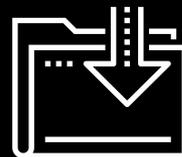


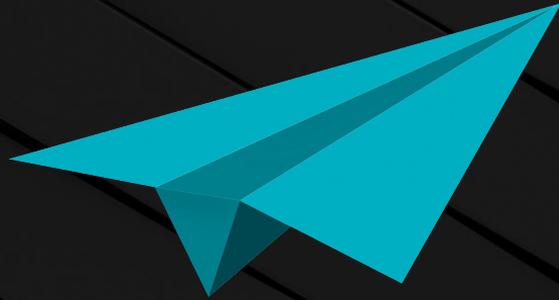


# Advanced ES6

Skills Bootcamp in Front-End Web Development

Lesson 11.3





**Office Hours**

---

**30 Minutes**



**WELCOME**

# Learning Objectives

---

By the end of class, you will be able to:



Identify and implement how and when to use `for...of` loops.



Identify and implement how and when to use the spread and rest operators.

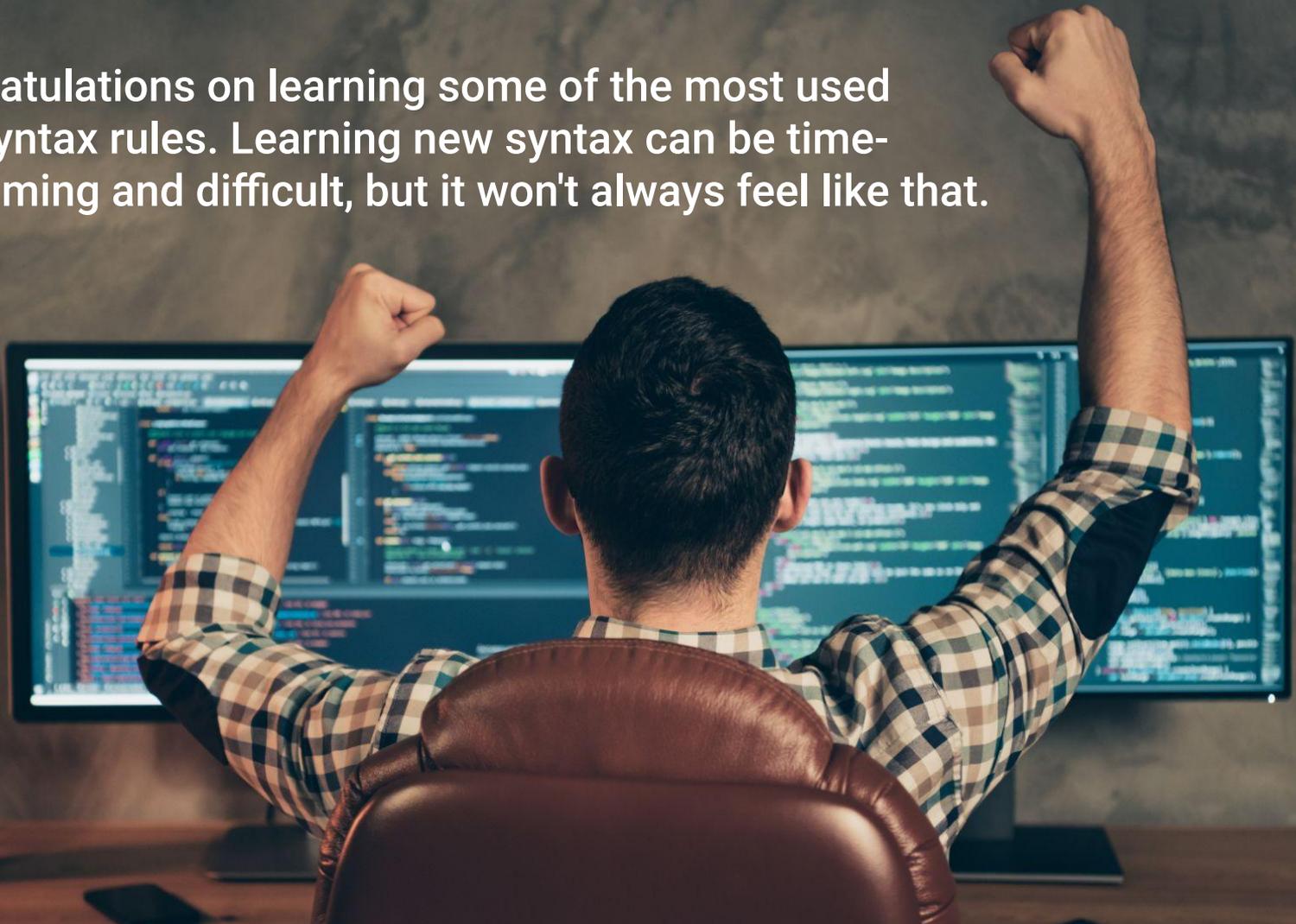


Use destructuring assignment syntax to unpack values from arrays, or properties from objects, into unique variables.



# Stoke Curiosity

Congratulations on learning some of the most used ES6 syntax rules. Learning new syntax can be time-consuming and difficult, but it won't always feel like that.



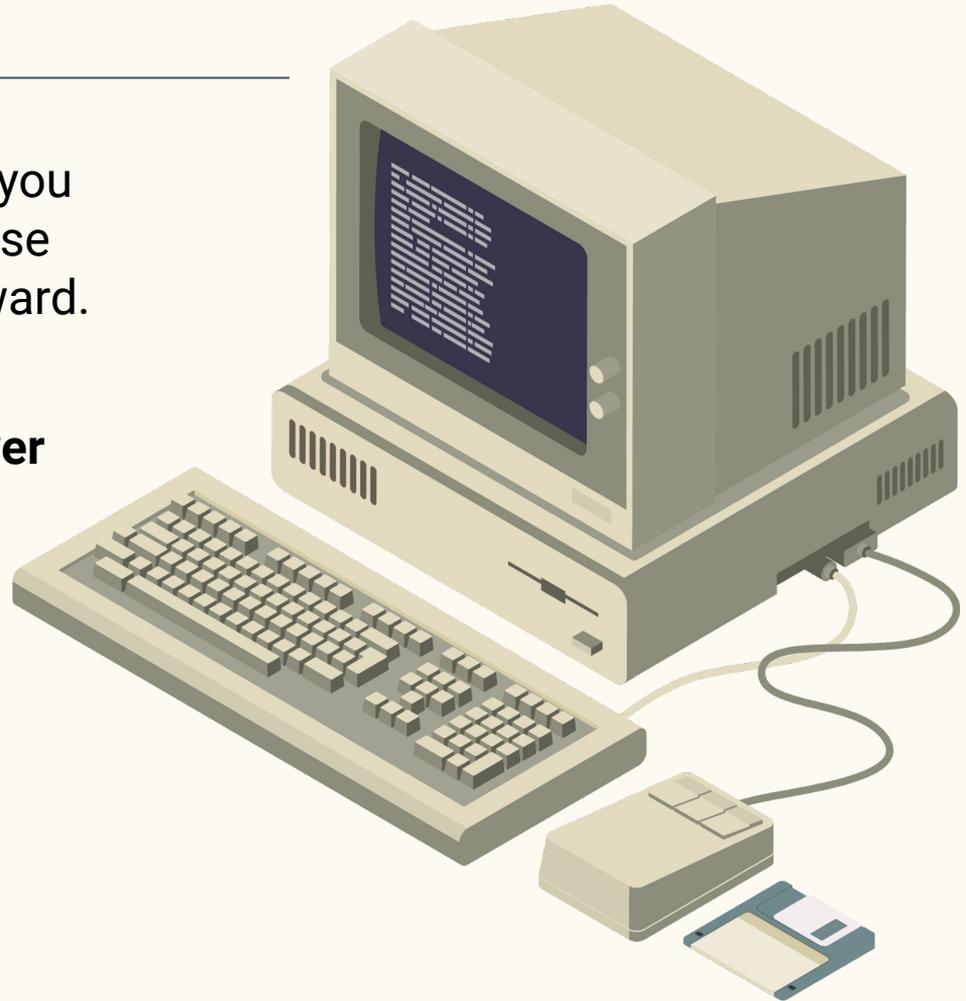
# Stoke Curiosity

---

Using these rules more will allow you to commit them to memory and use less mental bandwidth going forward.

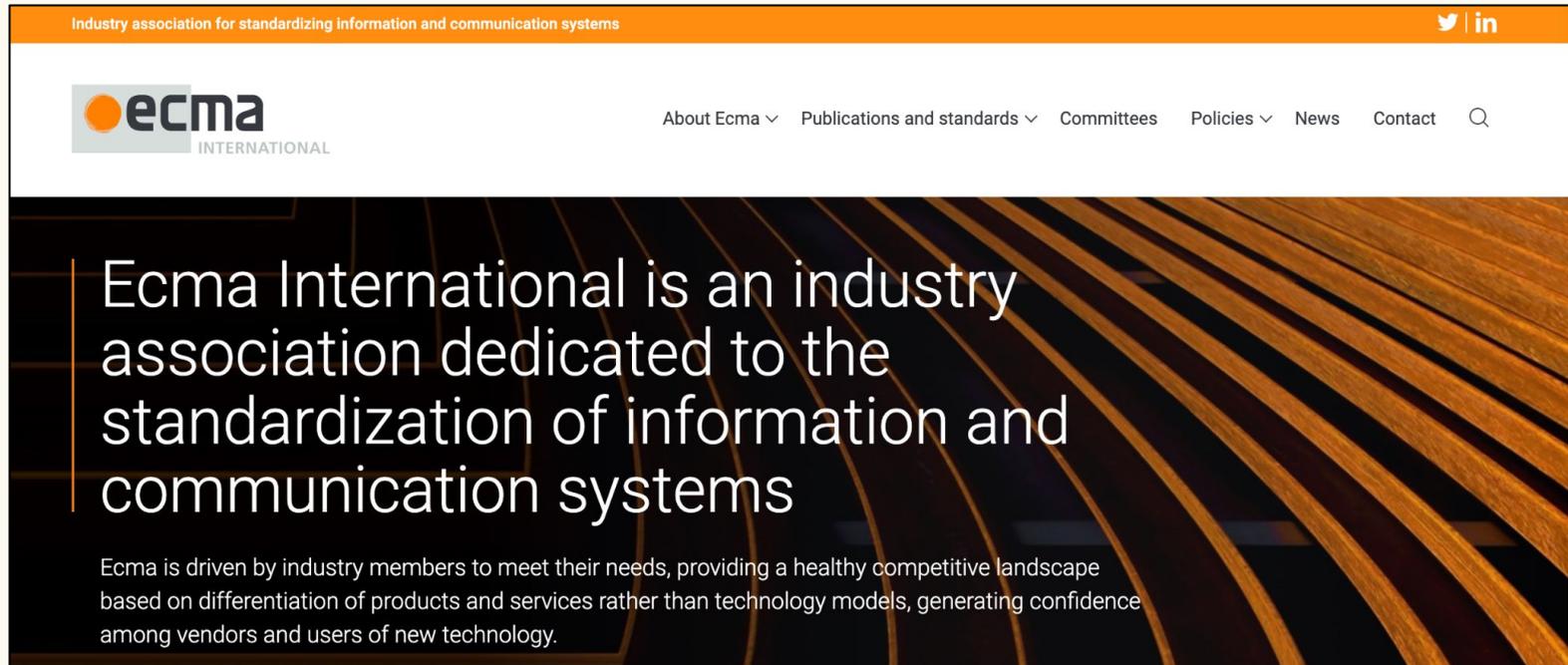
**One of the best ways to learn these new rules is to use the newer syntax whenever possible.**

The old syntax is still perfectly valid and students can always fall back on it.



# Stoke Curiosity

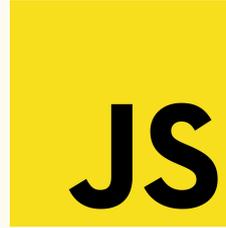
These newer syntax rules were not born in a vacuum. They are the result of years of revision by a standards organization called ECMA. It is important to have standards in web development to ensure maximum compatibility across a wide spectrum of web browsers.



The image shows a screenshot of the ECMA International website. At the top, there is an orange navigation bar with the text "Industry association for standardizing information and communication systems" on the left and social media icons for Twitter and LinkedIn on the right. Below this is a white header area containing the ECMA International logo on the left and a navigation menu with the following items: "About Ecma", "Publications and standards", "Committees", "Policies", "News", and "Contact", each followed by a downward arrow. A search icon is also present. The main content area features a dark background with a pattern of curved, glowing orange lines. The primary text reads: "Ecma International is an industry association dedicated to the standardization of information and communication systems". Below this, a smaller paragraph states: "Ecma is driven by industry members to meet their needs, providing a healthy competitive landscape based on differentiation of products and services rather than technology models, generating confidence among vendors and users of new technology."

# Stoke Curiosity

| Version    | Year           | Official Name   |
|------------|----------------|-----------------|
| <b>ES1</b> | 1997           | ECMAScript 1    |
| <b>ES2</b> | 1998           | ECMAScript 2    |
| <b>ES3</b> | 1999           | ECMAScript 3    |
| <b>ES4</b> | never released | ECMAScript 4    |
| <b>ES5</b> | 2009           | ECMAScript 5    |
| <b>ES6</b> | 2015           | ECMAScript 2015 |
|            | 2016           | ECMAScript 2016 |
|            | 2017           | ECMAScript 2017 |
|            | 2018           | ECMAScript 2018 |



- The ES in ES6 stands for ECMAScript.
- ECMAScript itself is a programming language, but as far as we are concerned, it is just a language from which syntax rules are inherited.
- We can find a more detailed history in the [Wikipedia article](#) on ECMAScript.



# Instructor Demonstration

---

for...of

# Demo: `for...of`

---

Notice when we run the code that we see each value in the songs array, as follows:

```
var songs = ['Bad Guy', 'Old Town Road', '7 Rings'];

for (const value of songs) {
  console.log(value);
}
```

After we comment in the second example, we use a `for...of` loop to iterate over an object or map, as shown in the following code:

```
const songs = new Map([[ 'Bad Guy', 1 ], [ 'Old Town Road', 2 ]]);

for (const [key, value] of songs) {
  console.log(`${key}'s chart position is ${value}`);
}
```

The `for...of` statement creates a loop iterating over objects, including `Array`, `Map`, `Set`, `String`, `TypedArray`.



How does the `for...of` seem to differ from a `forEach`?

The **forEach** method only applies to arrays, while the **for...of** is much more flexible.





# Activity: for...of

Suggested Time:

---

10 Minutes



**Time's Up! Let's Review.**

## Review: `for...of`

---

- The syntax for the `for...of` loop reads very similar to plain English, which helps conceptualize what is happening in the program.
- The syntax is relatively straightforward. The key takeaway is knowing when to use and what to use it for.
- When we open the `index.html` file, we notice that each line item in the unordered list has a green color.
- This is the result of using the `for...of` loop to iterate over each line item and add the class of green to the class list for the given element, as follows:

```
const songs = document.querySelectorAll("ul > li");  
  
for (const song of songs) {  
  song.classList.add("green");  
}
```

## Review: `for...of`

---

If you forget the syntax, VS Code can help you create these types of loops by offering a snippet to work from.

You can try it yourself by typing `forof` and simply pressing Enter, which will result in the following code:

```
for (const iterator of object) {  
  
}
```



How do you know when to use  
a `for...of` loop?



While it generally depends on the situation, `for...of` loops help most when you need to iterate through key-value pairs in an object.

What can we  
do if we don't  
completely  
understand this?



We can refer to supplemental material, read the [MDN Web Docs on for...of](#), and stick around for office hours to ask for help.

The screenshot shows the MDN Web Docs interface. At the top, there is a navigation bar with the MDN logo, links for Technologies, References & Guides, and Feedback, a search box, and a sign-in link. Below this is a breadcrumb trail: Web technology for developers > JavaScript > JavaScript reference > Statements and declarations > for...of. A 'Change language' button is also present. The main content area is titled 'for...of' and includes a description: 'The for...of statement creates a loop iterating over iterable objects, including: built-in String, Array, array-like objects (e.g., arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.' Below the text is a code block titled 'JavaScript Demo: Statement - For...Of' containing the following code: 

```
1 const array1 = ['a', 'b', 'c'];
2
3 for (const element of array1) {
4   console.log(element);
5 }
6
7 // expected output: "a"
8 // expected output: "b"
9 // expected output: "c"
10
```

**MDN Web Docs** moz://a

► Technologies    ► References & Guides    ► Feedback

Search MDN  Sign in

Web technology for developers > JavaScript > JavaScript reference > Statements and declarations > for...of Change language

## Table of contents

- Syntax
- Examples
- Specifications
- Browser compatibility
- See also

## Related Topics

**JavaScript**

**Tutorials:**

- Complete beginners
- JavaScript Guide
- Intermediate

# for...of

The **for...of** statement creates a loop iterating over [iterable objects](#), including: built-in [String](#), [Array](#), array-like objects (e.g., [arguments](#) or [NodeList](#)), [TypedArray](#), [Map](#), [Set](#), and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

### JavaScript Demo: Statement - For...Of

```
1 const array1 = ['a', 'b', 'c'];
2
3 for (const element of array1) {
4   console.log(element);
5 }
6
7 // expected output: "a"
8 // expected output: "b"
9 // expected output: "c"
10
```



# Instructor Demonstration

---

## Rest and Spread Operators

## Demo: Rest and Spread Operators

---

When we run the file, we get output for a few different operations: without the rest parameter, with the rest parameter, without spread operator, and with spread operator, as shown in the following example:

```
function add(x, y) {  
  return x + y;  
}  
  
console.log(add(1, 2, 3, 4, 5)) // => 3
```

It is possible to call a function with any number of arguments, but only the first two will be counted.

# Demo: Rest and Spread Operators

---

Let's examine this function using rest parameters, as follows:

```
function add(...nums) {  
  let sum = 0;  
  for (let num of nums) sum += num;  
  return sum;  
}
```

```
add(1) // => 1
```

```
add(3,3) // => 6
```

```
add(1, 1, 4, 5) // => 11
```

In this example, we use rest parameters `(...nums)` to collect all of the arguments into a `nums` array, enabling us to pass in as many arguments as we want.

# Demo: Rest and Spread Operators

---

Now let's review the following example:

```
function howManyArgs(...args) {  
  return `You passed ${args.length} arguments.`; // point out the template literal  
}  
  
console.log(howManyArgs(0, 1)); // You have passed 2 arguments.  
console.log(howManyArgs("argument!", null, ["one", 2, "three"], 4)); // You have passed 4 arguments.
```

The takeaway here is that variables are now available inside the array of the function. We can also pass as many in as we want.

The spread operator `...` allows iterables like arrays, objects, and strings to be expanded into single arguments or elements.

You can compare this to pouring items out of a cup.

The only difference is that the items are variables and the cup is an iterable.

**variable**



**iterable**



# Demo: Rest and Spread Operators

---

In the following example, we have expanded both arrays into a new array with all of the elements:

```
// Spread Operator

let dragons = ['Drogon', 'Viserion', 'Rhaegal'];
let weapons = ['dragonglass', ...dragons, 'wildfire']; // notice the spread operator ...dragons

console.log(weapons); // prints ["dragonglass", "Drogon", "Viserion", "Rhaegal", "wildfire"]
```



Why does the first example of the `add()` function only output `3`?



**Because only the first and second parameter get counted without the use of the rest operator.**



The syntax for spread and rest are similar, but what is the difference between the two?



The rest parameter allows us to pass in any number of arguments, while the spread operator allows us to spread out an iterable into unique variables.

# Questions?





**Pair Programming Activity:**

---

# **Rest and Spread Operators**

Suggested Time:

---

**15 Minutes**



**Time's Up! Let's Review.**

## Review: Rest and Spread Operators

---

In the first exercise, we are using the spread operator to copy the items in the `songs` array to the `new_songs` array.

Much like we would be dumping out the contents of a cup, we are populating the `new_songs` array with the items in `songs`, as shown in the following code:

```
const songs = ["Creep", "Everlong", "Bulls On Parade", "Song 2", "What I Got"];
const new_songs = [...songs];

console.log(new_songs); // => ["Creep", "Everlong", "Bulls On Parade", "Song 2", "What I Got"];
```

# Review: Rest and Spread Operators

---

In the second exercise, we use the `reduce()` method to execute a reducer function on each element of the array. In our case, the reducer function was adding all the numbers up.

```
const addition = (x, y, z) => {  
  const array = [x, y, z];  
  return array.reduce((a, b) => a + b, 0);  
};
```

Then we modified the `addition()` function to make use of the rest parameters.

The `additionSpread()` function allows us to pass in as many arguments as we need.

This is particularly useful in this case where we want to add as many numbers as necessary.

```
const additionSpread = (...array) => {  
  return array.reduce((a, b) => a + b, 0);  
};
```

# Review: Rest and Spread Operators

---

If we run `node index.js` in our command line, we will see the results of the `console.logs`.

```
console.log(addition(1, 2, 3)); // => 6  
console.log(additionSpread(1, 2, 3)); // => 6  
console.log(additionSpread(1, 2, 3, 4, 100)); // => 110
```



**This is just an introduction.** You will begin to pick up fluency as you get more practice.



What does the `reduce()` method help us with in this exercise?



The `reduce()` method reduces an array to a single value. It takes a callback function and runs that function for each value in the array starting from the left to the right.

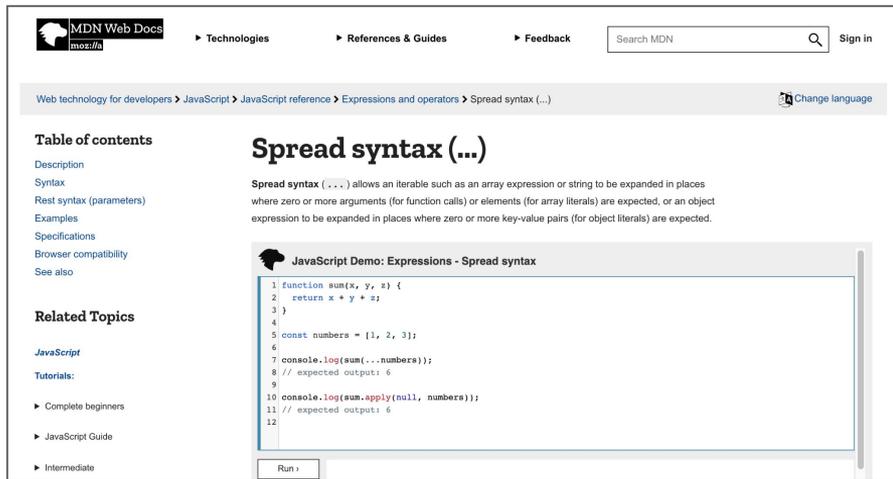


What can we  
do if we don't  
completely  
understand this?

# We can refer to supplemental material and stick around for office hours to ask for help.

Read the [MDN Web Docs on spread](#)

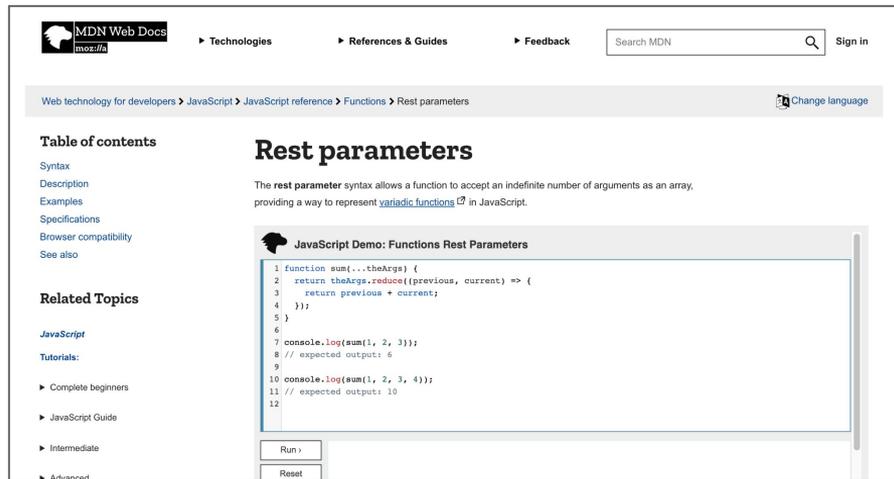
Read the [MDN Web Docs on rest](#)



The screenshot shows the MDN Web Docs page for "Spread syntax (...)". The page title is "Spread syntax (...)" and the breadcrumb trail is "Web technology for developers > JavaScript > JavaScript reference > Expressions and operators > Spread syntax (...)". The page includes a "Table of contents" on the left with sections for Description, Syntax, Rest syntax (parameters), Examples, Specifications, Browser compatibility, and See also. The main content area has a description: "Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected." Below the description is a "JavaScript Demo: Expressions - Spread syntax" with a code editor containing the following code:

```
1 function sum(x, y, z) {
2   return x + y + z;
3 }
4
5 const numbers = [1, 2, 3];
6
7 console.log(sum(...numbers));
8 // expected output: 6
9
10 console.log(sum.apply(null, numbers));
11 // expected output: 6
12
```

Below the code editor is a "Run" button.



The screenshot shows the MDN Web Docs page for "Rest parameters". The page title is "Rest parameters" and the breadcrumb trail is "Web technology for developers > JavaScript > JavaScript reference > Functions > Rest parameters". The page includes a "Table of contents" on the left with sections for Syntax, Description, Examples, Specifications, Browser compatibility, and See also. The main content area has a description: "The rest parameter syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent [variable functions](#) in JavaScript." Below the description is a "JavaScript Demo: Functions Rest Parameters" with a code editor containing the following code:

```
1 function sum(...theArgs) {
2   return theArgs.reduce((previous, current) => {
3     return previous + current;
4   });
5 }
6
7 console.log(sum(1, 2, 3));
8 // expected output: 6
9
10 console.log(sum(1, 2, 3, 4));
11 // expected output: 10
12
```

Below the code editor are "Run" and "Reset" buttons.

# Questions?





# Instructor Demonstration

---

## Object Destructuring

# Demo: Object Destructuring

---

Notice that when we run the `index.js` file we see several variables logged to the terminal. Each of these are different ways of accessing variables inside an object.

Also, in the file, we are using dot notation to access variables inside an object, as we have in the past. We are plucking off certain variables and setting them equal to the value of the object.

This is done with curly braces on the left side of the equals sign, as shown in the following example:

```
const arya = {
  name: 'Arya Stark',
  parents: ['Eddard Stark', 'Catelyn Stark'],
};

const { name, parents } = arya;
```

## Demo: Object Destructuring

---

You can now also use object destructuring as a way to pluck off certain variables from an object.

Consider the following example:

```
const betterLogCharacter = ({ name, parents }) =>
  console.log(`${name}'s parents are: ${parents[0]} and ${parents[1]}.`);

betterLogCharacter(jaime);
```

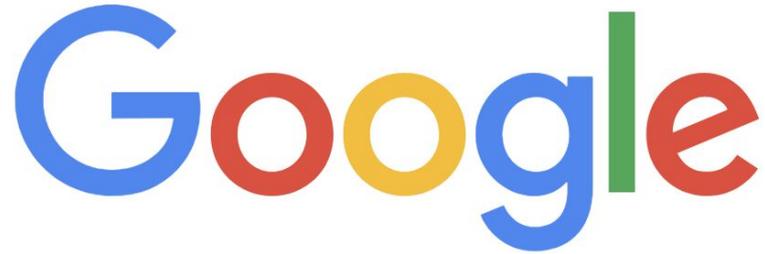
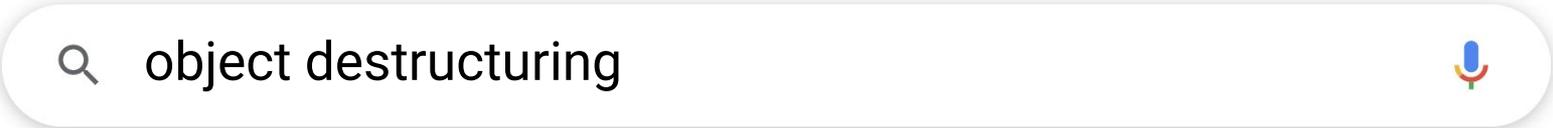


**How would we build this?**

# Review: Object Destructuring

---

We could look up examples of object destructuring and get a feel for the logic before attempting the activity.

The Google logo is displayed in its characteristic multi-colored font: 'G' is blue, the first 'o' is red, the second 'o' is yellow, 'g' is blue, 'l' is green, and 'e' is red.A white search bar with rounded ends is shown. On the left is a magnifying glass icon. The text 'object destructuring' is entered into the bar. On the right side of the bar is a small microphone icon.

# Questions?





# Activity: Object Destructuring

Suggested Time:

---

10 Minutes



**Time's Up! Let's Review.**

# Review: Object Destructuring

---



This exercise highlights how we can pluck off multiple properties at once, saving us a few lines of code.



We can also destructure function parameters. This allows us to name them directly and again save a few steps.



Now we can simply expect an object and pull the properties off without worrying about the order they're passed in or writing extra code to destructure them the old way.

## Review: Object Destructuring

---

In the past, if we wanted to cherry-pick an object's properties, we'd have to do something like the following example:

```
const nodejs = {
  name: 'Node.js',
  type: 'JavaScript runtime environment',
};

const nodejsName = nodejs.name;
const nodejsType = nodejs.type;

console.log(nodejsName); // <= Node.js
console.log(nodejsType); // <= JavaScript runtime environment
```

# Review: Object Destructuring

---

With ES6 object destructuring syntax, we can destructure data based on their property key names:

```
const { name, type } = nodejs;
console.log(name); // <= Node.js
console.log(type); // <= JavaScript runtime environment
```

For a nested object, we need to be more specific:

```
const { framework1, framework2 } = js.tools.frameworks;
console.log(framework1); // <= AngularJS
console.log(framework2); // <= Vue.js
```

For arrays, we can destructure data by the index:

```
const languages = ['HTML', 'CSS', 'JavaScript'];
const [markup, style, scripting] = languages;
console.log(markup, style, scripting); // <= HTML CSS JavaScript
console.log(markup); // <= HTML
```



**Does the order matter when  
passing destructured object  
properties into a function?**



**No! Because we are referring to the properties in the object by name, the key names will align with the correct value every time.**



What can we  
do if we don't  
completely  
understand this?

We can refer to supplemental material, read the [MDN Web Docs on object destructuring](#), and stick around for office hours to ask for help.

The screenshot shows the MDN Web Docs interface. At the top left is the MDN logo with 'moz://a' below it. Navigation links include 'Technologies', 'References & Guides', and 'Feedback'. A search bar contains 'Search MDN' and a 'Sign in' link. A breadcrumb trail reads: 'Web technology for developers > JavaScript > JavaScript reference > Expressions and operators > Destructuring assignment'. A 'Change language' icon is on the right. The main heading is 'Destructuring assignment'. Below it, a paragraph explains that the **destructuring assignment** syntax is used to unpack values from arrays or objects into distinct variables. A code demo titled 'JavaScript Demo: Expressions - Destructuring assignment' shows the following code:

```
1 let a, b, rest;
2 [a, b] = [10, 20];
3
4 console.log(a);
5 // expected output: 10
6
7 console.log(b);
8 // expected output: 20
9
10 [a, b, ...rest] = [10, 20, 30, 40, 50];
11
12 console.log(rest);
13 // expected output: Array [30,40,50]
14
```

On the left side, there is a 'Table of contents' with links for Syntax, Description, Examples, Specifications, Browser compatibility, and See also. Below that is a 'Related Topics' section with a link for 'JavaScript' and a 'Tutorials:' section containing links for 'Complete beginners', 'JavaScript Guide', and 'Intermediate'.

# Questions?



A close-up, high-angle photograph of a computer keyboard. The central focus is a single, large, white key with rounded corners. On this key, there is a dark blue icon of a coffee cup with three wavy lines above it representing steam. Below the icon, the word "Break" is printed in a dark blue, serif font. The key is set against a light-colored, textured keyboard surface. Other keys are visible in the background, including one with a double quote symbol and another with a dash/slash symbol, but they are out of focus.

Break



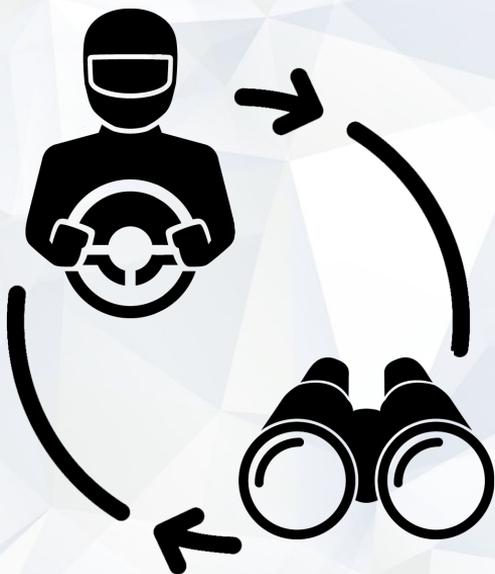
# Instructor Demonstration

---

## Mini-Project

# Questions?





## Pair Programming Activity:

---

# Mini-Project

In this activity, you'll work with a partner to build a command-line tool that generates an HTML portfolio page from user input.

Suggested Time:

---

40 Minutes

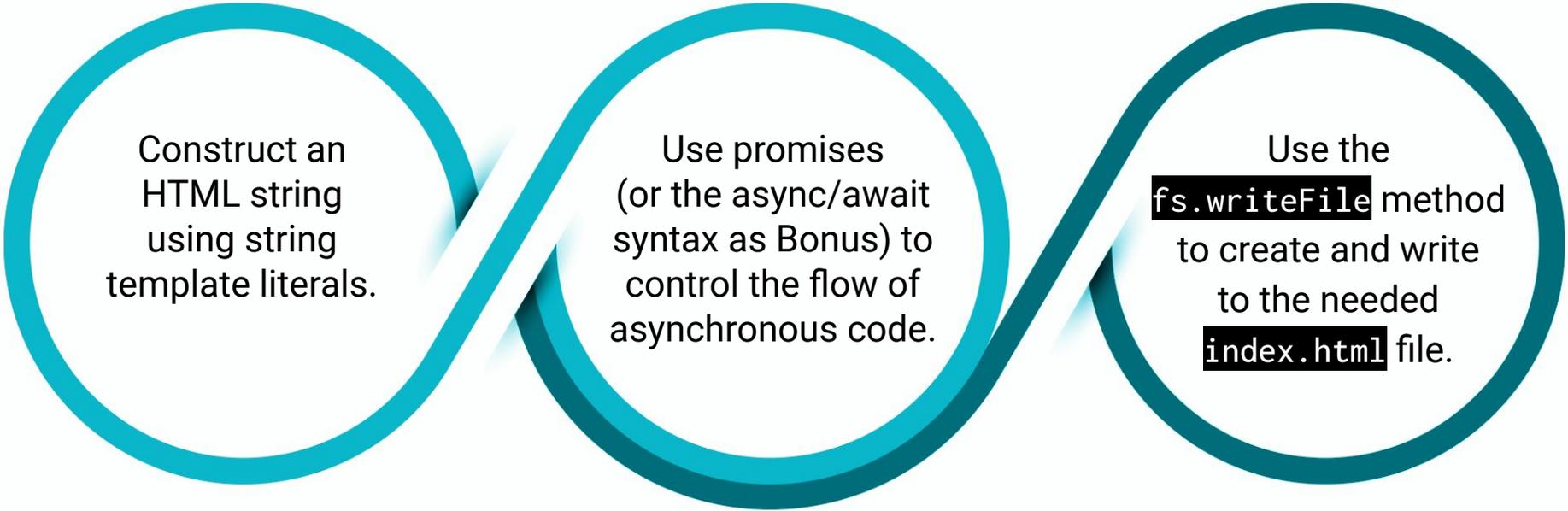


**Time's Up! Let's Review.**

# Review: Mini-Project

---

We can...



Construct an HTML string using string template literals.

Use promises (or the `async/await` syntax as Bonus) to control the flow of asynchronous code.

Use the `fs.writeFile` method to create and write to the needed `index.html` file.

# Review: Mini-Project

---

We import the required packages first, as follows:

```
const inquirer = require("inquirer");  
const fs = require("fs");  
const util = require("util");
```

We use the `util.promisify` method to take a function that uses Node style callbacks to create a new version of the function that now uses Promises.

As shown in the following example, it does exactly what it sounds like:

```
const WriteFileAsync = util.promisify(fs.writeFile);
```

# Review: Mini-Project

`inquirer.prompt({})` will collect the needed responses from the user and assign them to an object for us.

We called the object `answers`, as shown in the following example:

```
function promptUser() {
  return inquirer.prompt([
    {
      type: "input",
      name: "name",
      message: "What is your name?"
    },
    {
      type: "input",
      name: "location",
      message: "Where are you from?"
    },
    {
      type: "input",
      name: "hobby",
      message: "What is your favorite hobby?"
    },
    {
      type: "input",
      name: "food",
      message: "What is your favorite food?"
    },
    {
      type: "input",
      name: "github",
      message: "Enter your GitHub Username"
    },
    {
      type: "input",
      name: "linkedin",
      message: "Enter your LinkedIn URL."
    }
  ]);
}
```



You might be wondering what is the best way to actually make the HTML for this project?

# Review: Mini-Project

---

Here we use a helper function, `generateHTML` that will return a template string. We then inject the responses directly into that template string using the `${}` syntax, like in the following code:

```
function generateHTML(answers) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
  <title>Document</title>
</head>
<body>
  <div class="jumbotron jumbotron-fluid">
    <div class="container">
      <h1 class="display-4">Hi! My name is ${answers.name}</h1>
      <p class="lead">I am from ${answers.location}.</p>
      <h3>Example heading <span class="badge badge-secondary">Contact Me</span></h3>
      <ul class="list-group">
        <li class="list-group-item">My GitHub username is ${answers.github}</li>
        <li class="list-group-item">LinkedIn: ${answers.linkedin}</li>
      </ul>
    </div>
  </div>
</body>
</html>`;
}
```

# Review: Mini-Project

---

Finally, we call the `promptUser` function, and on success we generate the HTML file with these customized responses. We then create the file, appending the contents of the HTML template literal we created, like in the following example:

```
promptUser()
  .then(function(answers) {
    const html = generateHTML(answers);

    return writeFileAsync("index.html", html);
  })
  .then(function() {
    console.log("Successfully wrote to index.html");
  })
  .catch(function(err) {
    console.log(err);
  });
```

# Review: Mini-Project

---

**Let's take a quick look at the Bonus.** Code using the await syntax must be inside of a function declared with the async identifier. We're also using a try/catch block to handle any errors that might occur when using async/await.

You can see this in the following example:

```
async function init() {
  console.log("hi")
  try {
    const answers = await promptUser();

    const html = generateHTML(answers);

    await writeFileAsync("index.html", html);

    console.log("Successfully wrote to index.html");
  } catch(err) {
    console.log(err);
  }
}
```



**How can asynchronous code help  
developers write better code?**

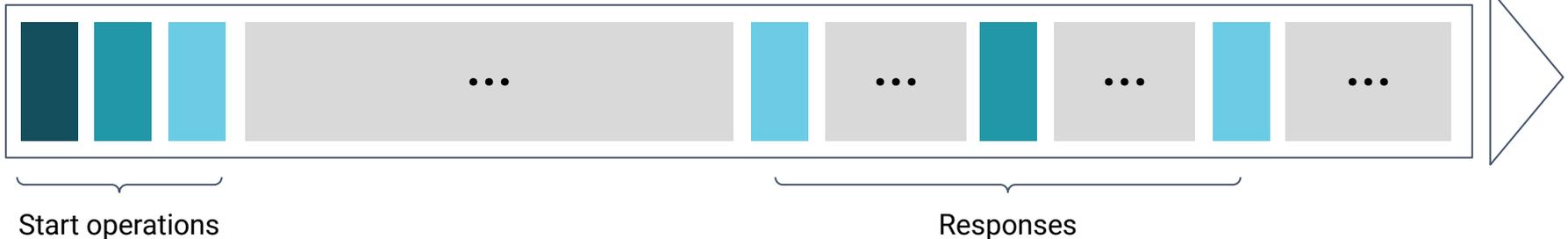
# Review: Mini-Project

Asynchronous programming allows the code to execute logic without blocking the rest of the application's functionality.

## Synchronous



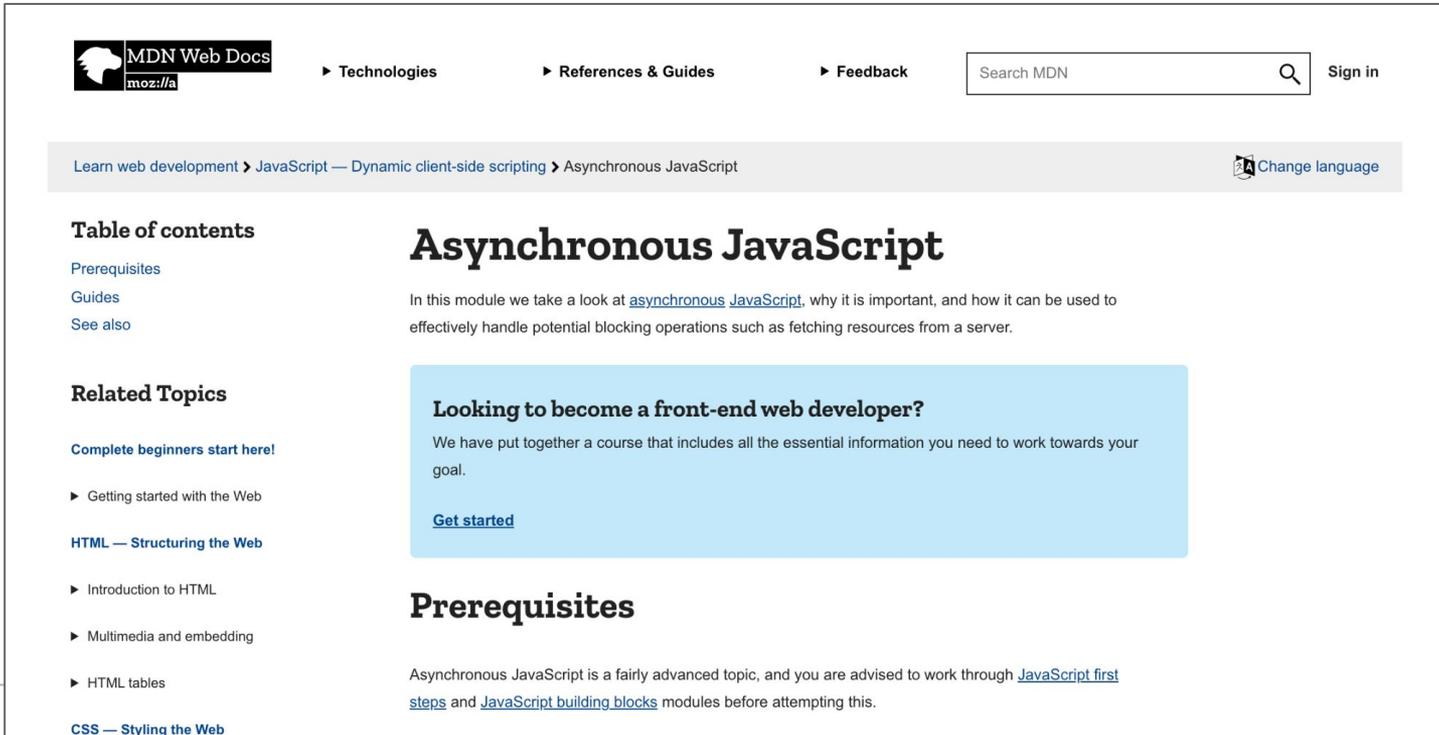
## Asynchronous



A woman with dark curly hair is sitting at a desk, looking at a laptop. She has a thoughtful expression, with her hand resting on her chin. The background is a bright, slightly blurred office or home workspace with a window and a small potted plant.

What can we  
do if we don't  
completely  
understand this?

We can refer to supplemental material, read the [MDN Web Docs on asynchronous JavaScript](#), and stick around for office hours to ask for help.



The screenshot shows the MDN Web Docs page for "Asynchronous JavaScript". The page layout includes a top navigation bar with the MDN logo, "Technologies", "References & Guides", "Feedback", a search box, and a "Sign in" link. Below the navigation bar is a breadcrumb trail: "Learn web development > JavaScript — Dynamic client-side scripting > Asynchronous JavaScript". A "Change language" button is also present. The main content area features a "Table of contents" sidebar with links for "Prerequisites", "Guides", and "See also". The "Related Topics" section includes "Complete beginners start here!" and a list of topics like "Getting started with the Web", "HTML — Structuring the Web", "Introduction to HTML", "Multimedia and embedding", and "HTML tables". The main heading is "Asynchronous JavaScript", followed by an introductory paragraph. A light blue callout box contains the text "Looking to become a front-end web developer?" and a "Get started" link. Below this is the "Prerequisites" section, which includes a paragraph about the advanced nature of the topic and links to "JavaScript first steps" and "JavaScript building blocks".

MDN Web Docs  
moz://a

▶ Technologies ▶ References & Guides ▶ Feedback

Search MDN Sign in

Learn web development > JavaScript — Dynamic client-side scripting > Asynchronous JavaScript

Change language

## Table of contents

- Prerequisites
- Guides
- See also

## Related Topics

Complete beginners start here!

- ▶ Getting started with the Web

HTML — Structuring the Web

- ▶ Introduction to HTML
- ▶ Multimedia and embedding
- ▶ HTML tables

CSS — Styling the Web

# Asynchronous JavaScript

In this module we take a look at [asynchronous JavaScript](#), why it is important, and how it can be used to effectively handle potential blocking operations such as fetching resources from a server.

**Looking to become a front-end web developer?**

We have put together a course that includes all the essential information you need to work towards your goal.

[Get started](#)

## Prerequisites

Asynchronous JavaScript is a fairly advanced topic, and you are advised to work through [JavaScript first steps](#) and [JavaScript building blocks](#) modules before attempting this.

The  
End