```
  Future<String> content = File('file.txt').readAsString();
  content.then((content) {
    print(content);
  });
}
```

readAsStringメソッドは非同期にファイルの内容を読み取り、文字列として返します。戻り値の型はFuture<String>型です。Futureのthenメソッドには処理が完了したときに呼び出されるコールバックを渡します。

Futureクラスはasync、awaitキーワードと組み合わせることで、同期的なコードのように記述できます。

```
import 'dart:io';

Future<void> main() async {
  String content = await File('file.txt').readAsString();
  print(content);
}
```

readAsStringメソッドの呼び出しにawaitキーワードを付与しました。これにより、readAsStringが終了するまで待機します。また、戻り値のFuture<String>型をString型に自動的に変換します。コールバックのネストが減り、コードが簡潔になります。

重要なポイントとしてawaitキーワードはasyncキーワードを付与したメソッド内でしか使えません。また、asyncキーワードを付与したメソッドの戻り値は暗黙的にFutureクラスでラップされます。main()の本体にasyncキーワードを付与し、戻り値はFuture<void>に変更しています。

エラーハンドリング

Future型のエラーハンドリングです。catchErrorメソッドで例外が発生したときに呼び出されるコールバックを登録します。

```
// 戻り値がFuture型、例外をスローする関数
Future<String> fetchUserName() {
  var str = Future.delayed(
      const Duration(seconds: 1),
      () => throw 'User not found.');
  return str;
}

fetchUserName()
  .then((name) {
```

```
    print('User name is $name');
  })
  .catchError((e) {
    print(e);
  });
// => User not found.
```

async-awaitで実行した非同期処理はtry-catch構文で例外を捕捉します。

```
// 戻り値がFuture型、例外をスローする関数
Future<String> fetchUserName() {
  var str = Future.delayed(
      const Duration(seconds: 1),
      () => throw 'User not found.');
  return str;
}

try {
  final name = await fetchUserName();
  print('User name = $name');
} catch (e) {
  print(e);
}
// => User not found.
```

例外発生時に返す代替の値がある場合はthenメソッドの引数onErrorで処理する方法があります。

```
// 戻り値がFuture型、例外をスローする関数
Future<String> fetchUserName() {
  var str = Future.delayed(
      const Duration(seconds: 1),
      () => throw 'User not found.');
  return str;
}

final result = await fetchUserName()
  .then((name) {
    // fetchUserName関数が正常終了した場合の値を返す
    return 'User name is $name';
  },
  onError: (e, st) {
    // fetchUserName関数で例外が発生した場合の値を返す
    return 'Unknown user';
  },
);
print(result);
// => Unknown user
```

Stream型

非同期に連続した値を扱うStream型です。

```
import 'dart:io';

void main() {
  final file = File('file.txt');
  final Stream<List<int>> stream = file.openRead();
  stream.listen((data) {
    print('data: ${data.length} bytes');
  });
}
```

openReadメソッドはファイルを読み込み、一定のサイズごとにデータを返します。戻り値の型はStream<List<int>>型です。Streamはlistenメソッドで購読し、データが通知されたときに呼び出されるコールバックを登録します。

Futureクラスと同様にasyncとawait forキーワードと組み合わせることで、同期的なコードのように記述できます。

```
import 'dart:io';

Future<void> main() async {
  final file = File('file.txt');
  final Stream<List<int>> stream = file.openRead();
  await for (final data in stream) {
    print('data: ${data.length} bytes');
  };
}
```

Streamをfor文に渡し、awaitキーワードを付与しました。for文の一時変数dataにはStreamの値であるList<int>型のデータが代入されます。こちらもawaitを使うため関数にasyncキーワードを付与する必要があります。

Streamの購読をキャンセル、一時停止する

listenメソッドの戻り値はStreamSubscription型です。cancelメソッドで購読をキャンセルできます。

```
import 'dart:io';

void main() {
  final file = File('file.txt');
  final subscription = file.openRead()
    .listen((data) {
```

```
      print('data: ${data.length} bytes');
    }
  );

  Future<void> result = subscription.cancel(); // 購読をキャンセル
}
```

cancelを呼び出すと、以降イベントが通知されなくなります。購読がキャンセルされることで、Streamでリソースの解放処理が発生する場合があります。解放処理の完了や例外を検知するためにcancelメソッドの戻り値はFuture型となっています。たとえば、ファイルをopenReadメソッドで読み込んだ後に削除するようなケースで利用します。

また、購読を一時停止するpauseメソッド、購読を再開するresumeメソッドがあります。

```
import 'dart:io';

Future<void> main() async {
  final file = File('file.txt');
  final subscription = file.openRead()
    .listen((data) {
      print('data: ${data.length} bytes');
    }
  );

  await Future.delayed(const Duration(seconds: 1));
  subscription.pause(); // 購読を一時停止
  await Future.delayed(const Duration(seconds: 4));
  subscription.resume(); // 購読を再開
}
```

Stream型を生成する関数

Stream型を返す関数を実装するにはasync*キーワードを使います。関数が呼び出されるとStreamが生成され、Streamが購読されると関数の本体が実行されます。

```
import 'dart:io';

Stream<String> languages() async* {
  await Future.delayed(const Duration(seconds: 1));
  yield 'Dart';
  await Future.delayed(const Duration(seconds: 1));
  yield 'Kotlin';
```

```
  await Future.delayed(const Duration(seconds: 1));
  yield 'Swift';
  await Future.delayed(const Duration(seconds: 1));
  yield* Stream.fromIterable(['JavaScript', 'C++', 'Go']);
}
```

String型のStreamを生成する関数の例です。yieldに続いてStringを記述すると、その値が戻り値のStreamに通知されます。yield*に続いてStreamを記述すると、そのStreamの値が戻り値のStreamに通知されます。購読がキャンセルされた際は、次のyield文が実行されると関数の実行が中断されます。

Streamの終わり

Streamの終了時に処理を実行するにはlistenメソッドのonDoneにコールバックを渡します。

```
Stream<String> languages() async* {
  await Future.delayed(const Duration(seconds: 1));
  yield 'Dart';
  await Future.delayed(const Duration(seconds: 1));
  yield 'Kotlin';
  await Future.delayed(const Duration(seconds: 1));
  yield 'Swift';
  await Future.delayed(const Duration(seconds: 1));
  yield* Stream.fromIterable(['JavaScript', 'C++', 'Go']);
}

void main() async {
  languages().listen((language) {
    print(language);
  }, onDone: () {
    print('Done');
  });
}
// => Dart
// => Kotlin
// => Swift
// => JavaScript
// => C++
// => Go
// => Done
```

async - await forを使った場合は、Streamが終了するとfor文から抜けます。Streamの終了時に実行する処理はシンプルにfor文のあとに書けばOKです。

```
Stream<String> languages() async* {
  await Future.delayed(const Duration(seconds: 1));
  yield 'Dart';
  await Future.delayed(const Duration(seconds: 1));
  yield 'Kotlin';
  await Future.delayed(const Duration(seconds: 1));
  yield 'Swift';
  await Future.delayed(const Duration(seconds: 1));
  yield* Stream.fromIterable(['JavaScript', 'C++', 'Go']);
}

Future<void> main() async {
  await for (final language in languages()) {
    print(language);
  }
  print('Done');
}
// => Dart
// => Kotlin
// => Swift
// => JavaScript
// => C++
// => Go
// => Done
```

Streamは購読をキャンセルしない限り終了しない特性を持ったものもあり得ます。たとえば、`Stream.periodic`コンストラクタから得られるStreamは一定の間隔で繰り返し値を通知するStreamを生成します。このような終了しないStreamで`async - await for`を用いると以降の処理が実行されないので注意が必要です。

```
Future<void> main() async {
  await for (final count in Stream<int>.periodic(const Duration(seconds: 1), (i)
=> i)) {
    print(count);
  }

  print('Done!'); // この行は実行されない
}
```

エラーハンドリング

Streamの例外を処理するには`listen`メソッドの`onError`にコールバックを渡します。

```
Stream<String> languages() async* {
  await Future.delayed(const Duration(seconds: 1));
  yield 'Dart';
  await Future.delayed(const Duration(seconds: 1));
  throw Exception('Some error');
}

void main() {
  languages().listen((language) {
    print(language);
  }, onError: (e) {
    print(e);
  });
}
// => Dart
// => Exception: Some error
```

async - await forを使った場合は、try-catch構文で例外を捕捉します。

```
Stream<String> languages() async* {
  await Future.delayed(const Duration(seconds: 1));
  yield 'Dart';
  await Future.delayed(const Duration(seconds: 1));
  throw Exception('Some error');
}

Future<void> main() async {
  try {
    await for (final language in languages()) {
      print(language);
    }
  } catch (e) {
    print(e);
  }
}
// => Dart
// => Exception: Some error
```

また、listenメソッドの第四引数cancelOnErrorはStreamで例外が発生した場合に購読を中止するかどうかを指定できます。デフォルト値はfalseで、例外が発生しても購読は継続されます。

StreamController

async*関数よりも簡単にStreamを生成する方法としてStreamControllerクラスがあります。

```
import 'dart:async';

class Counter {
  int _count = 0;
  StreamController<int> _controller = StreamController<int>();

  Stream<int> get stream => _controller.stream;

  void increment() {
    _count++;
    _controller.add(_count);
  }
}

Future<void> main() async {
  final counter = Counter();
  counter.increment();
  counter.increment();

  counter.stream.listen((i) {
    print('count: $i');
  });

  counter.increment();
}
// => count: 1
// => count: 2
// => count: 3
```

Counterクラスは内部にStreamControllerを持ち、incrementメソッドが呼び出されるとStreamControllerに値を送信します。StreamControllerへの値の送信はaddメソッドで行います。async*関数では関数内でyieldを使ってイベントを送信しましたが、StreamControllerでは外部からイベントを送信できるため、より柔軟にStreamを扱うことができます。

このほか、例外を送信するaddErrorメソッドや、購読されているかどうかを判定するhasListenerプロパティなどがあります。

async*は購読されるまで関数の本体が実行されません。しかし、StreamControllerは購読されていなくてもaddメソッドで値を送信することができ、その値はバッファリングされ購読されたとき一斉に通知されます(購読の一時停止時も同様にバッファリングされます)。そのため、用途によりメモリを消費する可能性があるので注意が必要です。