Introduction to the Dart Future

A future is an object that represents the result of an asynchronous operation. A future object will return a value at a later time.

A future has two states: uncompleted and completed. When a future completes, it has two possibilities:

- Completed with a value
- Failed with an error

Dart uses the `Future<T>` class for creating future objects. Because the `Future<T>` is a generic class, the value it returns can be any object.

In practice, you often consume future objects instead of creating them.

Dart Future example

The following example uses the `Future.delayed()` constructor to create a future object that returns the number 2 after one second:

```
var future = Future<int>.delayed(
 Duration(seconds: 1),
 () => 2
);
```

In this example:

- The `future` is an object of the `Future<int>` class.
- The first argument of the `Future<int>.delayed()` constructor is a `Duration` object. Dart will add the anonymous function in the second argument to the event queue and delay executing it by one second.
- The event loop will move the anonymous function `() => 2` to the main isolate and execute it.

When a future completes, you can get the return value. There are two ways to get the value after the future completes:

- Using a callback
- Using async and await keywords

Using callbacks

When a future completes, you can run a callback to handle the result.
The Future<T> class provides three methods for scheduling a callback:

- then()
- catchError()
- whenComplete()

If a future completes successfully with a value, you can get the result by adding a callback to the then() method.

If a future fails with an error, you can handle the error in the catchError() method.

Whether a future completes with a value or an error, you can schedule a callback in the whenComplete() method.

In other words, the callback that you pass to the whenComplete() method will always execute whether the future succeeds or not.

The following example illustrates how to use the above using the callback:

```
void main() {
 print(1);


 var future = Future<int>.delayed(Duration(seconds: 1), () => 2);
 future.then((value) => print(value));


 print(3);
}
```

Output:

```
1
3
2
```

In this example, the number 2 appears immediately after the number 3 because the future has zero delays.

Using async and await keywords

The `async` and `await` keywords provide a declarative way to define asynchronous functions.

When using `async` and `await` keywords, you should follow these rules:

- Place the `async` keyword before the function body to make the function asynchronous.
- The `await` keyword works only in `async` functions. In other words, if a function contains `await` keywords, you need to make it an `async` function.

The following example converts `main()` function from a synchronous to asynchronous function:

```
Future<void> main() async {
 print(1);


 var value = await Future<int>.delayed(
   Duration(seconds: 0),
   () => 2
 );
 print(value);


 print(3);
}
```

Output:

```
1
2
3
```

In this example:

- First, add the `async` keyword to the `main()` function to make it asynchronous. Since the `main()` function doesn't explicitly return a value, its return type is `Future<void>`.
- Second, add the `await` keyword in front of the future object. Once Dart encounters the `await` keyword, it sends all the code starting from that line to the event queue and waits for the future to complete. That's why you see the number 2 before 3.

Handling errors with try-catch block

If an asynchronous operation results in an error, you can use the try-catch block to handle it. For example:

```
Future<void> main() async {
 print(1);
 try {
  var value = await Future<int>.delayed(
    Duration(seconds: 0),
    () => throw Exception('An error from the future'));
  print(value);
 } catch (e) {
  print(e);
 }
 print(3);
}
```

Output:

```
1
Exception: Error
3
```

In this example, the `Future` object raises an exception. And we use the `try-catch` statement to catch it and display the error message.

Dart future practical example

The following example reads the contents of a text file and prints it out:

```
import 'dart:io';


Future<void> main() async {
 try {
   var file = File('readme1.txt');
   var contents = await file.readAsString();
   print(contents);
 } on FileSystemException catch (e) {
   print(e);
 }
}
```

How it works.

First, import the `dart:io` library:

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

Second, create a `File` object with a file path:

```
var file = File('readme.txt');
```

Third, read the contents of a file using the `readAsString()` method. The method returns a `Future<String>` which is the contents of the text file:

```
var contents = await file.readAsString();
```

If the read is successful, display the text file's contents:

```
print(contents);
```

If the file is not found or an error while reading a file, display the error message in the `catch` block.

Summary

- A future is an object that represents the result of an asynchronous operation.
- A future completes successfully with a value or fails with an error.
- Dart uses the Future<T> class for creating future objects.
- Use the `async` keyword to define asynchronous functions.
- Use the `await` keyword to wait for a future to complete.