Chapter 10: Asynchronous Programming

You've come a long way in your study of the Dart programming language. This chapter is an important one, as it fills in the remaining gaps needed to complete your apprenticeship. In it, you'll not only learn how to deal with code that takes a long time to complete, but along the way, you'll also see how to handle errors, connect to a remote server and read data from a file.

Your computer does a lot of work, and it does the work so fast that you don't usually realize how much it's actually doing. Every now and then, though, especially on an older computer or phone, you may notice an app slow down or even freeze. This may express itself as **jank** during an animation: that annoying stutter that happens when the device is doing so much work that some animation frames get dropped.

Tasks that take a long time generally fall into two categories: I/O tasks, and computationally intensive tasks. I/O, or input-output, includes things like reading and writing files, accessing a database, or downloading content from the internet. These all happen outside the CPU, so the CPU has to wait for them to complete. Computationally intensive tasks, on the other hand, happen inside the CPU. These tasks may include things like decrypting data, performing a mathematical calculation, or parsing JSON.

As a developer, you have to think about how your app, and in particular your UI, will respond when it meets these time-consuming tasks. Can you imagine if a user clicked a download button in your app, and the app simply froze until the 20 MB download was complete? You'd be collecting one-star reviews in a hurry.

Thankfully, Dart has a powerful solution baked into the very core of the language that allows you to gracefully handle delays without blocking the responsiveness of your app.

Concurrency in Dart

A **thread** is a sequence of commands that a computer executes. Some programming languages support multithreading, which is running multiple threads at the same time, while others do not. Dart, in particular, is a single-threaded language.

"What? Was it designed back in 1990 or something?"

No, Dart was actually created in 2011, well into the age of multicore CPUs.

"What a waste of all those other processing cores!"

Ah, but no. This choice to be single-threaded was made very deliberately and has some great advantages as you'll soon see.

Parallelism vs. concurrency

To understand Dart's model for handling long-running tasks, and also to see why the creators of Dart decided to make Dart single-threaded, it's helpful to understand the difference between parallelism and concurrency. In common English, these words mean approximately the same thing, but in computer science, there's a distinction.

Parallelism is when multiple tasks run *at the same time* on multiple processors or CPU cores. **Concurrency**, on the other hand, is when multiple tasks take turns running on a single CPU core. When a restaurant has a single person alternately taking orders and clearing tables, that's concurrency. But a restaurant that has one person taking orders and a different person clearing tables, that's parallelism.

"It seems like parallelism is better."

It can be — when there's a lot of work to do and that work is easy to split into independent tasks. However, there are some disadvantages with parallelism, too.

A problem with parallelism

Little Susie has four pieces of chocolate left in the box next to her bed. She used to have ten, but she's already eaten six of them. She's saved the best ones for last, because after school today, three of her friends are coming home with her. She can't wait to share her chocolates with them. Imagine her horror, though, when she gets home and finds only two pieces of chocolate left in the box! After a lengthy investigation, it turns out that Susie's brother had discovered her stash and helped himself to two of the chocolates. From

that day on, Susie always locked her box whenever she left home.

The same thing can happen in parallel threads that have access to the same memory. One thread saves a value in memory and expects the value to be the same when the thread checks the value later. However, if a second thread modifies the value, the first thread gets confused. It can be a major headache to track down those kinds of bugs because they come from a source completely separated from the code that reports the error. A language that supports multithreading needs to set up a system of locks so that values won't be changed at the wrong time. The cognitive load of designing, implementing, and debugging a system with multiple threads can be heavy, to say the least.

So the problem isn't so much with parallelism itself, but rather with multiple threads having access to the same state in memory.

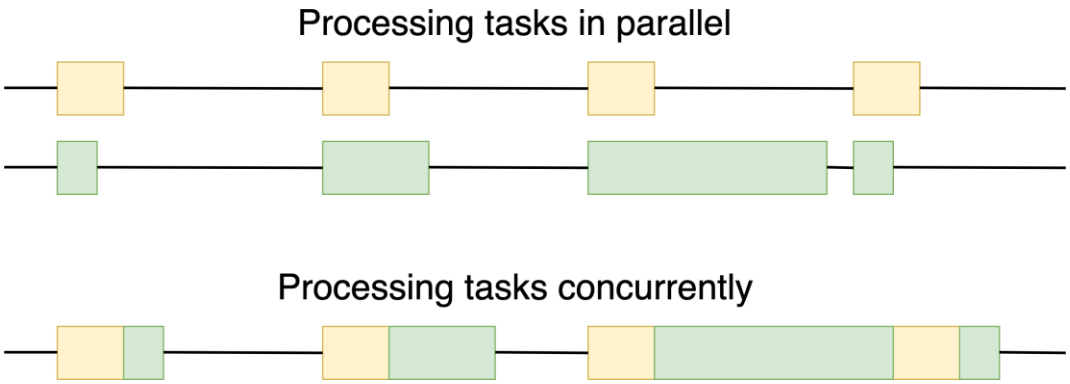
Dart isolates

Dart's single thread runs in what it calls an **isolate**. Each isolate has its own allocated memory area, which ensures that no isolate can access any other isolate's state. That means that there's no need for a complicated locking system. It also means that sensitive data is much more secure. Such a system greatly reduces the cognitive load on a programmer.

But isn't concurrency slow?

If you're running all of a program's tasks on a single thread, it seems like it would be really slow. However, it turns out

that that's not usually the case. In the following image, you can see tasks running on two threads in the top portion, and the same tasks running on a single thread in the bottom portion.



The concurrent version does take a little longer, but it isn't *much longer*. The reason is that the parallel threads were idle for much of the time. A single thread is usually more than enough to accomplish what needs to be done.

Flutter needs to update the UI 60 times a second, where each update timeslice is called a **frame**. That leaves about 16 milliseconds to redraw the UI on each frame. It doesn't take that long, normally, so that gives you additional time to perform other work while the thread is idle. As long as that work doesn't block Flutter from updating the UI on the next frame, the user won't notice any problems. The trick is to schedule tasks during the thread's downtimes.

Synchronous vs. asynchronous code

The word **synchronous** is composed of *syn*, meaning "together", and *chron*, meaning "time", thus *together in time*. Synchronous code is where each instruction is

executed in order, one line of code immediately following the previous one.

This is in contrast to **asynchronous** code, which means *not* together in time. That is, with asynchronous code, certain tasks are rescheduled to be run in the future when the thread isn't busy.

All of the code that you've written so far in the book has been synchronous. For example:

```
print("first");
print("second");
print("third");
```

Run that and it prints:

```
first
second
third
```

Since the code is executed synchronously, it'll never print in a different order like `third first second`.

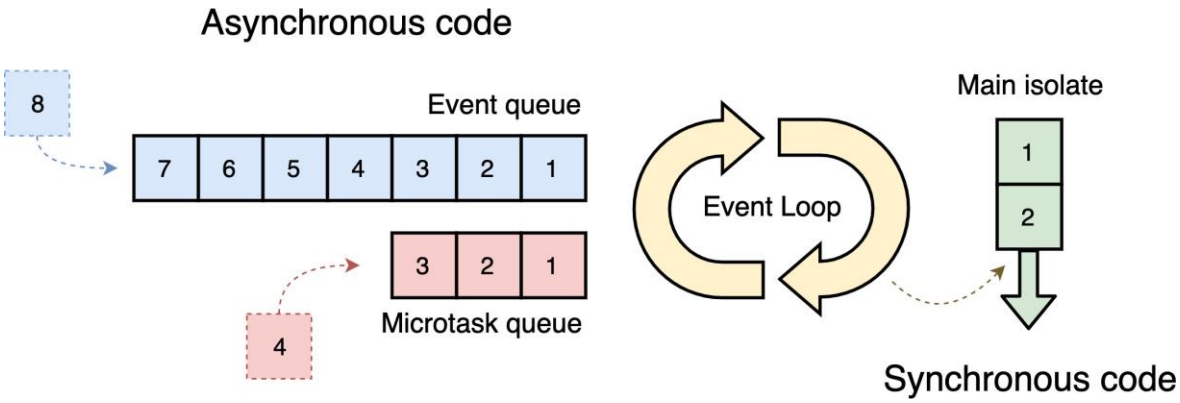
For many tasks, order matters. Multiplying before adding is different than adding before multiplying. You have to open the bottle before you can take a drink. For other tasks, though, the order doesn't matter. It doesn't matter if you brush your teeth first or wash your face first. It doesn't matter if you put a sock on the right foot first or the left foot first.

As in life, so it is with Dart. While some code needs to be executed in order, other tasks can be temporarily postponed. The postponable tasks are where the Dart event loop comes in.

The event loop

You've learned that Dart is based around concurrency on a single thread, but how does Dart manage to schedule tasks asynchronously? Dart uses what it calls an **event loop** to execute tasks that had previously been postponed.

The event loop has two queues: a **microtask queue** and an **event queue**. The microtask queue is mostly used internally by Dart. The event queue is for events like a user entering a keystroke or touching the screen, or data coming from a database, file, or remote server. Have a look at the following image:



- Synchronous tasks in the main isolate thread are always run immediately. You can't interrupt them.

- If Dart finds any long-running tasks that agree to be postponed, Dart puts them in the event queue.

- When Dart is finished running the synchronous tasks, the event loop checks the microtask queue. If the microtask queue has any tasks, the event loop puts them on the main thread to execute next. The event loop keeps checking the microtask queue until it's empty.

- If the synchronous tasks and microtask queue are both empty, then the event loop sends the next waiting task in the event queue to run in the main thread. Once it gets there, the code is executed synchronously. Just like any other synchronous code, nothing can interrupt it after it starts.

- If any new microtasks enter the microtask queue, the event loop always handles them before the next event in the event queue.

- This process continues until all of the queues are empty.

Running code in parallel

When people say Dart is single-threaded, they mean that Dart only runs on a single thread in the isolate. However, that *doesn't* mean you can't have tasks running on another thread. One example of this is when the underlying platform performs some work at the request of Dart. For example, when you ask to read a file on the system, that work isn't happening on the Dart thread. The system is doing the work inside its own process. Once the system finishes its work, it passes the result back to Dart, and Dart schedules some code

to handle the result in the event queue. A lot of the I/O work from the `dart:io` library happens in this way.

Another way to perform work on other threads is to create a new Dart isolate. The new isolate has its own memory and its own thread working in parallel with the main isolate. The two isolates are only able to communicate through messages, though. They have no access to each other's memory state. The idea is similar to messaging a friend. Sending Ray a text message doesn't give you access to the internal memory of his mobile device. He simply checks his messages and replies to you when he feels like it.

You won't often need to create a new isolate. However, if you have a task that's taking too long on your main isolate thread, which you'll notice as unresponsiveness or jank in the UI, then this work is likely a good candidate for handing it off to another isolate. The final section of this chapter will tell you how to do that.

Futures

You now know, at a high level, how Dart handles asynchronous code with its event loop. Now it's time to learn how to work with asynchronous code at a practical level.

The Future type

Dart has a type called `Future`, which is basically a promise to give you the value you really want later. Here's the signature of a method that returns a future:

```
Future<int> countTheAtoms();
```

Future itself is generic; it can provide any type. In this case, though, the future is promising to give you an integer. In your code, if you called countTheAtoms, Dart would quickly return an object of type Future<int>. In effect, this is saying, "Hey, I'll get back to you with that int sometime later. Carry on!", in which case you'd proceed to run whatever synchronous code is next.

Behind the scenes, Dart has passed your request on to, presumably, an atom counting machine, which runs independently of your main Dart isolate. At this point, there is nothing on the event queue, and your main thread is free to do other things. Dart knows about the uncompleted future, though. When the atom counting machine finishes its work, it tells Dart and Dart puts the result, along with any code you gave it to handle the result, on the event queue. Dart says, "Sorry that took so long. Who knew that there were 9.2 quintillion atoms in that little grain of sand! I'll put your handling code at the end of the event queue. Give the event loop a few milliseconds and then it'll be your turn."

Note: Since the largest an `int` can be on a 64 bit system is 9,223,372,036,854,775,807, or 2^63 − 1, it would be better to use `BigInt` as the return type of `countTheAtoms`. Although slower, `BigInt` can handle arbitrarily large numbers. When `int` values are too big at compile time, there's a compile-time error. However, at runtime, they overflow. That is, `9223372036854775807 + 1 == -9223372036854775808`.

States for a future

Before a future completes, there isn't really anything you can do with it, but after it completes it will have two possible results: the value you were asking for, or an error. This all works out to three different states for a future:

- Uncompleted
- Completed with a value
- Completed with an error

Example of a future

One easy way to see a future in action is with the `Future.delayed` constructor.

```
final myFuture = Future<int>.delayed(
  Duration(seconds: 1),
  () => 42,
);
```

Here's what's happening:

- `myFuture` is of type `Future<int>`.
- The first argument is a `Duration`. After a delay of `1` second, Dart will add the anonymous function in the second argument to the event queue.
- When the event loop gets to `() => 42` it will run that function in the main isolate, which results in the function returning the integer `42`.

In the future above, the value you really want is the `42`, but how do you get it? Your variable `myFuture` isn't `42`; it's a future that's a promise to return an `int` or an error. You can see that if you try to print `myFuture`:

```
print(myFuture);
```

The result is:

```
Instance of 'Future<int>'
```

There are two ways to get at the value after a future completes. One is with callbacks and the other is using the `async-await` syntax.

Getting the result with callbacks

A **callback** is an anonymous function that will run after some event has completed. In the case of a future, there are three callback opportunities: `then`, `catchError` and `whenComplete`.

Replace the body of the main function with the following code:

```
print('Before the future');

final myFuture = Future<int>.delayed(
  Duration(seconds: 1),
  () => 42,
)
    .then(
      (value) => print('Value: $value'),
    )
    .catchError(
      (error) => print('Error: $error'),
    )
    .whenComplete(
      () => print('Future is complete'),
    );

print('After the future');
```

You recall that a future will either give you a value or an error. If it completes with a value, you can get the value by adding a callback to the `then` method. The anonymous function provides the value as an argument so that you have access to it. On the other hand, if the future completes with an error, you can handle it in `catchError`. Either way, though, whether the future completes with a value or an error, you have the opportunity to run any final code in `whenComplete`.

Run the code above to see these results:

```
Before the future
After the future
Value: 42
Future is complete.
```

Were you surprised that "After the future" was printed before the future results? That `print` statement is synchronous, so it ran immediately. Even if the future didn't have a one-second delay, it would still have to go to the event queue and wait for all the synchronous code to finish.

Getting the result with async-await

Callbacks are pretty easy to understand, but they can be hard to read, especially if you nest them in multiple layers. A more readable way to write the code above is using the `async` and `await` syntax. This syntax makes futures look much more like synchronous code.

Replace the entire `main` function with the following:

```
Future<void> main() async {
  print('Before the future');

  final value = await Future<int>.delayed(
    Duration(seconds: 1),
    () => 42,
  );
  print('Value: $value');

  print('After the future');
}
```

There are a few changes this time:

- If a function uses the `await` keyword, then it must return a `Future` and add the `async` keyword before the function body. Using `async` clearly tells Dart that this is an asynchronous function, and that the results will go to the event queue. Since `main` doesn't return a value, you use `Future<void>`.

- In front of the future, you added the `await` keyword. Once Dart sees `await`, the rest of the function won't run until the future completes. If the future completes with a value, there are no callbacks. You have direct access to that value. Thus, the type of the `value` variable above is not `Future`, but `int`.

Run the code above to see the following results:

```
Before the future
Value: 42
After the future
```

This time, "After the future" gets printed last. That's because *everything* after the `await` keyword is sent to the event queue.

What if the future returns an error, though?

For that, you need to learn about an error handling feature of Dart called a `try-catch` block.

Handling errors with try-catch blocks

The syntax of a try-catch block looks like this:

```
try {

} catch (error) {

} finally {

}
```

If you're attempting an operation that might result in an error, you'll place it in the **try** block. If there is an error, Dart will give you a chance to handle it in the **catch** block. And whether there is an error or not, you can run some last code in the **finally** block.

Note: Dart has both an **Exception** type and an **Error** type. The words exception and error are often used interchangeably, but an **Exception** is something that you should expect and handle in the **catch** block. However, an **Error** is the result of a programming mistake. You should let the error crash your app as a sign that you need to fix whatever caused the error.

Try-catch blocks with async-await

Here's what the future looks like inside the try-catch block:

```
print('Before the future');

try {
  final value = await Future<int>.delayed(
    Duration(seconds: 1),
    () => 42,
  );
  print('Value: $value');
} catch (error) {
  print(error);
} finally {
  print('Future is complete');
}

print('After the future');
```

The catch and finally blocks correspond to the catchError and whenComplete callbacks that you saw earlier. If the future completes with an error, then the try block will immediately be aborted and the catch block will be called. But no matter whether the future completes with a value or an error, the finally block will always be called.

Run the code above to see the following result:

```
Before the future
Value: 42
Future is complete
After the future
```

The future finished with a value, so the catch block was not called.

Catching an error

In order to see what happens when there's an error, add the following line to the try block on the line immediately before `print("Value: $value")`:

```
throw Exception("There was an error");
```

The `throw` keyword is how you return an instance of `Exception` or `Error`.

Run the code again:

```
Before the future
Exception: There was an error
Future is complete
After the future
```

This time you can see that the `try` block never got a chance to print the value, but the `catch` block picked up the error message from the exception.

Asynchronous network requests

In the examples above, you used `Future.delayed` to simulate a task that takes a long time. Using `Future.delayed` is useful during app development for this same reason: You can implement an interface with a mock network request class to see how your UI will react while the app is waiting for a response.

As useful as `Future.delayed` is, though, eventually you'll need to implement the real network request class. The following example will show how to make an HTTP request to access a REST API. This example will make use of many of the concepts you've learned previously in this book.

> **Note**: HTTP, or **hypertext transfer protocol**, is a standard way of communicating with a remote server. REST, or **representational state transfer**, is an architectural style that includes commands like `GET`, `POST`, `PUT`, and `DELETE`. The API, or **application programing interface**, is similar in idea to the interfaces you made in Chapter 9. A remote server defines a specific API using REST commands which allow clients to access and modify resources on the server.

Creating a data class

The web API you're going to use will return some data about a todo list item. The data will be in JSON format, so in order to convert that into a more usable Dart object, you'll create a special class to hold the data.

Add the following code below the `main` function:

```
class Todo {
  Todo({
    required this.userId,
    required this.id,
    required this.title,
    required this.completed,
  });

  factory Todo.fromJson(Map<String, Object?> jso
nMap) {
    return Todo(
      userId: jsonMap['userId'] as int,
      id: jsonMap['id'] as int,
      title: jsonMap['title'] as String,
      completed: jsonMap['completed'] as bool,
    );
  }

  final int userId;
  final int id;
  final String title;
  final bool completed;

  @override
  String toString() {
    return 'userId: $userId\n'
        'id: $id\n'
        'title: $title\n'
        'completed: $completed';
  }
}
```

This is all content that you learned in Chapter 6. You'll use the `fromJson` factory constructor in a minute.

Adding the necessary imports

The `http` package from the Dart team lets you make a GET request to a real server. Make sure your project has a **pubspec.yaml** file, and then add the following dependency:

```
dependencies:
  http: ^0.13.1
```

Save the file, and if necessary, run `dart pub get` in the terminal to pull the `http` package from Pub.

Then at the top of the file with your `main` function, add the following imports:

```
import 'dart:convert';
import 'dart:io';
import 'package:http/http.dart' as http;
```

Here's what each import is for:

- The `dart:convert` library will give you `jsonDecode`, a function for converting a raw JSON string to a Dart map.
- The `dart:io` library has `HttpException` and `SocketException`, which you'll use shortly.
- The final import is the `http` library that you just added to `pubspec.yaml`. Note the `as http` at the end. This isn't necessary, but the `as` keyword lets you prefix any functions from the library with the name `http`. You don't need to call it `http` — any arbitrary name is fine.

Feel free to change the name to `pinkElephants` if you so desire. Providing a custom name can be useful for avoiding naming conflicts with other libraries or functions.

Making a GET request

Now that you have the necessary imports, replace your `main` function with the following code:

```
Future<void> main() async {
  final url = 'https://jsonplaceholder.typicode.com/todos/1';
  final parsedUrl = Uri.parse(url);
  final response = await http.get(parsedUrl);
  final statusCode = response.statusCode;
  if (statusCode == 200) {
    final rawJsonString = response.body;
    final jsonMap = jsonDecode(rawJsonString);
    final todo = Todo.fromJson(jsonMap);
    print(todo);
  } else {
    throw HttpException('$statusCode');
  }
}
```

There are a few new things here, so have a look at each of them:

- The URL address is for a server that provides an API that returns sample JSON for developers. It's very similar to the type of API you would make as a backend for a client app. `Uri.parse` converts the raw URL string to a format that `http.get` will recognize.

- You use `http.get` to make a `GET` request to the URL. Change `http` to `pinkElephants` if that's what you called it earlier. `GET` requests are the same kinds of requests that browsers make whenever you type a URL in the address bar.

- Since it takes time to contact a server that may exist in another continent, `http.get` returns a future. Dart passes off the work of contacting the remote server to the underlying platform, so you won't need to worry about it blocking your app while you wait. Since you are using the `await` keyword, the rest of the main method will be added to the event queue when the future completes. If the future completes with a value, the value will be an object of type `Response`, which includes information from the server.

- HTTP defines various three-digit status codes. A status code of `200` means `OK` — the request was successful and the server did what you asked of it. The common status code of `404`, on the other hand, means the server couldn't find what you were asking for. If that happens you'll throw an `HttpException`.

- The response body from this URL address includes a string in JSON format. You use `jsonDecode` from the `dart:convert` library to convert the raw JSON string into a Dart map.

- Once you have a Dart `map`, you can pass it into the `fromJson` factory constructor of your `Todo` class that you wrote earlier.

Make sure you have an internet connection, then run the code above. You'll see a printout from your `Todo` object's `toString` method:

```
userId: 1
id: 1
title: delectus aut autem
completed: false
```

The values of each field come from the remote server.

Handling errors

There are a few things that could go wrong with the code above, so you'll need to be ready to handle any errors that come up. First, surround all the code inside the body of the `main` function with a `try` block:

```
try {
  final url = 'https://jsonplaceholder.typicode.com/todos/1';
  // ...
}
```

Then below the `try` block, add the following `catch` blocks:

```
on SocketException catch (error) {
  print(error);
} on HttpException catch (error) {
  print(error);
} on FormatException catch (error) {
  print(error);
}
```

These `catch` blocks handle specific types of errors. You use the `on` keyword to specify the name of the exception. Here's what each one means:

- `SocketException`: You'll get this exception if there's no internet connection. The `http.get` method is the one to throw the exception.
- `HttpException`: You're throwing this exception yourself if the status code is not `200 OK`.
- `FormatException`: `jsonDecode` throws this exception if the JSON string from the server isn't in proper JSON format. It would be unwise to blindly trust whatever the server gives you.

> **Note**: It's good to be specific in your error catching. That way if a different kind of error comes up that you weren't expecting, your app will fail with a crash. That might sound bad, but it means you can fix the error right away instead of silently ignoring it as a generic `catch` block would do.

Testing a socket exception

Turn off your internet and run the code again. You should see the following output:

```
SocketException: Failed host lookup: 'jsonplaceholder.typicode.com'
```

In an actual app, instead of just printing a message to the console, you'd probably want to remind the user to turn on their internet.

Turn *your* internet back on and proceed to the next test.

Testing an HTTP exception

Change the URL to the following:

```
final url = 'https://jsonplaceholder.typicode.com/todos/pink-elephants';
```

Unless https://jsonplaceholder.typicode.com has recently added the `/pink-elephants` URL endpoint, you should get a 404 when you run the code again:

```
HttpException: 404
```

In a real app, you'd inform the user that whatever they were looking for isn't available.

Restore the URL as it was before:

```
final url = 'https://jsonplaceholder.typicode.com/todos/1';
```

Testing a JSON format exception

Replace the following line:

```
final rawJsonString = response.body;
```

with this:

```
final rawJsonString = "abc";
```

Since abc isn't properly formatted JSON, you'll see the following error when you run the code again:

```
FormatException: Unexpected character (at charac
ter 1)
abc
^
```

Nice work! You now know how to get the value from a future and handle any errors.

After you finish the following mini-exercises, you'll learn about streams, a concept closely related to futures.

Mini-exercises

1. Use the `Future.delayed` constructor to provide a string after two seconds that says "I am from the future."

2. Create a `String` variable named `message` that awaits the future to complete with a value.

3. Surround the code above with a `try-catch` block.

Streams

A future represents a single value that will arrive in the future. A **stream**, on the other hand, represents *multiple*

values that will arrive in the future. Think of a stream like a list of futures. You can imagine a stream meandering through the woods as the autumn leaves fall onto the surface of the water. Each time a leaf floats by, it's like the value that a Dart stream provides.

Streaming music online as opposed to downloading the song before playing it is another good comparison. When you stream music, you get lots of little chunks of data, but when you download the whole file, you only get a single value, which is the entire file — a little like what a future returns. In fact, the `http.get` command you used in the last section was actually implemented as a stream internally. However, Dart just waited until the stream finished and then returned all of the data at once in the form of a completed future.

Streams, which are of type `Stream`, are used extensively in Dart and Dart-based frameworks. Here are some examples:

- Reading a large file stored locally where new data from the file comes in chunks.
- Downloading a file from a remote server.
- Listening for requests coming into a server.
- Representing user events such as button clicks.
- Relaying changes in app state to the UI.

While it's possible to build your own streams from scratch, most of the time you don't need to do that. You only need to

be able to use the streams that Dart or a Dart package provides for you, which is what this section will teach you.

Subscribing to a stream

The `dart:io` library contains a `File` class which allows you to read data from a file. First, you'll read data the easy way using the `readAsString` method, which returns the contents of the file as a future. Then you'll do it again by reading the data as a stream of bytes.

Adding an assets file

You need a text file to work with, so you'll add that to your project now.

Create a new folder named **assets** in the root of your project. In that folder, create a file named **text.txt**. Add some text to the file. Although any text will work, *Lorem Ipsum* is a good standby:

```
Lorem ipsum dolor sit amet, consectetur adipisci
ng elit, sed do eiusmod tempor incididunt ut lab
ore et dolore magna aliqua. Ut enim ad minim ven
iam, quis nostrud exercitation ullamco laboris n
isi ut aliquip ex ea commodo consequat. Duis aut
e irure dolor in reprehenderit in voluptate veli
t esse cillum dolore eu fugiat nulla pariatur. E
xcepteur sint occaecat cupidatat non proident, s
unt in culpa qui officia deserunt mollit anim id
est laborum.
```

Then save the file.

Note: *Lorem Ipsum* is often used as filler text by graphic designers and app developers when the meaning of the text doesn't matter. The Latin words were taken from the writings of the Roman statesman and philosopher Cicero but modified so as to become essentially meaningless.

Reading as a string

Now that you've created the text file, replace your Dart code with the following:

```
import 'dart:io';

Future<void> main() async {
  final file = File('assets/text.txt');
  final contents = await file.readAsString();
  print(contents);
}
```

Here's what's new:

- `File` takes the relative path to your text file as the argument.
- `readAsString` returns `Future<String>`, but by using `await` you'll receive the string itself when it's ready.

`File` also has a `readAsStringSync` method, which would run synchronously and avoid awaiting a future. However, doing so would block your app if the reading takes a while. Many of

the methods on `File` have synchronous versions, but in order to prevent blocking your app, you should generally prefer the asynchronous versions.

Run the code above, and you'll see the contents of `text.txt` printed to the console.

Increasing the file size

If the file is large, you can read it as a stream. This allows you to start processing the data more quickly, since you don't have to wait to finish reading the entire file as you did in the last example.

When you read a file as a stream, it reads the file in chunks. The size of the chunks depends on how Dart is implemented on the system you're using, but it's probably 65,536 bytes per chunk as it was on the local machines used when writing this chapter. The `text.txt` file with *Lorem Ipsum* that you created earlier is only 445 bytes, so that means trying to stream that file would be no different than simply reading the whole thing as you did before.

In order to get a text file large enough to stream in chunks, create a new file in the **assets** folder called **text_long.txt.** Copy the *Lorem Ipsum* text and paste it in **text_long.txt** as new lines so that there are 1000 *Lorem Ipsum* copies. You can of course select-all and recopy from time to time, unless you find it therapeutic to paste things a thousand times. Save the file and you'll be ready to proceed.

Alternatively, you can find **text_long.txt** in the **assets** folder of the **final** project that comes with this chapter.

Reading from a stream

Replace the contents in the body of the `main` function with the following code:

```
final file = File('assets/text_long.txt');
final stream = file.openRead();
stream.listen(
  (data) {
    print(data.length);
  },
);
```

Here are a few points to note:

- Instead of calling `readAsString` on `file`, this time you're calling `openRead`, which returns an object of type `Stream<List<int>>`. That's a lot of angle brackets, but `Stream<List<int>>` simply means that it's a stream that periodically produces a list, and that list is a list of integers. The integers are the byte values, and the list is the chunk of data that's being passed in.

- To subscribe for notifications whenever there's new data coming in the stream, you call the `listen` method and pass it an anonymous function that takes a single parameter. The `data` parameter here is of type `List<int>`, so now you have access to the chunk of data coming in from the file.

- Since each integer in the list is one byte, calling `data.length` will tell you the number of bytes in the chunk.

Note: By default, only a single object can listen to a stream. This is known as a **single subscription stream**. If you want more than one object to be notified of stream events, you need to create a **broadcast stream**, which you could do like so:

```
final broadcastStream =
stream.asBroadcastStream();
```

Run the code in `main` and you should see something similar to the following:

```
65536
65536
65536
65536
65536
65536
52783
```

The data, at least on the computer used while writing this chapter, was all in 65,536-byte chunks until the final one, which was smaller since it didn't quite fill up the 65,536-byte buffer size. Your final chunk may be a different size than the one shown here, depending on how therapeutic your copy-and-paste session was.

Using an asynchronous for loop

Just as you can use callbacks or `async-await` to get the value of a future, there are also two ways to get the values of a

stream. In the example above, you used the `listen` callback. Here is the same example using an asynchronous for loop:

```
Future<void> main() async {
  final file = File('assets/text_long.txt');
  final stream = file.openRead();
  await for (var data in stream) {
    print(data.length);
  }
}
```

The `await for` keywords cause the loop to pause until the next data event comes in. Run this and you'll see the same results as before.

Error handling

Like futures, stream events can also produce an error rather than a value. You can handle errors using a callback or try-catch blocks.

Using a callback

One way to handle errors is to use the `onError` callback like so:

```
final file = File('assets/text_long.txt');
final stream = file.openRead();
stream.listen(
  (data) {
    print(data.length);
  },
  onError: (error) {
    print(error);
  },
  onDone: () {
    print('All finished');
  },
);
```

Here are a couple of points to note:

- When an error occurs, it won't cancel the stream, and you'll continue to receive more data events. If you actually did want to cancel the stream after an error, then `listen` also has a `cancelOnError` parameter which you can set to `true`.
- When a stream finishes sending all of its data, it'll fire a done event. This gives you a chance to respond with an `onDone` callback.

Using try-catch

The other way to handle errors on a stream is with a try-catch block in combination with `async-await`. Here is what that looks like:

```
try {
  final   file   =   File("assets/text_long.txt");
  final stream = file.openRead();
  await for (var data in stream) {
    print(data.length);
  }
} on Exception catch (error) {
    print(error);
} finally {
  print("All   finished");
}
```

In this example, you're catching all exceptions. A more robust solution would check for specific errors like `FileSystemException`, which would be thrown if the file didn't exist.

Run either the callback version, or the try-catch version, and you'll see the same chunk sizes as before, with the additional text "All finished" printed at the end.

Change the filename to something nonexistent, like `pink_elephants.txt`, and run the code again. Confirm that you have a `FileSystemException`.

```
FileSystemException: Cannot open file, path = "as
sets/pink_elephants.txt" (OS Error: No such file
or directory, errno = 2)
All  finished
```

Even with the exception, the `finally` block, or onDone callback if that's what you used, still printed "All finished".

Cancelling a stream

As mentioned above, you may use the `cancelOnError` parameter to tell the stream that you want to stop listening in the event of an error. However, even if there isn't an error, you should always cancel your subscription to a stream if you no longer need it. This allows Dart to clean up the memory the stream was using. Failing to do so can cause a memory leak.

Replace your Dart code with the following version:

```
import 'dart:async';
import 'dart:io';

Future<void> main() async {
  final file = File('assets/text_long.txt');
  final stream = file.openRead();
  StreamSubscription<List<int>>? subscription;
  subscription = stream.listen(
    (data) {
      print(data.length);
      subscription?.cancel();
    },
    cancelOnError: true,
    onDone: () {
      print('All finished');
    },
  );
}
```

Calling `listen` returns a `StreamSubscription`, which is part of the `dart:async` library. Keeping a reference to that in the `subscription` variable allows you to cancel the subscription

whenever you want. In this case, you cancel it after the first data event.

Run the code and you'll only see `65536` printed once. The `onDone` callback was never called because the stream never completed.

Transforming a stream

Being able to transform a stream as the data is coming in is very powerful. In the examples above, you never did anything with the data except print the length of the list of bytes. Those bytes represent text, though, so you're going to transform the data from numbers to text.

For this demonstration, there's no need to use a large text file so you'll switch back to the 445-byte version of *Lorem Ipsum* in **text.txt**.

Viewing the bytes

Replace the contents of `main` with the following code:

```
final file = File('assets/text.txt');
final stream = file.openRead();
stream.listen(
  (data) {
    print(data);
  },
);
```

Run that and you'll see a long list of bytes in decimal form:

```
[76, 111, 114, 101, ... ]
```

Although different computers encode text files using different encodings, the abbreviated list above is from a computer that uses UTF-8 encoding. You may recall that UTF-16 uses 16-bit, or 2-byte, code units to encode Unicode text. UTF-8 uses one to four 8-bit units to encode Unicode text. Since for values of 127 and below, UTF-8 and Unicode code points are the same, English text only takes one byte per letter. This makes file sizes smaller than UTF-16 encoding, which is beneficial when saving to disk or sending over a network.

If you look up 76 in Unicode you'll see that it's the capital letter L, 111 is o, and on it goes with `Lorem ipsum dolor sit`....

Decoding the bytes

Next, you'll take the UTF-8 bytes and convert them to a string.

Make sure you have the following imports and `main` method:

```
import 'dart:convert';
import 'dart:io';

Future<void> main() async {
  final file = File('assets/text.txt');
  final stream = file.openRead();
  await for (var data in stream.transform(utf8.d
ecoder)) {
    print(data);
  }
}
```

The difference here is that you added the `transform` method to the stream. This method takes the input from the original stream, transforms it with a `StreamTransformer`, and outputs a new stream, which you can listen to or loop over as before. In this case, the stream transformer was the dart:convert library's `utf8.decoder`, which takes a list of bytes and converts them to a string.

Run the code and you'll see the *Lorem Ipsum* passage printed in plain text.

Mini-exercises

The following code produces a stream that outputs an integer every second and then stops after the tenth time.

```
Stream<int>.periodic(
  Duration(seconds: 1),
  (value) => value,
).take(10);
```

1. Set the stream above to a variable named `myStream`.

2. Use `await for` to print the value of the integer on each data event coming from the stream.

Isolates

Most of the time it's fine to run your own code synchronously, and for long-running I/O tasks, you can use Dart libraries that return futures or streams. However you may sometimes discover that your code is too computationally expensive and degrades the performance of your app.

App stopping synchronous code

Have a look at this example:

```
String playHideAndSeekTheLongVersion() {
  var counting = 0;
  for (var i = 1; i <= 10000000000; i++) {
    counting = i;
  }
  return '$counting! Ready or not, here I come!'
;
}
```

Counting to ten billion takes a while — even for a computer! If you run that function in a Flutter app, your app's UI would freeze until the function finishes.

Try running that function from `main` like so:

```
print("OK, I'm counting...");
print(playHideAndSeekTheLongVersion());
```

You'll notice a significant wait until the counting finishes.

App stopping asynchronous code

Since you've read this far in the chapter, you should be aware that making the function asynchronous doesn't fix the problem:

```
Future<String> playHideAndSeekTheLongVersion() a
sync {
  var counting = 0;
  await Future(() {
    for (var i = 1; i <= 10000000000; i++) {
      counting = i;
    }
  });
  return "$counting! Ready or not, here I come!"
;
}
```

Run that using `await`:

```
print("OK, I'm counting...");
print(await  playHideAndSeekTheLongVersion());
```

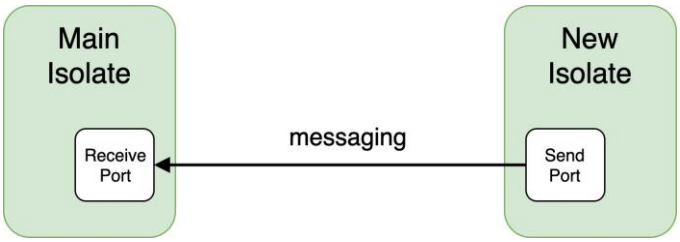
Adding the computationally intensive loop as an anonymous function in a `Future` constructor does indeed make it a future. However, think about what's going on here. Dart simply puts that anonymous function at the end of the event queue. True, all the events before it will get to go first, but

once the 10-billion-counter-loop gets to the end of the queue, it'll start running synchronously and block the app until it finishes. Using a future only delays the eventual block.

Spawning an isolate

When you're used to using futures from the Dart I/O libraries, it's easy to get lulled into thinking that futures always run in the background, but that's not the case. If you want to run some computationally intensive code on another thread, then you'll need to create a new isolate to do that.

The term for creating an isolate in Dart is called **spawning**. Since isolates don't share any memory with each other, they can only communicate by sending messages. When you spawn a new isolate, you give it a message communication object called a **send port**. The new isolate uses the send port to send messages back to a, which is listening on the main isolate.



In this example, the communication is only one way, although it's also possible to set up two-way communication between isolates.

Using a send port to return results

Add the new version of `playHideAndSeekTheLongVersion` as a top level method in your file:

```
import 'dart:isolate';

void playHideAndSeekTheLongVersion(SendPort send
Port) {
  var counting = 0;
  for (var i = 1; i <= 1000000000; i++) {
    counting = i;
  }
  sendPort.send('$counting! Ready or not, here I
come!');
}
```

Note that now it's a `void` function that takes a `SendPort` object as a parameter. `SendPort` is like one of those emergency mobile phones for kids where the phone can only call home. Home in this case is the main isolate. Instead of returning a string from the function like you were doing before, this time you're sending it as a message over the send port. Back in the main isolate, there will be a receive port listening for the message.

Spawning the isolate and listening for messages

Replace the main method with the following code:

```dart
Future<void> main() async {
  // 1
  final receivePort = ReceivePort();

  // 2
  final isolate = await Isolate.spawn(
    playHideAndSeekTheLongVersion,
    // 3
    receivePort.sendPort,
  );

  // 4
  receivePort.listen((message) {
    print(message);
    // 5
    receivePort.close();
    isolate.kill();
  });
}
```

Here's what you did:

1. You created a receive port to listen for messages from the new isolate.

2. Next, you spawned a new isolate and gave it two arguments. The first argument is the function that you want the isolate to execute. That function must be a top-level or static function. It must also take a single parameter. The second argument of `spawn` will be passed as the argument to `playHideAndSeekTheLongVersion`.

3. The `receivePort` has a `sendPort` that belongs to it. This is the part where Mommy gives little Timmy the phone

and says to call home if anything happens. The second parameter of `spawn` isn't actually required to be a `SendPort` object, but how is Timmy going to call home without a phone? If you want to pass additional parameters to the function, you can make the second parameter of `spawn` be a list or a map in which one of the elements is a `SendPort` and the other elements are additional arguments.

4. Finally, `receivePort.listen` gets a callback whenever `sendPort` sends a message. This is where Mommy carries her phone with her wherever she goes, always waiting for a call from Timmy.

5. In this example, the isolate is no longer needed after the work is done, so you can close the receive port and kill the isolate to free up the memory. This is where the Mommy-Timmy analogy fails. Mommy goes and saves Timmy before anything bad happens.

Note: The Flutter framework has a highly simplified way to start a new isolate, perform some work, and then return the result using a function called `compute`. Rather than passing the function a send port, you just pass it any values that are needed. In this case, you could just pass it the number to count to:

```
await compute(playHideAndSeekTheLongVersion,
10000000000);
```

That's enough to get you started on working with isolates. As a word of advice, though, don't feel like you need to preoptimize everything you think might be a computationally intensive task. Write your code as if it will all run on the main isolate. Only after you encounter performance problems will you need to start thinking about moving some code to a separate isolate.

Challenges

Before moving on, here are some challenges to test your knowledge of asynchronous programming. It's best if you try to solve them yourself, but if you get stuck, solutions are available in the `challenge` folder of this chapter.

Challenge 1: Whose turn is it?

This is a fun one and will test how well you understand how Dart handles asynchronous tasks. In what order will Dart print the text with the following `print` statements? Why?

```
void main() {
  print('1  synchronous');
  Future(() => print('2 event queue')).then(
    (value) => print('3 synchronous'),
  );
  Future.microtask(() => print('4 microtask queu
e'));
  Future.microtask(() => print('5 microtask queu
e'));
  Future.delayed(
    Duration(seconds: 1),
    () => print('6 event queue'),
  );
  Future(() => print('7 event queue')).then(
    (value) => Future(() => print('8 event queu
e')),
  );
  Future(() => print('9 event queue')).then(
    (value) => Future.microtask(
      () => print('10 microtask queue'),
    ),
  );
  print('11  synchronous');
}
```

Try to answer before checking. If you're right, give yourself a well-deserved pat on the back!

Challenge 2: Care to make a comment?

The following link returns a JSON list of comments:

```
https://jsonplaceholder.typicode.com/comments
```

Create a `Comment` data class and convert the raw JSON to a Dart list of type `List<Comment>`.

Challenge 3: Data stream

The following code allows you to stream content from the given URL:

```
final url = Uri.parse('https://raywenderlich.com'
);
final client = http.Client();
final request = http.Request('GET', url);
final response = await client.send(request);
final stream = response.stream;
```

Your challenge is to transform the stream from bytes to strings and see how many bytes each data chunk is. Add error handling, and when the stream is finished, close the client.

Challenge 4: Fibonacci from afar

In Challenge 4 of Chapter 4, you wrote some code to calculate the *n*th Fibonacci number. Repeat that challenge, but run the code in a separate isolate. Pass the value of *n* to the new isolate as an argument, and send the result back to the main isolate.

Key points

- Dart is single-threaded and handles asynchronous programming through concurrency, rather than through parallelism.

- Concurrency refers to rescheduling tasks to run later on the same thread, while parallelism refers to running tasks at the same time on different threads.

- The way Dart implements the scheduling of asynchronous tasks is by using an event loop, which has an event queue and a microtask queue.

- Synchronous code always runs first and cannot be interrupted. This is followed by anything in the microtask queue, and when these are completed, by any tasks in the event queue.

- You may run Dart code on another thread only by spawning a new isolate.

- Dart isolates do not share any memory state and may only communicate through messages.

- Using a future, which is of type `Future`, tells Dart that the requested task may be rescheduled on the event loop.

- When a future completes, it will contain either the requested value or an error.

- A method that returns a future doesn't necessarily run on a different process or thread. That depends entirely on the implementation.

- A stream, which is of type `Stream`, is a series of futures.

- Using a stream enables you to handle data events as they happen rather than waiting for them all to finish.

- You can handle errors on futures and streams with callbacks or `try-catch` blocks.

Where to go from here?

This chapter taught you how to use futures and streams, but a good next step would be learning how to create them yourself. There's also a lot more that you can do through stream manipulation.

If you enjoyed making HTTP requests to access resources from a remote server, you should consider server-side development with Dart as well. Being able to use a single language for both the frontend and the backend is nothing short of amazing. No cognitive switching is required, because everything you learned in this book applies to writing Dart code on the server.