# Chapter 8. Structured Concurrency with Coroutines

In the previous chapter, we introduced a new asynchronous programming paradigm—coroutines. When using coroutines, it's important to know how to use suspending functions appropriately; we'll cover that topic in this chapter. As most programs have to deal with exception handling and cancellation, we'll also cover these topics—and you'll see that, in this regard, coroutines have their own set of rules you should be aware of.

The first section of this chapter covers the idiomatic usage of suspending functions. We'll take the example of a hiking app to compare two implementations: one based on threads and the other one based on suspending functions and coroutines. You'll see how this comparison highlights the power of coroutines in some situations.

As is common for most mobile apps, the hiking example requires a *cancellation mechanism*. We'll cover all you need to know about cancellation with coroutines. In order to prepare for most situations, we'll then cover *parallel decomposition* and *supervison*. Using these concepts, you'll be able to implement complex concurrent logic if you need to.

Finally, this chapter ends with an explanation of exception handling with coroutines.

## Suspending Functions

Imagine that you're developing an application to help users plot, plan, track, draw, and share information about hiking. Your users should be able to navigate to any of the hikes they've already completed or that are in progress. Before going out for a given hike, some basic statistics are useful, like:

- Total distance
- The length of the last hike in both time and distance
- The current weather along the trail they chose
- Favorite hikes

Such an application would require various interactions between the client and server(s) for meteorological data and user information. How might we choose to store data for such an application?

We may choose to store this data locally for later use, or on remote servers (which is referred to as *persistence strategies*). Longer-running tasks, especially networking or IO tasks, can take shape with background jobs like reading from a database, a local file, or a protobuf; or querying a remote server. At its core, reading data from a host device will always be faster than reading the same data from the network.

So, the retrieved data may come at variable rates, depending on the nature of the query. Our worker logic must be resilient and flexible enough to support and survive this situation, and tough enough to handle circumstances beyond our control or even awareness.

## Set the Scene

You need to build out a feature that allows users to retrieve their favorite hikes along with the current weather for each of those hikes.

We've already gone ahead and provided some library code of the application described in the beginning of the chapter. The following is a set of classes and functions already made available to you:

```
data class Hike(
   val name: String,
   val miles: Float,
   val ascentInFeet: Int)

class Weather // Implementation removed for brevity

data class HikeData(val hike: Hike, val weather: Weather?)
```

`Weather` isn't a Kotlin data class, because we need a name for a type for the weather attribute for `HikeData` (if we had declared `Weather` as a data class without providing attributes, the code wouldn't compile).

A `Hike`, in this example, is only:

1. A name
2. A total number of miles
3. The total ascent in feet

A `HikeData` pairs a `Hike` object with a *nullable* `Weather` instance (if we couldn't get the weather data for some reason).

We are also provided with the methods to fetch the list of a `Hike` given a user id along with weather data for a hike:

```
fun fetchHikesForUser(userId: String): List<Hike> {
    // implementation removed for brevity
}

fun fetchWeather(hike: Hike): Weather {
    // implementation removed for brevity
}
```

Those two functions might be long-running operations—like querying a database or an API. In order to avoid blocking the UI thread while fetching the list of hikes or the current weather, we'll leverage suspending functions.

We believe that the best way to understand how to use suspending functions is to compare the following:

- A "traditional" approach using threads and `Handler`
- An implementation using suspending functions with coroutines

First we'll show you how the traditional approach has its limitations in some situations, and that it's not easy to overcome them. Then we'll show you how using suspending functions and coroutines changes the way we implement asynchronous logic and how we can solve all the problems we had with the traditional approach.

Let's start with the thread-based implementation.

## Traditional Approach Using java.util.concurrent.ExecutorService

`fetchHikesForUser` and `fetchWeather` functions should be invoked from a background thread. In Android, that might be done in any number of ways. Java has the traditional `Thread` library of course, and the `Executors` framework. The Android standard library has the (now legacy) `AsyncTask`, `HandlerThread`, as well as the `ThreadPoolExecutor` class.

Among all those possibilities, we want to take the best implementation in terms of expressiveness, readability, and control. For those reasons, we decided to leverage the `Executors` framework.

Inside a `ViewModel`, suppose you use one of the factory methods for `ExecutorService` from the `Executors` class to get back a `ThreadPoolExecutor` for performing asynchronous work using the traditional thread-based model.

In the following, we've chosen a *work-stealing* pool. Compared to a simple-thread pool with a blocking queue, a work-stealing pool can reduce contention while keeping a targeted number of threads active. The idea behind this is that enough work queues are maintained so that an overwhelmed worker[1] might have one of its tasks "stolen" by another worker which is less loaded:

```
class HikesViewModel : ViewModel() {
    private val ioThreadPool: ExecutorService =
        Executors.newWorkStealingPool(10)

    fun fetchHikesAsync(userId: String) {
        ioThreadPool.submit {
            val hikes = fetchHikesForUser(userId)
            onHikesFetched(hikes)
        }
    }

    private fun onHikesFetched(hikes: List<Hike>) {
        // Continue with the rest of the view-model logic
        // Beware, this code is executed from a background thread
    }
}
```

When performing IO operations, having 10 threads is reasonable, even on Android devices. In the case of `Executors.newWorkStealingPool`, the actual number of threads grows and shrinks dynamically, depending on the load. Do note, however, that a work-stealing pool makes no guarantees about the order in which submitted tasks are executed.

NOTE

We could also have leveraged the Android primitive `ThreadPoolExecutor` class. More specifically, we could have created our thread pool this way:

```
private val ioThreadPool: ExecutorService =
    ThreadPoolExecutor(
        4,   // Initial pool size
        10,  // Maximum pool size
        1L,
        TimeUnit.SECONDS,
        LinkedBlockingQueue()
    )
```

The usage is then exactly the same. Even if there are subtle differences with the work-stealing pool we initially created, what's important to notice here is how you can submit tasks to the thread pool.

Using a thread pool just for `fetchHikesForUser` could be overkill—especially if you don't invoke `fetchHikesForUser` for different users concurrently. Consider the rest of the implementation that uses an `ExecutorService` for more sophisticated concurrent work, as shown in the following code:

```
class HikesViewModel : ViewModel() {
    // other attributes
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()

    fun fetchHikesAsync(userId: String) { // content hidden }

    private fun onHikesFetched(hikes: List<Hike>) {
        hikes.forEach { hike  ->
            ioThreadPool.submit {
                val weather = fetchWeather(hike)          ❶
                val hikeData = HikeData(hike, weather)    ❷
                hikeDataList.add(hikeData)                ❸
                hikeLiveData.postValue(hikeDataList)      ❹
            }
        }
    }
}
```

For each `Hike`, a new task is submitted. This new task:

❶ Fetches weather information

❷ Stores `Hike` and `Weather` objects inside a `HikeData` container

❸ Adds the `HikeData` instance to an internal list

❹ Notifies the view that the `HikeData` list has changed, which will pass the newly updated state of that list data

We explicitly left a common mistake in the preceding code. Can you spot it? Although it runs fine as is, imagine that we add a public method to add a new hike:

```
fun addHike(hike: Hike) {
    hikeDataList.add(HikeData(hike, null))
    // then fetch Weather and notify view using hikeLiveData
}
```

In step 3 in the `onHikesFetched` method, we added a new element to `hikeDataList` from one of the background threads of `ioThreadPool`. What could go wrong with such a harmless method?

You could try to invoke `addHike` from the main thread while `hikeDataList` is being modified by a background thread.

Nothing enforces the thread from which the public `addHike` is going to be called. In Kotlin on the JVM, a mutable list is backed by an `ArrayList`. However, an `ArrayList` isn't *thread-safe*. Actually, this isn't the only mistake we've made. `hikeDataList` isn't correctly published—there's no guarantee that in step 4 the background thread sees an updated value for `hikeDataList`. There is no *happens before*[2] enforcement here from the Java memory model—the background thread might not see an up-to-date state of `hikeDataList`, even if the main thread put a new element in the list beforehand.

Consequently, the iterator within the `onHikesFetched` chain will throw a `ConcurrentModificationException` when it realizes the collection has been "magically" modified. Populating `hikeDataList` from a background thread isn't safe in this case (see Figure 8-1).

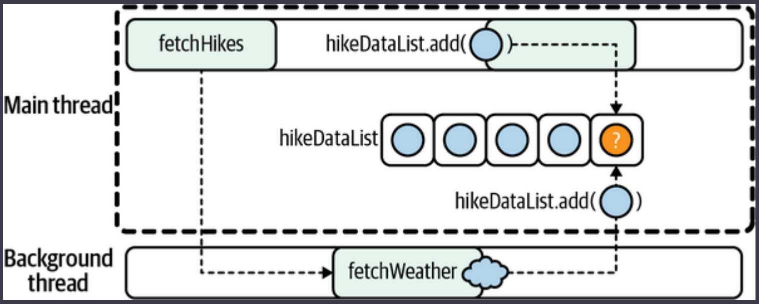

Figure 8-1. `addHike` adds to the existing `hikeDataList` that is already being modified in the background thread.

Falling into this pattern, even when safe, increases the likelihood that habit overtakes sensibility and that during the same day or week or month, this mistake repeats in a less safe circumstance. Consider other team members with edit access to the same codebase and you can see that we quickly lose control.

Thread safety matters anytime multiple threads are attempting to access the same resource at the same time, and it's hard to get right. This is why *defaulting to the main thread*[3] is considered a good practice.

So how would you do this? Are you able to get the background thread to tell the main thread "add this element to this list whenever you can, then notify the view with the updated list of `HikeData`"? For this purpose, you can use the handy `HandlerThread` and `Handler` classes.

## A Reminder About HandlerThread

A `HandlerThread` is a thread to which a “message loop” is attached. It’s an implementation of the producer-consumer design pattern, where the `HandlerThread` is the consumer. A `Handler` sits between the actual message queue and other threads that can send new messages. Internally, the loop that consumes the message queue is created using the `Looper` class (also called “looper”). A `HandlerThread` completes when you invoke its `quit` or `quickSafely` method. Paraphrasing Android’s documentation, the `quit` method causes the handler thread’s looper to terminate without processing any more messages in the message queue. The `quitSafely` method causes the handler thread’s looper to terminate as soon as all remaining messages in the message queue, that are already due to be delivered, have been handled.

Be really careful about remembering to stop a `HandlerThread`. For example, imagine you start a `HandlerThread` within the lifecycle of an activity (say, in an `onCreate` method of a fragment). If you rotate the device, the activity is destroyed and then re-created. A new `HandlerThread` instance is then created and started while the old one is still running, leading to a serious memory leak (see Figure 8-2)!

Figure 8-2. A `HandlerThread` consumes tasks coming from the `MessageQueue`.

On Android, the main thread is a `HandlerThread`. Because creating a `Handler` to post messages to the main thread is very common, a static method on the `Looper` class exists to get the reference on the main thread’s `Looper` instance. Using a `Handler`, you can post a `Runnable` to be executed on the thread that the `Looper` instance associated with the `Handler` is attached to. The Java signature is:

```
public final boolean post(@NonNull Runnable r) { ... }
```

Since a `Runnable` only has one abstract method, `run`, it can be nice and syntactically sweetened in Kotlin using a lambda, as shown in the following code:

```
// Direct translation in Kotlin (though not idiomatic)
handler.post(object: Runnable {
      override fun run() {
         // content of run
      }
   }
)

// ..which can be nicely simplified into:
handler.post {
    // content of `run` method
}
```

In practice, you just create it like this:

```
val handler: Handler = Handler(Looper.getMainLooper())
```

Then you can can utilize the loop handler in the previous example, as shown in the following code:

```
class HikesViewModel : ViewModel() {
    private val ioThreadPool: ExecutorService = Executors.newWorkStealingPool(10)
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()
    private val handler: Handler = Handler(Looper.getMainLooper())

    private fun onHikesFetched(hikes: List<Hike>) {
        hikes.forEach { hike  ->
            ioThreadPool.submit {
                val weather = fetchWeather(hike)
                val hikeData = HikeData(hike, weather)

                // Here we post a Runnable
                handler.post {
                    hikeDataList.add(hikeData)          ❶
                    hikeLiveData.value = hikeDataList   ❷
                }
            }
        }
    }

    // other methods removed for brevity
}
```

This time, we post a `Runnable` to the main thread, in which:

❶ A new `HideData` instance is added to `hikeDataList`.

❷ `hikeLiveData` is given the `hikeDataList` as an updated value. Notice that we can use the highly readable and intuitive assignment operator here: `hikeLiveData.value` `=` `..`, which is nicer than `hikeLiveData.postValue(..)`. This is because the `Runnable` will be executed from the main thread—`postValue` is only useful when updating the value of a `LiveData` from a background thread.

Doing this, all accessors of `hikeDataList` are *thread-confined* to the main thread (see Figure 8-3), eliminating all possible concurrency hazards.

Figure 8-3. The main thread can only access `hikeDataList`.

That's it for the "traditional" approach. Other libraries like *RxJava/RxKotlin* and *Arrow* could have been used to perform essentially the same thing. The logic is made of several steps. You start the first one, giving it a callback containing the set of instructions to run when the background job is done. Each step is connected to the next by the code inside the callbacks. We've discussed it in Chapter 6, and we hope that we've illuminated some potential pitfalls and given you the tools to avoid them.

Interestingly, callback complexity doesn't seem to be an issue in this example—everything is done with two methods, a `Handler` and a `ExecutorService`. However, an insidious situation arises in the following scenario:

A user navigates to a list of hikes, then `fetchHikesAsync` is called on the `ViewModel`. The user just installed the application on a new device;

thus the history isn't in cache, so the app has to access remote APIs to fetch fresh data from some remote service.

Let's assume that the wireless network is slow, but not so slow as to cause IO timeout errors. The view keeps showing that the list is updating, and the user might think that there is in fact a suppressed error, and retry the fetch (which might be available using some refresh UI like a `SwipeRefreshLayout`, an explicit refresh button, or even just using navigation to reenter the UI and presume a fetch will be called implicitly).

Unfortunately, nothing in our implementation anticipates this. When `fetchHikesAsync` is called, a workflow is launched and cannot be stopped. Imagining the worst case, every time a user navigates back and reenters in the hike list view, a new workflow is launched. This is clearly poor design.

A cancellation mechanism might be one possible solution. We might implement a cancellation mechanism by ensuring that every new call of `fetchHikesAsync` cancels any previous in-flight or pending call. Alternatively, you could discard new calls of `fetchHikesAsync` while a previous call is still running. Implementing that in this context requires thoughtfulness and deliberation.

A cancellation mechanism isn't as fire-and-forget as we might find in other flows, because you have to ensure that *every* background thread effectively stops their execution.

As you know from the previous chapter, coroutines and suspending functions can be a great fit here, and in similar circumstances. We chose this hiking app example because we have a great opportunity to use suspending functions.

## Using Suspending Functions and Coroutines

As a reminder, we'll now implement the exact same logic; but this time we'll be using suspending functions and coroutines.

You declare a suspending function when the function may not return immediately. Therefore, any blocking function is eligible to be rewritten as a suspending function.

The `fetchHikesForUser` function is a good example because it blocks the calling thread until it returns a list of `Hike` instances. Therefore, it can be expressed as a suspending function, as shown in the following code:

```
suspend fun hikesForUser(userId: String): List<Hike> {
    return withContext(Dispatchers.IO) {
        fetchHikesForUser(userId)
    }
}
```

We had to pick another name for the suspending function. In this example, blocking calls are prefixed with "fetch" by convention.

Similarly, as shown in Example 8-1, you can declare the equivalent for `fetchWeather`.

**Example 8-1. `fetchWeather` as suspending function**

```kotlin
suspend fun weatherForHike(hike: Hike): Weather {
    return withContext(Dispatchers.IO) {
        fetchWeather(hike)
    }
}
```

Those suspending functions are wrappers around their blocking counterpart. When invoked from inside a coroutine, the `Dispatcher` supplied to the `withContext` function determines which thread pool the blocking call is executed on. Here, `Dispatchers.IO` is a perfect fit and is very similar to the work-stealing pool seen earlier.

---

NOTE

Once you've wrapped blocking calls in suspending blocks like the suspending `weatherForHike` function, you're now ready to use those suspending functions inside coroutines—as you'll see shortly.

Actually, there's a convention with suspending functions to make everyone's life simpler: *a suspending function never blocks the calling thread.* In the case of `weatherForHike`, this is indeed the case, since regardless of which thread invokes `weatherForHike` from within a coroutine, the `withContext(Dispatchers.IO)` statement causes the execution to jump to another thread.[4]

---

Everything we've done using the callback pattern can now fit in a single public `update` method, which reads like procedural code. This is possible thanks to the suspending functions, as shown in Example 8-2.

**Example 8-2. Using suspending functions in the view-model**

```kotlin
class HikesViewModel : ViewModel() {
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()

    fun update() {
        viewModelScope.launch {                                       ❶
            /* Step 1: get the list of hikes */
            val hikes = hikesForUser("userId")                        ❷

            /* Step 2: for each hike, get the weather, wrap into a
             * container, update hikeDataList, then notify view
             * listeners by updating the corresponding LiveData */
            hikes.forEach { hike ->                                   ❸
                launch {
                    val weather = weatherForHike(hike)                ❹
                    val hikeData = HikeData(hike, weather)
                    hikeDataList.add(hikeData)
                    hikeLiveData.value = hikeDataList
                }
            }
        }
    }
}
```

We're going to provide the details of Example 8-2 step by step:

1. When `update` is called, it immediately starts a coroutine, using the `launch` coroutine builder. As you know, a coroutine is never launched out of the blue. As we've seen in Chapter 7, a coroutine must always be started within a `CoroutineScope`. Here, we're using `viewModelScope`.
   Where does this scope come from? The Android Jetpack team from Google know that using Kotlin and coroutines requires a `CoroutineScope`. To ease your life, they maintain Android KTX, which is a set of Kotlin extensions on the Android platform and other APIs. The goal is to use Kotlin idioms while still integrating nicely with the Android framework. They leverage extension functions, lambdas, parameter default values, and coroutines. Android KTX is made of several libraries. In this example, we used *lifecycle-viewmodel-ktx*. To use it in your app, you should add the following to your dependencies listed in your `build.gradle` (use a newer version if available): `implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0"`.
2. The line `val hikes = hikesForUser("userId")` is the first suspension point. The coroutine started by `launch` is stopped until `hikesForUser` returns.
3. You've got your list of `Hike` instances. Now you can *concurrently* fetch the weather data for each of them. We can use a loop and start a new coroutine for each hike using `launch`.
4. `val weather = weatherForHike(hike)` is another suspension point. Each of the coroutines started in the `for` loop will reach this suspension point.

Let's have a closer look at the coroutine started for each `Hike` instance in the following code:

```kotlin
launch {
    val weather = weatherForHike(hike)
    val hikeData = HikeData(hike, weather)
    hikeDataList.add(hikeData)
    hikeLiveData.value = hikeDataList
}
```

Since the parent scope (`viewModelScope`) defaults to the main thread, every single line inside the launch block is executed on the main thread, except the content of the suspending function `weatherForHike`, which uses `Dispatchers.IO` (see Example 8-1). The assignment of `weather` is done on the main thread. Therefore, the usages of `hikeDataList` are confined to the main thread—there are no thread-safety issues. As for `hikeLiveData`, you can use the setter of its `value` (and since we're in Kotlin, that means the assignment operator), instead of `postValue`, since we know we're calling this from the main thread.

WARNING

When using a coroutine scope, you should always be conscious of how it manages your coroutines, especially knowing what `Dispatcher` the scope uses. The following code shows how it's declared in the source code of the library:

```
val ViewModel.viewModelScope: CoroutineScope
  get() {
    val scope: CoroutineScope? = this.getTag(JOB_KEY)
    if (scope != null) {
       return scope
    }
    return setTagIfAbsent(
       JOB_KEY,
       CloseableCoroutineScope(
          SupervisorJob() +  Dispatchers.Main.immediate))
  }
```

As you can see in this example, `viewModelScope` is declared as an extension property on the `ViewModel` class. Even if the `ViewModel` class has absolutely no notion of `CoroutineScope`, declaring it in this manner enables the syntax in our example. Then, an internal store is consulted to check whether a scope has already been created or not. If not, a new one is created using `CloseableCoroutineScope(..)`.[5] For instance, don't pay attention to `SupervisorJob`—we'll explain its role later when we discuss cancellation. What's particularly relevant here is `Dispatchers.Main.immediate`, a variation of `Dispatcher.Main`, which executes coroutines immediately when they are launched from the main thread. Consequently, this scope defaults to the main thread. This is a critical piece of information that you'll need to know moving forward from here.

## Summary of Suspending Functions Versus Traditional Threading

Thanks to suspending functions, asynchronous logic can be written like procedural code. Since the Kotlin compiler generates all the necessary callbacks and boilerplate code under the hood, the code you write using a cancellation mechanism can be much more concise.[6] For example, a coroutine scope that uses `Dispatchers.Main` doesn't need `Handler`s or other communication primitives to pass data to and from a background thread to the main thread, as is still the case with purely multithreaded environments (without coroutines). Actually, all the problems we had in the thread-based approach are now nicely solved using coroutines—and that includes the cancellation mechanism.

Code using coroutines and suspending functions can also be more readable, as there can be far fewer implicit or indirect instructions (like nested calls, or SAM instances, as described in Chapter 6). Moreover, IntelliJ and Android Studio make those suspending calls stand out with a special icon in the margin.

In this section, we only scratched the surface of cancellation. The following section covers all you need to know about cancellation with coroutines.

## Cancellation

Handling task cancellation is a critical part of an Android application. When a user navigates for the first time to the view displaying the list of hikes along with statistics and weather, a decent number of coroutines are started from the view-model. If for some reason the user decides to leave the view, then the tasks launched by the view-model are probably running for nothing. Unless of course the user later navigates back to the view, but it's dangerous to assume that. To avoid wasting resources, a good practice in this scenario is to cancel all ongoing tasks related to views no longer needed. This is a good example of cancellation you might implement yourself, as part of your application design. There's another kind of cancellation: the one that happens when something bad happens. So we'll distinguish the two types here:

*Designed cancellation*

For example, a task that's cancelled after a user taps a "Cancel" button in a custom or arbitrary UI.

*Failure cancellation*

For example, a cancellation that's caused by exceptions, either intentionally (thrown) or unexpectedly (unhandled).

Keep those two types of cancellation in mind, as you'll see that the coroutine framework handles them differently.

## Coroutine Lifecycle

To understand how cancellation works, you need to be aware that a coroutine has a lifecycle, which is shown in Figure 8-4.

Figure 8-4. Coroutine lifecycle.

When a coroutine is created, for example, with the `launch {..}` function with no additional context or arguments, it's created in the `Active` state. That means it starts immediately when `launch` is called. This is also called *eagerly* started. In some situations, you might want to start a coroutine *lazily*, which means it won't do anything until you manually start it. To do this, `launch` and `async` can both take a named argument "start," of type `CoroutineStart`. The default value is `CoroutineStart.DEFAULT` (eager start), but you can use `CoroutineStart.LAZY`, as in the following code:

```
val job = scope.launch(start = CoroutineStart.LAZY) { ... }
// some work
job.start()
```

Don't forget to call `job.start()`! Because when started lazily, a coroutine needs to be explicitly started.[7] You don't have to do this by default, as a coroutine is created in the `Active` state.

When a coroutine is done with its work, it remains in the `Completing` state until all of its children reach the `Completed` state (see Chapter 7). Only then does it reach the `Completed` state. As usual, let's crack open the source code and take a look at the following:

```
viewModelScope.launch {
    launch {
        fetchData()   // might take some time
    }
    launch {
        fetchOtherData()
    }
}
```

This `viewModelScope.launch` completes its work almost instantly: it only starts two child coroutines and does nothing else on its own. It quickly reaches the `Completing` state and moves to the `Completed` state only when the child coroutines complete.

### Coroutine cancellation

While in `Active` or `Completing` state, if an exception is thrown or the logic calls `cancel()`, the coroutine transitions to `Cancelling` state. If required, this is when you perform necessary cleanup. The coroutine remains in this `Cancelling` state until the cleanup job is done with its work. Only then will the coroutine transition to the `Cancelled` state.

### Job holds the state

Internally, all those states of the lifecycle are held by the `Job` of the coroutine. The `Job` doesn't have a property named "state" (whose values would range from "NEW" to "COMPLETED"). Instead, the state is represented by three Booleans (flags): `isActive`, `isCancelled`, and `isCompleted`. Each state is represented by a combination of those flags, as you can see in Table 8-1.

Table 8-1. `Job` states

| **State** | `isActive` | `isCompleted` | `isCancelled` |
|---|---|---|---|
| New (optional initial state) | false | false | false |
| Active (default initial state) | true | false | false |
| Completing (transient state) | true | false | false |
| Cancelling (transient state) | false | false | true |
| Cancelled (final state) | false | true | true |
| Completed (final state) | false | true | false |

As you can see, there is no way to distinguish the `Completing` state from the `Active` state using only those Booleans. Anyway, in most cases what you will really care about is the value of a particular flag, rather than the state itself. For example, if you check for `isActive`, you're actually checking for `Active` and `Completing` states at the same time. More on that in the next section.

## Cancelling a Coroutine

Let's take a look at the following example, where we have a coroutine which simply prints on the console `"job: I'm working.."` twice per second. The parent coroutine waits a little before cancelling this coroutine:

```
val startTime = System.currentTimeMillis()
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    while (true) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
delay(1200)
println("main: I'm going to cancel this job")
job.cancel()
println("main: Done")
```

You can see that the instance of `Job` returned by `launch` has a `cancel()` method. As its name suggests, it cancels the running coroutine. By the way, a `Deferred` instance—which is returned by the `async` coroutine builder—also has this `cancel()` method since a `Deferred` instance is a specialized `Job`.

Back to our example: you might expect this little piece of code to print "job: I'm working.." three times. Actually, the output is:

```
job: I'm working..
job: I'm working..
job: I'm working..
main: I'm going to cancel this job
main: Done
job: I'm working..
job: I'm working..
```

So the child coroutine is still running despite the cancellation from the parent. This is because the child coroutine isn't cooperative with cancellation. There are several ways to change that. The first one is by periodically checking for the cancellation status of the coroutine, using `isActive`, as shown in the following code:

```
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

You can call `isActive` this way because it's an extension property on `CoroutineScope`, as shown in the following code:

```
/**
 * Returns true when the current Job is still active (has not
 * completed and was not cancelled yet).
 */
val CoroutineScope.isActive: Boolean (source)
```

Now that the code is cooperative with cancellation, the result is:

```
job: I'm working..
job: I'm working..
job: I'm working..
main: I'm going to cancel this job
main: Done
```

Using `isActive` is simply reading a Boolean value. Determining whether the job should be stopped, and both the setup and execution of that logic, is your r[.keep-together] esponsibility.

In lieu of `isActive`, `ensureActive` can be used. The difference between `isActive` and `ensureActive` is that the latter immediately throws a `CancellationException` if the job is no longer active.

So `ensureActive` is a drop-in replacement of the following code:

```
if (!isActive) {
    throw CancellationException()
}
```

Similarly to `Thread.yield()`, there is a third possibility: `yield()`, which is a suspending function. In addition to checking the cancellation status of the job, the underlying thread is released and is made available for other coroutines. This is especially useful when performing CPU-intensive computations inside a coroutine using `Dispatchers.Default` (or similar). Placing `yield()` at strategic places, you can avoid exhausting the thread pool. In other words, you probably don't want a coroutine to be too selfish, and keep a core busy with specific contextual responsibilities for an extended period of time, if those resources could be better served in another process. To be more cooperative, a greedy CPU-bound coroutine should `yield()` from time to time, giving other coroutines the opportunity to run.

Those ways of interrupting a coroutine are perfect when the cancellation is happening inside your code. What if you just delegated some work to a third-party library, like an HTTP client?

## Cancelling a Task Delegated to a Third-Party Library

`OkHttp` is a widely deployed HTTP client on Android. If you're not familiar with this library, the following is a snippet taken from the official documentation, to perform an synchronous GET:

```
fun run() {
    val request = Request.Builder()
        .url("https://publicobject.com/helloworld.txt")
        .build()

    client.newCall(request).execute().use { response ->
      if (!response.isSuccessful)
          throw IOException("Unexpected code $response")

      for ((name, value) in response.headers) {
        println("$name: $value")
      }

      println(response.body?.string())
    }
}
```

This example is pretty straightforward. `client.newCall(request)` returns an instance of `Call`. You enqueue an instance of `Callback` while your code proceeds unfazed. Is this cancellable? Yes. A `Call` can be manually cancelled using `call.cancel()`.

When using coroutines, the preceding example is the kind of code you might write inside a coroutine. It would be ideal if this cancellation was done automatically upon cancellation of the coroutine inside of which the HTTP request is done. Otherwise, the following shows what you would have to write:

```
if (!isActive) {
    call.cancel()
    return
}
```

The obvious caveat is that it pollutes your code—not to mention that you could forget to add this check, or have it at the wrong place. There must be a better solution to this.

Thankfully, the coroutine framework comes with functions specifically designed to turn a function that expects a callback into a suspending function. They come in several flavors including `suspendCancellableCoroutine`. The latter is designed to craft a suspending function which is *cooperative with cancellation*.

The following code shows how to create a suspending function as an extension function of `Call`, which is cancellable and suspends until you get the response of your HTTP request, or an exception occurs:

```
suspend fun Call.await() = suspendCancellableCoroutine<ResponseBody?> {
    continuation ->

    continuation.invokeOnCancellation {
        cancel()
    }

    enqueue(object : Callback {
        override fun onResponse(call: Call, response: Response) {
            continuation.resume(response.body)
        }
```

```
            override fun onFailure(call: Call, e: IOException) {
                continuation.resumeWithException(e)
            }
        })
    }
```

If you've never seen code like this, it's natural to be afraid of its off-putting complexity. The great news is that this function is fully generic—it only needs to be written once. You can have it inside a "util" package of your project if you want, or in your parallelism package; or just remember the basics and use some version of it when performing conversions like that.

Before showing the benefits of such a utility method, we owe you a detailed explanation.

In Chapter 7, we explained how the Kotlin compiler generates a `Continuation` instance for each suspending function. The `suspendCancellableCoroutine` function gives you the opportunity to use this instance of `Continuation`. It accepts a lambda with `CancellableContinuation` as receiver, as shown in the following code:

```
public suspend inline fun <T> suspendCancellableCoroutine(
    crossinline block: (CancellableContinuation<T>) -> Unit
): T
```

A `CancellableContinuation` is a `Continuation` that is cancellable. We can register a callback that will be invoked upon cancellation, using `invokeOnCancellation { .. }`. In this case, all we want is to cancel the `Call`. Since we're inside an extension function of `Call`, we add the following code:

```
continuation.invokeOnCancellation {
    cancel()   // Call.cancel()
}
```

After we've specified what should happen upon cancellation of the suspending function, we perform the actual HTTP request by invoking `Call.enqueue()`, giving a `Callback` instance. A suspending function "resumes" or "stops suspending" when the corresponding `Continuation` is resumed, with either `resume` or `resumeWithException`.

When you get the result of your HTTP request, either `onResponse` or `onFailure` will be called on the `Callback` instance you provided. If `onResponse` is called, this is the "happy path." You got a response and you should now resume the continuation with a result of your choice. As shown in Figure 8-5, we chose the body of the HTTP response. Meanwhile, on the "sad path," `onFailure` is called, and `OkHttp API` gives you an instance of an `IOException`.

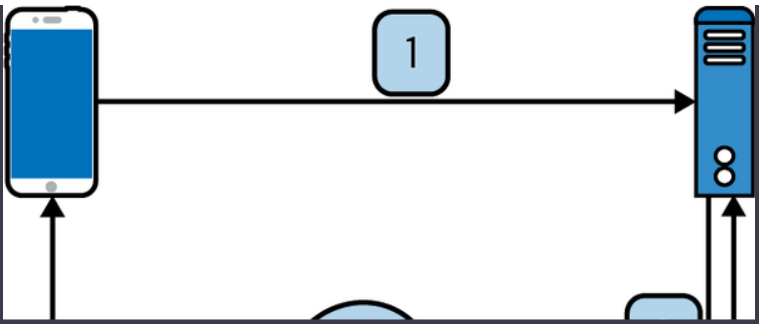

Figure 8-5. (1) First, a device will send an HTTP request to the server. (2) The type of the response being returned will determine what happens next. (3) If the request is a success, then `onResponse` is called. Otherwise, `onFailure` is executed.

It is important to resume the continuation with this exception, using `resumeWithException`. This way, the coroutine framework knows about the failure of this suspending function and will propagate this event all the way up the coroutine hierarchy.

Now, for the best part: a showcase of how to use it inside a coroutine, as shown in the following:

```
fun main() = runBlocking {
    val job = launch {                                          ❶
        val response = performHttpRequest()                     ❷
        println("Got response ${response?.string()}")
    }
    delay(200)                                                  ❸
    job.cancelAndJoin()                                         ❹
    println("Done")
}

val okHttpClient = OkHttpClient()
val request = Request.Builder().url(
    "http://publicobject.com/helloworld.txt"
).build()

suspend fun performHttpRequest(): ResponseBody? {
     return withContext(Dispatchers.IO) {
         val call = okHttpClient.newCall(request)
         call.await()
     }
}
```

❶ We start off by launching a coroutine with `launch`.

❷ Inside the coroutine returned by `launch`, we invoke a suspending function `performHttpRequest`, which uses `Dispatchers.IO`. This suspending function creates a new `Call` instance and then invokes our suspending `await()` on it. At this point, an HTTP request is performed.

3. Concurrently, and while step 2 is done on some thread of `Dispatchers.IO`, our main thread proceeds execution of the main method, and immediately encounters `delay(200)`. The coroutine running on the main thread is suspended for 200 ms.

4. After 200 ms have passed, we invoke `job.cancelAndJoin()`, which is a convenience method for `job.cancel()`, then `job.join()`. Consequently, if the HTTP request takes longer than 200 ms, the coroutine started by `launch` is still in the `Active` state. The suspending `performHttpRequest` hasn't returned yet. Calling `job.cancel()` cancels the coroutine. Thanks to structured concurrency, the coroutine knows about all of its children. The cancellation is propagated all the way down the hierarchy. The `Continuation` of `performHttpRequest` gets cancelled, and so does the HTTP request. If the HTTP request takes less than 200 ms, `job.cancelAndJoin()` has no effect.

No matter how deep in the coroutine hierarchy the HTTP request is performed, if our predefined `Call.await()` is used, the cancellation of the `Call` is triggered if a parent coroutine is cancelled.

## Coroutines That Are Cooperative with Cancellation

You've just seen the various techniques to make a coroutine cancellable. Actually, the coroutine framework has a convention: a well-behaved cancellable coroutine throws a `CancellationException` when it's cancelled. Why? Let's look at this suspending function in the following code:

```kotlin
suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

It is indeed cancellable thanks to the `isActive` check. Imagine that you need to do some cleanup when this function is cancelled. You know when this function is cancelled when `isActive == false`, so you can add a cleanup block at the end, as shown in the following:

```kotlin
suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }

    // cleanup
    if (!isActive) { .. }
}
```

Sometimes you'll need to have the cleanup logic outside the cancelled function; for example, when this function comes from an external dependency. So you need to find a way to notify the call stack that this function is cancelled. Exceptions are perfect for this. This is why the coroutine framework follows this convention of throwing a `CancellationException`. Actually, *all* suspending functions from the *kotlinx.coroutines* package are cancellable and throw `CancellationException` when cancelled. `withContext` is one of them, so you could react to `wasteCpu` cancellation higher in the call stack, as shown in the following code:

```
fun main() = runBlocking {
    val job = launch {
        try {
            wasteCpu()
        } catch (e: CancellationException) {
            // handle cancellation
        }
    }
    delay(200)
    job.cancelAndJoin()
    println("Done")
}
```

If you run this code, you'll find that a `CancellationException` is caught. Even though we never explicitly threw a `CancellationException` from inside `wasteCpu()`, `withContext` did it for us.

---

NOTE

By throwing `CancellationException` only in case of cancellation, the coroutine framework is able to differentiate a simple cancellation from a failure of a coroutine. In the latter case, an exception will be raised that isn't a subtype of `CancellationException`.

If you wish to investigate coroutine cancellation, you can *name* your coroutines and enable debugging of coroutines inside the IDE by adding the VM option `-Dkotlinx.coroutines.debug`. To name a coroutine, simply add a `CoroutineName` context element like so: `val job = launch(CoroutineName("wasteCpu")) {..}`. This way, when catching a `CancellationException`, the stacktrace is much more explicit and begins with the following line:

```
kotlinx.coroutines.JobCancellationException: StandaloneCoroutin
```

---

In the previous example, if you swap `wasteCpu()` with `performHttpRequest()`—the suspending function we made earlier with `suspendCancellableCoroutine`—you will also find that a `CancellationException` is caught. So a suspending function made with `suspendCancellableCoroutine` also throws a `CancellationException` when cancelled.

## delay Is Cancellable

Remember `delay()`? Its signature is shown in the following code:

```
public suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // don't delay
    return suspendCancellableCoroutine sc@ { .. }
}
```

`suspendCancellableCoroutine` again! So this means that anywhere you use `delay`, you're giving a coroutine or suspending function the opportunity to cancel. Building on this, we could rewrite `wasteCpu()` as in the following:

```
private suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
    while (true) {       ❶
        delay(10)        ❷
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

Notice that:

❶ We removed the `isActive` check.

❷ Then we added a simple `delay`, with a small enough sleep time (so the behavior is similar to the previous implementation).

This new version of `wasCpu` turns out to be cancellable just like the original, and throws `CancellationException` when cancelled. This is because this suspending function spends most of its time in the `delay` function.

TIP

To summarize this section, you should strive to make your suspending functions cancellable. A suspending function can be made of several suspending functions. All of them should be cancellable. For example, if you need to perform a CPU-heavy computation, then you should use `yield()` or `ensureActive()` at strategic places. For example:

```
suspend fun compute() = withContext(Dispatchers.Default) {
     blockingCall()  // a regular blocking call, hopefully not blocking too long
     yield()  // give the opportunity to cancel
     anotherBlockingCall()   // because why not
}
```

## Handling Cancellation

In the previous section, you learned that it is possible to react to cancellation using a try/catch statement. However, imagine that inside the code handling the cancellation, you need to call some other suspending functions. You could be tempted to implement the strategy shown in the following code:

```
launch {
    try {
        suspendCall()
    } catch (e: CancellationException) {
       // handle cancellation
       anotherSuspendCall()
   }
}
```

Sadly, the preceding code doesn't compile. Why? Because *a cancelled coroutine isn't allowed to suspend*. This is another rule from the coroutine framework. The solution is to use `withContext(NonCancellable)`, as shown in the following code:

```
launch {
    try {
        suspendCall()
    } catch (e: CancellationException) {
       // handle cancellation
       withContext(NonCancellable) {
            anotherSuspendCall()
       }
   }
}
```

`NonCancellable` is specifically designed for `withContext` to make sure the supplied block of code won't be cancelled.[8]

## Causes of Cancellation

As we've seen before, there are two kinds of cancellation: *by design* and *by failure*. Initially, we said that a failure is encountered when an exception is thrown. It was a bit of an overstatement. You've just seen that, when voluntarily cancelling a coroutine, a `CancellationException` is thrown. This is in fact what distinguishes the two kinds of cancellation.

When cancelling a coroutine `Job.cancel` (by design), the coroutine terminates without affecting its parent. If the parent also has other child coroutines, they also aren't affected by this cancellation. The following code illustrates this:

```
fun main() = runBlocking {
    val job = launch {
        val child1 = launch {
            delay(Long.MAX_VALUE)
        }
        val child2 = launch {
            child1.join()
            println("Child 1 is cancelled")

            delay(100)
            println("Child 2 is still alive!")
        }

        println("Cancelling child 1..")
        child1.cancel()
        child2.join()
        println("Parent is not cancelled")
    }
    job.join()
}
```

The output of this program is:

```
Cancelling child 1..
Child 1 is cancelled
Child 2 is still alive!
Parent is not cancelled
```

`child1` delays forever while `child2` waits for `child1` to proceed. The parent quickly cancels `child1`, and we can see that `child1` is indeed cancelled since `child2` continues its execution. Finally, the output "Parent is not cancelled" is proof that the parent wasn't affected by this cancellation (nor was `child2`, by the way).

On the other hand, in the case of a failure (if an exception different from `CancellationException` was thrown), the default behavior is that the parent gets cancelled with that exception. If the parent also has other child coroutines, they are also cancelled. Let's try to illustrate this. Spoiler alert—don't do what we show in the following:

```
fun main() = runBlocking {
    val scope = CoroutineScope(coroutineContext + Job())   ❶

    val job = scope.launch {                                ❷
        launch {
            try {
                delay(Long.MAX_VALUE)                       ❸
            } finally {
                println("Child 1 was cancelled")
            }
        }

        launch {
            delay(1000)                                     ❹
            throw IOException()
        }
    }
    job.join()                                              ❺
}
```

What we're trying to create is a circumstance in which a child fails after some time, and we want to check that it causes the parent to fail. Then we need to confirm that all other child coroutines of that parent should be cancelled too, assuming that's the cancellation policy we passed.

At first glance, this code looks OK:

❶ We're creating the parent scope.

❷ We're starting a new coroutine inside this scope.

❸ The first child waits indefinitely. If this child gets cancelled, it should print "Child 1 was cancelled" since a `CancellationException` would have been thrown from the `delay(Long.MAX_VALUE)`.

❹ Another child throws an `IOException` after a delay of 1 second.

❺

Wait for the coroutine started in step 2. If you don't do this, the execution of `runBlocking` terminates and the program stops.

Running this program, you indeed see "Child 1 was cancelled," though the program crashes right after with an uncaught `IOException`. Even if you surround `job.join()` with a `try`/`catch` block, you'll still get the crash.

What we're missing here is the origination of the exception. It was thrown from inside a `launch`, which propagates exceptions upward through the coroutine hierarchy until it reaches the parent scope. This behavior cannot be overridden. Once that `scope` sees the exception, it cancels itself and all its children, then propagates the exception to its parent, which is the scope of `runBlocking`.

It's important to realize that trying to catch the exception isn't going to change the fact that the root coroutine of `runBlocking` is going to be cancelled with that exception.

In some cases, you might consider this as an acceptable scenario: any unhandled exception leads to a program crash. However, in other scenarios you might prefer to prevent the failure of `scope` to propagate to the main coroutine. To this purpose, you need to register a `CoroutineExceptionHandler` (CEH):

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, exception ->
        println("Caught original $exception")
    }
    val scope = CoroutineScope(coroutineContext + ceh + Job())

    val job = scope.launch {
         // same as in the previous code sample
    }
}
```

A `CoroutineExceptionHandler` is conceptually very similar to `Thread.UncaughtExceptionHandler`—except it's intended for coroutines. It's a `Context` element, which should be added to the context of a scope or a coroutine. The scope should create its own `Job` instance, as a CEH only takes effect when installed at the top of a coroutine hierarchy. In the preceding example, we added the CEH to the context of the scope. We could very well have added it to the context of the first `launch`, like so:

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, exception ->
        println("Caught original $exception")
    }

    // The CEH can also be part of the scope
    val scope = CoroutineScope(coroutineContext + Job())

    val job = scope.launch(ceh) {
        // same as in the previous code sample
    }
}
```

Running this sample with the exception handler, the output of the program now is:

```
Child 1 was cancelled
Caught original java.io.IOException
```

The program no longer crashes. From inside the CEH implementation, you could retry the previously failed operations.

This example demonstrates that *by default*, the failure of a coroutine causes its parent to cancel itself along with all the other children of that parent. What if this behavior doesn't match your application design? Sometimes the failure of a coroutine is acceptable and doesn't require the cancellation of all other coroutines started inside the same scope. This is called *supervision* in the coroutine framework.

## Supervision

Consider the real-world example of loading a fragment's layout. Each child `View` might require some background processing to be fully constructed. Assuming you're using a scope which defaults to the main thread, and child coroutines for the background tasks, the failure of one of those tasks shouldn't cause the failure of the parent scope. Otherwise, the whole fragment would become unresponsive to the user.

To implement this cancellation strategy, you can use `SupervisorJob`, which is a `Job` for which the failure or cancellation of a child doesn't affect other children; *nor* does it affect the scope itself. A `SupervisorJob` is typically used as a drop-in replacement for `Job` when building a `CoroutineScope`. The resulting scope is then called a "supervisor scope." Such a scope propagates cancellation downward only, as shown in the following code:

```kotlin
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, e -> println("Handled $e") }
    val supervisor = SupervisorJob()
    val scope = CoroutineScope(coroutineContext + ceh + supervisor)
    with(scope) {
        val firstChild = launch {
            println("First child is failing")
            throw AssertionError("First child is cancelled")
        }

        val secondChild = launch {
            firstChild.join()

            delay(10) // playing nice with hypothetical cancellation
            println("First child is cancelled: ${firstChild.isCancelled}, but second one is s
        }

        // wait until the second child completes
        secondChild.join()
    }
}
```

The output of this sample is:

```
First child is failing
Handled java.lang.AssertionError: First child is cancell
First child is cancelled: true, but second one is still 
```

Notice that we've installed a CEH in the context of the scope. Why? The first child throws an exception that is never caught. Even if a supervisor scope isn't affected by the failure of a child, it still propagates unhandled exceptions—which, as you know, might cause the program to crash. This is precisely the purpose of a CEH: to handle uncaught exceptions. Interestingly enough, the CEH could also have been installed into the context of the first `launch`, with the same result, as shown in the following:

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, e -> println("Handled $e") }
    val supervisor = SupervisorJob()
    val scope = CoroutineScope(coroutineContext + supervisor)
    with(scope) {
        val firstChild = launch(ceh) {
            println("First child is failing")
            throw AssertionError("First child is cancelled")
        }

        val secondChild = launch {
            firstChild.join()

            delay(10)
            println("First child is cancelled: ${firstChild.isCancelled}, but second one is s
        }

        // wait until the second child completes
        secondChild.join()
    }
}
```

A CEH is intended to be installed at the top of a coroutine hierarchy, as this is the place where uncaught exceptions can be handled.

In this example, the CEH is installed on a direct child of the coroutine scope. You can install it on a nested coroutine, as in the following:

```
val firstChild = launch {
    println("First child is failing")
    launch(ceh) {
       throw AssertionError("First child is cancelled")
    }
}
```

In this case, the CEH isn't accounted for, and the program might crash.

## supervisorScope Builder

Similarly to `coroutineScope` builder—which inherits the current context and creates a new `Job`—`supervisorScope` creates a `SupervisorJob`. Just like `coroutineScope`, it waits for all children to complete. One crucial difference with `coroutineScope` is that it only propagates cancellation downward, and cancels all children only if it has failed itself. Another difference with `coroutineScope` is how exceptions are handled. We'll delve into that in the next section.

## Parallel Decomposition

Imagine that a suspending function has to run multiple tasks in parallel before returning its result. Take, for example, the suspending function `weatherForHike` from our hiking app at the beginning of this chapter. Fetching the weather could involve multiple APIs, depending on the nature of the data. Wind data and temperature could be fetched separately, from separate data sources.

Assuming you have suspending functions `fetchWind` and `fetchTemperatures`, you could implement `weatherForHike` as follows:

```
private suspend fun weatherForHike(hike: Hike): Weather =
        withContext(Dispatchers.IO) {
   val deferredWind = async { fetchWind(hike) }
   val deferredTemp = async { fetchTemperatures(hike) }
   val wind = deferredWind.await()
   val temperatures = deferredTemp.await()
   Weather(wind, temperatures) // assuming Weather can be built that way
}
```

`async` can also be used in this example because `withContext` provides a `CoroutineScope`—its last argument is a suspending lambda with `CoroutineScope` as the receiver. Without `withContext`, this sample wouldn't compile, because there wouldn't be any scope provided for `async`.

`withContext` is particularly useful when you need to change the dispatcher inside your suspending function. What if you don't need to change your dispatcher? The suspending `weatherForHike` could very well be called from a coroutine which is already dispatched to the IO dispatcher. Then, using `withContext(Dispatchers.IO)` would be redundant. In such situations, you could use `coroutineScope` instead of or in conjunction with `withContext`. It's a `CoroutineScope` builder, which you use as in the following:

```
private suspend fun weatherForHike(hike: Hike): Weather = coroutineScope {
    // Wind and temperature fetch are performed concurrently
    val deferredWind = async(Dispatchers.IO) {
        fetchWind(hike)
    }
    val deferredTemp = async(Dispatchers.IO) {
        fetchTemperatures(hike)
    }
   val wind = deferredWind.await()
   val temperatures = deferredTemp.await()
   Weather(wind, temperatures) // assuming Weather can be built that way
}
```

Here, `coroutineScope` replaces `withContext`. What does this `coroutineScope` do? First of all, have a look at its signature:

```
public suspend fun <R> coroutineScope(block: suspend CoroutineScope.() -> R): R
```

From the official documentation, this function creates a `CoroutineScope` and calls the specified `suspend` block with this scope. The provided scope inherits its `coroutineContext` from the outer scope, but overrides the context's `Job`.

This function is designed for *parallel decomposition* of work. When any child coroutine in this scope fails, this scope fails and all the rest of the children are cancelled (for a different behavior, use supervisorScope). This function returns as soon as the given block and all its child coroutines are completed.

## Automatic Cancellation

Applied to our example, if `fetchWind` fails, the scope provided by `coroutineScope` fails and `fetchTemperatures` is subsequently cancelled. If `fetchTemperatures` involves allocating heavy objects, you can see the benefit of the cancellation.

`coroutineScope` really shines when you need to *perform several tasks concurrently*.

## Exception Handling

Exception handling is an important part of your application design. Sometimes you will just catch exceptions immediately after they're raised, while other times you'll let them bubble up the hierarchy until the dedicated component handles it. To that extent, the language construct `try`/`catch` is probably what you've used so far. However, in the coroutine framework, there's a catch (pun intended). We could have started this chapter with it, but we needed to introduce you to *supervision* and `CoroutineExceptionHandler` first.

### Unhandled Versus Exposed Exceptions

When it comes to exception propagation, uncaught exceptions can be treated by the coroutine machinery as on of the following:

*Unhandled to the client code*

> *Unhandled* exceptions can only be handled by a `CoroutineExceptionHandler`.

*Exposed to the client code*

> *Exposed* exceptions are the ones the client code can handle using `try`/`catch`.

In this matter, we can distinguish two categories of coroutine builders based on how they treat uncaught exceptions:

- Unhandled (`launch` is one of them)
- Exposed (`async` is one of them)

First of all, do note that we're talking about uncaught exceptions. If you catch an exception *before* it is handled by a coroutine builder, everything works as usual—you catch it, so the coroutine machinery isn't aware of it. The following shows an example with `launch` and `try`/`catch`:

```
scope.launch {
    try {
        regularFunctionWhichCanThrowException()
    } catch (e: Exception) {
        // handle exception
    }
}
```

This example works as you would expect, *if* `regularFunctionWhichCanThrowException` is, as its name suggests, a regular function which does not involve, directly or indirectly, other coroutine builders—in which case, special rules can apply (as we'll see later in this chapter).

The same idea applies to the `async` builder, as shown in the following:

```
fun main() = runBlocking {

    val itemCntDeferred = async {
        try {
            getItemCount()
        } catch (e: Exception) {
            // Something went wrong. Suppose you don't care and consider it should return 0.
            0
        }
    }

    val count = itemCntDeferred.await()
    println("Item count: $count")
}

fun getItemCount(): Int {
    throw Exception()
    1
}
```

The output of this program is, as you can easily guess:

```
Item count: 0
```

Alternatively, instead of `try`/`catch`, you could use `runCatching`. It allows for a nicer syntax if you consider that the happy path is when no exception is thrown:

```
scope.launch {
     val result = runCatching {
           regularFunctionWhichCanThrowException()
     }

     if (result.isSuccess) {
         // no exception was thrown
     } else {
         // exception was thrown
```

```
        }
    }
```

Under the hood, `runCatching` is nothing but a `try`/`catch`, returning a `Result` object, which offers some sugar methods like `getOrNull()` and `exceptionOrNull()`, as in the following:

```
/**
 * Calls the specified function [block] with `this` value as its receiver
 * and returns its encapsulated result if invocation was successful,
 * catching and encapsulating any thrown exception as a failure.
 */
public inline fun <T, R> T.runCatching(block: T.() -> R): Result<R> {
    return try {
        Result.success(block())
    } catch (e: Throwable) {
        Result.failure(e)
    }
}
```

Some extension functions are defined on the `Result` and available out of the box, like `getOrDefault` which returns the encapsulated value of the `Result` instance if `Result.isSuccess` is `true` or a provided default value otherwise.

## Exposed Exceptions

As we stated before, you can catch *exposed* exceptions using built-in language support: `try`/`catch`. The following code shows where we have created our own scope inside of which two concurrent tasks, `task1` and `task2`, are started in a `supervisorScope`. `task2` immediately fails:

```
fun main() = runBlocking {

    val scope = CoroutineScope(Job())

    val job = scope.launch {
        supervisorScope {
            val task1 = launch {
                // simulate a background task
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                // try to fetch some count, but it fails
                throw Exception()
                1
            }

            try {
                task2.await()
            } catch (e: Exception) {
                println("Caught exception $e")
            }
            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

The output of this program is:

```
Caught exception java.lang.Exception
Done background task
Program ends
```

This example demonstrates that inside a `supervisorScope`, `async` *exposes* uncaught exceptions in the `await` call. If you don't surround the `await` call with a `try`/`catch` block, then the scope of `supervisorScope` fails and cancels `task1`, then *exposes* to its parent the exception that caused its failure. So this means that even when using a `supervisorScope`, unhandled exceptions in a scope lead to the cancellation of the entire coroutine hierarchy beneath that scope—and the exception is propagated up. By handling the exception the way we did in this example, task 2 fails while task 1 isn't affected.

Interestingly enough, if you don't invoke `task2.await()`, the program executes as if no exception was ever—thrown`task2` silently fails.

Now we'll use the exact same example, but with a `coroutineScope` instead of `supervisorScope`:

```
fun main() = runBlocking {

    val scope = CoroutineScope(Job())

    val job = scope.launch {
        coroutineScope {
            val task1 = launch {
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                throw Exception()
                1
            }

            try {
                task2.await()
            } catch (e: Exception) {
                println("Caught exception $e")
            }
            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

The output of this program is:

```
Caught exception java.lang.Exception
```

Then the program crashes on Android due to `java.lang.Exception`—we'll explain this shortly.

From this you can learn that inside a `coroutineScope`, `async` *exposes* uncaught exceptions but also notifies its parent. If you don't call `task2.await()`, the program still crashes because `coroutineScope` fails and *exposes* to its parent the exception that caused its failure. Then, `scope.launch` treats this exception as *unhandled*.

## Unhandled Exceptions

The coroutine framework treats unhandled exceptions in a specific way: it tries to use a CEH if the coroutine context has one. If not, it delegates to the *global handler*. This handler calls a customizable set of CEH *and* calls the standard mechanism of unhandled exceptions: `Thread.uncaughtExceptionHandler`. By default on Android, the previously mentioned set of handlers is only made of a single CEH which prints the stacktrace of the unhandled exception. However, it is possible to register a custom handler which will be called in addition to the one that prints the stacktrace. So you should remember that if you don't handle an exception, the `Thread.uncaughtExceptionHandler` *will* be invoked.

The default `UncaughtExceptionHandler` on Android makes your application crash, while on the JVM,[9] the default handler prints the stacktrace to the console. Consequently, if you execute this program not on Android but on the JVM, the output is:[10]

```
Caught exception java.lang.Exception
(stacktrace of java.lang.Exception)
Program ends
```

Back to Android. How could you handle this exception? Since `coroutineScope` *exposes* exceptions, you could wrap `coroutineScope` inside a `try`/`catch` statement. Alternatively, if you don't handle it correctly, the preceding `coroutineScope`, `scope.launch`, treats this exception as unhandled. Then your last chance to handle this exception is to register a CEH. There are at least two reasons you would do that: first, to stop the exception's propagation and avoid a program crash; and second, to notify your crash analytics and rethrow the exception—potentially making the application crash. In any case, we're not advocating for silently catching exceptions. If you do want to use CEH, there are a couple of things you should know. A CEH only works when registered to:

- `launch` (not `async`) when `launch` is a root coroutine builder[11]
- A scope
- `supervisorScope`'s direct child

In our example, the CEH should be registered either on `scope.launch` or on the scope itself. The following code shows this on the root coroutine:

```
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
```

```kotlin
    }

    val scope = CoroutineScope(Job())

    val job = scope.launch(ceh) {
        coroutineScope {
            val task1 = launch {
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                throw Exception()
                1
            }

            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

The output of this program is:

```
Caught exception java.lang.Exception
CEH handle java.lang.Exception
Program ends
```

Here is the same example, this time with the CEH registered on the scope:

```kotlin
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
    }

    val scope = CoroutineScope(Job() + ceh)

    val job = scope.launch {
       // same as previous example
    }
}
```

Finally, we illustrate the use of a CEH on a `supervisorScope` direct child:

```kotlin
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
    }

    val scope = CoroutineScope(Job())

    val job = scope.launch {
        supervisorScope {
            val task1 = launch {
                // simulate a background task
                delay(1000)
                println("Done background task")
            }

            val task2 = launch(ceh) {
                // try to fetch some count, but it fails
                throw Exception()
            }

            task1.join()
            task2.join()
        }
    }

    job.join()
    println("Program ends")
}
```

Notice that the coroutine builder on which the CEH is registered is a `launch`. It wouldn't have been taken into account with an `async`, which *exposes* uncaught exceptions, which can be handled with `try`/`catch`.

## Summary

- When a function might not return immediately, it's a good candidate to be implemented as a suspending function. However, the `suspend` modifier doesn't magically turn a blocking call into a nonblocking one. Use `withContext` along with the appropriate `Dispatcher`, and/or call other suspending functions.
- A coroutine can be deliberately cancelled using `Job.cancel()` for `launch`, or `Deferred.cancel()` for `async`. If you need to call some suspending functions inside your cleanup code, make sure you wrap your cleanup logic inside a `withContext(NonCancellable) { .. }` block. The cancelled coroutine will remain in the cancelling state until the cleanup exits. After the cleanup is done, the aforementioned coroutine goes to the cancelled state.
- A coroutine always waits for its children to complete before completing itself. So cancelling a coroutine also cancels all of its children.
- Your coroutines should be cooperative with cancellation. All suspending functions from the *kotlinx.coroutines* package are cancellable. This notably includes `withContext`. If you're implementing your own suspending function, make sure it is cancellable by checking `isActive` or calling `ensureActive()` or `yield()` at appropriate steps.
- There are two categories of coroutine scope: the scopes using `Job` and the ones using `SupervisorJob` (also called supervisor scopes). They differ in how cancellation is performed and in exception handling. If the failure of a child should also cancel other children, use a regular scope. Otherwise, use a supervisor scope.
- `launch` and `async` differ in how they treat uncaught exceptions. `async` *exposes* exceptions, which can be caught by wrapping the `await` call in a `try`/`catch`. On the other hand, `launch` treats uncaught exceptions as unhandled, which can be handled using a CEH.
- A CEH is optional. It should only be used when you really need to do something with unhandled exceptions. Unhandled exceptions typically should make your application crash. Or, at least, recovering from some exceptions might leave your application in an undetermined state. Nevertheless, if you decide to use a CEH, then it should

be installed at the top of the coroutine hierarchy—typically into the topmost scope. It can also be installed on a `supervisorScope` direct child.
- If a coroutine fails because of an uncaught exception, it gets cancelled along with all of its children and the exceptions propagate up.

## Closing Thoughts

You learned how to write your own suspending functions, and how to use them inside coroutines. Your coroutines live within scopes. In order to implement the desired cancellation policy, you know how to choose between `coroutineScope` and `supervisorScope`. The scopes you create are children of other scopes higher in the hierarchy. In Android, those “root” scopes are library-provided—you don’t create them yourself. A good example is the `viewModelScope` available in any `ViewModel` instance.

Coroutines are a perfect fit for one-time or repetitive tasks. However, we often have to work with asynchronous streams of data. `Channel`s and `Flow`s are designed for that, and will be covered in the next two chapters.

1 When performing CPU-bound tasks, a worker is bound to a CPU core.

2 See *Java Concurrency in Practice* (Addison-Wesley), Brian Goetz et al., 16.2.2.

3 We mentioned this in Chapter 5. In this case, it means that we add a new element to `hikeDataList` from the main thread.

4 Unless the `Dispatchers.IO` suffers from thread starvation, which is highly unlikely.

5 It’s just a subclass of the regular `CoroutineScope`, which invokes `coroutineContext.cancel()` inside its `close()` method.

6 Notice that the material on the suspending functions approach is relatively shorter (three and a half pages compared to seven pages for the traditional approach)—probably because suspending functions is an easier (and easier-to-explain) solution.

7 When started lazily, a coroutine is in the `New` state. Only after invoking `job.start()` does the coroutine move to the `Active state`. Calling `job.join()` also starts the coroutine.

8 `NonCancellable` is actually a special implementation of `Job` which is always in `Active` state. So suspending functions that use `ensureActive()` under this context are never cancelled.

9 By JVM, we mean on a desktop application, or on the server side.

10 “Program ends” is printed because the *unhandled* exception makes `scope` fail, not the scope from `runBlocking`.

11 A root coroutine builder is a scope’s direct child. In the previous example, at the line `val job = scope.launch {..}`, `launch` is a root coroutine builder.