Asynchronous programming: Streams

What's the point?

- Streams provide an asynchronous sequence of data.
- Data sequences include user-generated events and data read from files.
- You can process a stream using either **await for** or `listen()` from the Stream API.
- Streams provide a way to respond to errors.
- There are two kinds of streams: single subscription or broadcast.

Asynchronous programming in Dart is characterized by the Future and Stream classes.

A Future represents a computation that doesn't complete immediately. Where a normal function returns the result, an asynchronous function returns a Future, which will eventually contain the result. The future will tell you when the result is ready.

A stream is a sequence of asynchronous events. It is like an asynchronous Iterable—where, instead of getting the next event when you ask for it, the stream tells you that there is an event when it is ready.

Receiving stream events

Streams can be created in many ways, which is a topic for another article, but they can all be used in the same way: the *asynchronous for loop* (commonly just called **await for**) iterates over the events of a stream like the **for loop** iterates over an Iterable. For example:

```dart
Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  await for (final value in stream) {
    sum += value;
  }
  return sum;
}
```

This code simply receives each event of a stream of integer events, adds them up, and returns (a future of) the sum. When the loop body ends, the function is paused until the next event arrives or the stream is done.

The function is marked with the `async` keyword, which is required when using the **await for** loop.

The following example tests the previous code by generating a simple stream of integers using an `async*` function:

> **Note**
>
> This page uses embedded DartPads to display runnable examples. If you see empty boxes instead of DartPads, go to the DartPad troubleshooting page.

> **Note**
>
> Click **Run** to see the result in the **Console**.

Error events

Streams are done when there are no more events in them, and the code receiving the events is notified of this just as it is notified that a new event arrives. When reading events using an **await for** loop, the loops stops when the stream is done.

In some cases, an error happens before the stream is done; perhaps the network failed while fetching a file from a remote server, or perhaps the code creating the events has a bug, but someone needs to know about it.

Streams can also deliver error events like it delivers data events. Most streams will stop after the first error, but it is possible to have streams that deliver more than one error, and streams that deliver more data after an error event. In this document we only discuss streams that deliver at most one error.

When reading a stream using **await for**, the error is thrown by the loop statement. This ends the loop, as well. You can catch the error using **try-catch**. The following example throws an error when the loop iterator equals 4:

> **Note**
>
> Click **Run** to see the result in the **Console**.

Working with streams

The Stream class contains a number of helper methods that can do common operations on a stream for you, similar to the methods on an Iterable. For example, you can find the last positive integer in a stream using `lastWhere()` from the Stream API.

```dart
Future<int> lastPositive(Stream<int> stream) =>
    stream.lastWhere((x) => x >= 0);
```

Two kinds of streams

There are two kinds of streams.

Single subscription streams

The most common kind of stream contains a sequence of events that are parts of a larger whole. Events need to be delivered in the correct order and without missing any of them. This is the kind of stream you get when you read a file or receive a web request.

Such a stream can only be listened to once. Listening again later could mean missing out on initial events, and then the rest of the stream makes no sense. When you start listening, the data will be fetched and provided in chunks.

Broadcast streams

The other kind of stream is intended for individual messages that can be handled one at a time. This kind of stream can be used for mouse events in a browser, for example.

You can start listening to such a stream at any time, and you get the events that are fired while you listen. More than one listener can listen at the same time, and you can listen again later after canceling a previous subscription.

Methods that process a stream

The following methods on Stream<T> process the stream and return a result:

```dart
Future<T> get first;
Future<bool> get isEmpty;
Future<T> get last;
Future<int> get length;
Future<T> get single;
Future<bool> any(bool Function(T element) test);
Future<bool> contains(Object? needle);
Future<E> drain<E>([E? futureValue]);
Future<T> elementAt(int index);
Future<bool> every(bool Function(T element) test);
Future<T> firstWhere(bool Function(T element) test, {T Function()? orElse});
Future<S> fold<S>(S initialValue, S Function(S previous, T element) combine);
Future forEach(void Function(T element) action);
Future<String> join([String separator = '']);
Future<T> lastWhere(bool Function(T element) test, {T Function()? orElse});
Future pipe(StreamConsumer<T> streamConsumer);
Future<T> reduce(T Function(T previous, T element) combine);
Future<T> singleWhere(bool Function(T element) test, {T Function()? orElse});
Future<List<T>> toList();
Future<Set<T>> toSet();
```

All of these functions, except `drain()` and `pipe()`, correspond to a similar function on Iterable. Each one can be written easily by using an `async` function with an **await for** loop (or just using one of the other methods). For example, some implementations could be:

```dart
Future<bool> contains(Object? needle) async {
  await for (final event in this) {
    if (event == needle) return true;
  }
  return false;
}

Future forEach(void Function(T element) action) async {
  await for (final event in this) {
    action(event);
  }
}

Future<List<T>> toList() async {
  final result = <T>[];
  await forEach(result.add);
  return result;
}

Future<String> join([String separator = '']) async =>
    (await toList()).join(separator);
```

(The actual implementations are slightly more complex, but mainly for historical reasons.)

Methods that modify a stream

The following methods on Stream return a new stream based on the original stream. Each one waits until someone listens on the new stream before listening on the original.

```dart
Stream<R> cast<R>();
Stream<S> expand<S>(Iterable<S> Function(T element) convert);
Stream<S> map<S>(S Function(T event) convert);
Stream<T> skip(int count);
Stream<T> skipWhile(bool Function(T element) test);
Stream<T> take(int count);
Stream<T> takeWhile(bool Function(T element) test);
Stream<T> where(bool Function(T event) test);
```

The preceding methods correspond to similar methods on Iterable which transform an iterable into another iterable. All of these can be written easily using an `async` function with an **await for** loop.

```dart
Stream<E> asyncExpand<E>(Stream<E>? Function(T event) convert);
Stream<E> asyncMap<E>(FutureOr<E> Function(T event) convert);
Stream<T> distinct([bool Function(T previous, T next)? equals]);
```

The `asyncExpand()` and `asyncMap()` functions are similar to `expand()` and `map()`, but allow their function argument to be an asynchronous function. The `distinct()` function doesn't exist on `Iterable`, but it could have.

```dart
Stream<T> handleError(Function onError, {bool Function(dynamic error)? test});
Stream<T> timeout(Duration timeLimit,
    {void Function(EventSink<T> sink)? onTimeout});
Stream<S> transform<S>(StreamTransformer<T, S> streamTransformer);
```

The final three functions are more special. They involve error handling which an **await for** loop can't do—the first error reaching the loops will end the loop and its subscription on the stream. There is no recovering from that. The following code shows how to use `handleError()` to remove errors from a stream before using it in an **await for** loop.

```dart
Stream<S> mapLogErrors<S, T>(
  Stream<T> stream,
  S Function(T event) convert,
) async* {
  var streamWithoutErrors = stream.handleError((e) => log(e));
  await for (final event in streamWithoutErrors) {
    yield convert(event);
  }
}
```

The transform() function

The `transform()` function is not just for error handling; it is a more generalized "map" for streams. A normal map requires one value for each incoming event. However, especially for I/O streams, it might take several incoming events to produce an output event. A StreamTransformer can work with that. For example, decoders like Utf8Decoder are transformers. A transformer requires only one function, bind(), which can be easily implemented by an `async` function.

Reading and decoding a file

The following code reads a file and runs two transforms over the stream. It first converts the data from UTF8 and then runs it through a LineSplitter. All lines are printed, except any that begin with a hashtag, `#`.

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

void main(List<String> args) async {
  var file = File(args[0]);
  var lines = utf8.decoder
      .bind(file.openRead())
      .transform(const LineSplitter());
  await for (final line in lines) {
    if (!line.startsWith('#')) print(line);
  }
}
```

The listen() method

The final method on Stream is `listen()`. This is a "low-level" method—all other stream functions are defined in terms of `listen()`.

```dart
StreamSubscription<T> listen(void Function(T event)? onData,
    {Function? onError, void Function()? onDone, bool? cancelOnError});
```

To create a new `Stream` type, you can just extend the `Stream` class and implement the `listen()` method—all other methods on `Stream` call `listen()` in order to work.

The `listen()` method allows you to start listening on a stream. Until you do so, the stream is an inert object describing what events you want to see. When you listen, a StreamSubscription object is returned which represents the active stream producing events. This is similar to how an `Iterable` is just a collection of objects, but the iterator is the one doing the actual iteration.

The stream subscription allows you to pause the subscription, resume it after a pause, and cancel it completely. You can set callbacks to be called for each data event or error event, and when the stream is closed.

Other resources

Read the following documentation for more details on using streams and asynchronous programming in Dart.

- Creating Streams in Dart, an article about creating your own streams
- Futures and Error Handling, an article that explains how to handle errors using the Future API
- Asynchrony support, a section in the language tour
- Stream API reference