concurrent*, which means they remove and execute multiple pieces of work at a time. Either way work will start in the order it was added to the queue unless we specifically say something has a high or low priority.

You might look at that and wonder why you even need serial queues – surely running one thing at a time is what we're trying to avoid? Well, no. In fact, there are lots of times when having the predictability of a serial queue is important.

As a simple example, your user might want to batch convert a collection of videos from one format to another. Video conversion is a really intense operation and is already highly optimized to take full advantage of multi-core CPUs, so it's more efficient to convert one video fully, then the next, then a third, and so on until you reach the end, rather than trying to convert four at once. This kind of work is perfect for a serial queue.

More importantly, sometimes serial queues are *required* to ensure our data is safe. For example, manipulating your user's data is exactly the kind of work you'd want to do on a serial queue, because it stops you from trying to read the data at the same time some other part of your program is trying to write new data.

Putting this all together, threads are the individual slices of a program that do pieces of work, whereas queues are like pipelines of execution where we can request that work be done at some point. Queues are easier to think about than threads because they focus on what matters:

we don't care how some code runs on the CPU, as long as it either runs in a particular order (serially) or not (concurrently). A lot of the time we don't even create the queue – we just use one of the built-in queues and let the system figure out how it should happen.

Tip: Sometimes Apple's frameworks will help you a little here. For example, even though using the **@State** property wrapper in a view will cause the body to be refreshed when the property is changed, this property wrapper is designed to be safe to call on any thread.

Main thread and main queue: what's the difference?

The main thread is the one that starts our program, and it's also the one where all our UI work must happen. However, there is also a main *queue*, and although sometimes we use the terms "main thread" and "main queue" interchangeably, they aren't quite the same thing.

It's a subtle distinction, but it can sometimes matter: although your main queue will always execute on the main thread (and is therefore where you'll be doing your UI work!), it's also possible that other queues might sometimes run on the main thread – the system is free to move things around in whatever way is most efficient.

So, if you're on the main queue then you're definitely on the main thread, but being on the main thread doesn't automatically mean you're on the main queue – a different queue could temporarily be running on the main thread.

At this point you're very likely staring at the screen wondering when this would ever be a problem, or perhaps even rereading what I said like it's a cryptic riddle. Trust me, if you ever hit a problem where the main thread vs main queue matters, you'll be glad you read this!

Where is Swift concurrency supported?

When it was originally announced, Swift concurrency required at least iOS 15, macOS 12, watchOS 8, tvOS 15, or on other platforms at least Swift 5.5.

However, if you're building your code using Xcode 13.2 or later you can back deploy to older versions of each of those operating systems: iOS 13, macOS 10.15, watchOS 6, and tvOS 13 are all supported. This offers the full range of Swift functionality, including actors, async/await, the task APIs, and more.

Important: This backwards compatibility applies only to Swift language features, not to any APIs built using those language features. This means you can write your own code to use async/await, actors, and so on, but you won't automatically gain access to the new Foundation APIs using those – things like the new **URLSession** APIs that use async/await still require iOS 15.

If you are keen to use the newer APIs in your project while also preserving backwards compatibility for older OS releases, your best bet is to add a runtime version check for iOS 15 then wrap the older APIs with continuations. This kind of hybrid solution allows you to keep using async/await elsewhere in your project – you get all the benefits of concurrency for the vast majority of your code, while keeping your backwards deployment shims neatly organized in one place so they can be removed in a year or two.

Dedication

This book is dedicated to the many people who helped read through earlier editions, reporting typos, suggesting improvements, and helping make this book as comprehensive as it can be – I'm really grateful they took the time to provide so much help and support!

In alphabetical order, they are: Abdul Ajetunmobi, Alejandro Martinez, Andrew Cowley, Chad Naeger, Chris Rivers, Darius Dunlap, Devanshi Modha, Diego Freniche Brito, Eric Schofield, Glenn Sequeira, Gustavo Diel, Hamish Allan, Horacio Chavez, Howard Katz, Isa Hashim, Jana Grill, Jason Bruder, Jeff Terry, Lars Christiansen, Laurent Brusa, Tor Rafsol Løseth, Mats Braa, Michael M. Mayer, Paul Williamson, Phil Martin, Piers Ebdon, Rick Gigger, Sarah Reichelt, Thomas Alvarez, Tom Comer, Vikash Anand, and Zack Apiratitham.

Chapter 2
Async/await

What is a synchronous function?

By default, all Swift functions are *synchronous*, but what does that mean?

A synchronous function is one that executes all its work in a simple, straight line on a single thread. Although the function itself can interact with other threads – it can request that work happens elsewhere, for example – the function itself always runs on a single thread.

This has a number of advantages, not least that synchronous functions are very easy to think about: when you call function A, it will carry on working until all its work is done, then return a value. If while working, function A calls function B, and perhaps functions C, D, and E as well, it doesn't matter – they all will execute on the same thread, and run one by one until the work completes.

Internally this is handled as a function stack: whenever one function calls another, the system creates what's called a *stack frame* to store all the data required for that new function – that's things like its local variables, for example. That new stack frame gets pushed on top of the previous one, like a stack of Lego bricks, and if that function calls a third function then another stack frame is created and added above the others. Eventually the functions finish, and their stack frame is removed and destroyed in a process we call *popping*, and control goes back to whichever function the code was called from.

Synchronous functions have an important downside, which is that they are *blocking*. If function A calls function B and needs to know what its return value is, then function A must wait for function B to finish before it can continue. This sounds self-evident, I know, but blocking code is problematic because now you've blocked a whole thread – it might be sitting around for a second or more waiting for some data to return.

You're probably thinking that waiting for a second is nothing at all, but in computing terms that's an ice age! Keep in mind that the Neural Engine in Apple's A14 Bionic CPU – just one part of the chip that powers the iPhone 12 – is capable of doing 11 *trillion* things per second, so if you block a thread for even a second that's 11,000,000,000,000 things you could have done but didn't.

One solution is to say “Well, if we’ve got a thread that’s currently blocked, we should just launch a new thread.” That’s definitely *a* solution, but it’s also a path towards thread explosion and we’ve already covered why that’s every bit as bad as it sounds.

So, although synchronous functions are easy to think about and work with, they aren’t very efficient for certain kinds of tasks. To make our code more flexible and more efficient, it’s possible to create *asynchronous* functions instead.

What is an asynchronous function?

Although Swift functions are synchronous by default, we can make them *asynchronous* by adding one keyword: **async**. Inside asynchronous functions, we can call other asynchronous functions using a second keyword: **await**. As a result, you'll often hear Swift developers talk about async/await as a way of coding.

So, this is a synchronous function that rolls a virtual dice and returns its result:

```
func randomD6() -> Int {
  Int.random(in: 1...6)
}

let result = randomD6()
print(result)
```

And this is an *asynchronous* or *async* function:

```
func randomD6() async -> Int {
  Int.random(in: 1...6)
}

let result = await randomD6()
print(result)
```

The only part of the code that changed is adding the **async** keyword before the return type and the **await** keyword before calling it, but those changes tell us three important things about async functions.

First, **async** is part of the function's *type*. The original, synchronous function returns an integer, which means we can't use it in a place that expects it to return a string. However, by

marking the code **async** we've now made it an *asynchronous* function that returns an integer, which means we can't use it in a place that expects a *synchronous* function that returns an integer.

This is what I mean when I say that the async nature of the function is part of its type: it affects the way we refer to the function everywhere else in our code. This is exactly how **throws** works – you can't use a throwing function in a place that expects a non-throwing function.

Second, notice that the work inside our function hasn't actually changed. The same work is being done as before: this function doesn't actually use the **await** keyword at all, and that's okay. You see, marking a function with **async** means it *might* do asynchronous work, not that it must. Again, the same is true of **throws** – some paths through a function might throw, but others might not.

A third key difference arises when we *call* **randomD6()**, because we need to do so asynchronously. Swift provides a few ways we can do this, but in our example we used **await**, which means "run this function asynchronously and wait for its result to come back before continuing."

So, what's the *actual difference* between synchronous and asynchronous functions? To answer that, I want to show you a real function that does some async work to fetch a file from a web server:

```
func fetchNews() async -> Data? {
  do {
    let url = URL(string: "https://hws.dev/news-1.json")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return data
  } catch {
    print("Failed to fetch data")
    return nil
  }
}
```

```
if let data = await fetchNews() {
  print("Downloaded \(data.count) bytes")
} else {
  print("Download failed.")
}
```

Later on we'll be digging in more to how that actually works, but for now what matters is that the **URLSession.shared.data(from:)** method we call is asynchronous – its job is to fetch some data from a web server, without causing the whole program to freeze up.

We've already seen that synchronous functions cause *blocking*, which leads to performance problems. Async functions do *not* block: when we call them with **await** we are marking a suspension point, which is a place where the function can suspend itself – literally stop running – so that other work can happen. At some point in the future the function's work completes, and Swift will wake it back up out of its "suspended animation"-like existence and it will carry on working.

This might sound simple on paper, but in practice it's very clever.

First, when an async function is suspended, all the async functions that called it are also suspended; they all wait quietly while the async work happens, then resume later on. This is really important: async functions have this special ability to be suspended that regular synchronous functions do not. It's for this reason that synchronous functions cannot call async functions directly – they don't know how to suspend themselves.

Second, a function can be suspended as many times as is needed, but it won't happen without you writing **await** there – functions won't suspend themselves by surprise.

Third, a function that is suspended does *not* block the thread it's running on, and instead it gives up that thread so that Swift can do other work instead. Note: Although we can tell Swift how important many tasks are, we don't get to decide exactly how the system schedules our work – it automatically takes care of all the threads working under the hood. This means if we

call async function A without waiting for its result, then a moment later call async function B, it's entirely possible B will start running before A does.

Fourth, when the function resumes, it might be running on the same thread as before, but it might not; Swift gets to choose, and you shouldn't make any assumptions here. This means by the time your function resumes all sorts of things might have changed in your program – a few milliseconds might have passed, or perhaps 20 seconds or more.

And finally, I know I'm repeating myself, but this matters: just because a function is async doesn't mean it *will* suspend – the **await** keyword only marks a *potential suspension point*. Most of the time Swift knows perfectly well that the function we're calling is async, so this **await** keyword is as much for us as it is for the compiler – it's a way of clearly marking which parts of the function might suspend, so you can know for sure which parts of the function run as one atomic chunk. ("Atomic" is a fancy word meaning "indivisible" – a chunk of work where all lines of code will execute without being interrupted by other code running.) This requirement for **await** is identical to the requirement for **try**, where we must mark each line of code that might throw errors.

So, async functions are like regular functions, except they have a superpower: if they need to, they can suspend themselves and all their callers, freeing up their thread to do other work.

How to create and call an async function

Using async functions in Swift is done in two steps: declaring the function itself as being **async**, then calling that function using **await**.

For example, if we were building an app that wanted to download a whole bunch of temperature readings from a weather station, calculate the average temperature, then upload those results, then we might want to make all three of those async:

1. Downloading data from the internet should always be done asynchronously, even a very small download can take a long time if the user has a bad cellphone connection.
2. Doing lots of mathematics might run quickly if the system is doing nothing else, but it might also take a long time if you have complex work and the system is busy doing something else.
3. Uploading data *to* the internet suffers from the same networking problems as downloading, and should always be done asynchronously.

To actually *use* those functions we would then need to write a fourth function that calls them one by one and prints the response. This function also needs to be async, because in theory the three functions it calls could suspend and so it might also need to be suspended.

I'm not going to do the actual networking code here, because we'll be looking a lot at networking later on. Instead, I want to focus on the structure of our functions so you can see how they fit together, so we'll be using mock data here – random numbers for the weather data, and the string "OK" for our server response.

Here's the code:

```
func fetchWeatherHistory() async -> [Double] {
  (1...100_000).map { _ in Double.random(in: -10...30) }
}
```

```
func calculateAverageTemperature(for records: [Double]) async
-> Double {
  let total = records.reduce(0, +)
  let average = total / Double(records.count)
  return average
}

func upload(result: Double) async -> String {
  "OK"
}

func processWeather() async {
  let records = await fetchWeatherHistory()
  let average = await calculateAverageTemperature(for: records)
  let response = await upload(result: average)
  print("Server response: \(response)")
}

await processWeather()
```

So, we have three simple async functions that fit together to form a sequence: download some data, process that data, then upload the result. That all gets stitched together into a cohesive flow using the **processWeather()** function, which can then be called from elsewhere.

That's not a lot of code, but it *is* a lot of functionality:

- Every one of those **await** calls is a potential suspension point, which is why we marked it explicitly. Like I said, one async function can suspend as many times as is needed.
- Swift will run each of the **await** calls in sequence, waiting for the previous one to complete. This is *not* going to run several things in parallel.
- Each time an **await** call finishes, its final value gets assigned to one of our constants – **records**, **average**, and **response**. Once created this is just regular data, no different from if we had created it synchronously.

- Because it calls async functions using **await**, it is *required* that **processWeather()** be itself an async function. If you remove that Swift will refuse to build your code.

When reading async functions like this one, it's good practice to look for the **await** calls because they are all places where unknown other amounts of work might take place before the next line of code executes.

Think of it a bit like this:

```
func processWeather() async {
  let records = await fetchWeatherHistory()
  // anything could happen here
  let average = await calculateAverageTemperature(for: records)
  // or here
  let response = await upload(result: average)
  // or here
  print("Server response: \(response)")
}
```

We're only using local variables inside this function, so they are safe. However, if you were relying on properties from a class, for example, they might have changed between each of those **await** lines.

Swift provides ways of protecting against this using a system known as *actors* – more on that much later.

How to call async throwing functions

Just like their synchronous counterparts, Swift's async functions can be throwing or non-throwing depending on how you want them to behave. However, there is a twist: although we mark the function as being **async throws**, we *call* the function using **try await** – the keyword order is flipped.

To demonstrate this in action, here's a **fetchFavorites()** method that attempts to download some JSON from my server, decode it into an array of integers, and return the result:

```
func fetchFavorites() async throws -> [Int] {
  let url = URL(string: "https://hws.dev/user-favorites.json")!
  let (data, _) = try await URLSession.shared.data(from: url)
  return try JSONDecoder().decode([Int].self, from: data)
}

if let favorites = try? await fetchFavorites() {
  print("Fetched \(favorites.count) favorites.")
} else {
  print("Failed to fetch favorites.")
}
```

Both fetching data and decoding it are throwing functions, so we need to use **try** in both those places. Those errors aren't being handled in the function, so we need to mark **fetchFavorites()** as also being throwing so Swift can let any errors bubble up to whatever called it.

Notice that the function is marked **async throws** but the function *calls* are marked **try await** – like I said, the keyword order gets reversed. So, it's "asynchronous, throwing" in the function definition, but "throwing, asynchronous" at the call site. Think of it as unwinding a stack.

Not only does **try await** read more easily than **await try**, but it's also more reflective of what's

actually happening when our code executes: we're waiting for some work to complete, and when it does complete we'll check whether it ended up throwing an error or not.

Of course, the other possibility is that they could have **try await** at the call site and **throws async** on the function definition, and here's what the Swift team says about that:

"This order restriction is arbitrary, but it's not harmful, and it eliminates the potential for stylistic debates."

Personally, if I saw **throws async** I'd be more likely to write it as "this thing throws an async error" or similar, but as the quote above says ultimately the order is just arbitrary.

What calls the first async function?

You can only call async functions from other async functions, because they might need to suspend themselves and everything that is waiting for them. This leads to a bit of a chicken and egg problem: if only async functions can call other async functions, what starts it all – what calls the very first async function?

Well, there are three main approaches you'll find yourself using.

First, in simple command-line programs using the **@main** attribute, you can declare your **main()** method to be async. This means your program will immediately launch into an async function, so you can call other async functions freely.

Here's how that looks in code:

```
func processWeather() async {
  // Do async work here
}

@main
struct MainApp {
  static func main() async {
    await processWeather()
  }
}
```

Second, in apps built with something like SwiftUI the framework itself has various places that can trigger an async function. For example, the **refreshable()** and **task()** modifiers can both call async functions freely.

Using the **task()** modifier we could write a simple "View Source" app that fetches the content of a website when our view appears:

```
struct ContentView: View {
  @State private var sourceCode = ""

  var body: some View {
    ScrollView {
      Text(sourceCode)
    }
    .task {
      await fetchSource()
    }
  }

  func fetchSource() async {
    do {
      let url = URL(string: "https://apple.com")!

      let (data, _) = try await URLSession.shared.data(from:
url)
      sourceCode = String(decoding: data, as:
UTF8.self).trimmingCharacters(in: .whitespacesAndNewlines)
    } catch {
      sourceCode = "Failed to fetch apple.com"
    }
  }
}
```

Tip: Using **task()** will almost certainly run our code away from the main thread, but the **@State** property wrapper has specifically been written to allow us to modify its value on any thread.

The third option is that Swift provides a dedicated **Task** API that lets us call async functions from a synchronous function. Now, you might think "wait a minute – how can a synchronous

function call an asynchronous function?" Well, it *can't* – at least not *directly*. Remember, async functions might need to suspend themselves in the future, and synchronous functions don't know how to do that.

When you use something like **Task** you're asking Swift to run some async code. If you don't care about the result you have nothing to wait *for* – the task will start running immediately while your own function continues, and it will always run to completion even if you don't store the active task somewhere. This means you're not awaiting the result of the task, so you won't run the risk of being suspended. Of course, when you actually want to *use* any returned value from your task, that's when **await** is required.

We'll be looking at Swift's **Task** API in detail later on, but for now we could quickly upgrade our little website source code viewer to work with any URL. This time we're going to trigger the network fetch using a button press, which is *not* asynchronous by default, so we're going to wrap our work in a **Task**. This is possible because we don't need to wait for the task to complete – it will always run to completion as soon as it is made, and will take care of updating the UI for us.

Here's how that looks:

```
struct ContentView: View {
  @State private var site = "https://"
  @State private var sourceCode = ""

  var body: some View {
    VStack {
      HStack {
        TextField("Website address", text: $site)
          .textFieldStyle(.roundedBorder)
        Button("Go") {
          Task {
            await fetchSource()
          }
```

```
        }
      }
      .padding()

      ScrollView {
        Text(sourceCode)
      }
    }
  }

  func fetchSource() async {
    do {
      let url = URL(string: site)!
      let (data, _) = try await URLSession.shared.data(from:
url)
      sourceCode = String(decoding: data, as:
UTF8.self).trimmingCharacters(in: .whitespacesAndNewlines)
    } catch {
      sourceCode = "Failed to fetch \(site)"
    }
  }
}
```

What's the performance cost of calling an async function?

Whenever we use **await** to call an async function, we mark a potential suspension point in our code – we're acknowledging that it's entirely possible our function will be suspended, along with all its callers, while the work completes. In terms of performance, this is *not* free: synchronous and asynchronous functions use a different calling convention internally, with the asynchronous variant being slightly less efficient.

The important thing to understand here is that Swift cannot tell at compile time whether an **await** call will suspend or not, and so the same (slightly) more expensive calling convention is used regardless of what actually takes place at runtime.

However, what happens at runtime depends on whether the call suspends or not:

- If a suspension happens, then Swift will pause the function and all its callers, which has a small performance cost. These will then be resumed later, and ultimately whatever performance cost you pay for the suspension is like a rounding error compared to the performance gain provided by async/await even existing.
- If a suspension does *not* happen, no pause will take place and your function will continue to run with the same efficiency and timings as a synchronous function.

That last part carries an important side effect: using **await** will not cause your code to wait for one runloop to go by before continuing.

It's a common joke that many coding problems can be fixed by waiting for one runloop tick to pass before trying again – usually seen as **DispatchQueue.main.async { … }** in Swift projects – but that will *not* happen when using **await**, because the code will execute immediately.

So, if your code doesn't actually suspend, the only cost to calling an asynchronous function is the slightly more expensive calling convention, and if your code *does* suspend then any cost is more or less irrelevant because you've gained so much extra performance thanks to the

suspension happening in the first place.

How to create and use async properties

Just as Swift's functions can be asynchronous, computed properties can also be asynchronous: attempting to access them must also use **await** or similar, and may also need **throws** if errors can be thrown when computing the property. This is what allows things like the **value** property of **Task** to work – it's a simple property, but we must access it using **await** because it might not have completed yet.

Important: This is only possible on read-only computed properties – attempting to provide a setter will cause a compile error.

To demonstrate this, we could create a **RemoteFile** struct that stores a URL and a type that conforms to **Decodable**. This struct won't actually fetch the URL when the struct is created, but will instead dynamically fetch the content's of the URL every time the property is requested so that we can update our UI dynamically.

Tip: If you use **URLSession.shared** to fetch your data it will automatically be cached, so we're going to create a custom URL session that always ignores local and remote caches to make sure our remote file is always fetched.

Here's the code:

```
// First, a URLSession instance that never uses caches
extension URLSession {
  static let noCacheSession: URLSession = {
    let config = URLSessionConfiguration.default
    config.requestCachePolicy
= .reloadIgnoringLocalAndRemoteCacheData
    return URLSession(configuration: config)
  }()
}
```

```
// Now our struct that will fetch and decode a URL every
// time we read its `contents` property
struct RemoteFile<T: Decodable> {
  let url: URL
  let type: T.Type

  var contents: T {
    get async throws {
      let (data, _) = try await
URLSession.noCacheSession.data(from: url)
      return try JSONDecoder().decode(T.self, from: data)
    }
  }
}
```

So, we're fetching the URL's contents every time **contents** is access, as opposed to storing the URL's contents when a **RemoteFile** instance is created. As a result, the property is marked both **async** and **throws** so that callers must use **await** or similar when accessing it.

To try that out with some real SwiftUI code, we could write a view that fetches messages. We don't ever want stale data, so we're going to point our **RemoteFile** struct at a particular URL and tell it to expect an array of **Message** objects to come back, then let it take care of fetching and decoding those while also bypassing the **URLSession** cache:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let user: String
  let text: String
}

struct ContentView: View {
  let source = RemoteFile(url: URL(string: "https://hws.dev/
inbox.json")!, type: [Message].self)
```

```
  @State private var messages = [Message]()

  var body: some View {
    NavigationView {
      List(messages) { message in
        VStack(alignment: .leading) {
          Text(message.user)
            .font(.headline)
          Text(message.text)
        }
      }
      .navigationTitle("Inbox")
      .toolbar {
        Button(action: refresh) {
          Label("Refresh", systemImage: "arrow.clockwise")
        }
      }
      .onAppear(perform: refresh)
    }
  }

  func refresh() {
    Task {
      do {
        // Access the property asynchronously
        messages = try await source.contents
      } catch {
        print("Message update failed.")
      }
    }
  }
}
```

That call to **source.contents** is where the real action happens – it's a property, yes, but it must also be accessed asynchronously so that it can do its work of fetching and decoding without blocking the UI.

How to call an async function using async let

Sometimes you want to run several async operations at the same time then wait for their results to come back, and the easiest way to do that is with **async let**. This lets you start several async functions, all of which begin running immediately – it's much more efficient than running them sequentially.

A common example of where this is useful is when you have to make two or more network requests, none of which relate to each other. That is, if you need to get Thing X and Thing Y from a server, but you don't need to wait for X to return before you start fetching Y.

To demonstrate this, we could define a couple of structs to store data – one to store a user's account data, and one to store all the messages in their inbox:

```
struct User: Decodable {
  let id: UUID
  let name: String
  let age: Int
}

struct Message: Decodable, Identifiable {
  let id: Int
  let from: String
  let message: String
}
```

These two things can be fetched independently of each other, so rather than fetching the user's account details *then* fetching their message inbox we want to get them both together.

In this instance, rather than using a regular **await** call a better choice is **async let**, like this:

```
func loadData() async {
```

```
    async let (userData, _) = URLSession.shared.data(from:
URL(string: "https://hws.dev/user-24601.json")!)

    async let (messageData, _) = URLSession.shared.data(from:
URL(string: "https://hws.dev/user-messages.json")!)

    // more code to come
}
```

That's only a small amount of code, but there are three things I want to highlight in there:

- Even though the **data(from:)** method is async, we don't need to use **await** before it because that's implied by **async let**.
- The **data(from:)** method is also throwing, but we don't need to use **try** to execute it because that gets pushed back to when we actually want to read its return value.
- Both those network calls start immediately, but might complete in any order.

Okay, so now we have two network requests in flight. The next step is to wait for them to complete, decode their returned data into structs, and use that somehow.

There are two things you need to remember:

- Both our **data(from:)** calls might throw, so when we read those values we need to use **try**.
- Both our **data(from:)** calls are running concurrently while our main **loadData()** function continues to execute, so we need to read their values using **await** in case they aren't ready yet.

So, we could complete our function by using **try await** for each of our network requests in turn, then print out the result:

```
func loadData() async {
    async let (userData, _) = URLSession.shared.data(from:
URL(string: "https://hws.dev/user-24601.json")!)
```

```
    async let (messageData, _) = URLSession.shared.data(from:
URL(string: "https://hws.dev/user-messages.json")!)

    do {
        let decoder = JSONDecoder()
        let user = try await decoder.decode(User.self, from:
userData)
        let messages = try await decoder.decode([Message].self,
from: messageData)
        print("User \(user.name) has \(messages.count)
message(s).")
    } catch {
        print("Sorry, there was a network problem.")
    }
}

await loadData()
```

The Swift compiler will automatically track which **async let** constants could throw errors and will enforce the use of **try** when reading their value. It doesn't matter which form of **try** you use, so you can use **try**, **try?** or **try!** as appropriate.

Tip: If you never try to read the value of a throwing **async let** call – i.e., if you've started the work but don't care what it returns – then you don't need to use **try** at all, which in turn means the function running the **async let** code might not need to handle errors at all.

Although both our network requests are happening at the same time, we still need to wait for them to complete in some sort of order. So, if you wanted to update your user interface as soon as either **user** or **messages** arrived back **async let** isn't going to help by itself – you should look at the dedicated **Task** type instead.

One complexity with **async let** is that it captures any values it uses, which means you might

accidentally try to write code that isn't safe. Swift helps here by taking some steps to enforce that you aren't trying to modify data unsafely.

As an example, if we wanted to fetch the favorites for a user, we might have a function such as this one:

```
func fetchFavorites(for user: User) async -> [Int] {
  print("Fetching favorites for \(user.name)…")

  do {
    async let (favorites, _) = URLSession.shared.data(from:
URL(string: "https://hws.dev/user-favorites.json")!)
    return try await JSONDecoder().decode([Int].self, from:
favorites)
  } catch {
    return []
  }
}

let user = User(id: UUID(), name: "Taylor Swift", age: 26)
async let favorites = fetchFavorites(for: user)
await print("Found \(favorites.count) favorites.")
```

That function accepts a **User** parameter so it can print a status message. But what happens if our **User** was created as a variable and captured by **async let**? You can see this for yourself if you change the user:

```
var user = User(id: UUID(), name: "Taylor Swift", age: 26)
```

Even though it's a struct, the **user** variable will be *captured* rather than *copied* and so Swift will throw up the build error "Reference to captured var 'user' in concurrently-executing code."

To fix this we need to make it clear the struct cannot change by surprise, even when captured,

by making it a constant rather than a variable.

What's the difference between await and async let?

Swift lets us perform async operations using both **await** and **async let**, but although they both run some async code they don't quite run the same: **await** immediately waits for the work to complete so we can read its result, whereas **async let** does not.

Which you choose depends on the kind of work you're trying to do. For example, if you want to make two network requests where one relates to the other, you might have code like this:

```
let first = await requestFirstData()
let second = await requestSecondData(using: first)
```

There the call to **requestSecondData()** cannot start until the call to **requestFirstData()** has completed and returned its value – it just doesn't make sense for those two to run simultaneously.

On the other hand, if you're making several completely different requests – perhaps you want to download the latest news, the weather forecast, and check whether an app update was available – then those things do *not* rely on each other to complete and would be great candidates for **async let**:

```
func getAppData() -> ([News], [Weather], Bool) {
  async let news = getNews()
  async let weather = getWeather()
  async let hasUpdate = getAppUpdateAvailable()
  return await (news, weather, hasUpdate)
}
```

So, use **await** when it's important you have a value before continuing, and **async let** when your work can continue without the value for the time being – you can always use **await** later on when it's actually needed.

Why can't we call async functions using async var?

Swift's **async let** syntax provides short, helpful syntax for running lots of work concurrently, allowing us to wait for them all later on. However, it only works as **async let** – it's not possible to use **async var**.

If you think about it, this restriction makes sense. Consider pseudocode like this:

```
func fetchUsername() async -> String {
  // complex networking here
  "Taylor Swift"
}

async var username = fetchUsername()
username = "Justin Bieber"
print("Username is \(username)")
```

That attempts to create a variable asynchronously, then writes to it directly. Have we cancelled the async work? If not, when the async work completes will it overwrite our new value? Do we still need to use **await** when reading the value even after we've explicitly set it?

This kind of code would create all sorts of confusion, so it's just not allowed – **async let** is our only option.

How to use continuations to convert completion handlers into async functions

Older Swift code uses completion handlers for notifying us when some work has completed, and sooner or later you're going to have to use it from an **async** function – either because you're using a library someone else created, or because it's one of your own functions but updating it to async would take a lot of work.

Swift uses *continuations* to solve this problem, allowing us to create a bridge between older functions with completion handlers and newer async code.

To demonstrate this problem, here's some code that attempts to fetch some JSON from a web server, decode it into an array of **Message** structs, then send it back using a completion handler:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let from: String
  let message: String
}

func fetchMessages(completion: @escaping ([Message]) -> Void) {
  let url = URL(string: "https://hws.dev/user-messages.json")!

  URLSession.shared.dataTask(with: url) { data, response, error
in
    if let data = data {
      if let messages = try?
JSONDecoder().decode([Message].self, from: data) {
        completion(messages)
```

```
            return
        }
    }

    completion([])
  }.resume()
}
```

Although the **dataTask(with:)** method does run our code on its own thread, this is *not* an async function in the sense of Swift's async/await feature, which means it's going to be messy to integrate into other code that *does* use modern async Swift.

To fix this, Swift provides us with *continuations*, which are special objects we pass into the completion handlers as captured values. Once the completion handler fires, we can either return the finished value, throw an error, or send back a **Result** that can be handled elsewhere.

In the case of **fetchMessages()**, we want to write a new async function that calls the original, and in its completion handler we'll return whatever value was sent back:

```
func fetchMessages() async -> [Message] {
  await withCheckedContinuation { continuation in
    fetchMessages { messages in
      continuation.resume(returning: messages)
    }
  }
}

let messages = await fetchMessages()
print("Downloaded \(messages.count) messages.")
```

As you can see, starting a continuation is done using the **withCheckedContinuation()** function, which passes into itself the continuation we need to work with. It's called a "checked" continuation because Swift checks that we're using the continuation correctly,

which means abiding by one very simple, very important rule:

Your continuation must be resumed exactly once. Not zero times, and not twice or more times – exactly once.

If you call the checked continuation twice or more, Swift will cause your program to halt – it will just crash. I realize this sounds bad, but when the alternative is to have some bizarre, unpredictable behavior, crashing doesn't sound so bad.

On the other hand, if you fail to resume the continuation at all, Swift will print out a large warning in your debug log similar to this: "SWIFT TASK CONTINUATION MISUSE: fetchMessages() leaked its continuation!" This is because you're leaving the task suspended, causing any resources it's using to be held indefinitely.

You might think these are easy mistakes to avoid, but in practice they can occur in all sorts of places if you aren't careful.

For example, in our original **fetchMessages()** method we used this:

```
if let data = data {
  if let messages = try? JSONDecoder().decode([Message].self,
from: data) {
    completion(messages)
    return
  }
}

completion([])
```

That checks for data coming back, and checks that it can be decoded correctly, before completing and returning, but if either of those two checks fail then the completion handler is called with an empty array – no matter what happens, the completion handler gets called.

But what if we had written something different? See if you can spot the problem with this

alternative:

```
if let data = data {
  if let messages = try? JSONDecoder().decode([Message].self,
from: data) {
    completion(messages)
  }
} else {
  completion([])
}
```

That attempts to decode the JSON into a **Message** array and send back the result using the completion handler, or send back an empty array if nothing came back from the server. However, it has a mistake that will cause problems with continuations: if some valid data comes back but *can't* be decoded into an array of messages, the completion handler will never be called and our continuation will be leaked.

These two code samples are fairly similar, which shows how important it is to be careful with your continuations. However, if you have checked your code carefully and you're sure it is correct, you can if you want replace the **withCheckedContinuation()** function with a call to **withUnsafeContinuation()**, which works exactly the same way but doesn't add the runtime cost of checking you've used the continuation correctly.

I say you can do this *if you want*, but I'm dubious about the benefit. It's easy to say "I know my code is safe, go for it!" but I'd be wary about moving across to unsafe code unless you had profiled your code using Instruments and were quite sure Swift's extra checks were causing a performance problem.

How to create continuations that can throw errors

Swift provides **withCheckedContinuation()** and **withUnsafeContinuation()** to let us create continuations that can't throw errors, but if the API you're using *can* throw errors you should use their throwing equivalents: **withCheckedThrowingContinuation()** and **withUnsafeThrowingContinuation()**.

Both of these replacement functions work identically to their non-throwing counterparts, except now you need to catch any errors thrown inside the continuation.

To demonstrate this, here's the same **fetchMessages()** function we used previously, built without async/await in mind:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let from: String
  let message: String
}

func fetchMessages(completion: @escaping ([Message]) -> Void) {
  let url = URL(string: "https://hws.dev/user-messages.json")!

  URLSession.shared.dataTask(with: url) { data, response, error
in
    if let data = data {
      if let messages = try?
JSONDecoder().decode([Message].self, from: data) {
        completion(messages)
        return
      }
    }
```

```
    completion([])
  }.resume()
}
```

If we wanted to wrap that using a continuation, we might decide that having zero messages is an error we should throw rather than just sending back an empty array. That thrown error would then need to be handled outside the continuation somehow.

So, first we'd define the errors we want to throw, then we'd write a newer async version of **fetchMessages()** using **withCheckedThrowingContinuation()**, and handling the "no messages" error using whatever code we wanted:

```
// An example error we can throw
enum FetchError: Error {
  case noMessages
}

func fetchMessages() async -> [Message] {
  do {
    return try await withCheckedThrowingContinuation
{ continuation in
      fetchMessages { messages in
        if messages.isEmpty {
          continuation.resume(throwing: FetchError.noMessages)
        } else {
          continuation.resume(returning: messages)
        }
      }
    }
  } catch {
    return [
      Message(id: 1, from: "Tom", message: "Welcome to MySpace!
```

```
I'm your new friend.")
    ]
  }
}

let messages = await fetchMessages()
print("Downloaded \(messages.count) messages.")
```

As you can see, that detects a lack of messages and sends back a welcome message instead, but you could also let the error propagate upwards by removing **do**/**catch** and making the new **fetchMessages()** function throwing.

Tip: Using **withUnsafeThrowingContinuation()** comes with all the same warnings as using **withUnsafeContinuation()** – you should only switch over to it if it's causing a performance problem.

How to store continuations to be resumed later

Many of Apple's frameworks report back success or failure using multiple different delegate callback methods rather than completion handlers, which means a simple continuation won't work.

As a simple example, if you were implementing **WKNavigationDelegate** to handle navigating around a **WKWebView** you would implement methods like this:

```
func webView(_ webView: WKWebView, didFinish navigation:
WKNavigation!) {
  // our work succeeded
}

func webView(WKWebView, didFail: WKNavigation!, withError:
Error) {
  // our work failed
}
```

So, rather than receiving the result of our work through a single completion closure, we instead get the result in two different places. In this situation we need to do a little more work to create async functions using continuations, because we need to be able to resume the continuation in either method.

To solve this problem you need to know that continuations are just structs with a specific generic type. For example, a checked continuation that succeeds with a string and never throws an error has the type **CheckedContinuation<String, Never>**, and an unchecked continuation that returns an integer array and can throw errors has the type **UnsafeContinuation<[Int], Error>**.

All this is important because to solve our delegate callback problem we need to stash away a

continuation in one method – when we trigger some functionality – then resume it from different methods based on whether our code succeeds or fails.

I want to demonstrate this using real code, so we're going to create an **ObservableObject** to wrap Core Location, making it easier to request the user's location.

First, add these imports to your code so we can read their location, and also use SwiftUI's **LocationButton** to get standardized UI:

```
import CoreLocation
import CoreLocationUI
```

Second, we're going to create a small part of a **LocationManager** class that has two properties: one for storing a continuation to track whether we have their location coordinate or an error, and one to track an instance of **CLLocationManager** that does the work of finding the user. This also needs a small initializer so the **CLLocationManager** knows to report location updates to us.

Add this class now:

```
@MainActor
class LocationManager: NSObject, ObservableObject,
CLLocationManagerDelegate {
  var locationContinuation:
CheckedContinuation<CLLocationCoordinate2D?, Error>?
  let manager = CLLocationManager()

  override init() {
    super.init()
    manager.delegate = self
  }

  // More code to come
}
```

Tip: Because that observable object is used with SwiftUI, I've marked it with the **@MainActor** attribute to avoid updating the user interface on a background thread. The **@MainActor** attribute is covered much later in this book.

Third, we need to add an async function that requests the user's location. This needs to be wrapped inside a **withCheckedThrowingContinuation()** call, so that Swift creates a continuation we can stash away and use later.

Add this method to the class now:

```
func requestLocation() async throws -> CLLocationCoordinate2D?
{
  try await withCheckedThrowingContinuation { continuation in
    locationContinuation = continuation
    manager.requestLocation()
  }
}
```

And finally we need to implement the two methods that might be called after we request the user's location: **didUpdateLocations** will be called if their location was received, and **didFailWithError** otherwise. Both of these need to resume our continuation, with the former sending back the first location coordinate we were given, and the latter throwing whatever error occurred:

Add these last two methods to the class now:

```
func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
  locationContinuation?.resume(returning:
locations.first?.coordinate)
}

func locationManager(_ manager: CLLocationManager,
```

```
didFailWithError error: Error) {
  locationContinuation?.resume(throwing: error)
}
```

So, by storing our continuation as a property we're able to resume it in two different places – once where things go to plan, and once where things go wrong for whatever reason. Either way, no matter what happens our continuation resumes exactly once.

At this point our continuation wrapper is complete, so we can use it inside a SwiftUI view. If we put everything together, here's the end result:

```
@MainActor
class LocationManager: NSObject, ObservableObject,
CLLocationManagerDelegate {
  var locationContinuation:
CheckedContinuation<CLLocationCoordinate2D?, Error>?
  let manager = CLLocationManager()

  override init() {
    super.init()
    manager.delegate = self
  }

  func requestLocation() async throws ->
CLLocationCoordinate2D? {
    try await withCheckedThrowingContinuation { continuation in
      locationContinuation = continuation
      manager.requestLocation()
    }
  }

  func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
```

```
    locationContinuation?.resume(returning:
locations.first?.coordinate)
  }

  func locationManager(_ manager: CLLocationManager,
didFailWithError error: Error) {
    locationContinuation?.resume(throwing: error)
  }
}

struct ContentView: View {
  @StateObject private var locationManager = LocationManager()

  var body: some View {
    LocationButton {
      Task {
        if let location = try? await
locationManager.requestLocation() {
          print("Location: \(location)")
        } else {
          print("Location unknown.")
        }
      }
    }
    .frame(height: 44)
    .foregroundColor(.white)
    .clipShape(Capsule())
    .padding()
  }
}
```

How to fix the error “async call in a function that does not support concurrency”

This error occurs when you’ve tried to call an async function from a synchronous function, which is not allowed in Swift – asynchronous functions must be able to suspend themselves and their callers, and synchronous functions simply don’t know how to do that.

If your asynchronous work needs to be waited for, you don’t have much of a choice but to mark your current code as also being **async** so that you can use **await** as normal. However, sometimes this can result in a bit of an “async infection” – you mark one function as being async, which means its caller needs to be async too, as does *its* caller, and so on, until you’ve turned one error into 50.

In this situation, you can create a dedicated **Task** to solve the problem. We’ll be covering this API in more detail later on, but here’s how it would look in your code:

```
func doAsyncWork() async {
  print("Doing async work")
}

func doRegularWork() {
  Task {
    await doAsyncWork()
  }
}

doRegularWork()
```

Tasks like this one are created and run immediately. We aren’t waiting for the task to complete, so we shouldn’t use **await** when creating it.

Chapter 3

Sequences and streams

What's the difference between Sequence, AsyncSequence, and AsyncStream?

Swift provides several ways of receiving a potentially endless flow of data, allowing us to read values one by one, or loop over them using **for**, **while**, or similar.

The simplest is the **Sequence** protocol, which continually returns values until the sequence is terminated by returning **nil**. Lots of things conform to **Sequence**, including arrays, strings, ranges, **Data**, and more. Through protocol extensions **Sequence** also gives us access to a variety of methods, including **contains()**, **filter()**, **map()**, and others.

The **AsyncSequence** protocol is almost identical to **Sequence**, with the important exception that each element in the sequence is returned asynchronously. I realize that sounds obvious, but it actually has two major impacts on the way they work.

First, reading a value from the async sequence must use **await** so the sequence can suspend itself while reading its next value. This might be performing some complex work, for example, or perhaps fetching data from a server.

Second, more advanced async sequences known as *async streams* might generate values faster than you can read them, in which case you can either discard the extra values or buffer them to be read later on.

So, in the first case think of it like your code wanting values faster than the async sequence can make them, whereas in the second case it's more like the async sequence generating data faster than than your code can read them.

Otherwise, **Sequence** and **AsyncSequence** have lots in common: the code to create a custom one yourself is almost the same, both can throw errors if you want, both get access to common functionality such as **map()**, **filter()**, **contains()**, and **reduce()**, and you can also use **break** or **continue** to exit loops over either of them.

How to loop over an AsyncSequence using for await

You can loop over an **AsyncSequence** using Swift's regular loop types, **for**, **while**, and **repeat**, but whenever you read a value from the async sequence you must use either **await** or **try await** depending on whether it can throw errors or not.

As an example, **URL** has a built-in **lines** property that generates an async sequence of all the lines directly from a URL. This does a *lot* of work internally: making the network request, fetching part of the data, converting it into a string, sending it back for us to use, then repeating fetch, convert, and send again and again until the server stops sending back data.

All that functionality boils down to just a handful of lines of code:

```
func fetchUsers() async throws {
  let url = URL(string: "https://hws.dev/users.csv")!

  for try await line in url.lines {
    print("Received user: \(line)")
  }
}

try? await fetchUsers()
```

Notice how we must use **try** along with **await**, because fetching data from the network might throw errors.

Using **lines** returns a specialized type called **AsyncLineSequence**, which returns individual lines from the download as strings. Because our source happens to be a comma-separated values file (CSV), we can write code to read the values from each line easily enough:

```
func printUsers() async throws {
  let url = URL(string: "https://hws.dev/users.csv")!
```

```
    for try await line in url.lines {
        let parts = line.split(separator: ",")
        guard parts.count == 4 else { continue }

        guard let id = Int(parts[0]) else { continue }
        let firstName = parts[1]
        let lastName = parts[2]
        let country = parts[3]

        print("Found user #\(id): \(firstName) \(lastName) from \
(country)")
    }
}

try? await printUsers()
```

Just like a regular sequence, using an async sequence in this way effectively generates an iterator then calls **next()** on it repeatedly until it returns **nil**, at which point the loop finishes.

If you want more control over how the sequence is read, you can of course create your own iterator then call **next()** whenever you want and as often as you want. Again, it will send back **nil** when the sequence is empty, at which point you should stop calling it.

For example, we could read the first user from our CSV and treat them specially, then read the next four users and do something specific for them, then finally reduce all the remainder down into an array of other users:

```
func printUsers() async throws {
    let url = URL(string: "https://hws.dev/users.csv")!

    var iterator = url.lines.makeAsyncIterator()
```

```
    if let line = try await iterator.next() {
        print("The first user is \(line)")
    }

    for i in 2...5 {
        if let line = try await iterator.next() {
            print("User #\(i): \(line)")
        }
    }

    var remainingResults = [String]()

    while let result = try await iterator.next() {
        remainingResults.append(result)
    }

    print("There were \(remainingResults.count) other users.")
}

try? await printUsers()
```

How to manipulate an AsyncSequence using map(), filter(), and more

AsyncSequence has implementations of many of the same methods that come with **Sequence**, but *how* they operate varies: some return a single value that fulfills your request, such as requesting the first value from the async sequence, and others return a new kind of async sequence, such as filtering values as they arrive.

This distinction in turn affects how they are called: returning a single value requires you to await at the call site, whereas returning a new async sequence requires you to await later on when you're reading values from the new sequence.

To demonstrate this difference, let's try out a few common operations, starting with a call to **map()**. Mapping an async sequence creates a new async sequence with the type **AsyncMapSequence**, which stores both your original async sequence and also the transformation function you want to use. So, instead of waiting for all elements to be returned and transforming them at once, you've effectively put the transformation into a chain of work: rather than fetching an item and sending it back, the sequence now fetches an item, transforms it, *then* sends it back.

So, we could map over the lines from a **URL** to make each line uppercase, like this:

```
func shoutQuotes() async throws {
  let url = URL(string: "https://hws.dev/quotes.txt")!
  let uppercaseLines = url.lines.map(\.localizedUppercase)

  for try await line in uppercaseLines {
    print(line)
  }
}
```

```
try? await shoutQuotes()
```

This also works for converting between types using **map()**, like this:

```
struct Quote {
  let text: String
}

func printQuotes() async throws {
  let url = URL(string: "https://hws.dev/quotes.txt")!

  let quotes = url.lines.map(Quote.init)

  for try await quote in quotes {
    print(quote.text)
  }
}

try? await printQuotes()
```

Alternatively, we could use **filter()** to check every line with a predicate, and process only those that pass. Using our quotes, we could print only anonymous quotes like this:

```
func printAnonymousQuotes() async throws {
  let url = URL(string: "https://hws.dev/quotes.txt")!
  let anonymousQuotes = url.lines.filter
{ $0.contains("Anonymous") }

  for try await line in anonymousQuotes {
    print(line)
  }
}
```

```
try? await printAnonymousQuotes()
```

Or we could use **prefix()** to read just the first five values from an async sequence:

```
func printTopQuotes() async throws {
  let url = URL(string: "https://hws.dev/quotes.txt")!
  let topQuotes = url.lines.prefix(5)

  for try await line in topQuotes {
    print(line)
  }
}

try? await printTopQuotes()
```

And of course you can also combine these together in varying ways depending on what result you want. For example, this will filter for anonymous quotes, pick out the first five, then make them uppercase:

```
func printQuotes() async throws {
  let url = URL(string: "https://hws.dev/quotes.txt")!

  let anonymousQuotes = url.lines.filter
{ $0.contains("Anonymous") }
  let topAnonymousQuotes = anonymousQuotes.prefix(5)
  let shoutingTopAnonymousQuotes =
topAnonymousQuotes.map(\.localizedUppercase)

  for try await line in shoutingTopAnonymousQuotes {
    print(line)
  }
}
```

```
try? await printQuotes()
```

Just like using a regular **Sequence**, the order you apply these transformations matters – putting **prefix()** before **filter()** will pick out the first five quotes *then* select only the ones that are anonymous, which might produce fewer results.

Each of these transformation methods returns a new type specific to what the method does, so calling **map()** returns an **AsyncMapSequence**, calling **filter()** returns an **AsyncFilterSequence**, and calling **prefix()** returns an **AsyncPrefixSequence**.

When you stack multiple transformations together – for example, a filter, then a prefix, then a map, as in our previous example – this will inevitably produce a fairly complex return type, so if you intend to send back one of the complex async sequences you should consider an opaque return type like this:

```
func getQuotes() async -> some AsyncSequence {
  let url = URL(string: "https://hws.dev/quotes.txt")!
  let anonymousQuotes = url.lines.filter
{ $0.contains("Anonymous") }
  let topAnonymousQuotes = anonymousQuotes.prefix(5)
  let shoutingTopAnonymousQuotes =
topAnonymousQuotes.map(\.localizedUppercase)
  return shoutingTopAnonymousQuotes
}

let result = await getQuotes()

do {
  for try await quote in result {
    print(quote)
  }
} catch {
```

```
    print("Error fetching quotes")
}
```

All the transformations so far have created new async sequences and so we haven't needed to use them with **await**, but many also produce a single value. These *must* use **await** in order to suspend until all parts of the sequence have been returned, and may also need to use **try** if the sequence is throwing.

For example, **allSatisfy()** will check whether all elements in an async sequence pass a predicate of your choosing:

```
func checkQuotes() async throws {
    let url = URL(string: "https://hws.dev/quotes.txt")!
    let noShortQuotes = try await url.lines.allSatisfy { $0.count
> 30 }
    print(noShortQuotes)
}

try? await checkQuotes()
```

Important: As with regular sequences, in order to return a correct value **allSatisfy()** must have fetched every value in the sequence first, and therefore using it with an infinite sequence will never return a value. The same is true of other similar functions, such as **min()**, **max()**, and **reduce()**, so be careful.

You can of course combine methods that create new async sequences and return a single value, for example to fetch lots of random numbers, convert them to integers, then find the largest:

```
func printHighestNumber() async throws {
    let url = URL(string: "https://hws.dev/random-numbers.txt")!

    if let highest = try await
url.lines.compactMap(Int.init).max() {
```

```
    print("Highest number: \(highest)")
  } else {
    print("No number was the highest.")
  }
}

try? await printHighestNumber()
```

Or to sum all the numbers:

```
func sumRandomNumbers() async throws {
  let url = URL(string: "https://hws.dev/random-numbers.txt")!
  let sum = try await url.lines.compactMap(Int.init).reduce(0,
+)
  print("Sum of numbers: \(sum)")
}

try? await sumRandomNumbers()
```

How to create a custom AsyncSequence

There are only three differences between creating an **AsyncSequence** and creating a regular **Sequence**:

1. We need to conform to the **AsyncSequence** and **AsyncIteratorProtocol** protocols.
2. The **next()** method of our iterator must be marked **async**.
3. We need to create a **makeAsyncIterator()** method rather than **makeIterator()**.

That last point technically allows us to create one type that is both a synchronous and asynchronous sequence, although I'm not sure when that would be a good idea.

We're going to build two async sequences so you can see how they work, one simple and one more realistic. First, the simple one, which is an async sequence that doubles numbers every time **next()** is called:

```
struct DoubleGenerator: AsyncSequence, AsyncIteratorProtocol {
  typealias Element = Int
  var current = 1

  mutating func next() async -> Element? {
    defer { current &*= 2 }

    if current < 0 {
      return nil
    } else {
      return current
    }
  }

  func makeAsyncIterator() -> DoubleGenerator {
```

```
    self
  }
}

let sequence = DoubleGenerator()

for await number in sequence {
  print(number)
}
```

Tip: In case you haven't seen it before, **&*=** multiples with overflow, meaning that rather than running out of room when the value goes beyond the highest number of a 64-bit integer, it will instead flip around to be negative. We use this to our advantage, returning **nil** when we reach that point.

If you prefer having a separate iterator struct, that also works as with **Sequence** and you don't need to adjust the calling code:

```
struct DoubleGenerator: AsyncSequence {
  typealias Element = Int

  struct AsyncIterator: AsyncIteratorProtocol {
    var current = 1

    mutating func next() async -> Element? {
      defer { current &*= 2 }

      if current < 0 {
        return nil
      } else {
        return current
      }
    }
```

```
    }

    func makeAsyncIterator() -> AsyncIterator {
        AsyncIterator()
    }
}

let sequence = DoubleGenerator()

for await number in sequence {
    print(number)
}
```

Now let's look at a more complex example, which will periodically fetch a URL that's either local or remote, and send back any values that have changed from the previous request.

This is more complex for various reasons:

1. Our **next()** method will be marked **throws**, so callers are responsible for handling loop errors.
2. Between checks we're going to sleep for some number of seconds, so we don't overload the network. This will be configurable when creating the watcher, but internally it will use **Task.sleep()**.
3. If we get data back and it hasn't changed, we go around our loop again – wait for some number of seconds, re-fetch the URL, then check again.
4. Otherwise, if there *has* been a change between the old and new data, we overwrite our old data with the new data and send it back.
5. If no data is returned from our request, we immediately terminate the iterator by sending back **nil**.
6. This is important: once our iterator ends, any further attempt to call **next()** must also return **nil**. This is part of the design of **AsyncSequence**, so stick to it.

To add to the complexity a little, **Task.sleep()** measures its time in nanoseconds, so to sleep for one second you should specify 1 billion as the sleep amount.

Like I said, this is more complex, but it's also a useful, real-world example of **AsyncSequence**. It's also particularly powerful when combined with SwiftUI's **task()** modifier, because the network fetches will automatically start when a view is shown and cancelled when it disappears. This allows you to constantly watch for new data coming in, and stream it directly into your UI.

Anyway, here's the code – it creates a **URLWatcher** struct that conforms to the **AsyncSequence** protocol, along with an example of it being used to display a list of users in a SwiftUI view:

```
struct URLWatcher: AsyncSequence, AsyncIteratorProtocol {
  typealias Element = Data

  let url: URL
  let delay: Int
  private var comparisonData: Data?
  private var isActive = true

  init(url: URL, delay: Int = 10) {
    self.url = url
    self.delay = delay
  }

  mutating func next() async throws -> Element? {
    // Once we're inactive always return nil immediately
    guard isActive else { return nil }

    if comparisonData == nil {
      // If this is our first iteration, return the initial
value
```

```
      comparisonData = try await fetchData()
    } else {
      // Otherwise, sleep for a while and see if our data
changed
      while true {
        try await Task.sleep(nanoseconds: UInt64(delay) *
1_000_000_000)
        let latestData = try await fetchData()

        if latestData != comparisonData {
          // New data is different from previous data,
          // so update previous data and send it back
          comparisonData = latestData
          break
        }
      }
    }

    if comparisonData == nil {
      isActive = false
      return nil
    } else {
      return comparisonData
    }
  }

  private func fetchData() async throws -> Element {
    let (data, _) = try await URLSession.shared.data(from: url)
    return data
  }

  func makeAsyncIterator() -> URLWatcher {
```

```
    self
  }
}

// As an example of URLWatcher in action, try something like
this:
struct User: Identifiable, Decodable {
  let id: Int
  let name: String
}

struct ContentView: View {
  @State private var users = [User]()

  var body: some View {
    List(users) { user in
      Text(user.name)
    }
    .task {
      // continuously check the URL watcher for data
      await fetchUsers()
    }
  }

  func fetchUsers() async {
    let url = URL(fileURLWithPath: "FILENAMEHERE.json")
    let urlWatcher = URLWatcher(url: url, delay: 3)

    do {
      for try await data in urlWatcher {
        try withAnimation {
          users = try JSONDecoder().decode([User].self, from:
```

```
data)
            }
        }
    } catch {
        // just bail out
    }
  }
}
```

To make that work in your own project, replace “FILENAMEHERE” with the location of a local file you can test with. For example, I might use **/Users/twostraws/users.json**, giving that file the following example contents:

```
[
  {
    "id": 1,
    "name": "Paul"
  }
]
```

When the code first runs the list will show Paul, but if you edit the JSON file and re-save with extra users, they will just slide into the SwiftUI list automatically.

How to convert an AsyncSequence into a Sequence

Swift does not provide a built-in way of converting an **AsyncSequence** into a regular **Sequence**, but often you'll want to make this conversion yourself so you don't need to keep awaiting results to come back in the future.

The easiest thing to do is call **reduce(into:)** on the sequence, appending each item to an array of the sequence's element type. To make this more reusable, I'd recommend adding an extension such as this one:

```
extension AsyncSequence {
  func collect() async rethrows -> [Element] {
    try await reduce(into: [Element]()) { $0.append($1) }
  }
}
```

With that in place, you can now call **collect()** on any async sequence in order to get a simple array of its values. Because this is an async operation, you must call it using **await** like so:

```
func getNumberArray() async throws -> [Int] {
  let url = URL(string: "https://hws.dev/random-numbers.txt")!
  let numbers = url.lines.compactMap(Int.init)
  return try await numbers.collect()
}

if let numbers = try? await getNumberArray() {
  for number in numbers {
    print(number)
  }
}
```

Tip: Because we've made **collect()** use **rethrows**, you only need to call it using **try** if the call to **reduce()** would normally throw, so if you have an async sequence that doesn't throw errors you can skip **try** entirely.

Chapter 4

Tasks and task groups

What are tasks and task groups?

Using async/await in Swift allows us to write asynchronous code that is easy to read and understand, but by itself it doesn't enable us to run anything concurrently – even with several CPU cores working hard, async/await code would still execute sequentially.

To create actual concurrency – to provide the ability for multiple pieces of work to run at the same time – Swift provides us with two specific types for constructing and managing concurrency in a way that makes it easier to use: **Task** and **TaskGroup**. Although the types themselves aren't *complex*, they unlock a lot of power and flexibility, and sit at the core of how we use concurrency with Swift.

Which you choose – **Task** or **TaskGroup** – depends on the goal of your work: if you want one or two independent pieces of work to start, then **Task** is the right choice. If you want to split up one job into several concurrent operations then **TaskGroup** is a better fit. Task groups work best when their individual operations return exactly the same kind of data, but with a little extra effort you can coerce them into supporting heterogenous data types.

Although you might not realize it, you're using tasks every time you write any async code in Swift. You see, all async functions run as part of a task whether or not we explicitly ask for it to happen. Even using **async let** is *syntactic sugar* for creating a task then waiting for its result – special syntax that makes a particular piece of code easier to write. This is why if you use multiple sequential **async let** calls they will all start executing immediately while the rest of your code continues.

Both **Task** and **TaskGroup** can be created with one of four priority levels: **high** is the most important, then **medium**, **low**, and finally **background** at the bottom. Task priorities allow the system to adjust the order in which it executes work, meaning that important work can happen before unimportant work.

Tip: If you've been doing iOS programming for a while, you may prefer to use the more familiar quality of service priorities from **DispatchQueue**, which are **userInitiated** and **utility** in place of **high** and **low** respectively. There is no equivalent to the old **userInteractive**

priority, which is now exclusively reserved for the user interface.

How to create and run a task

Swift's **Task** struct lets us start running some work immediately, and optionally wait for the result to be returned. And it *is* optional: sometimes you don't care about the result of the task, or sometimes the task automatically updates some external value when it completes, so you can just use them as "fire and forget" operations if you need to. This makes them a great way to run async code from a synchronous function.

First, let's look at an example where we create two tasks back to back, then wait for them both to complete. This will fetch data from two different URLs, decode them into two different structs, then print a summary of the results, all to simulate a user starting up a game – what are the latest news updates, and what are the current highest scores?

Here's how that looks:

```
struct NewsItem: Decodable {
  let id: Int
  let title: String
  let url: URL
}

struct HighScore: Decodable {
  let name: String
  let score: Int
}

func fetchUpdates() async {
  let newsTask = Task { () -> [NewsItem] in
    let url = URL(string: "https://hws.dev/headlines.json")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode([NewsItem].self, from:
data)
  }
```

```
    let highScoreTask = Task { () -> [HighScore] in
        let url = URL(string: "https://hws.dev/scores.json")!
        let (data, _) = try await URLSession.shared.data(from: url)
        return try JSONDecoder().decode([HighScore].self, from:
data)
    }

    do {
        let news = try await newsTask.value
        let highScores = try await highScoreTask.value
        print("Latest news loaded with \(news.count) items.")

        if let topScore = highScores.first {
            print("\(topScore.name) has the highest score with \
(topScore.score), out of \(highScores.count) total results.")
        }
    } catch {
        print("There was an error loading user data.")
    }
}

await fetchUpdates()
```

Let's unpick the key parts:

1. Creating and running a task is done by using its initializer, passing in the work you want to do.
2. Tasks don't always need to return a value, but when they do chances are you'll need to declare exactly what as you create the task – I've said **() -> [NewsItem] in**, for example.
3. As soon as you create the task it will start running – there's no **start()** method or similar.
4. The entire task is run concurrently with your other code, which means it might be able to

run in parallel too. In our case, that means fetching and decoding the data happens inside the task, which keeps our main **fetchUpdates()** function free.

5. If you want to read the return value of a task, you need to access its **value** property using **await**. In our case our task could also throw errors because we're accessing the network, so we need to use **try** as well.
6. Once you've copied out the value from your task you can use that normally without needing **await** or **try** again, although subsequent accesses to the task itself – e.g. **newsTask.value** – *will* need **try await** because Swift can't statically determine that the value is already present.

Both tasks in that example returned a value, but that's not a requirement – the "fire and forget" approach allows us to create a task without storing it, and Swift will ensure it runs until completion correctly.

To demonstrate this, we could make a small SwiftUI program to fetch a user's inbox when a button is pressed. Button actions are *not* async functions, so we need to launch a new task inside the action. The task *can* call async functions, but in this instance we don't actually care about the result so we're not going to store the task – the function it calls will handle updating our SwiftUI view.

Here's the code:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let from: String
  let text: String
}

struct ContentView: View {
  @State private var messages = [Message]()

  var body: some View {
    NavigationView {
```

```
        Group {
          if messages.isEmpty {
            Button("Load Messages") {
              Task {
                await loadMessages()
              }
            }
          } else {
            List(messages) { message in
              VStack(alignment: .leading) {
                Text(message.from)
                  .font(.headline)

                Text(message.text)
              }
            }
          }
        }
        .navigationTitle("Inbox")
    }
  }

  func loadMessages() async {
    do {
      let url = URL(string: "https://hws.dev/messages.json")!
      let (data, _) = try await URLSession.shared.data(from:
url)
      messages = try JSONDecoder().decode([Message].self, from:
data)
    } catch {
      messages = [
        Message(id: 0, from: "Failed to load inbox.", text:
```

```
"Please try again later.")
        ]
    }
  }
}
```

Even though that code isn't so different from the previous example, I still want to pick out a few things:

1. Creating the new task is what allows us to start calling an async function even though the button's action is a synchronous function.
2. The lifetime of the task is *not* bound by the button's action closure. So, even though the closure will finish immediately, the task it created will carry on running to completion.
3. We aren't trying to read a return value from the task, or storing it anywhere. This task doesn't actually return anything, and doesn't need to.

I know it's a not a lot of code, but between **Task**, async/await, and SwiftUI a lot of work is happening on our behalf. Remember, when we use **await** we're signaling a potential suspension point, and when our functions resume they might be on the same thread as before or they might not.

In this case there are potentially four thread swaps happening in our code:

- All UI work runs on the main thread, so the button's action closure will fire on the main thread.
- Although we create the task on the main thread, the work we pass to it will execute on a background thread.
- Inside **loadMessages()** we use **await** to load our URL data, and when that resumes we have another potential thread switch – we might be on the same background thread as before, or on a different background thread.
- Finally, the **messages** property uses the **@State** property wrapper, which will automatically update its value on the main thread. So, even though we assign to it on a background thread, the actual update will get silently pushed back to the main thread.

Best of all, we don't have to care about this – we don't need to know how the system is balancing the threads, or even that the threads exist, because Swift and SwiftUI take care of that for us. In fact, the concept of tasks is so thoroughly baked into SwiftUI that there's a dedicated **task()** modifier that makes them even easier to use.

What's the difference between a task and a detached task?

If you create a new task using the regular **Task** initializer, your work starts running immediately and inherits the priority of the caller, any task local values, and its actor context. On the other hand, detached tasks also start work immediately, but do *not* inherit the priority or other information from the caller.

I'm going to explain in more detail why these differences matter, but first I want to mention this very important quote from the Swift Evolution proposal for **async let**: "**Task.detached** most of the time should not be used at all." I'm getting that out of the way up front so you don't spend time learning about detached tasks, only to realize you probably shouldn't use them!

Still here? Okay, let's dig in to our three differences: priority, task local values, and actor isolation.

The priority part is straightforward: if you're inside a user-initiated task and create a new task, it will also have a priority of user-initiated, whereas creating a new detached task would give a nil priority unless you specifically asked for something.

The task local values part is a little more complex, but to be honest probably isn't going to be of interest to most people. Task local values allow us to share a specific value everywhere inside one specific task – they are like static properties on a type, except rather than everything sharing that property, each task has its own value. Detached tasks do *not* inherit the task local values of their parent because they do not have a parent.

The *actor context* part is more important and more complex. When you create a regular task from inside an actor it will be isolated to that actor, which means you can use other parts of the actor synchronously:

```
actor User {
  func login() {
```

```
    Task {
      if authenticate(user: "taytay89", password: "n3wy0rk") {
        print("Successfully logged in.")
      } else {
        print("Sorry, something went wrong.")
      }
    }
  }

  func authenticate(user: String, password: String) -> Bool {
    // Complicated logic here
    return true
  }
}

let user = User()
await user.login()
```

In comparison, a detached task runs concurrently with all other code, including the actor that created it – it effectively has no parent, and therefore has greatly restricted access to the data inside the actor.

So, if we were to rewrite the previous actor to use a detached task, it would need to call **authenticate()** like this:

```
actor User {
  func login() {
    Task.detached {
      if await self.authenticate(user: "taytay89", password:
"n3wy0rk") {
        print("Successfully logged in.")
      } else {
        print("Sorry, something went wrong.")
```

```
        }
      }
    }

    func authenticate(user: String, password: String) -> Bool {
      // Complicated logic here
      return true
    }
}

let user = User()
await user.login()
```

This distinction is particularly important when you are running on the main actor, which will be the case if you're responding to a button click for example. The rules here might not be immediately obvious, so I want to show you some examples of what is allowed and what is *not* allowed, and more importantly explain why each is the case.

First, if you're changing the value of an **@State** property, you can do so using a regular task like this:

```
struct ContentView: View {
  @State private var name = "Anonymous"

  var body: some View {
    VStack {
      Text("Hello, \(name)!")
      Button("Authenticate") {
        Task {
          name = "Taylor"
        }
      }
    }
```

```
    }
}
```

Note: The **Task** here is of course not needed because we're just setting a local value; I'm just trying to illustrate how regular tasks and detached tasks are different.

In fact, because **@State** guarantees it's safe to change its value on any thread, we can use a detached task instead even though it won't inherit actor isolation:

```
struct ContentView: View {
    @State private var name = "Anonymous"

    var body: some View {
        VStack {
            Text("Hello, \(name)!")
            Button("Authenticate") {
                Task.detached {
                    name = "Taylor"
                }
            }
        }
    }
}
```

That's the easy part. The rules change when we switch to an observable object that publishes changes. As soon as you add any **@ObservedObject** or **@StateObject** property wrappers to a view, Swift will automatically infer that the whole view must also run on the main actor.

This makes sense if you think about it: changes published by observable objects must update the UI on the main thread, and because any part of the view might try to adjust your object the only safe approach is for the whole view to run on the main actor.

So, this means we can modify a view model from inside a task created inside a SwiftUI view:

```
class ViewModel: ObservableObject {
  @Published var name = "Hello"
}

struct ContentView: View {
  @StateObject private var model = ViewModel()

  var body: some View {
    VStack {
      Text("Hello, \(model.name)!")
      Button("Authenticate") {
        Task {
          model.name = "Taylor"
        }
      }
    }
  }
}
```

However, we *cannot* use **Task.detached** here – Swift will throw up an error that a property isolated to global actor 'MainActor' can not be mutated from a non-isolated context. In simpler terms, our view model updates the UI and so must be on the main actor, but our detached task does not belong to that actor.

At this point, you might wonder why detached tasks would have any use. Well, consider this code:

```
class ViewModel: ObservableObject { }

struct ContentView: View {
  @StateObject private var model = ViewModel()

  var body: some View {
```

```
    Button("Authenticate", action: doWork)
  }

  func doWork() {
    Task {
      for i in 1...10_000 {
        print("In Task 1: \(i)")
      }
    }

    Task {
      for i in 1...10_000 {
        print("In Task 2: \(i)")
      }
    }
  }
}
```

That's the simplest piece of code that demonstrates the usefulness of detached tasks: a SwiftUI view monitoring an empty view model, plus a button that launches a couple of tasks to print out text.

When that runs, you'll see "In Task 1" printed 10,000 times, then "In Task 2" printed 10,000 times – even though we have created two tasks, they are executing sequentially. This happens because our **@StateObject** view model forces the entire view onto the main actor, meaning that it can only do one thing at a time.

In contrast, if you change both **Task** initializers to **Task.detached**, you'll see "In Task 1" and "In Task 2" get intermingled as both execute at the same time. Without any need for actor isolation, Swift can run those tasks concurrently – using a detached task has allowed us to shed our attachment to the main actor.

Although detached tasks do have very specific uses, generally I think they should be your last

port of call – use them only if you've tried both a regular task and **async let**, and neither solved your problem.

How to get a Result from a task

If you want to read the return value from a **Task** directly, you should read its **value** using **await**, or use **try await** if it has a throwing operation. However, all tasks also have a **result** property that returns an instance of Swift's **Result** struct, generic over the type returned by the task as well as whether it might contain an error or not.

To demonstrate this, we could write some code that creates a task to fetch and decode a string from a URL. To start with we're going to make this task throw errors if the download fails, or if the data can't be converted to a string.

Here's the code:

```
enum LoadError: Error {
  case fetchFailed, decodeFailed
}

func fetchQuotes() async {
  let downloadTask = Task { () -> String in
    let url = URL(string: "https://hws.dev/quotes.txt")!
    let data: Data

    do {
      (data, _) = try await URLSession.shared.data(from: url)
    } catch {
      throw LoadError.fetchFailed
    }

    if let string = String(data: data, encoding: .utf8) {
      return string
    } else {
      throw LoadError.decodeFailed
    }
```

```
    }

    let result = await downloadTask.result

    do {
        let string = try result.get()
        print(string)
    } catch LoadError.fetchFailed {
        print("Unable to fetch the quotes.")
    } catch LoadError.decodeFailed {
        print("Unable to convert quotes to text.")
    } catch {
        print("Unknown error.")
    }
}

await fetchQuotes()
```

There's not a lot of code there, but there are a few things I want to point out as being important:

1. Our task might return a string, but also might throw one of two errors. So, when we ask for its **result** property we'll be given a **Result<String, Error>**.
2. Although we need to use **await** to get the result, we *don't* need to use **try** even though there could be errors there. This is because we're just reading out the result, not trying to read the successful value.
3. We call **get()** on the **Result** object to read the successful, but *that's* when **try** is needed because it's when Swift checks whether an error occurred or not.
4. When it comes to catching errors, we need a "catch everything" block at the end, even though we know we'll only throw **LoadError**.

That last point hits us because Swift isn't able to evaluate the task to see exactly what kinds of

error are thrown inside, and there's no way of adding that annotation ourself because Swift doesn't support typed throws.

If you don't care what errors are thrown, or don't mind digging through Foundation's various errors yourself, you can avoid handling errors in the task and just let them propagate up:

```
func fetchQuotes() async {
  let downloadTask = Task { () -> String in
    let url = URL(string: "https://hws.dev/quotes.txt")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return String(decoding: data, as: UTF8.self)
  }

  let result = await downloadTask.result

  do {
    let string = try result.get()
    print(string)
  } catch {
    print("Unknown error.")
  }
}

await fetchQuotes()
```

The main take aways here are:

1. All tasks can return a **Result** if you want.
2. For the error type, the **Result** will either contain **Error** or **Never**.
3. Although we need to use **await** to get the result, we *don't* need to use **try** until we try to get the success value inside.

Many places where **Result** was useful are now better served through async/await, but **Result** is

still useful for storing in a single value the success or failure of some operation. Yes, in the code above we evaluate the result immediately for brevity, but the power of **Result** is that it's value you can pass around elsewhere in your code to deal with at a later time.

How to control the priority of a task

Swift tasks can have a priority attached to them, such as **.high** or **.background**, but the priority can also be **nil** if no specific priority was assigned. This priority can be used by the system to determine which task should be executed next, but this isn't guaranteed – think of it as a suggestion rather than a rule.

Creating a task with a priority look like this:

```
func fetchQuotes() async {
  let downloadTask = Task(priority: .high) { () -> String in
    let url = URL(string: "https://hws.dev/chapter.txt")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return String(decoding: data, as: UTF8.self)
  }

  do {
    let text = try await downloadTask.value
    print(text)
  } catch {
    print(error.localizedDescription)
  }
}

await fetchQuotes()
```

Although you *can* directly assign a priority to a task when it's created, if you don't then Swift will follow three rules for deciding the priority automatically:

1. If the task was created from another task, the child task will inherit the priority of the parent task.

2. If the new task was created directly from the main thread as opposed to a task, it's automatically assigned the highest priority of **.userInitiated**.
3. If the new task wasn't made by another task or the main thread, Swift will try to query the priority of the thread or give it a **nil** priority.

This means not specifying an exact priority is often a good idea because Swift will do The Right Thing.

However, like I said you can also specify an exact priority from one of the following:

- The highest priority is **.high**, which is synonymous with **.userInitiated**. As the name implies, this should be used only for tasks that the user specifically started and is actively waiting for.
- Next highest is **medium**, and again as the name implies this is a great choice for most of your tasks that the user isn't actively waiting for.
- Next is **.low**, which is synonymous with **.utility**. This is the best choice for anything long enough to require a progress bar to be displayed, such as copying files or importing data.
- The lowest priority is **.background**, which is for any work the user can't see, such as building a search index. This could in theory take hours to complete.

Like I said, priority inheritance helps get us a sensible priority by default, particularly when creating tasks in response to a user interface action.

For example, we could build a simple SwiftUI app using a single task, and we don't need to provide a specific priority –it will automatically run as high priority because it was started from our UI:

```
struct ContentView: View {
  @State private var jokeText = ""

  var body: some View {
    VStack {
      Text(jokeText)
```

```
      Button("Fetch new joke", action: fetchJoke)
    }
  }

  func fetchJoke() {
    Task {
      let url = URL(string: "https://icanhazdadjoke.com")!
      var request = URLRequest(url: url)
      request.setValue("Swift Concurrency by Example",
forHTTPHeaderField: "User-Agent")
      request.setValue("text/plain", forHTTPHeaderField:
"Accept")

      let (data, _) = try await URLSession.shared.data(for:
request)

      if let jokeString = String(data: data, encoding: .utf8) {
        jokeText = jokeString
      } else {
        jokeText = "Load failed."
      }
    }
  }
}
```

Any task can query its current priority using **Task.currentPriority**, but this works from anywhere – if it's called in a function that is not currently part of a task, Swift will query the system for an answer or send back **.medium**.

Understanding how priority escalation works

Every task can be created with a specific priority level, or it can inherit a priority from somewhere else. But in two specific circumstances, Swift will *raise* the priority of a task so it's able to complete faster.

This always happens because of some specific action from us:

1. If higher-priority task A starts waiting for the result of lower-priority task B, task B will have its priority elevated to the same priority as task A.
2. If lower-priority task A has started running on an actor, and higher-priority task B has been enqueued on that same actor, task A will have its priority elevated to match task B.

In both cases, Swift is trying to ensure the higher priority task gets the quality of service it needs to run quickly – if something very important can only complete when something less important is also complete, then the less important task *becomes* very important.

For the most part, this isn't something we need to worry about in our code – think of it as a bonus feature provided automatically by Swift's tasks.

However, there is *one* place where priority escalation might surprise you, and it's worth at least being aware of it: in our first situation, where a high-priority task uses **await** on a low-priority task, using **Task.currentPriority** will report the *escalated* priority rather than the original priority. So, you might create a task with a low priority, but when you query it a minute later it might have moved up to be a high priority.

At the time of writing, this is the only real "gotcha" moment with task escalation. The other situation – if you queue a high-priority task on the same actor where a low-priority task is already running – will also involve priority escalation, but *won't* change the value of **currentPriority**. This means your task will run a little faster and it might not be obvious why, but honestly it's unlikely you'll even notice this.

How to cancel a Task

Swift's tasks use *cooperative cancellation*, which means that although we can tell a task to stop work, the task itself is free to completely ignore that instruction and carry on for as long as it wants. This is a feature rather than a bug: if cancelling a task made it stop work immediately, the task might leave your program in an inconsistent state.

There are seven things to know when working with task cancellation:

1. You can explicitly cancel a task by calling its **cancel()** method.
2. Any task can check **Task.isCancelled** to determine whether the task has been cancelled or not.
3. You can call the **Task.checkCancellation()** method, which will throw a **CancellationError** if the task has been cancelled or do nothing otherwise.
4. Some parts of Foundation automatically check for task cancellation and will throw their own cancellation error even without your input.
5. If you're using **Task.sleep()** to wait for some amount of time to pass, that will *not* honor cancellation requests – the task will still sleep even when cancelled.
6. If the task is part of a group and any part of the group throws an error, the other tasks will be cancelled and awaited.
7. If you have started a task using SwiftUI's **task()** modifier, that task will automatically be canceled when the view disappears.

We can explore a few of these in code. First, here's a function that uses a task to fetch some data from a URL, decodes it into an array, then returns the average:

```
func getAverageTemperature() async {
  let fetchTask = Task { () -> Double in
    let url = URL(string: "https://hws.dev/readings.json")!
    let (data, _) = try await URLSession.shared.data(from: url)
    let readings = try JSONDecoder().decode([Double].self,
from: data)
    let sum = readings.reduce(0, +)
```

```
    return sum / Double(readings.count)
  }

  do {
    let result = try await fetchTask.value
    print("Average temperature: \(result)")
  } catch {
    print("Failed to get data.")
  }
}

await getAverageTemperature()
```

Now, there is no explicit cancellation in there, but there *is* implicit cancellation because the **URLSession.shared.data(from:)** call will check to see whether its task is still active before continuing. If the task has been cancelled, **data(from:)** will automatically throw a **URLError** and the rest of the task won't execute.

However, that implicit check happens *before* the network call, so it's unlikely to be an actual cancellation point in practice. As most of our users are likely to be using mobile network connections, the network call is likely to take most of the time of this task, particularly if the user has a poor connection.

So, we could upgrade our task to explicitly check for cancellation *after* the network request, using **Task.checkCancellation()**. This is a static function call because it will always apply to whatever task it's called inside, and it needs to be called using **try** so that it can throw a **CancellationError** if the task *has* been cancelled.

Here's the new function:

```
func getAverageTemperature() async {
  let fetchTask = Task { () -> Double in
    let url = URL(string: "https://hws.dev/readings.json")!
```

```
    let (data, _) = try await URLSession.shared.data(from: url)
    try Task.checkCancellation()
    let readings = try JSONDecoder().decode([Double].self,
from: data)
    let sum = readings.reduce(0, +)
    return sum / Double(readings.count)
  }

  do {
    let result = try await fetchTask.value
    print("Average temperature: \(result)")
  } catch {
    print("Failed to get data.")
  }
}

await getAverageTemperature()
```

As you can see, it just takes one call to **Task.checkCancellation()** to make sure our task isn't wasting time calculating data that's no longer needed.

If you want to handle cancellation yourself – if you need to clean up some resources or perform some other calculations, for example – then instead of calling **Task.checkCancellation()** you should check the value of **Task.isCancelled** instead. This is a simple Boolean that returns the current cancellation state, which you can then act on however you want.

To demonstrate this, we could rewrite our function a third time so that cancelling the task or failing to fetch data returns an average temperature of 0. This time we're going to cancel the task ourselves as soon as it's created, but because we're always returning a default value we no longer need to handle errors when reading the task's result:

```
func getAverageTemperature() async {
```

```
    let fetchTask = Task { () -> Double in
        let url = URL(string: "https://hws.dev/readings.json")!

        do {
            let (data, _) = try await URLSession.shared.data(from:
url)
            if Task.isCancelled { return 0 }

            let readings = try JSONDecoder().decode([Double].self,
from: data)
            let sum = readings.reduce(0, +)
            return sum / Double(readings.count)
        } catch {
            return 0
        }
    }

    fetchTask.cancel()

    let result = await fetchTask.value
    print("Average temperature: \(result)")
}

await getAverageTemperature()
```

Now we have one implicit cancellation point with the **data(from:)** call, and an explicit one with the check on **Task.isCancelled**. If either one is triggered, the task will return 0 rather than throw an error.

Tip: You can use both **Task.checkCancellation()** and **Task.isCancelled** from both synchronous and asynchronous functions. Remember, async functions can call synchronous functions freely, so checking for cancellation can be just as important to avoid doing

unnecessary work.

How to make a task sleep

Swift's **Task** struct has a static **sleep()** method that will cause the current task to be suspended for at least some number of nanoseconds. Yes, nanoseconds: you need to write 1_000_000_000 to get 1 second. You need to call **Task.sleep()** using **await** as it will cause the task to be suspended, and you also need to use **try** because **sleep()** will throw an error if the task is cancelled.

For example, this will make the current task sleep for at least 3 seconds:

```
try await Task.sleep(nanoseconds: 3_000_000_000)
```

Important: Calling **Task.sleep()** will make the current task sleep for *at least* the amount of time you ask, not *exactly* the time you ask. There is a little drift involved because the system might be busy doing other work when the sleep ends, but you are at least guaranteed it won't end *before* your time has elapsed.

Using nanoseconds is a bit clumsy, but Swift doesn't have an alternative at this time – the plan seems to be to wait for a more thorough review of managing time in the language before committing to specific API.

In the meantime, we can add small **Task** extensions to make sleeping easier to accomplish. For example, this lets us sleep using seconds as a floating-point number:

```
extension Task where Success == Never, Failure == Never {
  static func sleep(seconds: Double) async throws {
    let duration = UInt64(seconds * 1_000_000_000)
    try await Task.sleep(nanoseconds: duration)
  }
}
```

With that in place, you can now write **Task.sleep(seconds: 0.5)** or similar.

Calling **Task.sleep()** automatically checks for cancellation, meaning that if you cancel a

sleeping task it will be woken and throw a **CancellationError** for you to catch.

Tip: Unlike making a thread sleep, **Task.sleep()** does *not* block the underlying thread, allowing it pick up work from elsewhere if needed.

How to voluntarily suspend a task

If you're executing a long-running task that has few if any suspension points, for example if you're repeatedly iterating over an intensive loop, you can call **Task.yield()** to *voluntarily* suspend the current task so that Swift can give other tasks the chance to proceed a little if needed.

To demonstrate this, we could write a simple function to calculate the factors for a number – numbers that divide another number equally. For example, the factors for 12 are 1, 2, 3, 4, 6, and 12. A simple version of this function might look like this:

```
func factors(for number: Int) async -> [Int] {
  var result = [Int]()

  for check in 1...number {
    if number.isMultiple(of: check) {
      result.append(check)
    }
  }

  return result
}

let factors = await factors(for: 120)
print("Found \(factors.count) factors for 120.")
```

Despite being a pretty inefficient implementation, in release builds that will still execute quite fast even for numbers such as 100,000,000. But if you try something even bigger you'll notice it struggles – running hundreds of millions of checks is really going to make the task chew up a lot of CPU time, which might mean other tasks are left sitting around unable to make even the slightest progress forward.

Keep in mind our other tasks might be able to kick off some work then suspend immediately,

such as making network requests. A simple improvement is to force our **factors()** method to pause every so often so that Swift can run other tasks if it wants – we're effectively asking it to come up for air and let another task have a go.

So, we could modify the function so that it calls **Task.yield()** every 100,000 numbers, like this:

```
func factors(for number: Int) async -> [Int] {
  var result = [Int]()

  for check in 1...number {
    if check.isMultiple(of: 100_000) {
      await Task.yield()
    }

    if number.isMultiple(of: check) {
      result.append(check)
    }
  }

  return result
}

let factors = await factors(for: 120)
print("Found \(factors.count) factors for 120.")
```

However, that has the downside of now having twice as much work in the loop. As an alternative, you could try yielding only when a multiple is actually found, like this:

```
func factors(for number: Int) async -> [Int] {
  var result = [Int]()

  for check in 1...number {
    if number.isMultiple(of: check) {
```

```
            result.append(check)
            await Task.yield()
        }
    }

    return result
}

let factors = await factors(for: 120)
print("Found \(factors.count) factors for 120.")
```

That offers Swift the chance to pause every time a multiple is found. Yes, it will be called a lot in the first few iterations, but fewer multiples will be found over time and so it probably won't yield as often as the previous example – it could well defeat the point of using **yield()** in the first place.

Calling `yield()` does not always mean the task will stop running: if it has a higher priority than other tasks that are waiting, it's entirely possible your task will just immediately resume its work. Think of this as *guidance* – we're giving Swift the chance to execute other tasks temporarily rather than forcing it to do so.

Think of calling **Task.yield()** as the equivalent of calling a fictional **Task.doNothing()** method – it gives Swift the chance to adjust the execution of its tasks without actually creating any real work.

How to create a task group and add tasks to it

Swift's task groups are collections of tasks that work together to produce a single result. Each task inside the group must return the same kind of data, but if you use enum associated values you can make them send back different kinds of data – it's a little clumsy, but it works.

Creating a task group is done in a very precise way to avoid us creating problems for ourselves: rather than creating a **TaskGroup** instance directly, we do so by calling the **withTaskGroup(of:)** function and telling it the data type the task group will return. We give this function the code for our group to execute, and Swift will pass in the **TaskGroup** that was created, which we can then use to add tasks to the group.

First, I want to look at the simplest possible example of task groups, which is returning 5 constant strings, adding them into a single array, then joining that array into a string:

```
 func printMessage() async {
  let string = await withTaskGroup(of: String.self) { group ->
String in
    group.addTask { "Hello" }
    group.addTask { "From" }
    group.addTask { "A" }
    group.addTask { "Task" }
    group.addTask { "Group" }

    var collected = [String]()

    for await value in group {
      collected.append(value)
    }

    return collected.joined(separator: " ")
```

```
    }

    print(string)
}

await printMessage()
```

I know it's trivial, but it demonstrates several important things:

1. We must specify the exact type of data our task group will return, which in our case is **String.self** so that each child task can return a string.
2. We need to specify exactly what the return value of the group will be using **group -> String in** – Swift finds it hard to figure out the return value otherwise.
3. We call **addTask()** once for each task we want to add to the group, passing in the work we want that task to do.
4. Task groups conform to **AsyncSequence**, so we can read all the values from their children using **for await**, or by calling **group.next()** repeatedly.
5. Because the whole task group executes asynchronously, we must call it using **await**.

However, there's one other thing you *can't* see in that code sample, which is that our task results are sent back in *completion* order and not creation order. That is, our code above might send back "Hello From A Task Group", but it also might send back "Task From A Hello Group", "Group Task A Hello From", or any other possible variation – the return value could be different every time.

Tasks created using **withTaskGroup()** cannot throw errors. If you *want* them to be able to throw errors that bubble upwards – i.e., that are handled outside the task group – you should use **withThrowingTaskGroup()** instead. To demonstrate this, and also to demonstrate a more real-world example of **TaskGroup** in action, we could write some code that fetches several news feeds and combines them into one list:

```
struct NewsStory: Identifiable, Decodable {
    let id: Int
```

```
  let title: String
  let strap: String
  let url: URL
}

struct ContentView: View {
  @State private var stories = [NewsStory]()

  var body: some View {
    NavigationView {
      List(stories) { story in
        VStack(alignment: .leading) {
          Text(story.title)
            .font(.headline)

          Text(story.strap)
        }
      }
      .navigationTitle("Latest News")
    }
    .task {
      await loadStories()
    }
  }

  func loadStories() async {
    do {
      stories = try await withThrowingTaskGroup(of:
[NewsStory].self) { group -> [NewsStory] in
        for i in 1...5 {
          group.addTask {
            let url = URL(string: "https://hws.dev/news-\
```

```
(i).json")!
            let (data, _) = try await
URLSession.shared.data(from: url)
            return try JSONDecoder().decode([NewsStory].self,
from: data)
          }
        }

        let allStories = try await group.reduce(into:
[NewsStory]()) { $0 += $1 }
        return allStories.sorted { $0.id > $1.id }
      }
    } catch {
      print("Failed to load stories")
    }
  }
}
```

In that code you can see we have a simple struct that contains one news story, a SwiftUI view showing all the news stories we fetched, plus a **loadStories()** method that handles fetching and decoding several news feeds into a single array.

There are four things in there that deserve special attention:

1. Fetching and decoding news items might throw errors, and those errors are *not* handled inside the tasks, so we need to use **withThrowingTaskGroup()** to create the group.
2. One of the main advantages of task groups is being able to add tasks inside a loop – we can loop from 1 through 5 and call **addTask()** repeatedly.
3. Because the task group conforms to **AsyncSequence**, we can call its **reduce()** method to boil all its task results down to a single value, which in this case is a single array of news stories.
4. As I said earlier, tasks in a group can complete in any order, so we sorted the resulting array of news stories to get them all in a sensible order.

Regardless of whether you're using throwing or non-throwing tasks, all tasks in a group must complete before the group returns. You have three options here:

1. Awaiting all individual tasks in the group.
2. Calling **waitForAll()** will automatically wait for tasks you have not explicitly awaited, discarding any results they return.
3. If you do not explicitly await any child tasks, they will be *implicitly* awaited – Swift will wait for them anyway, even if you aren't using their return values.

Of the three, I find myself using the first most often because it's the most explicit – you aren't leaving folks wondering why some or all of your tasks are launched then ignored.

How to cancel a task group

Swift's task groups can be cancelled in one of three ways:

1. If the parent task of the task group is cancelled.
2. If you explicitly call **cancelAll()** on the group.
3. If one of your child tasks throws an uncaught error, all remaining tasks will be implicitly cancelled.

The first of those happens outside of the task group, but the other two are worth investigating.

First, calling **cancelAll()** will cancel all remaining tasks. As with standalone tasks, cancelling a task group is *cooperative*: your child tasks can check for cancellation using **Task.isCancelled** or **Task.checkCancellation()**, but they can ignore cancellation entirely if they want.

I'll show you a real-world example of **cancelAll()** in action in a moment, but before that I want to show you some toy examples so you can see how it works.

We could write a simple **printMessage()** function like this one, creating three tasks inside a group in order to generate a string:

```
func printMessage() async {
  let result = await withThrowingTaskGroup(of: String.self)
{ group -> String in
    group.addTask {
      return "Testing"
    }

    group.addTask {
      return "Group"
    }

    group.addTask {
      return "Cancellation"
```

```
    }

    group.cancelAll()
    var collected = [String]()

    do {
      for try await value in group {
        collected.append(value)
      }
    } catch {
      print(error.localizedDescription)
    }

    return collected.joined(separator: " ")
  }

  print(result)
}

await printMessage()
```

As you can see, that calls **cancelAll()** immediately after creating all three tasks, and yet when the code is run you'll still see all three strings printed out. I've said it before, but it bears repeating and this time in bold: **cancelling a task group is cooperative, so unless the tasks you add implicitly or explicitly check for cancellation calling `cancelAll()` by itself won't do much.**

To see **cancelAll()** actually working, try replacing the first **addTask()** call with this:

```
group.addTask {
  try Task.checkCancellation()
  return "Testing"
}
```

And now our behavior will be different: you might see “Cancellation” by itself, “Group” by itself, “Cancellation Group”, “Group Cancellation”, or nothing at all.

To understand why, keep the following in mind:

1. Swift will start all three tasks immediately. They might all run in parallel; it depends on what the system thinks will work best at runtime.
2. Although we immediately call **cancelAll()**, some of the tasks might have started running.
3. All the tasks finish in completion order, so when we first loop over the group we might receive the result from any of the three tasks.

When you put those together, it’s entirely possible the first task to complete is the one that calls **Task.checkCancellation()**, which means our loop will exit, we’ll print an error message, and send back an empty string. Alternatively, one or both of the other tasks might run first, in which case we’ll get our other possible outputs.

Remember, calling **cancelAll()** only cancels *remaining* tasks, meaning that it won’t undo work that has already completed. Even then the cancellation is cooperative, so you need to make sure the tasks you add to the group check for cancellation.

With that toy example out of the way, here’s a more complex demonstration of **cancelAll()** that builds on an example from an earlier chapter. This code attempts to fetch, merge, and display using SwiftUI the contents of five news feeds. If any of the fetches throws an error the whole group will throw an error and end, but if a fetch somehow succeeds while ending up with an empty array it means our data quota has run out and we should stop trying any other feed fetches.

Here’s the code:

```
struct NewsStory: Identifiable, Decodable {
  let id: Int
  let title: String
  let strap: String
```

```
  let url: URL
}

struct ContentView: View {
  @State private var stories = [NewsStory]()

  var body: some View {
    NavigationView {
      List(stories) { story in
        VStack(alignment: .leading) {
          Text(story.title)
            .font(.headline)

          Text(story.strap)
        }
      }
      .navigationTitle("Latest News")
    }
    .task {
      await loadStories()
    }
  }

  func loadStories() async {
    do {
      try await withThrowingTaskGroup(of: [NewsStory].self)
{ group -> Void in
        for i in 1...5 {
          group.addTask {
            let url = URL(string: "https://hws.dev/news-\
(i).json")!
            let (data, _) = try await
```

```
URLSession.shared.data(from: url)
            try Task.checkCancellation()
            return try JSONDecoder().decode([NewsStory].self,
from: data)
          }
        }

        for try await result in group {
          if result.isEmpty {
            group.cancelAll()
          } else {
            stories.append(contentsOf: result)
          }
        }

        stories.sort { $0.id < $1.id }
      }
    } catch {
      print("Failed to load stories: \
(error.localizedDescription)")
    }
  }
}
```

As you can see, that calls **cancelAll()** as soon as any feed sends back an empty array, thus aborting all remaining fetches. Inside the child tasks there is an explicit call to **Task.checkCancellation()**, but the **data(from:)** also runs check for cancellation to avoid doing unnecessary work.

The other way task groups get cancelled is if one of the tasks throws an uncaught error. We can write a simple test for this by creating two tasks inside a group, both of which sleep for a little time. The first task will sleep for 1 second then throw an example error, whereas the

second will sleep for 2 seconds then print the value of **Task.isCancelled**.

Here's how that looks:

```
enum ExampleError: Error {
  case badURL
}

func testCancellation() async {
  do {
    try await withThrowingTaskGroup(of: Void.self) { group ->
Void in
      group.addTask {
        try await Task.sleep(nanoseconds: 1_000_000_000)
        throw ExampleError.badURL
      }

      group.addTask {
        try await Task.sleep(nanoseconds: 2_000_000_000)
        print("Task is cancelled: \(Task.isCancelled)")
      }

      try await group.next()
    }
  } catch {
    print("Error thrown: \(error.localizedDescription)")
  }
}

await testCancellation()
```

Note: Just throwing an error inside **addTask()** isn't enough to cause other tasks in the group to be cancelled – this only happens when you access the value of the throwing task using **next()**

or when looping over the child tasks. This is why the code sample above specifically waits for the result of a task, because doing so will cause **ExampleError.badURL** to be rethrown and cancel the other task.

Calling **addTask()** on your group will unconditionally add a new task to the group, even if you have already cancelled the group. If you want to avoid adding tasks to a cancelled group, use the **addTaskUnlessCancelled()** method instead – it works identically except will do nothing if called on a cancelled group. Calling **addTaskUnlessCancelled()** returns a Boolean that will be true if the task was successfully added, or false if the task group was already cancelled.

How to handle different result types in a task group

Each task in a Swift task group must return the same type of data as all the other tasks in the group, which is often problematic – what if you need one task group to handle several different types of data?

In this situation you should consider using **async let** for your concurrency if you can, because every **async let** expression can return its own unique data type. So, the first might result in an array of strings, the second in an integer, and so on, and once you've awaited them all you can use them however you please.

However, if you *need* to use task groups – for example if you need to create your tasks in a loop – then there is a solution: create an enum with associated values that wrap the underlying data you want to return. Using this approach, each of the tasks in your group still return a single data type – one of the cases from your enum – but *inside* those cases you can place the unique data types you're actually using.

This is best demonstrated with some example code, but because it's quite a lot I'm going to add inline comments so you can see what's going on:

```
// A struct we can decode from JSON, storing one message from a
contact.
struct Message: Decodable {
  let id: Int
  let from: String
  let message: String
}

// A user, containing their name, favorites list, and messages
array.
struct User {
```

```
  let username: String
  let favorites: Set<Int>
  let messages: [Message]
}

// A single enum we'll be using for our tasks, each containing
a different associated value.
enum FetchResult {
  case username(String)
  case favorites(Set<Int>)
  case messages([Message])
}

func loadUser() async {
  // Each of our tasks will return one FetchResult, and the
whole group will send back a User.
  let user = await withThrowingTaskGroup(of: FetchResult.self)
{ group -> User in
    // Fetch our username string
    group.addTask {
      let url = URL(string: "https://hws.dev/username.json")!
      let (data, _) = try await URLSession.shared.data(from:
url)
      let result = String(decoding: data, as: UTF8.self)

      // Send back FetchResult.username, placing the string
inside.
      return .username(result)
    }

    // Fetch our favorites set
    group.addTask {
```

```
            let url = URL(string: "https://hws.dev/user-
favorites.json")!
            let (data, _) = try await URLSession.shared.data(from:
url)
            let result = try JSONDecoder().decode(Set<Int>.self,
from: data)

            // Send back FetchResult.favorites, placing the set
inside.
            return .favorites(result)
        }

        // Fetch our messages array
        group.addTask {
            let url = URL(string: "https://hws.dev/user-
messages.json")!
            let (data, _) = try await URLSession.shared.data(from:
url)
            let result = try JSONDecoder().decode([Message].self,
from: data)

            // Send back FetchResult.messages, placing the message
array inside
            return .messages(result)
        }

        // At this point we've started all our tasks,
        // so now we need to stitch them together into
        // a single User instance. First, we set
        // up some default values:
        var username = "Anonymous"
        var favorites = Set<Int>()
```

```
    var messages = [Message]()

    // Now we read out each value, figure out
    // which case it represents, and copy its
    // associated value into the right variable.
    do {
      for try await value in group {
        switch value {
        case .username(let value):
          username = value
        case .favorites(let value):
          favorites = value
        case .messages(let value):
          messages = value
        }
      }
    } catch {
      // If any of the fetches went wrong, we might
      // at least have partial data we can send back.
      print("Fetch at least partially failed; sending back what
we have so far. \(error.localizedDescription)")
    }

    // Send back our user, either filled with
    // default values or using the data we
    // fetched from the server.
    return User(username: username, favorites: favorites,
messages: messages)
  }

  // Now do something with the finished user data.
  print("User \(user.username) has \(user.messages.count)
```

```
messages and \(user.favorites.count) favorites.")
}

await loadUser()
```

I know it's a lot of code, but really it boils down to two things:

1. Creating an enum with one case for each type of data you're expecting, with each case having an associated value of that type.
2. Reading the results from your group's tasks using a **switch** block that reads each case from your enum, extracts the associated value inside, and acts on it appropriately.

So, it's not impossible to handle heterogeneous results in a task group, it just requires a little extra thinking.

What's the difference between async let, tasks, and task groups?

Swift's **async let**, **Task**, and task groups all solve a similar problem: they allow us to create concurrency in our code so the system is able to run them efficiently. Beyond that, the way they work is quite different, and which you'll choose depends on your exact scenario.

To help you understand how they differ, and provide some guidance on where each one is a good idea, I want to walk through the key behaviors of each of them.

First, **async let** and **Task** are designed to create specific, individual pieces of work, whereas task groups are designed to run multiple pieces of work at the same time and gather the results. As a result, **async let** and **Task** have no way to express a dynamic amount of work that should run in parallel.

For example, if you had an array of URLs and wanted to fetch them all in parallel, convert them into arrays of weather readings, then average them to a single **Double**, task groups would be a great choice because you won't know ahead of time how m any URLs are in your array. Trying to write this using **async let** or **Task** just wouldn't work, because you'd have to hard-code the exact number of **async let** lines rather than just loop over an array.

Second, task groups automatically let us process results from child tasks in the order they complete, rather than in an order we specify. For example, if we wanted to fetch five pieces of data, task groups allow us to use **group.next()** to read whichever of the five comes back first, whereas using **async let** and **Task** would require us to await values in a specific, fixed order.

That alone is a helpful feature of task groups, but in some situations it goes from helpful to *crucial*. For example, if you have three possible servers for some data and want to use whichever one responds fastest, task groups are perfect – you can use **addTask()** once for each server, then call **next()** only once to read whichever one responded fastest.

Third, although all three forms of concurrency will automatically be marked as cancelled if their parent task is cancelled, only **Task** and task group can be cancelled directly, using

cancel() and **cancelAll()** respectively. There is no equivalent for **async let**.

Fourth, because **async let** doesn't give us a handle to the underlying task it creates for us, it's not possible to pass that task elsewhere – we can't start an **async let** task in one function then pass that task to a different function. On the other hand, if you create a task that returns a string and never throws an error, you can pass that **Task<String, Never>** object around as needed.

And finally, although task groups *can* work with heterogeneous results – i.e., child tasks that return different types of data – it takes the extra work of making an enum to wrap the data. **async let** and **Task** do not suffer from this problem because they always return a single result type, so each result can be different.

By sheer volume of advantages you might think that **async let** is clearly much less useful than both **Task** and task groups, but not all those points carry equal weight in real-world code. In practice, I would suggest you're likely to:

- Use **async let** the most; it works best when there is a fixed amount of work to do.
- Use **Task** for some places where **async let** doesn't work, such as passing an incomplete value to a function.
- Use task groups least commonly, or at least use them *directly* least commonly – you might build other things on top of them.

I find that order is pretty accurate in practice, for a number of reasons:

1. I normally want results from all the work I start, so being able to skip some or get results in completion order is less important.
2. It's surprisingly common to want to work with different data types, which is clumsy with task groups.
3. If I need to be able to cancel tasks, **Task** is similar enough to **async let** that it's easy to move across to **Task** without going all the way to a task group.

So, again I would recommend you start with **async let**, move to **Task** if needed, then go to task groups only if there's something specific they offer that you need.

How to make async command-line tools and scripts

If you're writing a command-line tool, you can use **async** in conjunction with the **@main** attribute to launch your app into an async context immediately. To do this, first create the static **main()** method as you normally would with **@main**, then add **async** to it. You can optionally also add **throws** if you don't intend to handle errors there.

For example, we could write a small command-line tool that fetches data from a URL and prints it out:

```
@main
struct UserFetcher {
  static func main() async throws {
    let url = URL(string: "https://hws.dev/users.csv")!

    for try await line in url.lines {
      print("Received user: \(line)")
    }
  }
}
```

Tip: Just like using the **@main** attribute with a synchronous **main()** method, you should not include a main.swift file in your command-line project.

Using **async** and **@main** together benefits from the full range of Swift concurrency features. Behind the scenes, Swift will automatically create a new task in which it runs your **main()** method, then terminate the program when that task finishes.

Although it doesn't work in the current Xcode release, the goal is for Swift to support async calls in top-level code. This would mean you could use main.swift files and remove most of the code in the previous sample – you could just go ahead and make async calls outside of a

function.

How to create and use task local values

Swift lets us attach metadata to a task using *task-local values*, which are small pieces of information that any code inside a task can read. For example, you've already seen how we can read **Task.isCancelled** to see whether the current task is cancelled or not, but that's not a true static property – it's scoped to the current task, rather than shared across all tasks. This is the power of task-local values: the ability to create static-like properties inside a task.

Important: Most people will not want to use task-local values – if you're just curious you're welcome to read on and explore how task-local values work, but honestly they are useful in only a handful of very specific circumstances and if you find them complex I wouldn't worry too much.

Task-local values are analogous to thread-local values in an old-style multithreading environment: we attach some metadata to our task, and any code running inside that task can read that data as needed. Swift's implementation is carefully scoped so that you create contexts where the data is available, rather than just injecting it directly into the task, which makes it possible to adjust your metadata over time. However, *inside* that context all code is able to read your task-local values, regardless of how it's used.

Using task-local values happens in four steps:

1. Creating a type that has one or more properties we want to make into task-local values. This can be an enum, struct, class, or even actor if you want, but I'd suggest starting with an enum so it's clear you don't intend to make instances of the type.
2. Marking each of your task-local values with the **@TaskLocal** property wrapper. These properties can be any type you want, including optionals, but must be marked as **static**.
3. Starting a new task-local scope using **YourType.$yourProperty.withValue(someValue) { … }**.

Inside the task-local scope, any time you read **YourType.yourProperty** you will receive the

task-local value for that property – it's not a regular static property that has a single value shared between all parts of your program, but instead it can return a different value depending on which task tries to read it.

To demonstrate task-local values in action, I want to give you two examples: the first is a simple toy example that demonstrates the code required to use them and how they work, but the second is a more real-world example that's actually useful.

First, our simple example. This will create a **User** enum with a **id** property that is marked **@TaskLocal**, then it will launch a couple of tasks with different values for that user ID. Each task will do exactly the same thing: print the user ID, sleep for a small amount of time, then print the user ID again, which will allow you to see both tasks running at the same time while having their own unique task-local user ID.

Here's the code:

```
enum User {
  @TaskLocal static var id = "Anonymous"
}

@main
struct App {
  static func main() async throws {
    Task {
      try await User.$id.withValue("Piper") {
        print("Start of task: \(User.id)")
        try await Task.sleep(nanoseconds: 1_000_000)
        print("End of task: \(User.id)")
      }
    }

    Task {
      try await User.$id.withValue("Alex") {
```

```
            print("Start of task: \(User.id)")
            try await Task.sleep(nanoseconds: 1_000_000)
            print("End of task: \(User.id)")
        }
    }

    print("Outside of tasks: \(User.id)")
  }
}
```

When that code runs it will print:

- Start of task: Alex
- Start of task: Piper
- Outside of tasks: Anonymous
- End of task: Alex
- End of task: Piper

Of course, because the two tasks run independently of each other you might also find that the order of Piper and Alex switch. The important thing is that each task has its own value for **User.id** even as they overlap, and code outside the task will continue to use the original value.

As you can see, Swift makes it impossible to forget about a task-local value you've set, because it only exists for the work inside **withValue()**. This scoping approach also means it's possible to nest multiple task locals as needed, and you can even shadow task locals – start a scope for one, do some work, then start another nested scope for that same property. so that it temporarily has a different value.

In real-world code, task-local values are useful for places where you need to repeatedly pass values around inside your tasks – values that need to be shared within the task, but not across your whole program like a singleton might be. For example, the Swift Evolution proposal for task-local values (**https://github.com/apple/swift-evolution/blob/main/proposals/0311-task-locals.md**) suggests examples such as tracing, mocking, progress monitoring, and more.

As a more complex example, we could create a simple **Logger** struct that writes out messages depending on the current level of logging: debug being the lowest log level, then info, warn, error, and finally fatal at the highest level. If we make the log level – which messages to print – be a task-local value, then each of our tasks can have whatever level of logging they want, regardless of what other tasks are doing.

To make this work we need three things:

1. An enum to describe the five levels of logging.
2. A **Logger** struct that is a singleton.
3. A task-local property inside **Logger** to store the current log level. (Even though the logger is a singleton, the log *level* is task-local.)

On top of that, we need a couple more things to actually demonstrate the logger in action: a **fetch()** method that downloads data from a URL and creates various logging messages, and a couple of tasks that call **fetch()** with different task-local log settings so we can see exactly how it all works.

Here's the code:

```
// Our five log levels, marked Comparable so we can use < and >
with them.
enum LogLevel: Comparable {
  case debug, info, warn, error, fatal
}

struct Logger {
  // The log level for an individual task
  @TaskLocal static var logLevel = LogLevel.info

  // Make this struct a singleton
  private init() { }
  static let shared = Logger()
```

```
  // Print out a message only if it meets or exceeds our log
level.
  func write(_ message: String, level: LogLevel) {
    if level >= Logger.logLevel {
      print(message)
    }
  }
}

@main
struct App {
  // Returns data from a URL, writing log messages along the
way.
  static func fetch(url urlString: String) async throws ->
String? {
    Logger.shared.write("Preparing request: \(urlString)",
level: .debug)

    if let url = URL(string: urlString) {
      let (data, _) = try await URLSession.shared.data(from:
url)
      Logger.shared.write("Received \(data.count) bytes",
level: .info)
      return String(decoding: data, as: UTF8.self)
    } else {
      Logger.shared.write("URL \(urlString) is invalid",
level: .error)
      return nil
    }
  }
```

```
  // Starts a couple of fire-and-forget tasks with different
log levels.
  static func main() async throws {
    Task {
      try await Logger.$logLevel.withValue(.debug) {
        try await fetch(url: "https://hws.dev/news-1.json")
      }
    }

    Task {
      try await Logger.$logLevel.withValue(.error) {
        try await fetch(url: "https:\\hws.dev/news-1.json")
      }
    }
  }
}
```

When that runs you'll see "Preparing request: **https://hws.dev/news-1.json**" as the first task starts, then "URL https:\hws.dev/news-1.json is invalid" as the second task starts (I used a back slash rather than forward slash), then "Received 8075 bytes" as the first task finishes downloading its data.

So, here our **fetch()** method doesn't even need to know that a task-local value is being used – it just calls the **Logger** singleton, which in turn refers to the task-local value.

To finish up, I want to leave you with a few important tips for using task-local values:

1. It's okay to access a task-local value outside of a **withValue()** scope – you'll just get back whatever default value you gave it.
2. Although regular tasks inherit task-local values of their parent task, detached tasks do *not* because they don't have a parent.
3. Task-local values are read-only; you can only modify them by calling **withValue()** as shown above.

And finally, one important quote from the Swift Evolution proposal for this feature: “please be careful with the use of task-locals and don't use them in places where plain-old parameter passing would have done the job.” Put more plainly, if task locals are the answer, there’s a very good chance you’re asking the wrong question.

How to run tasks using SwiftUI's task() modifier

SwiftUI provides a **task()** modifier that starts a new detached task as soon as a view appears, and automatically cancels the task when the view disappears. This is sort of the equivalent of starting a task in **onAppear()** then cancelling it **onDisappear()**, although **task()** has an extra ability to track an identifier and restart its task when the identifier changes.

In the simplest scenario – and probably the one you're going to use the most – **task()** is the best way to load your view's initial data, which might be loaded from local storage or by fetching and decoding a remote URL.

For example, this downloads data from a server and decodes it into an array for display in a list:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let from: String
  let text: String
}

struct ContentView: View {
  @State private var messages = [Message]()

  var body: some View {
    NavigationView {
      List(messages) { message in
        VStack(alignment: .leading) {
          Text(message.from)
            .font(.headline)

          Text(message.text)
```

```
            }
        }
        .navigationTitle("Inbox")
        .task {
            await loadMessages()
        }
    }
  }

  func loadMessages() async {
    do {
      let url = URL(string: "https://hws.dev/messages.json")!
      let (data, _) = try await URLSession.shared.data(from:
url)
      messages = try JSONDecoder().decode([Message].self, from:
data)
    } catch {
      messages = [
        Message(id: 0, from: "Failed to load inbox.", text:
"Please try again later.")
      ]
    }
  }
}
```

Important: The **task()** modifier is a great place to load the data for your SwiftUI views. Remember, they can be recreated many times over the lifetime of your app, so you should avoid putting this kind of work into their initializers if possible.

A more advanced usage of **task()** is to attach some kind of **Equatable** identifying value – when that value changes SwiftUI will automatically cancel the previous task and create a new task with the new value. This might be some shared app state, such as whether the user is

logged in or not, or some local state, such as what kind of filter to apply to some data.

As an example, we could upgrade our messaging view to support both an Inbox and a Sent box, both fetched and decoded using the same **task()** modifier. By setting the message box type as the identifier for the task with **.task(id: selectedBox)**, SwiftUI will automatically update its message list every time the selection changes.

Here's how that looks in code:

```
struct Message: Decodable, Identifiable {
  let id: Int
  let user: String
  let text: String
}

// Our content view is able to handle two kinds of message box
now.
struct ContentView: View {
  @State private var messages = [Message]()
  @State private var selectedBox = "Inbox"
  let messageBoxes = ["Inbox", "Sent"]

  var body: some View {
    NavigationView {
      List {
        Section {
          ForEach(messages) { message in
            VStack(alignment: .leading) {
              Text(message.user)
                .font(.headline)

              Text(message.text)
            }
```

```
            }
          }
        }
        .listStyle(.insetGrouped)
        .navigationTitle(selectedBox)

        // Our task modifier will recreate its fetchData() task
whenever selectedBox changes
        .task(id: selectedBox) {
          await fetchData()
        }
        .toolbar {
          // Switch between our two message boxes
          Picker("Select a message box", selection: $selectedBox)
{
            ForEach(messageBoxes, id: \.self, content: Text.init)
          }
          .pickerStyle(.segmented)
        }
      }
    }

    // This is almost the same as before, but now loads the
selectedBox JSON file rather than always loading the inbox.
    func fetchData() async {
      do {
        let url = URL(string: "https://hws.dev/\
(selectedBox.lowercased()).json")!
        let (data, _) = try await URLSession.shared.data(from:
url)
        messages = try JSONDecoder().decode([Message].self, from:
data)
```

```
        } catch {
            messages = [
                Message(id: 0, user: "Failed to load message box.",
text: "Please try again later.")
            ]
        }
    }
}
```

Tip: That example uses the shared **URLSession**, which means it will cache its responses and so load the two inboxes only once. If that's what you want you're all set, but if you want it to always fetch the files make sure you create your own session configuration and disable caching.

One particularly interesting use case for **task()** is with **AsyncSequence** collections that continuously generate values. This might be a server that maintains an open connection while sending fresh content, it might be the **URLWatcher** example we looked at previously, or perhaps just a local value. For example, we could write a simple random number generator that regularly emits new random numbers – with the **task()** modifier we can constantly watch that for changes, and stream the results into a SwiftUI view.

To bring this example to life, we're going to add one more thing: the random number generator will print a message every time a number is generated, and the resulting number list will be shown inside a detail view. Both of these are done so you can see how **task()** automatically cancels its work: the numbers will automatically start streaming when the detail view is shown, and stop streaming when the view is dismissed.

Here's the code:

```
// A simple random number generator sequence
struct NumberGenerator: AsyncSequence, AsyncIteratorProtocol {
    typealias Element = Int
    let delay: Double
```

```
  let range: ClosedRange<Int>

  init(in range: ClosedRange<Int>, delay: Double = 1) {
    self.range = range
    self.delay = delay
  }

  mutating func next() async -> Int? {
    // Make sure we stop emitting numbers when our task is
cancelled
    while Task.isCancelled == false {
      try? await Task.sleep(nanoseconds: UInt64(delay) *
1_000_000_000)
      print("Generating number")
      return Int.random(in: range)
    }

    return nil
  }

  func makeAsyncIterator() -> NumberGenerator {
    self
  }
}

// This exists solely to show DetailView when requested.
struct ContentView: View {
  var body: some View {
    NavigationView {
      NavigationLink(destination: DetailView()) {
        Text("Start Generating Numbers")
      }
```

```
    }
  }
}

// This generates and displays all the random numbers we've
generated.
struct DetailView: View {
  @State private var numbers = [String]()
  let generator = NumberGenerator(in: 1...100)

  var body: some View {
    List(numbers, id: \.self, rowContent: Text.init)
      .task {
        await generateNumbers()
      }
  }

  func generateNumbers() async {
    for await number in generator {
      numbers.insert("\(numbers.count + 1). \(number)", at: 0)
    }
  }
}
```

Notice how the **generateNumbers()** method at the end doesn't actually have any way of exiting? That's because it will exit automatically when **generator** stops returning values, which will happen when the task is cancelled, and *that* will happen when **DetailView** is dismissed – it takes no special work from us.

Tip: The **task()** modifier accepts a **priority** parameter if you want fine-grained control over your task's priority. For example, use **.task(priority: .low)** to create a low-priority task.

Is it efficient to create many tasks?

Previously I talked about the concept of *thread explosion*, which is when you create many more threads than CPU cores and the system struggles to manage them effectively. However, Swift's tasks are implemented very differently from threads, and so are significantly less likely to cause performance problems when used in large numbers, and in fact one of the Swift team who worked on it said that unless you're creating over 10,000 tasks **it's not worth worrying about the impact of so many tasks**.

That doesn't mean creating so many tasks is necessarily the best idea. You might think that's hard to do, but a task group calling **addTask()** inside a loop might create several hundred or even several *thousand* depending on what you were looping over. And that's okay. Again, even if you're creating well over 10,000 tasks it's not likely to cause a problem as long as you *know* that's what you're doing – if that's the architectural choice you're making after evaluating the alternative.

So, broadly speaking you should feel free to create as many tasks as you want, but if you ever find yourself creating tasks to transform elements inside huge arrays you might want to double-check your performance using something like Instruments.

Chapter 5
Actors

What is an actor and why does Swift have them?

Swift's actors are conceptually like classes that are safe to use in concurrent environments. This safety is made possible because Swift automatically ensures no two pieces of code attempt to access an actor's data at the same time – it is made impossible by the compiler, rather than requiring developers to write boilerplate code using systems such as locks.

In the following chapters we're going to explore more about how actors work and when you should use them, but here is the least you need to know:

1. Actors are created using the **actor** keyword. This is a concrete nominal type in Swift, like structs, classes, and enums.
2. Like classes, actors are reference types. This makes them useful for sharing state in your program.
3. They have many of the same features as classes: you can give them properties, methods (async or otherwise), initializers, and subscripts, they can conform to protocols, and they can be generic.
4. Actors do not support inheritance, so they cannot have convenience initializers, and do not support either **final** or **override**.
5. All actors automatically conform to the **Actor** protocol, which no other type can use. This allows you to write code restricted to work only with actors.

As well as those, there is one more behavior of actors that lies at the center of their existence: if you're attempting to read a variable property or call a method on an actor, and you're doing it from outside the actor itself, you must do so asynchronously using **await**.

I'll explain why in a moment, but I want to show you a brief snippet of code first so you can see what I mean:

```
actor User {
  var score = 10
```

```
  func printScore() {
    print("My score is \(score)")
  }

  func copyScore(from other: User) async {
    score = await other.score
  }
}

let actor1 = User()
let actor2 = User()

await print(actor1.score)
await actor1.copyScore(from: actor2)
```

You can see several things in action there:

1. The new **User** type is created using the **actor** keyword rather than **struct** or **class**.
2. It can have properties and methods just like structs or classes.
3. The **printScore()** method can access the local **score** property just fine, because it's our actor's method reading its own property.
4. But in **copyScore(from:)** we're attempting to read the score from another user, and we can't read their **score** property without marking the request with **await**.
5. Code from *outside* the actor also needs to use **await**.

The reason the **await** call is needed in **copyScore(from:)** is central to the reasons actors are needed at all. You see, rather than just letting us poke around in an actor's mutable state, Swift silently translates that request into what is effectively a message that goes into the actor's message inbox: "please let me know your score as soon as you can."

If the actor is currently idle it can respond with the score straight away and our code continues

no different from if we had used a class or a struct. But the actor might also have multiple other messages waiting in its inbox that it needs to deal with first, so our **score** request has to wait. Eventually our request makes it to the top of the inbox and it will be dealt with, and the **copyScore(from:)** method will continue.

This means several things:

1. Actors are effectively operating a private serial queue for their message inbox, taking requests one at a time and fulfilling them. This executes requests in the order they were received, but you can also use task priority to escalate requests.
2. Only one piece of code at a time can access an actor's mutable state unless you specifically mark some things as being unprotected. Swift calls this *actor isolation*.
3. Just like regular **await** calls, reading an actor's property or method marks a potential suspension point – we might get a value back immediately, but it might also take a little time.
4. Any state that is *not* mutable – i.e., a constant property – can be accessed without **await**, because it's always going to be safe.

In practice, the rule to remember is this: if you are writing code inside an actor's method, you can read other properties on that actor and call its synchronous methods without using **await**, but if you're trying to use that data from *outside* the actor **await** is required even for synchronous properties and methods. Think of situations where using **self** is possible: if you could **self** you don't need **await**.

Writing properties from outside an actor is not allowed, with or without `await`.

Of course, the real question here is why Swift needs actors at all – what's their fundamental purpose? And the answer is straightforward: if you ever need to make sure that access to some object is restricted to a single task at a time, actors are perfect.

This is more common than you might think – yes, UI work should be restricted to the main thread, but you probably also want to restrict database access so that you can't write conflicting data, for example. There are also times when simultaneous concurrent access to

data can cause *data races* – when the outcome of your work depends on the order in which tasks complete. These errors are particularly nasty to find and fix, but with actors such data races become impossible.

Tip: Creating an instance of an actor has no extra performance cost compared to creating an instance of a class; the only performance difference comes when trying to access the protected state of an actor, which might trigger task suspension.

How to create and use an actor in Swift

Creating and using an actor in Swift takes two steps: create the type using **actor** rather than **class** or **struct**, then use **await** when accessing its properties or methods externally. Swift takes care of everything else for us, including ensuring that properties and methods must be accessed safely.

Let's look at a simple example: a URL cache that remembers the data for each URL it downloads. Here's how that would be created and used:

```
actor URLCache {
  private var cache = [URL: Data]()

  func data(for url: URL) async throws -> Data {
    if let cached = cache[url] {
      return cached
    }

    let (data, _) = try await URLSession.shared.data(from: url)
    cache[url] = data
    return data
  }
}

@main
struct App {
  static func main() async throws {
    let cache = URLCache()

    let url = URL(string: "https://apple.com")!
    let apple = try await cache.data(for: url)
```

```
        let dataString = String(decoding: apple, as: UTF8.self)
        print(dataString)
    }
}
```

I marked its internal **cache** dictionary as **private**, so the only way we can access cached data is using the **data(for:)** method. This provides some degree of safety, because we might do some sort of special work inside the method that would be bypassed by accessing the property directly.

However, the *real* protection here is that the property and method are both encapsulated inside an *actor*, which means only a single thread can use them at any given time. In practice, this avoids two problems:

1. Attempting to read from a dictionary at the same time we're writing to it, which can cause your app to crash.
2. Two or more simultaneous requests for the same uncached URL coming in, forcing our code to fetch and store the same data repeatedly. This is a data race: whether we make two requests or one depends on the exact way our code is executed.

If we didn't have an actor here – if we had used **class URLCache** or **struct URLCache** – then we would need to solve those two problems ourselves. It's not *hard*, at least not in a simple way, but it's error-prone and boring to do, so it's great to be able to hand this work over to the Swift compiler to do for us.

However, this ease of use does come with some extra responsibility: it's really important you keep in mind the serial queue behavior of actors, because it's entirely possible you can create massive speed bumps in your code just because you wrote **actor** rather than **class**. Think about the URL cache we just made, for example – just by using **actor** rather than **class** when we made it, we forced it to load only a single URL at a time. If that's what you want then you're all set, but if not then you'll be wondering why all its requests are handled one by one.

The canonical example of why data races are problematic – the one that is often taught in

computer science degrees – is about *bank accounts*, because here data races can result in serious real-world problems. To see why, here's an example **BankAccount** class that handles sending and receiving money:

```
class BankAccount {
  var balance: Decimal

  init(initialBalance: Decimal) {
    balance = initialBalance
  }

  func deposit(amount: Decimal) {
    balance = balance + amount
  }

  func transfer(amount: Decimal, to other: BankAccount) {
    // Check that we have enough money to pay
    guard balance > amount else { return }

    // Subtract it from our balance
    balance = balance - amount

    // Send it to the other account
    other.deposit(amount: amount)
  }
}

let firstAccount = BankAccount(initialBalance: 500)
let secondAccount = BankAccount(initialBalance: 0)
firstAccount.transfer(amount: 500, to: secondAccount)
```

That's a class, so Swift won't do anything to stop us from accessing the same piece of data

multiple times. So, what could actually happen here?

Well, in the worst case two parallel calls to **transfer()** would be called on the same **BankAccount** instance, and the following would occur:

1. The first would check whether the balance was sufficient for the transfer. It is, so the code would continue.
2. The second would also check whether the balance was sufficient for the transfer. It still is, so the code would continue.
3. The first would then subtract the amount from the balance, and deposit it in the other account.
4. The second would then subtract the amount from the balance, and deposit it in the other account.

Do you see the problem there? Well, what happens if the account we're transferring from contains $100, and we're asked to transfer $80 to the other account? If we follow the steps above, both calls to **transfer()** will happen in parallel and see that there was enough for the transfer to take place, then both will transfer the money across. The end result is that our check for sufficient funds wasn't useful, and one account ends up with -$60 – something that might incur fees, or perhaps not even be allowed depending on the type of account they have.

If we switch this type to be an actor, that problem goes away. This means using **actor BankAccount** rather than **class BankAccount**, but also using **async** and **await** because we can't directly call **deposit()** on the other bank account and instead need to post the request as a message to be executed later.

Here's how that looks:

```
actor BankAccount {
  var balance: Decimal

  init(initialBalance: Decimal) {
    balance = initialBalance
```

```
    }

    func deposit(amount: Decimal) {
        balance = balance + amount
    }

    func transfer(amount: Decimal, to other: BankAccount) async {
        // Check that we have enough money to pay
        guard balance > amount else { return }

        // Subtract it from our balance
        balance = balance - amount

        // Send it to the other account
        await other.deposit(amount: amount)
    }
}

let firstAccount = BankAccount(initialBalance: 500)
let secondAccount = BankAccount(initialBalance: 0)
await firstAccount.transfer(amount: 500, to: secondAccount)
```

With that change, our bank accounts can no longer fall into negative values by accident, which avoids a potentially nasty result.

In other places, actors can prevent bizarre results that ought to be impossible. For example, what would happen if our example was a basketball team rather than a bank account, and we were transferring players rather than money? Without actors we could end up in the situation where we transfer the same player twice – Team A would end up without them, and Team B would have them twice!

How to make function parameters isolated

Any properties and methods that belong to an actor are isolated to that actor, but you can make *external* functions isolated to an actor if you want. This allows the function to access actor-isolated state as if it were inside that actor, without needing to use **await**.

Here's a simple example so you can see what I mean:

```
actor DataStore {
  var username = "Anonymous"
  var friends = [String]()
  var highScores = [Int]()
  var favorites = Set<Int>()

  init() {
    // load data here
  }

  func save() {
    // save data here
  }
}

func debugLog(dataStore: isolated DataStore) {
  print("Username: \(dataStore.username)")
  print("Friends: \(dataStore.friends)")
  print("High scores: \(dataStore.highScores)")
  print("Favorites: \(dataStore.favorites)")
}

let data = DataStore()
```

```
await debugLog(dataStore: data)
```

That creates a **DataStore** actor with various properties plus a couple of placeholder methods, then creates a **debugLog()** method that prints those *without* using **await** – they can be accessed directly. Notice the addition of the **isolated** keyword in the function signature; that's what allows this direct access, and it even allows the function to *write* to those properties too.

Using **isolated** like this does *not* bypass any of the underlying safety or implementation of actors – there can still only be one thread accessing the actor at any one time. What we've done just pushes that access out by a level, because now the whole function must be run on that actor rather than just individual lines inside it. In practice, this means **debugLog(dataStore:)** needs to be called using **await**.

This approach has an important side effect: because the whole function is now isolated to the actor, it must be called using **await** even though it isn't marked as async. This makes the function itself a single potential suspension point rather than individual accesses to the actor being suspension points.

In case you were wondering, you can't have two isolation parameters, because it wouldn't really make sense – which one is executing the function?

How to make parts of an actor nonisolated

All methods and mutable properties inside an actor are isolated to that actor by default, which means they cannot be accessed directly from code that's external to the actor. Access to constant properties is automatically allowed because they are inherently safe from race conditions, but if you want you can make some methods excepted by using the **nonisolated** keyword.

Actor methods that are non-isolated can access other non-isolated state, such as constant properties or other methods that are marked non-isolated. However, they *cannot* directly access *isolated state* like an isolated actor method would; they need to use **await** instead.

To demonstrate non-isolated methods, we could write a **User** actor that has three properties: two constant strings for their username and password, and a variable Boolean to track whether they are online. Because **password** is constant, we could write a non-isolated method that returns the hash of that password using CryptoKit, like this:

```
import CryptoKit
import Foundation

actor User {
  let username: String
  let password: String
  var isOnline = false

  init(username: String, password: String) {
    self.username = username
    self.password = password
  }

  nonisolated func passwordHash() -> String {
```

```
        let passwordData = Data(password.utf8)
        let hash = SHA256.hash(data: passwordData)
        return hash.compactMap { String(format: "%02x",
$0) }.joined()
    }
}

let user = User(username: "twostraws", password: "s3kr1t")
print(user.passwordHash())
```

I'd like to pick out a handful of things in that code:

1. Marking **passwordHash()** as **nonisolated** means that we can call it externally without using **await**.
2. We can also use **nonisolated** with computed properties, which in the previous example would have made **nonisolated var passwordHash: String**. Stored properties may not be non-isolated.
3. Non-isolated properties and methods can access only other non-isolated properties and methods, which in our case is a constant property. Swift will not let you ignore this rule.

Non-isolated methods are particularly useful when dealing with protocol conformances such as **Hashable** and **Codable**, where we must implement methods to be run from outside the actor.

For example, if we wanted to make our **User** actor conform to **Codable**, we'd need to implement **encode(to:)** ourselves as a non-isolated method like this:

```
actor User: Codable {
    enum CodingKeys: CodingKey {
        case username, password
    }

    let username: String
    let password: String
```

```
  var isOnline = false

  init(username: String, password: String) {
    self.username = username
    self.password = password
  }

  nonisolated func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(username, forKey: .username)
    try container.encode(password, forKey: .password)
  }
}

let user = User(username: "twostraws", password: "s3kr1t")

if let encoded = try? JSONEncoder().encode(user) {
  let json = String(decoding: encoded, as: UTF8.self)
  print(json)
}
```

How to use @MainActor to run code on the main queue

@MainActor is a global actor that uses the main queue for executing its work. In practice, this means methods or types marked with **@MainActor** can (for the most part) safely modify the UI because it will always be running on the main queue, and calling **MainActor.run()** will push some custom work of your choosing to the main actor, and thus to the main queue. At the simplest level both of these features are straightforward to use, but as you'll see there's a lot of complexity behind them.

First, let's look at using **@MainActor**, which automatically makes a single method or all methods on a type run on the main actor. This is particularly useful for any types that exist to update your user interface, such as **ObservableObject** classes.

For example, we could create a observable object with two **@Published** properties, and because they will both update the UI we would mark the whole class with **@MainActor** to ensure these UI updates always happen on the main actor:

```
@MainActor
class AccountViewModel: ObservableObject {
  @Published var username = "Anonymous"
  @Published var isAuthenticated = false
}
```

In fact, this set up is so central to the way **ObservableObject** works that SwiftUI bakes it right in: whenever you use **@StateObject** or **@ObservedObject** inside a view, Swift will ensure that the whole view runs on the main actor so that you can't accidentally try to publish UI updates in a dangerous way. Even better, no matter what property wrappers you use, the **body** property of your SwiftUI views is always run on the main actor.

Does that mean you don't need to explicitly add **@MainActor** to observable objects? Well, no – there are still benefits to using **@MainActor** with these classes, not least if they are using

async/**await** to do their own asynchronous work such as downloading data from a server.

So, my recommendation is simple: even though SwiftUI ensures main-actor-ness when using **@ObservableObject**, **@StateObject**, and SwiftUI view **body** properties, it's a good idea to add the **@MainActor** attribute to all your observable object classes to be absolutely sure all UI updates happen on the main actor. If you need certain methods or computed properties to opt out of running on the main actor, use **nonisolated** as you would do with a regular actor.

Important: I've said it previously, but it's worth repeating: you should *not* attempt to use actors for your observable objects, because they must do their UI updates on the main actor rather than a custom actor.

More broadly, *any* type that has **@MainActor** objects as properties will also implicitly be **@MainActor** using *global actor inference* – a set of rules that Swift applies to make sure global-actor-ness works without getting in the way too much. I'll cover these rules in the next chapter, because they are quite precise.

The magic of **@MainActor** is that it automatically forces methods or whole types to run on the main actor, a lot of the time without any further work from us. Previously we needed to do it by hand, remembering to use code like **DispatchQueue.main.async()** or similar every place it was needed, but now the compiler does it for us automatically.

Be careful: **@MainActor** is really helpful to make code run on the main actor, but it's not *foolproof*. For example, if you have a **@MainActor** class then in theory all its methods will run on the main actor, but one of those methods could trigger code to run on a background task. For example, if you're using Face ID and call **evaluatePolicy()** to authenticate the user, the completion handler will be called on a background thread even though that code is still within the **@MainActor** class.

If you *do* need to spontaneously run some code on the main actor, you can do that by calling **MainActor.run()** and providing your work. This allows you to safely push work onto the main actor no matter where your code is currently running, like this:

```
func couldBeAnywhere() async {
```

```
  await MainActor.run {
    print("This is on the main actor.")
  }
}

await couldBeAnywhere()
```

You can send back nothing from **run()** if you want, or send back a value like this:

```
func couldBeAnywhere() async {
  let result = await MainActor.run { () -> Int in
    print("This is on the main actor.")
    return 42
  }

  print(result)
}

await couldBeAnywhere()
```

Even better, if that code was already running on the main actor then the code is executed immediately – it won't wait until the next run loop in the same way that **DispatchQueue.main.async()** would have done.

If you wanted the work to be sent off to the main actor *without* waiting for its result to come back, you can place it in a new task like this:

```
func couldBeAnywhere() {
  Task {
    await MainActor.run {
      print("This is on the main actor.")
    }
  }
```

```
  // more work you want to do
}

couldBeAnywhere()
```

Or you can also mark your task's closure as being **@MainActor**, like this:

```
func couldBeAnywhere() {
  Task { @MainActor in
    print("This is on the main actor.")
  }

  // more work you want to do
}

couldBeAnywhere()
```

This is particularly helpful when you're inside a synchronous context, so you need to push work to the main actor without using the **await** keyword.

Important: If your function is already running on the main actor, using **await MainActor.run()** will run your code immediately without waiting for the next run loop, but using **Task** as shown above *will* wait for the next run loop.

You can see this in action in the following snippet:

```
@MainActor class ViewModel: ObservableObject {
  func runTest() async {
    print("1")

    await MainActor.run {
      print("2")
```

```
        Task { @MainActor in
            print("3")
        }

        print("4")
    }

    print("5")
  }
}
```

That marks the whole type as using the main actor, so the call to **MainActor.run()** will run immediately when **runTest()** is called. However, the inner **Task** will *not* run immediately, so the code will print 1, 2, 4, 5, 3.

Although it's possible to create your own global actors, I think we should probably avoid doing so until we've had sufficient chance to build apps using what we already have.

Understanding how global actor inference works

Apple explicitly annotates many of its types as being **@MainActor**, including most UIKit types such as **UIView** and **UIButton**. However, there are many places where types gain main-actor-ness *implicitly* through a process called *global actor inference* – Swift applies **@MainActor** automatically based on a set of predetermined rules.

There are five rules for global actor inference, and I want to tackle them individually because although they start easy they get more complex.

First, if a class is marked **@MainActor**, all its subclasses are also automatically **@MainActor**. This follows the principle of least surprise: if you inherit from a **@MainActor** class it makes sense that your subclass is also **@MainActor**.

Second, if a method in a class is marked **@MainActor**, any overrides for that method are also automatically **@MainActor**. Again, this is a natural thing to expect – you overrode a **@MainActor** method, so the only safe way Swift can call that override is if it's also **@MainActor**.

Third, any struct or class using a property wrapper with **@MainActor** for its wrapped value will automatically be **@MainActor**. This is what makes **@StateObject** and **@ObservedObject** convey main-actor-ness on SwiftUI views that use them – if you use either of those two property wrappers in a SwiftUI view, the whole view becomes **@MainActor** too. At the time of writing Xcode's generated interface for those two property wrappers don't show that they are annotated as **@MainActor**, but I've been assured they definitely are – hopefully Xcode can make that work better in the future.

The fourth rule is where the difficulty ramps up a little: if a protocol declares a method as being **@MainActor**, any type that conforms to that protocol will have that same method automatically be **@MainActor** *unless* you separate the conformance from the method.

What this means is that if you make a type conform to a protocol with a **@MainActor** method,

and add the required method implementation at the same time, it is implicitly **@MainActor**. However, if you separate the conformance and the method implementation, you need to add **@MainActor** by hand.

Here's that in code:

```
// A protocol with a single `@MainActor` method.
protocol DataStoring {
  @MainActor func save()
}

// A struct that does not conform to the protocol.
struct DataStore1 { }

// When we make it conform and add save() at the same time, our
method is implicitly @MainActor.
extension DataStore1: DataStoring {
  func save() { } // This is automatically @MainActor.
}

// A struct that conforms to the protocol.
struct DataStore2: DataStoring { }

// If we later add the save() method, it will *not* be
implicitly @MainActor so we need to mark it as such ourselves.
extension DataStore2 {
  @MainActor func save() { }
}
```

As you can see, we need to explicitly use **@MainActor func save()** in **DataStore2** because the global actor inference does not apply there. Don't worry about forgetting it, though – Xcode will automatically check the annotation is there, and offer to add **@MainActor** if it's missing.

The fifth and final rule is most complex of all: if the whole protocol is marked with **@MainActor**, any type that conforms to that protocol will also automatically be **@MainActor** *unless* you put the conformance separately from the main type declaration, in which case only the methods are **@MainActor**.

In attempt to make this clear, here's what I mean:

```
// A protocol marked as @MainActor.
@MainActor protocol DataStoring {
  func save()
}

// A struct that conforms to DataStoring as part of its primary
type definition.
struct DataStore1: DataStoring { // This struct is
automatically @MainActor.
  func save() { } // This method is automatically @MainActor.
}

// Another struct that conforms to DataStoring as part of its
primary type definition.
struct DataStore2: DataStoring { } // This struct is
automatically @MainActor.

// The method is provided in an extension, but it's the same as
if it were in the primary type definition.
extension DataStore2 {
  func save() { } // This method is automatically @MainActor.
}

// A third struct that does *not* conform to DataStoring in its
primary type definition.
```

```
struct DataStore3 { } // This struct is not @MainActor.

// The conformance is added as an extension
extension DataStore3: DataStoring {
  func save() { } // This method is automatically @MainActor.
}
```

I realize that might sound obscure, but it makes sense if you put it into a real-world context. For example, let's say you were working with the **DataStoring** protocol we defined above – what would happen if you modified one of Apple's types so that it conformed to it?

If conformance to a **@MainActor** protocol retroactively made the whole of Apple's type **@MainActor** then you would have dramatically altered the way it worked, probably breaking all sorts of assumptions made elsewhere in the system. If it's *your* type – a type you're creating from scratch in your own code – then you *can* add the protocol conformance as you make the type and therefore isolate the entire type to **@MainActor**, because it's your choice.

What is actor hopping and how can it cause problems?

When a thread pauses work on one actor to start work on another actor instead, we call it *actor hopping*, and it will happen any time one actor calls another.

Behind the scenes, Swift manages a group of threads called the *cooperative thread pool*, creating as many threads as there are CPU cores so that we can't be hit by thread explosion. Actors guarantee that they can be running only one method at a time, but they don't care which thread they are running on – they will automatically move between threads as needed in order to balance system resources.

Actor hopping with the cooperative pool is fast – it will happen automatically, and we don't need to worry about it. However, the main thread is *not* part of the cooperative thread pool, which means actor code being run from the main actor will require a context switch, which will incur a performance penalty if done too frequently.

You can see the problem caused by frequent actor hopping in this toy example code:

```
actor NumberGenerator {
  var lastNumber = 1

  func getNext() -> Int {
    defer { lastNumber += 1 }
    return lastNumber
  }

  @MainActor func run() async {
    for _ in 1...100 {
      let nextNumber = await getNext()
      print("Loading \(nextNumber)")
    }
```

```
    }
}

let generator = NumberGenerator()
await generator.run()
```

In that code, the **run()** method must take place on the main actor because it has the **@MainActor** attribute attached to it, however the **getNext()** method will run somewhere on the cooperative pool, meaning that Swift will need to perform frequent context switching from to and from the main actor inside the loop.

In practice, your code is more likely to look like this:

```
// An example piece of data we can show in our UI
struct User: Identifiable {
    let id: Int
}

// An actor that handles serial access to a database
actor Database {
    func loadUser(id: Int) -> User {
        // complex work to load a user from the database
        // happens here; we'll just send back an example
        User(id: id)
    }
}

// An observable object that handles updating our UI
@MainActor
class DataModel: ObservableObject {
    @Published var users = [User]()
    var database = Database()
```

```
  // Load all our users, updating the UI as each one
  // is successfully fetched
  func loadUsers() async {
    for i in 1...100 {
      let user = await database.loadUser(id: i)
      users.append(user)
    }
  }
}

// A SwiftUI view showing all the users in our data model
struct ContentView: View {
  @StateObject var model = DataModel()

  var body: some View {
    List(model.users) { user in
      Text("User \(user.id)")
    }
    .task {
      await model.loadUsers()
    }
  }
}
```

When that runs, the **loadUsers()** method will run on the main actor, because the whole **DataModel** class must run there – it has been annotated with **@MainActor** to avoid publishing changes from a background thread. However, the database's **loadUser()** method will run somewhere on the cooperative pool: it might run on thread 3 the first time it's called, thread 5 the second time, thread 8 the third time, and so on; Swift will take care of that for us.

This means when our code runs it will repeatedly hop to and from the main actor, meaning there's a significant performance cost introduced by all the context switching.

The solution here is to avoid all the switches by running operations in batches – hop to the cooperative thread pool once to perform all the actor work required to load many users, then process those batches on the main actor. The batch size could potentially load all users at once depending on your need, but even batch sizes of two would halve the context switches compared to individual fetches.

For example, we could rewrite our previous example like this:

```
struct User: Identifiable {
  let id: Int
}

actor Database {
  func loadUsers(ids: [Int]) -> [User] {
    // complex work to load users from the database
    // happens here; we'll just send back examples
    ids.map { User(id: $0) }
  }
}

@MainActor
class DataModel: ObservableObject {
  @Published var users = [User]()
  var database = Database()

  func loadUsers() async {
    let ids = Array(1...100)

    // Load all users in one hop
    let newUsers = await database.loadUsers(ids: ids)

    // Now back on the main actor, update the UI
    users.append(contentsOf: newUsers)
```

```
    }
}

struct ContentView: View {
    @StateObject var model = DataModel()

    var body: some View {
        List(model.users) { user in
            Text("User \(user.id)")
        }
        .task {
            await model.loadUsers()
        }
    }
}
```

Notice how the SwiftUI view is identical – we're just rearranging our internal data access to be more efficient.

What's the difference between actors, classes, and structs?

Swift provides four concrete nominal types for defining custom objects: actors, classes, structs, and enums. Each of these works a little differently from the others, but the first three might seem similar so it's worth spending a little time clarifying what they have in common and where their differences are.

Tip: Ultimately, which you use depends on the exact context you're working in, and you will need them all at some point.

Actors:

1. Are reference types, so are good for shared mutable state.
2. Can have properties, methods, initializers, and subscripts.
3. Do not support inheritance.
4. Automatically conform to the **Actor** protocol.
5. Automatically conform to the **AnyObject** protocol, and can therefore conform to **Identifiable** without adding an explicit **id** property.
6. Can have a deinitializer.
7. Cannot have their public properties and methods directly accessed externally; we must use **await**.
8. Can execute only one method at a time, regardless of how they are accessed.

Classes:

1. Are reference types, so are good for shared mutable state.
2. Can have properties, methods, initializers, and subscripts.
3. Support inheritance.
4. Cannot conform to the **Actor** protocol.
5. Automatically conform to the **AnyObject** protocol, and can therefore conform to **Identifiable** without adding an explicit **id** property.

6. Can have a deinitializer.
7. Can have their public properties and methods directly accessed externally.
8. Can potentially be executing severals methods at a time.

Structs:

1. Are value types, so are copied rather than shared.
2. Can have properties, methods, initializers, and subscripts.
3. Do not support inheritance.
4. Cannot conform to the **Actor** protocol.
5. Cannot conform to the **AnyObject** protocol; if you want to add **Identifiable** conformance you must add an **id** property yourself.
6. Cannot have a deinitializer.
7. Can have their public properties and methods directly accessed externally.
8. Can potentially be executing severals methods at a time.

You might think the advantages of actors are such that they should be used everywhere classes are currently used, but that is a bad idea. Not only do you lose the ability for inheritance, but you'll also cause a huge amount of pain for yourself because every single external property access needs to use **await**.

However, there are certainly places where actors are a natural fit. For example, if you were previously creating serial queues to handle specific workflows, they can be replaced almost entirely with actors – while also benefiting from increased safety and performance. So, if you have some work that absolutely must work one at a time, such as accessing a database, then trying converting it into something like a database actor.

There is one area in particular where using actors rather than classes is going to cause problems, so I really can't say this clearly enough:

Do not use actors for your SwiftUI data models. You should use a class that conforms to the **ObservableObject** protocol instead. If needed, you can optionally also mark that class with **@MainActor** to ensure it does any UI work safely, but keep in mind that using **@StateObject**

or **@ObservedObject** automatically makes a view's code run on the main actor. If you desperately need to be able to carve off some async work safely, you can create a sibling actor – a separate actor that does not use **@MainActor**, but does not directly update the UI.

Important: Do not use an actor for your SwiftUI data models

Swift's actors allow us to share data in multiple parts of our app *without* causing problems with concurrency, because they automatically ensure two pieces of code cannot simultaneously access the actor's protected data.

Actors are an important addition to our toolset, and help us guarantee safe access to data in concurrent environments. However, if you've ever wondered "should I use an actor for my SwiftUI data?", let me answer that as clearly as I can: actors are a *really* bad choice for any data models you use with SwiftUI – anything that conforms to the **ObservableObject** protocol.

SwiftUI updates its user interface on the main actor, which means when we make a class conform to **ObservableObject** we're agreeing that all our work will happen on the main actor. As an example, any time we modify an **@Published** property that *must* happen on the main actor, otherwise we'll be asking for changes to be made somewhere that isn't allowed.

Now think about what would happen if you tried to use a custom actor for your data. Not only would any data writes need to happen on that actor rather than the main actor (thus forcing the UI to update away from the main actor), but any data *reads* would need to happen there too – every time you tried to bind a string to a **TextField**, for example, you'd be asking Swift to simultaneously use the main actor and your custom actor, which doesn't make sense.

The correct solution here is to use a class that conforms to **ObservableObject**, then annotate it with **@MainActor** to ensure it does any UI work safely. If you still find that you need to be able to carve off some async work safely, you can create a sibling actor – a separate actor that does not use **@MainActor**, but does not directly update the UI.

Chapter 6
Solutions

How to download JSON from the internet and decode it into any Codable type

Fetching JSON from the network and using **Codable** to convert it into native Swift objects is probably the most common task for any Swift developer, usually followed by displaying that data in a **List** or **UITableView** depending on whether they are using SwiftUI or UIKit.

Well, using Swift's concurrency features we can write a small but beautiful extension for **URLSession** that makes such work just a single line of code – you just tell iOS what data type to expect and the URL to fetch, and it will do the rest. To add some extra flexibility, we can also provide options to customize decoding strategies for keys, data, and dates, providing sensible defaults for each one to keep our call sites clear for the most common usages.

Here's how it's done:

```
extension URLSession {
  func decode<T: Decodable>(
    _ type: T.Type = T.self,
    from url: URL,
    keyDecodingStrategy: JSONDecoder.KeyDecodingStrategy
= .useDefaultKeys,
    dataDecodingStrategy: JSONDecoder.DataDecodingStrategy
= .deferredToData,
    dateDecodingStrategy: JSONDecoder.DateDecodingStrategy
= .deferredToDate
  ) async throws  -> T {
    let (data, _) = try await data(from: url)

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = keyDecodingStrategy
```

```
        decoder.dataDecodingStrategy = dataDecodingStrategy
        decoder.dateDecodingStrategy = dateDecodingStrategy

        let decoded = try decoder.decode(T.self, from: data)
        return decoded
    }
}
```

That does several things:

1. It's an extension on **URLSession**, so you can go ahead and create your own custom session with a unique configuration if needed.
2. It uses generics, so that it will work with anything that conforms to the **Decodable** protocol – that's half of **Codable**, so if you use **Codable** it will work there too.
3. It uses **T.self** for the default data type, so if Swift can infer your type then you don't need to repeat yourself.
4. It allows all errors to propane to your call site, so you can handle networking and/or decoding errors as needed.

To use the extension in your own code, first define a type you want to work with, then go ahead and call **decode()** in whichever way you need:

```
struct User: Codable {
    let id: UUID
    let name: String
    let age: Int
}

struct Message: Codable {
    let id: Int
    let user: String
    let text: String
}
```

```
do {
  // Fetch and decode a specific type
  let url1 = URL(string: "https://hws.dev/user-24601.json")!
  let user = try await URLSession.shared.decode(User.self,
from: url1)
  print("Downloaded \(user.name)")

  // Infer the type because Swift has a type annotation
  let url2 = URL(string: "https://hws.dev/inbox.json")!
  let messages: [Message] = try await
URLSession.shared.decode(from: url2)
  print("Downloaded \(messages.count) messages")

  // Create a custom URLSession and decode a Double array from
that
  let config = URLSessionConfiguration.default
  config.requestCachePolicy
= .reloadIgnoringLocalAndRemoteCacheData
  let session = URLSession(configuration: config)

  let url3 = URL(string: "https://hws.dev/readings.json")!
  let readings = try await session.decode([Double].self, from:
url3)
  print("Downloaded \(readings.count) readings")
} catch {
  print("Download error: \(error.localizedDescription)")
}
```

As you can see, with that small extension in place it becomes trivial to fetch and decode any type of **Codable** data with just one line of Swift.