Asynchronous Functions and async / await

async functions

An `async function` is a special type of function in JavaScript that allows you to write asynchronous and promise-based code in a more readable and organized way. You can use the `await` keyword inside an async function to pause its execution until a promise is resolved, making it easier to work with asynchronous tasks without dealing with complex promise chains. In other words, async functions make writing asynchronous code cleaner and more straightforward.

Syntax

```
async function name(param0) {
  statements
}
async function name(param0, param1) {
  statements
}
async function name(param0, param1, /* …, */ paramN) {
  statements
}
```

Parameters

`name`

The function's name.

`param`

Optional, the name of a formal parameter for the function. These parameters act as placeholders for values that you can pass into the function when calling it.

`statements`

Optional, statements refer to the code inside the async function that defines its actions. You can use the await mechanism within these statements to handle asynchronous tasks, allowing you to control the flow of your function based on the resolution of promises.

Description

An `async function` declaration creates an `AsyncFunction` object. Each time when an async function is called, it returns a new `Promise` which will be resolved with the value returned by the async function, or rejected with an exception uncaught within the async function.

Async functions can contain zero or more `await` expressions. Await expressions make promise-returning functions behave as though they're synchronous by suspending execution until the returned promise is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression. Use of `async` and `await` enables the use of ordinary `try / catch` blocks around asynchronous code.

The `await` keyword is only valid inside async functions within regular JavaScript code. If you use it outside of an async function's body, you will get a SyntaxError.

Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.

For example, consider the following code:

```
async function foo() {
  return 1;
}
```

It is similar to:

```
function foo() {
  return Promise.resolve(1);
}
```

Note: Even though the return value of an async function behaves as if it's wrapped in a `Promise.resolve`, they are not equivalent. An async function will return a different reference, whereas `Promise.resolve` returns the same reference if the given value is a promise.

The body of an async function can be thought of as being split by zero or more await expressions. Top-level code, up to and including the first await expression (if there is one), is run synchronously. In this way, an async function without an await expression will run synchronously. If there is an await expression inside the function body, however, the async function will always complete asynchronously.

For example:

```
async function foo() {
  return 1;
}
```

It is similar to:

```
function foo() {
  return Promise.resolve(1).then(() => undefined);
}
```

Code after each `await` expression can be thought of as existing in a `.then` callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.

In the following example, we successively await two promises. Progress moves through function `foo` in three stages.

1. The first line of the body of function `foo` is executed synchronously, with the await expression configured with the pending promise. Progress through `foo` is then suspended and control is yielded back to the function that called `foo`.

2. Some time later, when the first promise has either been fulfilled or rejected, control moves back into `foo`. The result of the first promise fulfillment (if it was not rejected) is returned from the await expression. Here `1` is assigned to `result1`. Progress continues, and the second await expression is evaluated. Again, progress through `foo` is suspended and control is yielded.

3. Some time later, when the second promise has either been fulfilled or rejected, control re-enters `foo`. The result of the second promise resolution is returned from the second await expression. Here `2` is assigned to `result2`. Control moves to the return expression (if any). The default return value of `undefined` is returned as the resolution value of the current promise.

```
async function foo() {
  const result1 = await new Promise((resolve) =>
    setTimeout(() => resolve("1")),
  );
```

```
    const result2 = await new Promise((resolve) =>
      setTimeout(() => resolve("2")),
    );
  }
  foo();
```

Note how the promise chain is not built-up in one go. Instead, the promise chain is constructed in stages as control is successively yielded from and returned to the async function. As a result, we must be mindful of error handling behavior when dealing with concurrent asynchronous operations.

For example, in the following code an unhandled promise rejection error will be thrown, even if a `.catch` handler has been configured further along the promise chain. This is because `p2` will not be "wired into" the promise chain until control returns from `p1`.

```
  async function foo() {
    const p1 = new Promise((resolve) => setTimeout(() => resolve("1"), 1000));
    const p2 = new Promise((_, reject) => setTimeout(() => reject("2"), 500));
    const results = [await p1, await p2]; // Do not do this! Use Promise.all
                                             or Promise.allSettled instead.
  }
  foo().catch(() => {}); // Attempt to swallow all errors...
```

`async function` declarations behave similar to function declarations — they are hoisted to the top of their scope and can be called anywhere in their scope, and they can be redeclared only in certain contexts.

Examples

Fetching Data

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Failed to fetch data');
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('An error occurred:', error);
    throw error;
  }
}
```

```
fetchData()
  .then(data => {
    console.log('Data:', data);
  })
  .catch(error => {
    console.error('Failed to fetch data:', error);
  });
```

In this example, `fetchData` asynchronously fetches data from an API using the `fetch` API and handles potential errors.

Parallel Execution

```
async function parallelTasks() {
  const task1 = taskAsync1();
  const task2 = taskAsync2();

  const result1 = await task1;
  const result2 = await task2;

  console.log('Result 1:', result1);
  console.log('Result 2:', result2);
}

parallelTasks();
```

In this example, two asynchronous tasks `taskAsync1` and `taskAsync2` are executed in parallel, and the results are awaited individually. This demonstrates parallel execution of tasks.

Using async/await with Promises

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Resolved after 2 seconds"), 2000);
  });
}

async function awaitPromise() {
  const result = await resolveAfter2Seconds();
  console.log(result);
}

awaitPromise();
```

This example demonstrates how you can use `await` with an existing Promise, showing that async/await can work seamlessly with Promises.

Error Handling

```
async function performAsyncTask() {
  try {
    const result = await asyncOperation();
    console.log('Result:', result);
  } catch (error) {
    console.error('An error occurred:', error);
  }
}

performAsyncTask();
```

In this example, error handling is demonstrated within an async function using a try-catch block.

Async Function for Sequential File Operations

```
const fs = require('fs').promises;

async function performFileOperations() {
  try {
    const data1 = await fs.readFile('file1.txt', 'utf-8');
    console.log('File 1 data:', data1);

    const data2 = await fs.readFile('file2.txt', 'utf-8');
    console.log('File 2 data:', data2);
  } catch (error) {
    console.error('An error occurred:', error);
  }
}

performFileOperations();
```

In this example, async functions are used for sequential file operations, reading the content of two files one after the other.

These examples provide a broader view of how `async functions` that use `async/await` for handling asynchronous operations can be used in various scenarios, including fetching data, parallel execution, working with Promises, error handling, and sequential file operations.

async

The `async` **keyword** is a fundamental feature in JavaScript that **defines a function as asynchronous**. It plays a central role in modern JavaScript development by simplifying the handling of asynchronous tasks, such as network requests, file operations, or database queries.

When a function is declared as `async`, it signifies that the function will always return a Promise, which makes it easier to work with asynchronous code in a clean and structured manner.

Here's a more detailed exploration of the `async` keyword:

Syntax

To declare a function as asynchronous, you use the `async` keyword before the `function` keyword. An asynchronous function can have parameters and a function body like a regular function.

```
async function fetchData() {
  // Asynchronous code here
} }
```

Return Value

An asynchronous function always returns a Promise. If you explicitly return a value from the function, that value will be implicitly wrapped in a resolved Promise. For example:

```
async function getData() {
  return "Hello, world!";
}
```

Usage

The primary use of `async` functions is to simplify the handling of asynchronous operations. When you mark a function as `async`, it allows you to use the `await` keyword within the function. `await` is used to pause the execution of the function until a Promise is resolved. This sequential execution of code in an `async` function makes it easier to work with asynchronous operations and improves the logical flow of your code.

Error Handling

Asynchronous functions simplify error handling by allowing you to use standard `try...catch` blocks. If an error occurs within an `async` function, you can catch it using regular error-handling techniques. This is a significant improvement over traditional callback-based error handling.

Control Flow

Inside an `async` function, control flow behaves differently when compared to regular functions. When an `await` statement is encountered, the function's execution is paused, and the control is handed back to the event loop. This enables non-blocking execution, ensuring that your application remains responsive.

Top-Level `async`

In addition to being used within functions, `async` can also be used at the top level of a module. This allows modules to wait for their dependencies to execute before they themselves run. This is particularly valuable in modular architectures.

Compatibility

While `async` and `await` are widely supported in modern JavaScript environments, it's important to consider compatibility with older environments or browsers. In such cases, transpilers like Babel can be used to ensure compatibility.

In summary, the `async` keyword is a fundamental building block for writing clean and structured asynchronous code in JavaScript. It simplifies the management of asynchronous tasks, enhances error handling, and offers a more intuitive way to work with asynchronous operations. By using `async` and `await`, you can create code that is easier to read, understand, and maintain, ultimately leading to more efficient and reliable JavaScript applications.

`await`

The `await` keyword is used to pause the execution of an async function until a Promise is resolved. It can only be used inside an `async function` or at the top level of a module.

Syntax

The `await` keyword is used to pause the execution of an async function until a Promise is resolved. It can only be used inside an async function or at the top level of a module. The `await` keyword is followed by an expression, which can be any value, but it is most commonly a Promise. The await expression is used within an async function, and it instructs the function to pause its execution until the Promise is settled, meaning it's either fulfilled (resolved) or rejected. The basic syntax is as follows:

```
await expression
```

Parameters

`expression`

A Promise, a thenable object, or any value to wait for.

Return value

The fulfillment value of the promise or thenable object, or, if the expression is not thenable, the expression's own value.

Exceptions

Throws the rejection reason if the promise or thenable object is rejected.

Return Value

When the awaited Promise is resolved, the `await` expression returns the fulfillment value of that Promise. This allows you to work with the result of the asynchronous operation just as if it were a synchronous value. If the expression is not a Promise or a thenable, the `await` operator simply returns the value of the expression itself.

Usage

`await` is usually used to unwrap promises by passing a `Promise` as the `expression`. Using `await` pauses the execution of its surrounding `async` function until the promise is settled (that is, fulfilled or rejected). When execution resumes, the value of the `await` expression becomes that of the fulfilled promise.

If the promise is rejected, the `await` expression throws the rejected value. The function containing the `await` expression will appear in the stack trace of the error. Otherwise, if the rejected promise is not awaited or is immediately returned, the caller function will not appear in the stack trace.

The `expression` is resolved in the same way as `Promise.resolve()`: it's always converted to a native `Promise` and then awaited. If the `expression` is a:

- Native `Promise` (which means expression belongs to `Promise` or a subclass, and `expression.constructor === Promise`): The promise is directly used and awaited natively, without calling `then()`.
- Thenable object (including non-native promises, polyfill, proxy, child class, etc.): A new promise is constructed with the native `Promise()` constructor by calling the object's `then()` method and passing in a handler that calls the `resolve` callback.
- Non-thenable value: An already-fulfilled `Promise` is constructed and used.

Even when the used promise is already fulfilled, the async function's execution still pauses until the next tick. In the meantime, the caller of the async function resumes execution.

Because await is only valid inside async functions and modules, which themselves are asynchronous and return promises, the await expression never blocks the main thread and only defers execution of code that actually depends on the result, i.e. anything after the await expression.

Error Handling

One of the powerful features of `await` is its ability to handle errors. If the awaited Promise is rejected (meaning the asynchronous operation encountered an error), the `await` expression will throw an error, including the rejection reason. This error handling mechanism is significantly more readable and manageable compared to traditional callback-based error handling.

Control Flow

Inside an `async function`, the `await` keyword has a significant control flow effect. When `await` is encountered, the JavaScript engine will execute the expression while pausing the execution of the surrounding function. Any code that depends on the value of the `await` expression is placed in the microtask queue, allowing the main thread to continue executing other tasks. This asynchronous execution model enhances the responsiveness of your application.

For example:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function asyncFunction() {
  console.log("Start");
```

```
    await delay(2000);
    console.log("After 2 seconds");
    await delay(1000);
    console.log("After another 1 second");
    console.log("End");
  }

  console.log("Program start");
  asyncFunction().then(() => {
    console.log("Program end");
  });
```

When you run this code, you'll see that the "`Start`" message is logged first, followed by "`After 2 seconds`" after a 2-second delay and "`After another 1 second`" after another 1-second delay. Finally, the "`End`" message is logged, showing the control flow of the `async/await` operations.

Top-Level await

In addition to its use within `async functions`, JavaScript now supports the use of `await` at the top level of a module. This means that you can use `await` to ensure that modules wait for their dependencies to execute before they themselves run. This can be particularly valuable in modular application architectures.

Here is an example of a simple module using the Fetch API and specifying await within the export statement. Any modules that include this will wait for the fetch to resolve before running any code.

```
// fetch request
const colors = fetch("../data/colors.json").then((response) => response.json());

export default await colors;
```

In summary, the `await` keyword is a fundamental component of modern JavaScript for handling asynchronous code. Its usage simplifies the management of asynchronous operations, improves error handling, and enhances the readability of code. By making asynchronous code appear more like synchronous code, it provides a more intuitive and structured way to work with asynchronous tasks, ultimately resulting in cleaner and more maintainable code.

Examples

Awaiting a promise to be fulfilled

If a Promise is passed to an await expression, it waits for the Promise to be fulfilled and returns the fulfilled value.

```
function resolveAfter2Seconds(x) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  const x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}

f1();
```

Thenable objects

Thenable objects are resolved just the same as actual Promise objects.

```
async function f() {
  const thenable = {
    then(resolve, _reject) {
      resolve("resolved!");
    },
  };
  console.log(await thenable); // "resolved!"
}

f();
```

They can also be rejected:

```
 async function f() {
   const thenable = {
     then(resolve, reject) {
       reject(new Error("rejected!"));
     },
   };
   await thenable; // Throws Error: rejected!
 }

 f();
```

Conversion to promise

If the value is not a Promise, await converts the value to a resolved Promise, and waits for it. The awaited value's identity doesn't change as long as it doesn't have a then property that's callable.

```
async function f3() {
  const y = await 20;
  console.log(y); // 20

  const obj = {};
  console.log((await obj) === obj); // true
}

f3();
```

Handling rejected promises

If the Promise is rejected, the rejected value is thrown.

```
async function simpleRejectedPromiseHandling() {
  const rejectedPromise = Promise.reject("Oops, something went wrong!");
  try {
    const result = await rejectedPromise;
    console.log("This will not be executed");
  } catch (error) {
    console.error("Error caught:", error);
  }
}

simpleRejectedPromiseHandling();
```

In this example, we create a rejected promise using Promise.reject, and then we use await to handle it. If the promise is rejected, the code in the catch block will be executed, and the error message will be logged. This is a simplified example of how to handle rejected promises with await and a try...catch block.

This lesson was adapted from ChatGPT and MDN Web Docs:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await