Chapter 8: Designing for Concurrency

Concurrent design patterns help us to manage many tasks at once and structure their life cycle. By using these patterns efficiently, we can avoid problems such as resource leaks and deadlocks.

In this chapter, we'll discuss concurrent design patterns and how they are implemented in **Kotlin**. To do this, we'll be using the building blocks from previous chapters: coroutines, channels, flows, and concepts from **functional programming**.

We will be covering the following topics in this chapter:

- Deferred value
- Barrier
- Scheduler
- Pipeline
- Fan out
- Fan in
- Racing
- Mutex
- Sidekick channel

After completing this chapter, you'll be able to work with asynchronous values efficiently, coordinate the work of different coroutines, and distribute and aggregate work, as well as have the tools needed to resolve any concurrency problems that may arise in the process.

Technical requirements

In addition to the technical requirements from the previous chapters, you will also need a **Gradle**-enabled Kotlin project to be able to add the required dependencies.

You can find the source code used in this chapter on **GitHub** at the following location:

https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter08

Deferred Value

The goal of the **Deferred Value** design pattern is to return a reference to a result of an asynchronous computation. A **Future** in **Java** and **Scala**, and a **Promise** in **JavaScript** are both implementations of the Deferred Value design pattern.

We've already discussed **deferred values** in *Chapter 6*, *Threads and Coroutines*. We've seen that the **`async()`** function returns a type called **`Deferred`**, which is also an implementation of this design pattern.

Interestingly enough, the **`Deferred`** value itself is an implementation of both the **Proxy** design pattern that we've seen in *Chapter 3*, *Understanding Structural Patterns*, and the **State** design pattern from *Chapter 4*, *Getting Familiar with Behavioral Patterns*.

We can create a new container for the result of an asynchronous computation using the **`CompletableDeferred`** constructor:

```
val deferred = CompletableDeferred<String>()
```

To populate the **`Deferred`** value with a result, we use the **`complete()`** function, and if an error occurs in the process, we can use the **`completeExceptionally()`** function to pass the exception to the caller. To understand it better, let's write a function that returns an asynchronous result. Half of

the time the result will contain **`OK`**, and the other half of the time it will contain an exception.

```
suspend fun valueAsync(): Deferred<String> =
coroutineScope {
    val deferred = CompletableDeferred<String>()
    launch {
        delay(100)
        if (Random.nextBoolean()) {
            deferred.complete("OK")
        }
        else {
            deferred.completeExceptionally(
              RuntimeException()
            )
        }
    }
    deferred
}
```

You can see that we return the **`Deferred`** value almost immediately, then we start an asynchronous computation using **`launch`** and simulate some computation using the **`delay()`** function.

Since the process is asynchronous, the results won't be ready immediately. To wait for the results, we can use the **`await()`** function that we've already discussed in *Chapter 6, Threads and Coroutines*:

```
runBlocking {
    val value = valueAsync()
    println(value.await())
}
```

It's important to make sure that you always complete your **`Deferred`** value by calling either of the **`complete()`** or **`completeExceptionally()`** functions. Otherwise, your program may wait indefinitely for the results. It is also

possible to cancel **`deferred`** if you're no longer interested in its results. To do this, simply call **`cancel()`** on it:

```
deferred.cancel()
```

You'll rarely need to create your own deferred value. Usually, you would work with the one returned from the **`async()`** function.

Next, let's discuss how to wait for multiple asynchronous results at once.

Barrier

The **Barrier** design pattern provides us with the ability to wait for multiple concurrent tasks to complete before proceeding further. A common use case for this is composing objects from different sources.

For example, take the following class:

```
data class FavoriteCharacter(
    val name: String,
    val catchphrase: String,
    val picture: ByteArray = Random.nextBytes(42)
)
```

Let's assume that the **`catchphrase`** data comes from one service and the **`picture`** data comes from another. We would like to fetch these two pieces of data concurrently:

```
fun CoroutineScope.getCatchphraseAsync
(
    characterName: String
) = async { … }
fun CoroutineScope.getPicture
(
    characterName: String
) = async { … }
```

The most basic way to implement concurrent fetching would be as follows:

```
suspend fun fetchFavoriteCharacter(name: String) =
coroutineScope {
    val catchphrase =
getCatchphraseAsync(name).await()
    val picture = getPicture(name).await()
    FavoriteCharacter(name, catchphrase, picture)
}
```

But this solution has a major problem – we don't start fetching the **`picture`** data until the **`catchphrase`** data was fetched. In other words, the code is unnecessarily *sequential*. Let's see how this can be improved.

Using data classes as barriers

We can slightly alter the previous code to achieve the concurrency we want:

```
suspend fun fetchFavoriteCharacter(name: String) =
coroutineScope {
    val catchphrase = getCatchphraseAsync(name)
    val picture = getPicture(name)
    FavoriteCharacter(name, catchphrase.await(),
      picture.await())
}
```

Moving the **`await`** function into the invocation of the data class constructor allows us to start all of the coroutines at once and then wait for them to complete, just as we wanted.

The additional benefit of using data classes as barriers is the ability to *destructure* them easily:

```
val (name, catchphrase, _) =
fetchFavoriteCharacter("Inigo Montoya")
```

```
println("$name says: $catchphrase")
```

This works well if the type of data we receive from different asynchronous tasks is heterogeneous. In some cases, we receive the same types of data from different sources.

For example, let's ask **`Michael`** (our canary product owner), **`Taylor`** (our barista), and **`Me`** who our favorite movie character is:

```
object Michael {
    suspend fun getFavoriteCharacter() =
coroutineScope {
        async {
            FavoriteCharacter("Terminator",
              "Hasta la vista, baby")
        }
    }
}
object Taylor {
    suspend fun getFavoriteCharacter() =
coroutineScope {
        async {
            FavoriteCharacter("Don Vito Corleone",
"I'm
              going to make him an offer he can't
refuse")
        }
    }
}
object Me {
    suspend fun getFavoriteCharacter() =
coroutineScope {
        async {
            // I already prepared the answer!
            FavoriteCharacter("Inigo Montoya",
              "Hello, my name is...")
```

```
            }
        }
    }
```

Here, we have three very similar objects that differ only in the contents of the asynchronous results they return.

In this case, we can use a list to gather the results:

```
    val characters: List<Deferred<FavoriteCharacter>>
    =    listOf(
            Me.getFavoriteCharacter(),
            Taylor.getFavoriteCharacter(),
            Michael.getFavoriteCharacter(),
        )
```

Notice the type of the list. It's a collection of the **`Deferred`** elements of the **`FavoriteCharacter`** type. On such collections, there's an **`awaitAll()`** function available that acts as a barrier as well:

```
    println(characters.awaitAll())
```

When working with a set of homogenous asynchronous results and you need all of them to complete before proceeding further, use **`awaitAll()`**.

The Barrier design pattern creates a rendezvous point for multiple asynchronous tasks. The next pattern will help us abstract the execution of those tasks.

Scheduler

The goal of the **Scheduler** design pattern is to decouple *what* is being run from *how* it's being run and optimize the use of resources when doing so.

In Kotlin, **Dispatchers** are an implementation of the Scheduler design pattern that decouple the coroutine (that is, the *what*) from underlying thread pools (that is, the *how*).

We've already seen dispatchers briefly in *Chapter 6, Threads and Coroutines*.

To remind you, the coroutine builders such as **`launch()`** and **`async()`** can specify which dispatcher to use. Here's an example of how you specify it explicitly:

```
runBlocking {
    // This will use the Dispatcher from the parent
    // coroutine
    launch {
        // Prints: main
        println(Thread.currentThread().name)
    }
    launch(Dispatchers.Default) {
        // Prints DefaultDispatcher-worker-1
        println(Thread.currentThread().name)
    }
}
```

The default dispatcher creates as many threads as you have CPUs in the underlying thread pool. Another dispatcher that is available to you is the **IO Dispatcher**:

```
async(Dispatchers.IO) {
    for (i in 1..1000) {
        println(Thread.currentThread().name)
        yield()
    }
}
```

This will output the following:

```
> …
> DefaultDispatcher-worker-2
> DefaultDispatcher-worker-1
> DefaultDispatcher-worker-1
```

```
> DefaultDispatcher-worker-1
> DefaultDispatcher-worker-3
> DefaultDispatcher-worker-3
> ...
```

The IO Dispatcher is used for potentially long-running or blocking operations and will create up to 64 threads for that purpose. Since our example code doesn't do much, the IO Dispatcher doesn't need to create many threads. That's why you'll see only a small number of workers used in this example.

Creating your own schedulers

We are not limited to the dispatchers Kotlin provides. We can also define dispatchers of our own.

Here is an example of creating a dispatcher that would use a dedicated thread pool of **`4`** threads based on **`ForkJoinPool`**, which is efficient for *divide-and-conquer* tasks:

```
val forkJoinPool =
ForkJoinPool(4).asCoroutineDispatcher()

repeat(1000) {
    launch(forkJoinPool) {
        println(Thread.currentThread().name)
    }
}
```

If you create your own dispatcher, make sure that you either release it with **`close()`** or reuse it, as creating a new dispatcher and holding to it is expensive in terms of resources.

Pipeline

The **Pipeline** design pattern allows us to scale heterogeneous work, consisting of multiple steps of varying complexity across multiple CPUs, by breaking the work into smaller, concurrent pieces. Let's look at the following example to understand it better.

Back in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, we wrote an HTML page parser. It was assumed that the HTML pages themselves were already fetched for us, though. What we would like to design now is a process that would create a possibly infinite stream of pages.

First, we would like to fetch news pages once in a while. For that, we'll have a producer:

```
fun CoroutineScope.producePages() = produce {
    fun getPages(): List<String> {
        // This should actually fetch something
        return listOf(
            "<html><body><h1>
               Cool stuff</h1></body></html>",
            "<html><body><h1>
               Even more stuff</h1></body></html>"
        )
    }
    val pages = getPages()

    while (this.isActive) {
        for (p in pages) {
            send(p)
        }
    }
}
```

The **`isActive`** flag will be true as long as the coroutine is running and hasn't been canceled. It is a good practice to check this property in loops that may run for a long time so they can be stopped between iterations if needed.

Each time we receive new titles, we send them downstream. Since tech news isn't updated very often, we can check for updates only once in a while by using **`delay()`**. In the actual code, the delay would probably be minutes, if not hours.

The next step is creating a **Document Object Model** (**DOM**) out of those raw strings containing HTML. For that, we'll have a second producer, with this one receiving a channel that connects it to the first one:

```
fun CoroutineScope.produceDom(pages:
ReceiveChannel<String>) = produce {
    fun parseDom(page: String): Document {
        // In reality this would use a DOM library to
parse
        // string to DOM
        return Document(page)
    }
    for (p in pages) {
        send(parseDom(p))
    }
}
```

We can use the **`for`** loop to iterate over the channel as long as it's still open. This is a very elegant way of consuming data from an asynchronous source without the need to define callbacks.

We'll have a third function that receives the parsed documents and extracts the title out of each one:

```
fun CoroutineScope.produceTitles(parsedPages:
ReceiveChannel<Document>) = produce {
    fun getTitles(dom: Document): List<String> {
        return dom.getElementsByTagName("h1").map {
            it.toString()
        }
    }
```

```
    for (page in parsedPages) {
        for (t in getTitles(page)) {
            send(t)
        }
    }
}
```

We're looking for the headers, and so we use **`getElementsByTagName("H1")`**. For each header found, we turn it into its string representation.

Now, we will move on toward composing our coroutines into pipelines.

Composing a pipeline

Now that we've familiarized ourselves with the components of the pipeline, let's see how we can combine multiple components together:

```
runBlocking {
    val pagesProducer = producePages()
    val domProducer = produceDom(pagesProducer)
    val titleProducer = produceTitles(domProducer)
    titleProducer.consumeEach {
        println(it)
    }
}
```

The resulting pipeline will look as follows:

```
Input=>pagesProducer=>domProducer=>titleProducer=>Out
put
```

A pipeline is a great way to break a long process into smaller steps. Note that each resulting coroutine is a *pure function*, so it's also easy to test and reason about.

The entire pipeline could be stopped by calling **`cancel()`** on the first coroutine in line.

Fan Out

The goal of the **Fan Out** design pattern is to distribute work between multiple concurrent processors, also known as *workers*. To understand it better, let's look again at the previous section but consider the following problem:

What if the amount of work at the different steps in our pipeline is very different?

For example, it takes a lot more time to *fetch* the HTML content than to *parse* it. In such a case, we may want to distribute that heavy work between multiple coroutines. In the previous example, only a single coroutine was reading from each channel. But multiple coroutines can consume from a single channel too, thus dividing the work.

To simplify the problem we're about to discuss, let's have only one coroutine producing some results:

```
fun CoroutineScope.generateWork() = produce {
    for (i in 1..10_000) {
        send("page$i")
    }
    close()
}
```

And we'll have a function that creates a new coroutine that reads those results:

```
fun CoroutineScope.doWork(
    id: Int,
    channel: ReceiveChannel<String>
) = launch(Dispatchers.Default) {
    for (p in channel) {
        println("Worker $id processed $p")
```

```
    }
}
```

This function will generate a coroutine that is executed on the **`Default`** dispatcher. Each coroutine will listen to a channel and print every message it receives to the console.

Now, let's start our producer. Remember that all the following pieces of code need to be wrapped in the **`runBlocking`** function, but for simplicity, we omitted that part:

```
val workChannel = generateWork()
```

Then, we can create multiple workers that distribute the work between themselves by reading from the same channel:

```
val workers = List(10) { id ->
    doWork(id, workChannel)
}
```

Let's now examine a part of the output of this program:

```
> ...
> Worker 4 processed page9994
> Worker 8 processed page9993
> Worker 3 processed page9992
> Worker 6 processed page9987
```

Note that no two workers receive the same message and the messages are not being printed in the order they were sent. The Fan Out design pattern allows us to efficiently distribute the work across a number of coroutines, threads, and CPUs.

Next, let's discuss an accompanying design pattern that often goes hand-in-hand with Fan Out.

Fan In

The goal of the **Fan In** design pattern is to combine results from multiple workers. This design pattern is helpful when our workers produce results and we need to gather them.

This design pattern is the opposite of the Fan Out design pattern we discussed in the previous section. Instead of multiple coroutines *reading* from the same channel, multiple coroutines can *write* their results to the same channel.

Combining the Fan Out and Fan In design patterns is a good base for **MapReduce** algorithms. To demonstrate this, we'll slightly change the workers from the previous example, as follows:

```
private fun CoroutineScope.doWorkAsync(
    channel: ReceiveChannel<String>,
    resultChannel: Channel<String>
) = async(Dispatchers.Default) {
    for (p in channel) {
        resultChannel.send(p.repeat(2))
    }
}
```

Now, once done, each worker sends the results of its calculation to **`resultChannel`**.

Note that this pattern is different from the actor and producer builders we've seen before. Actors each have their own channels, while in this case, **`resultChannel`** is shared across all the workers.

To collect the results from the workers, we'll use the following code:

```
runBlocking {
    val workChannel = generateWork()
    val resultChannel = Channel<String>()
    val workers = List(10) {
        doWorkAsync(workChannel, resultChannel)
```

```
        }
        resultChannel.consumeEach {
            println(it)
        }
    }
```

Let's now clarify what this code does:

1. First, we create **`resultChannel`**, which all our workers will share.
2. Then, we supply it to each worker. We have ten workers in total. Each worker repeats the message it received twice and sends it on **`resultChannel`**.
3. Finally, we consume the results from the channel in our main coroutine. This way, we accumulate results from multiple concurrent workers in the same place.

Here's a sample of the output from the preceding code:

```
> ...
> page9995page9995
> page9996page9996
> page9997page9997
> page9999page9999
> page9998page9998
> page10000page10000
```

Next, let's discuss another design pattern, which will help us improve the responsiveness of our code in some cases.

Racing

Racing is a design pattern that runs multiple jobs concurrently, picking the result that returns first as the *winner* and discarding others as *losers*.

We can implement Racing in Kotlin using the **`select()`** function on channels.

Let's imagine you are building a weather application. For redundancy, you fetch the weather from two different sources, *Precise Weather* and *Weather Today*. We'll describe them as two producers that return their name and temperature.

If we have more than one producer, we can subscribe to their channels and take the first result that is available.

First, let's declare the two weather producers:

```
fun CoroutineScope.preciseWeather() = produce {
    delay(Random.nextLong(100))
    send("Precise Weather" to "+25c")
}
fun CoroutineScope.weatherToday() = produce {
    delay(Random.nextLong(100))
    send("Weather Today" to "+24c")
}
```

Their logic is pretty much the same. Both wait for a random number of milliseconds and then return a temperature reading and the name of the source.

We can listen to both channels simultaneously using the **`select`** expression:

```
runBlocking {
  val winner = select<Pair<String, String>> {
    preciseWeather().onReceive { preciseWeatherResult
->
            preciseWeatherResult
        }
        weatherToday().onReceive { weatherTodayResult
->
            weatherTodayResult
        }
```

```
    }
    println(winner)
}
```

Using the **`onReceive()`** function allows us to listen to multiple channels simultaneously.

Running this code multiple times will randomly print **`(Precise Weather, +25c)`** and **`(Weather Today, +24c)`**, as there is an equal chance for both of them to arrive first.

Racing is a very useful concept when you are willing to sacrifice resources in order to get the most responsiveness from your system and we achieved that using Kotlin's **`select`** expression. Now, let's explore the **`select`** expression a little further to discover another concurrent design pattern that it implements.

Unbiased select

When using the **`select`** clause, the order is important. Because it is inherently biased, if two events happen at the same time, it will select the first clause.

Let's see what that means in the following example.

We'll have only one producer this time, which sends over a channel which movie we should watch next:

```
fun CoroutineScope.fastProducer(
    movieName: String
) = produce(capacity = 1) {
    send(movieName)
}
```

Since we defined a non-zero capacity on the channel, the value will be available as soon as this coroutine runs.

Now, let's start the two producers and use a **`select`** expression to see which of the two movies will be selected:

```
runBlocking {
    val firstOption = fastProducer("Quick&Angry 7")
    val secondOption = fastProducer(
      "Revengers: Penultimatum")
    delay(10)
    val movieToWatch = select<String> {
        firstOption.onReceive { it }
        secondOption.onReceive { it }
    }
    println(movieToWatch)
}
```

No matter how many times you run this code, the winner will always be the same: **`Quick&Angry 7`**. This is because if both values are ready at the same time, the **`select`** clause will always pick the first channel available in the order they are declared.

Now, let's use **`selectUnbiased`** instead of the **`select`** clause:

```
...
val movieToWatch = selectUnbiased<String> {
    firstOption.onReceive { it }
    secondOption.onReceive { it }
}
...
```

Running this code now will sometimes produce **`Quick&Angry 7`** and sometimes produce **`Revengers: Penultimatum`**. Unlike the regular **`select`** clause, **`selectUnbiased`** doesn't care about the order. If more than one result is available, it will pick one randomly.

Mutex

Also known as **mutual exclusions**, **mutex** provides a means to protect a shared state that can be accessed by multiple coroutines at once.

Let's start with the same old dreaded **`counter`** example, where multiple concurrent tasks try to update the same **`counter`**:

```
var counter = 0
val jobs = List(10) {
    async(Dispatchers.Default) {
        repeat(1000) {
            counter++
        }
    }
}
jobs.awaitAll()
println(counter)
```

As you've probably guessed, the result that is printed is less than 10,000 – *totally embarrassing!*

To solve this, we can introduce a locking mechanism that will allow only a single coroutine to interact with the variable at once, making the operation *atomic*.

Each coroutine will try to obtain the ownership of the **`counter`**. If another coroutine is updating the **`counter`**, our coroutine will wait patiently and then try to acquire the lock again. Once updated, it must release the lock so that other coroutines can proceed:

```
var counter = 0
val mutex = Mutex()
val jobs = List(10) {
    launch {
        repeat(1000) {
            mutex.lock()
            counter++
```

```
                mutex.unlock()
            }
        }
    }
```

Now, our example always prints the correct number: **`10,000`**.

Mutex in Kotlin is different from the Java mutex. In Java, **`lock()`** on a mutex blocks the thread, until the lock can be acquired. A Kotlin mutex suspends the coroutine instead, providing better concurrency. Locks in Kotlin are cheaper.

This is good for simple cases. But what if the code within the critical section, that is, between **`lock()`** and **`unlock()`**, throws an exception?

We would have to wrap our code in **`try...catch`**, which is not very convenient:

```
try {
    mutex.lock()
    counter++
}
finally {
    mutex.unlock()
}
```

However, if we omit the **`finally`** block, our lock will never be released and it will block all other coroutines from proceeding and creating a deadlock.

Exactly for this purpose, Kotlin also introduces **`withLock()`**:

```
mutex.withLock {
    counter++
}
```

Notice how much more concise this syntax is compared with the previous example.

Sidekick channel

The **Sidekick channel** design pattern allows us to offload some work from our main worker to a *back worker*.

Up until now, we've only discussed the use of **`select`** as a *receiver*. But we can also use **`select`** to *send* items to another channel. Let's look at the following example.

First, we'll declare **`batman`** as an actor coroutine that processes 10 messages per second:

```
val batman = actor<String> {
    for (c in channel) {
        println("Batman is beating some sense into
$c")
        delay(100)
    }
}
```

Next, we'll declare **`robin`** as another actor coroutine that is a bit slower and processes only four messages per second:

```
val robin = actor<String> {
    for (c in channel) {
        println("Robin is beating some sense into
$c")
        delay(250)
    }
}
```

So, we have a superhero and his sidekick as two actors. Since the superhero is more experienced, it usually takes him less time to beat the villain he's facing.

But in some cases, he still has his hands full, so a sidekick needs to step in. We'll throw five villains at the pair with a few delays and see how they fare:

```
val epicFight = launch {
    for (villain in listOf("Jocker", "Bane",
"Penguin",       "Riddler", "Killer Croc")) {
        val result = select<Pair<String, String>> {
            batman.onSend(villain) {
                "Batman" to villain
            }
            robin.onSend(villain) {
                "Robin" to villain
            }
        }
        delay(90)
        println(result)
    }
}
```

Notice that the type of parameter for **`select`** refers to what is *returned* from the block and not what is being *sent* to the channels. That's the reason we use **`Pair<String, String>`** here.

This code prints the following:

```
> Batman is beating some sense into Jocker
> (Batman, Jocker)
> Robin is beating some sense into Bane
> (Robin, Bane)
> Batman is beating some sense into Penguin
> (Batman, Penguin)
> Batman is beating some sense into Riddler
> (Batman, Riddler)
> Robin is beating some sense into Killer Croc
> (Robin, Killer Croc)
```

Using a sidekick channel is a useful technique to provide fallback values. Consider using one in cases when you need to consume a consistent stream of data and cannot easily scale your consumers.

Summary

In this chapter, we covered various design patterns related to *concurrency* in Kotlin. Most of them are based on coroutines, channels, deferred values, or a combination of these building blocks.

Deferred values are used as placeholders for asynchronous values. The Barrier design pattern allows multiple asynchronous tasks to rendezvous before proceeding further. The Scheduler design pattern decouples the code of tasks from the way they are executed at runtime.

The Pipeline, Fan In, and Fan Out design patterns help us distribute the work and collect the results. Mutex helps us to control the number of tasks that are being executed at the same time. The Racing design pattern allows us to improve the responsiveness of our application. Finally, the Sidekick Channel design pattern offloads work onto a backup task in case the main task is not able to process the incoming events quickly enough.

All of these patterns should help you to manage the concurrency of your application in an efficient and extensible manner. In the next chapter, we'll discuss Kotlin's idioms and best practices, as well as some of the anti-patterns that emerged with the language.

Questions

1. What does it mean when we say that the **`select`** expression in Kotlin is *biased*?
2. When should you use a *mutex* instead of a *channel*?
3. Which of the concurrent design patterns could help you implement a MapReduce or divide-and-conquer algorithm efficiently?