Asynchronous programming: futures, async, await

This tutorial teaches you how to write asynchronous code using futures and the `async` and `await` keywords. Using embedded DartPad editors, you can test your knowledge by running example code and completing exercises.

To get the most out of this tutorial, you should have the following:

- Knowledge of basic Dart syntax.
- Some experience writing asynchronous code in another language.
- The `discarded_futures` and `unawaited_futures` lints enabled.

This tutorial covers the following material:

- How and when to use the `async` and `await` keywords.
- How using `async` and `await` affects execution order.
- How to handle errors from an asynchronous call using `try-catch` expressions in `async` functions.

Estimated time to complete this tutorial: 40-60 minutes.

ⓘ Note

This page uses embedded DartPads to display examples and exercises. If you see empty boxes instead of DartPads, go to the DartPad troubleshooting page.

The exercises in this tutorial have partially completed code snippets. You can use DartPad to test your knowledge by completing the code and clicking the **Run** button. **Don't edit the test code in the `main` function or below**.

If you need help, expand the **Hint** or **Solution** dropdown after each exercise.

Why asynchronous code matters

Asynchronous operations let your program complete work while waiting for another operation to finish. Here are some common asynchronous operations:

- Fetching data over a network.
- Writing to a database.
- Reading data from a file.

Such asynchronous computations usually provide their result as a `Future` or, if the result has multiple parts, as a `Stream`. These computations introduce asynchrony into a program. To accommodate that initial asynchrony, other plain Dart functions also need to become asynchronous.

To interact with these asynchronous results, you can use the `async` and `await` keywords. Most asynchronous functions are just async Dart functions that depend, possibly deep down, on an inherently asynchronous computation.

Example: Incorrectly using an asynchronous function

The following example shows the wrong way to use an asynchronous function (`fetchUserOrder()`). Later you'll fix the example using `async` and `await`. Before running this example, try to spot the issue -- what do you think the output will be?

Here's why the example fails to print the value that `fetchUserOrder()` eventually produces:

- `fetchUserOrder()` is an asynchronous function that, after a delay, provides a string that describes the user's order: a "Large Latte".
- To get the user's order, `createOrderMessage()` should call `fetchUserOrder()` and wait for it to finish. Because `createOrderMessage()` does *not* wait for `fetchUserOrder()` to finish, `createOrderMessage()` fails to get the string value that `fetchUserOrder()` eventually provides.
- Instead, `createOrderMessage()` gets a representation of pending work to be done: an uncompleted future. You'll learn more about futures in the next section.
- Because `createOrderMessage()` fails to get the value describing the user's order, the example fails to print "Large Latte" to the console, and instead prints "Your order is: Instance of '_Future<String>'".

In the next sections you'll learn about futures and about working with futures (using `async` and `await`) so that you'll be able to write the code necessary to make `fetchUserOrder()` print the desired value ("Large Latte") to the console.

Key terms

- **synchronous operation**: A synchronous operation blocks other operations from executing until it completes.
- **synchronous function**: A synchronous function only performs synchronous operations.
- **asynchronous operation**: Once initiated, an asynchronous operation allows other operations to execute before it completes.
- **asynchronous function**: An asynchronous function performs at least one asynchronous operation and can also perform *synchronous* operations.

What is a future?

A future (lower case "f") is an instance of the Future (capitalized "F") class. A future represents the result of an asynchronous operation, and can have two states: uncompleted or completed.

ⓘ Note

Uncompleted is a Dart term referring to the state of a future before it has produced a value.

Uncompleted

When you call an asynchronous function, it returns an uncompleted future. That future is waiting for the function's asynchronous operation to finish or to throw an error.

Completed

If the asynchronous operation succeeds, the future completes with a value. Otherwise, it completes with an error.

Completing with a value

A future of type `Future<T>` completes with a value of type `T`. For example, a future with type `Future<String>` produces a string value. If a future doesn't produce a usable value, then the future's type is `Future<void>`.

Completing with an error

If the asynchronous operation performed by the function fails for any reason, the future completes with an error.

Example: Introducing futures

In the following example, `fetchUserOrder()` returns a future that completes after printing to the console. Because it doesn't return a usable value, `fetchUserOrder()` has the type `Future<void>`. Before you run the example, try to predict which will print first: "Large Latte" or "Fetching user order...".

In the preceding example, even though `fetchUserOrder()` executes before the `print()` call on line 8, the console shows the output from line 8("Fetching user order...") before the output from `fetchUserOrder()` ("Large Latte"). This is because `fetchUserOrder()` delays before it prints "Large Latte".

Example: Completing with an error

Run the following example to see how a future completes with an error. A bit later you'll learn how to handle the error.

In this example, `fetchUserOrder()` completes with an error indicating that the user ID is invalid.

You've learned about futures and how they complete, but how do you use the results of asynchronous functions? In the next section you'll learn how to get results with the `async` and `await` keywords.

Quick review

- A Future<T> instance produces a value of type `T`.
- If a future doesn't produce a usable value, then the future's type is `Future<void>`.

- A future can be in one of two states: uncompleted or completed.
- When you call a function that returns a future, the function queues up work to be done and returns an uncompleted future.
- When a future's operation finishes, the future completes with a value or with an error.

Key terms:

- **Future**: the Dart Future class.
- **future**: an instance of the Dart `Future` class.

Working with futures: async and await

The `async` and `await` keywords provide a declarative way to define asynchronous functions and use their results. Remember these two basic guidelines when using `async` and `await`:

- **To define an async function, add `async` before the function body:**
- **The `await` keyword works only in `async` functions.**

Here's an example that converts `main()` from a synchronous to asynchronous function.

First, add the `async` keyword before the function body:

```dart
void main() async { ··· }
```

If the function has a declared return type, then update the type to be `Future<T>`, where `T` is the type of the value that the function returns. If the function doesn't explicitly return a value, then the return type is `Future<void>`:

```dart
Future<void> main() async { ··· }
```

Now that you have an `async` function, you can use the `await` keyword to wait for a future to complete:

```dart
print(await createOrderMessage());
```

As the following two examples show, the `async` and `await` keywords result in asynchronous code that looks a lot like synchronous code. The only differences are highlighted in the asynchronous example, which—if your window is wide enough—is to the right of the synchronous example.

Example: synchronous functions

```dart
String createOrderMessage() {
  var order = fetchUserOrder();
  return 'Your order is: $order';
}

Future<String> fetchUserOrder() =>
    // Imagine that this function is
    // more complex and slow.
    Future.delayed(
      const Duration(seconds: 2),
      () => 'Large Latte',
    );

void main() {
  print('Fetching user order...');
  print(createOrderMessage());
}
```

```
Fetching user order...
Your order is: Instance of 'Future<String>'
```

As shown in following two examples, it operates like synchronous code.

Example: asynchronous functions

```dart
Future<String> createOrderMessage() async {
  var order = await fetchUserOrder();
  return 'Your order is: $order';
}

Future<String> fetchUserOrder() =>
    // Imagine that this function is
    // more complex and slow.
    Future.delayed(
      const Duration(seconds: 2),
      () => 'Large Latte',
    );

Future<void> main() async {
  print('Fetching user order...');
  print(await createOrderMessage());
}
```

```
Fetching user order...
Your order is: Large Latte
```

The asynchronous example is different in three ways:

- The return type for `createOrderMessage()` changes from `String` to `Future<String>`.
- The **`async`** keyword appears before the function bodies for `createOrderMessage()` and `main()`.
- The **`await`** keyword appears before calling the asynchronous functions `fetchUserOrder()` and `createOrderMessage()`.

Key terms

- **async**: You can use the `async` keyword before a function's body to mark it as asynchronous.
- **async function**: An `async` function is a function labeled with the `async` keyword.
- **await**: You can use the `await` keyword to get the completed result of an asynchronous expression. The `await` keyword only works within an `async` function.

Execution flow with async and await

An `async` function runs synchronously until the first `await` keyword. This means that within an `async` function body, all synchronous code before the first `await` keyword executes immediately.

Example: Execution within async functions

Run the following example to see how execution proceeds within an `async` function body. What do you think the output will be?

After running the code in the preceding example, try reversing lines 2 and 3:

```dart
var order = await fetchUserOrder();
print('Awaiting user order...');
```

Notice that timing of the output shifts, now that `print('Awaiting user order')` appears after the first `await` keyword in `printOrderMessage()`.

Exercise: Practice using async and await

The following exercise is a failing unit test that contains partially completed code snippets. Your task is to complete the exercise by writing code to make the tests pass. You don't need to implement `main()`.

To simulate asynchronous operations, call the following functions, which are provided for you:

Function	Type signature	Description
fetchRole()	`Future<String> fetchRole()`	Gets a short description of the user's role.
fetchLoginAmount()	`Future<int> fetchLoginAmount()`	Gets the number of times a user has logged in.

Part 1: `reportUserRole()`

Add code to the `reportUserRole()` function so that it does the following:

- Returns a future that completes with the following string: `"User role: <user role>"`
  - Note: You must use the actual value returned by `fetchRole()`; copying and pasting the example return value won't make the test pass.
  - Example return value: `"User role: tester"`
- Gets the user role by calling the provided function `fetchRole()`.

Part 2: `reportLogins()`

Implement an `async` function `reportLogins()` so that it does the following:

- Returns the string `"Total number of logins: <# of logins>"`.
  - Note: You must use the actual value returned by `fetchLoginAmount()`; copying and pasting the example return value won't make the test pass.
  - Example return value from `reportLogins()`: `"Total number of logins: 57"`
- Gets the number of logins by calling the provided function `fetchLoginAmount()`.

▶ Hint

▶ Solution

Handling errors

To handle errors in an `async` function, use try-catch:

```dart
try {
  print('Awaiting user order...');
  var order = await fetchUserOrder();
} catch (err) {
  print('Caught error: $err');
}
```

Within an `async` function, you can write try-catch clauses the same way you would in synchronous code.

Example: async and await with try-catch

Run the following example to see how to handle an error from an asynchronous function. What do you think the output will be?

Exercise: Practice handling errors

The following exercise provides practice handling errors with asynchronous code, using the approach described in the previous section. To simulate asynchronous operations, your code will call the following function, which is provided for you:

Function	Type signature	Description
fetchNewUsername()	`Future<String> fetchNewUsername()`	Returns the new username that you can use to replace an old one.

Use `async` and `await` to implement an asynchronous `changeUsername()` function that does the following:

- Calls the provided asynchronous function `fetchNewUsername()` and returns its result.
  - Example return value from `changeUsername()`: `"jane_smith_92"`
- Catches any error that occurs and returns the string value of the error.
  - You can use the toString() method to stringify both Exceptions and Errors.

▶ Hint

▶ Solution

Exercise: Putting it all together

It's time to practice what you've learned in one final exercise. To simulate asynchronous operations, this exercise provides the asynchronous functions `fetchUsername()` and `logoutUser()`:

Function	Type signature	Description
fetchUsername()	`Future<String> fetchUsername()`	Returns the name associated with the current user.
logoutUser()	`Future<String> logoutUser()`	Performs logout of current user and returns the username that was logged out.

Write the following:

Part 1: `addHello()`

- Write a function `addHello()` that takes a single `String` argument.
- `addHello()` returns its `String` argument preceded by `'Hello '`.
  Example: `addHello('Jon')` returns `'Hello Jon'`.

Part 2: `greetUser()`

- Write a function `greetUser()` that takes no arguments.
- To get the username, `greetUser()` calls the provided asynchronous function `fetchUsername()`.
- `greetUser()` creates a greeting for the user by calling `addHello()`, passing it the username, and returning the result.
  Example: If `fetchUsername()` returns `'Jenny'`, then `greetUser()` returns `'Hello Jenny'`.

Part 3: `sayGoodbye()`

- Write a function `sayGoodbye()` that does the following:
    - Takes no arguments.
    - Catches any errors.
    - Calls the provided asynchronous function `logoutUser()`.
- If `logoutUser()` fails, `sayGoodbye()` returns any string you like.
- If `logoutUser()` succeeds, `sayGoodbye()` returns the string `'<result> Thanks, see you next time'`, where `<result>` is the string value returned by calling `logoutUser()`.

▶ Hint

▶ Solution

Which lints work for futures?

To catch common mistakes that arise while working with async and futures, enable the following lints:

- `discarded_futures`
- `unawaited_futures`

What's next?

Congratulations, you've finished the tutorial! If you'd like to learn more, here are some suggestions for where to go next:

- Play with DartPad.
- Try another tutorial.
- Learn more about futures and asynchronous code in Dart:
    - Streams tutorial: Learn how to work with a sequence of asynchronous events.
    - Concurrency in Dart: Understand and learn how to implement concurrency in Dart.
    - Asynchrony support: Dive in to Dart's language and library support for asynchronous coding.
    - Dart videos from Google: Watch one or more of the videos about asynchronous coding.
- Get the Dart SDK!