What is JavaScript?

- A high-level definition
- So what can it really do?
- What is JavaScript doing on your page?
  - Browser security
  - JavaScript running order
  - Interpreted versus compiled code
  - Server-side versus client-side code
  - Dynamic versus static code
- How do you add JavaScript to your page?
  - Internal JavaScript
  - External JavaScript
  - Inline JavaScript handlers
  - Using addEventListener instead
  - Script loading strategies
- Comments
- Summary

Welcome to the MDN beginner's JavaScript course! In this article we will look at JavaScript from a high level, answering questions such as "What is it?" and "What can you do with it?", and making sure you are comfortable with JavaScript's purpose.

Prerequisites:	An understanding of HTML and the fundamentals of CSS.
Learning objectives:	• What is JavaScript, and how does it fit into a website. • What you can do with JavaScript. • Adding JavaScript to a web page. • Writing comments inside JavaScript.

A high-level definition

JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved.

It is the third layer of the layer cake of standard web technologies, two of which (HTML and CSS) we have covered in much more detail in other parts of the Learning Area.

The three layers of standard web technologies; HTML, CSS and JavaScript

- HTML is the markup language that we use to structure and give meaning to our web content, for example defining paragraphs, headings, and data tables, or embedding images and videos in the page.
- CSS is a language of style rules that we use to apply styling to our HTML content, for example setting background colors and fonts, and laying out our content in multiple columns.
- JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else. (Okay, not everything, but it is amazing what you can achieve with a few lines of JavaScript code.)

The three layers build on top of one another nicely. Let's take a button as an example. We can mark it up using HTML to give it structure and purpose:

```
html {
  height: 100%;
}

body {
  height: inherit;
  display: flex;
  align-items: center;
  justify-content: center;
}

button {
  font-size: 1.4em;
}
```

```
<button>Player 1: Chris</button>
```

Then we can add some CSS into the mix to get it looking nice:

```
button {
  font-family: "Helvetica Neue", "Helvetica", sans-serif;
  letter-spacing: 1px;
  text-transform: uppercase;
  border: 2px solid rgb(200 200 0 / 60%);
  background-color: rgb(0 217 217 / 60%);
  color: rgb(100 0 0 / 100%);
  box-shadow: 1px 1px 2px rgb(0 0 200 / 40%);
  border-radius: 10px;
  padding: 3px 10px;
  cursor: pointer;
}
```

And finally, we can add some JavaScript to implement dynamic behavior:

```
function updateName() {
  const name = prompt("Enter a new name");
  button.textContent = `Player 1: ${name}`;
}

const button = document.querySelector("button");

button.addEventListener("click", updateName);
```

Try clicking on the text label, entering a name into the dialog box that opens, and pressing the OK button.

JavaScript can do a lot more than that — let's explore what in more detail.

> **NOTE:** Before moving on, why not jump in and get your hands dirty with a challenge from Scrimba at this early stage? Check out Render a welcome message *MDN learning partner*. If you don't know how to write this code, don't worry at all; you could try doing some web searches to find some answers, or view the solution at the end of the scrim.

So what can it really do?

The core client-side JavaScript language consists of some common programming features that allow you to do things like:

- Store useful values inside variables. In the above example for instance, we ask for a new name to be entered then store that name in a variable called `name`.
- Operations on pieces of text (known as "strings" in programming). In the above example we take the string "Player 1: " and join it to the `name` variable to create the complete text label, e.g., "Player 1: Chris".
- Running code in response to certain events occurring on a web page. We used a click event in our example above to detect when the button is clicked and then run the code that updates the text label.
- And much more!

What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called **Application Programming Interfaces** (**APIs**) provide you with extra superpowers to use in your JavaScript code.

APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement. They do the same thing for programming that ready-made furniture kits do for home building — it is much easier to take ready-cut panels and screw them together to make a bookshelf than it is to work out the design yourself, go and find the correct wood, cut all the panels to the right size and shape, find the correct-sized screws, and *then* put them together to make a bookshelf.

They generally fall into two categories.

Two categories of API; 3rd party APIs are shown to the side of the browser and browser APIs are in tl

Browser APIs are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example:

- The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed (as we saw above in our simple demo) for example, that's the DOM in action.
- The Geolocation API retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.
- The Canvas and WebGL APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies — see Chrome Experiments and webglsamples.
- Audio and Video APIs like HTMLMediaElement and WebRTC allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer (try our simple Snapshot demo to get the idea).

Third-party APIs are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example:

- The Bluesky API allows you to do things like displaying your latest posts on your website.
- The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

> **NOTE:** These APIs are advanced, and we'll not be covering any of these in this module. You can find out much more about these in our Client-side web APIs module.

There's a lot more available, too! However, don't get over excited just yet. You won't be able to build the next Facebook, Google Maps, or Instagram after studying JavaScript for 24 hours — there are a lot of basics to cover first. And that's why you're here — let's move on!

What is JavaScript doing on your page?

Here we'll actually start looking at some code, and while doing so, explore what actually happens when you run some JavaScript in your page.

Let's briefly recap the story of what happens when you load a web page in a browser (first talked about in our What is CSS? article). When you load a web page in your browser, you are running your code (the HTML, CSS, and JavaScript) inside an execution environment (the browser tab). This is like a factory that takes in raw materials (the code) and outputs a product (the web page).

HTML, CSS and JavaScript code come together to create the content in the browser tab when the pa

A very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (as mentioned above).

Browser security

Each browser tab has its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure — if this were not the case, then pirates

could start writing code to steal information from other websites, and other such bad things.

> **NOTE:** There are ways to send code and data between different websites/tabs in a safe manner, but these are advanced techniques that we won't cover in this course.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

```
function updateName() {
  const name = prompt("Enter a new name");
  button.textContent = `Player 1: ${name}`;
}

const button = document.querySelector("button");

button.addEventListener("click", updateName);
```

Here, we first define a code block called `updateName()` (these types of reusable code blocks are called **functions**), which asks the user for a new name and inserts that name into the text of a button. We then store a reference to a button using `document.querySelector` and attach an event listener to it using `addEventListener` so that when the button is clicked, the `updateName()` function is run.

If you were to swap the order of the `const button = ...` and `button.addEventListener(...)` lines, the code would no longer work — instead, you'd get an error returned in the browser developer console — `Uncaught ReferenceError: Cannot access 'button' before initialization`. This means that the `button` object has not been initialized yet, so we can't add an event listener to it.

> **NOTE:** It is not always true that JavaScript runs exactly in order from top to bottom, due to behaviors like hoisting, but for now, bear in mind that generally items need to be defined before you can use them. This is a common source of errors.

Interpreted versus compiled code

You might hear the terms **interpreted** and **compiled** in the context of programming. In interpreted languages, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it. The code is received in its programmer-friendly text form and processed directly from that.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example, C/C++ are compiled into machine code that is then run by the computer. The program is executed from a binary format, which was generated from the original program source code.

JavaScript is a lightweight interpreted programming language. The web browser receives the JavaScript code in its original text form and runs the script from that. From a technical standpoint, most modern JavaScript interpreters actually use a technique called **just-in-time compiling** to improve performance; the JavaScript source code gets compiled into a faster, binary format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still considered an interpreted language, since the compilation is handled at run time, rather than ahead of time.

There are advantages to both types of language, but we won't discuss them right now.

Server-side versus client-side code

You might also hear the terms **server-side** and **client-side** code, especially in the context of web development. Client-side code is code that is run on the user's computer — when a web page is viewed, the page's client-side code is downloaded, then run and displayed by the browser. In this module we are explicitly talking about **client-side JavaScript**.

Server-side code on the other hand is run on the server, then its results are downloaded and displayed in the browser. Examples of popular server-side web languages include PHP, Python, Ruby, C#, and even JavaScript! JavaScript can also be used as a server-side language, for example in the popular Node.js environment — you can find out more about server-side JavaScript in our Dynamic Websites – Server-side programming topic.

Dynamic versus static code

The word **dynamic** is used to describe both client-side JavaScript, and server-side languages — it refers to the ability to update the display of a web page/app to show different things in different circumstances, generating new content as required. Server-side code dynamically generates new content on the server, e.g., pulling data from a database, whereas client-side JavaScript

dynamically generates new content inside the browser on the client, e.g., creating a new HTML table, filling it with data requested from the server, then displaying the table in a web page shown to the user. The meaning is slightly different in the two contexts, but related, and both approaches (server-side and client-side) usually work together.

A web page with no dynamically updating content is referred to as **static** — it just shows the same content all the time.

How do you add JavaScript to your page?

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses link elements to apply external stylesheets and style elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the script element. Let's learn how this works.

> **NOTE:** Scrimba's Setting up our JavaScript file *MDN learning partner* interactive tutorial walks through a couple of different ways to add JavaScript to your HTML.

Internal JavaScript

1. First of all, create a new HTML file on your local file system and add the following content to it:

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>Apply JavaScript example</title>
  </head>
  <body>
    <button>Click me</button>
  </body>
</html>
```

2. Open the file in your web browser and in your text editor. You'll see that the HTML creates a simple web page containing a clickable button.
3. Next, go to your text editor and add the following at the bottom of your body — just before your closing `</body>` tag:

   ```
   <script>
     // JavaScript goes here
   </script>
   ```

   Note that the code in your web documents is generally loaded and executed in the order it appears on the page. By placing the JavaScript at the bottom, we ensure that all HTML elements are loaded. (See also Script loading strategies below.)

4. Now we'll add some JavaScript inside our script element to make the page do something more interesting — add the following code just below the "// JavaScript goes here" line:

   ```
   function createParagraph() {
     const para = document.createElement("p");
     para.textContent = "You clicked the button!";
     document.body.appendChild(para);
   }

   const buttons = document.querySelectorAll("button");

   for (const button of buttons) {
     button.addEventListener("click", createParagraph);
   }
   ```

5. Save your file and refresh the browser — now you should see that when you click the button, a new paragraph is generated and placed below.

The example should render like so:

```
<button>Click me</button>

<script>
  function createParagraph() {
    const para = document.createElement("p");
    para.textContent = "You clicked the button!";
    document.body.appendChild(para);
  }

  const buttons = document.querySelectorAll("button");
```

```
  for (const button of buttons) {
    button.addEventListener("click", createParagraph);
  }
</script>
```

If your example doesn't seem to work, go through the steps again and check that you did everything right.

- Did you save your local file as a `.html` file?
- Did you add your script element just before the `</body>` tag?
- Did you enter the JavaScript exactly as shown? **JavaScript is case sensitive, and very fussy, so you need to enter the syntax exactly as shown, otherwise it may not work.**

External JavaScript

This works great, but what if we wanted to put our JavaScript in an external file? Let's explore this now.

1. First, create a new file in the same directory as your HTML file. Call it `script.js` — make sure it has that `.js` filename extension, as that's how it is recognized as JavaScript.
2. Remove your script element from the HTML file, then add the following just before the closing `</head>` tag (that way the browser can start loading the file sooner than when it's at the bottom):

   ```
   <script type="module" src="script.js"></script>
   ```

3. Inside `script.js`, add the following JavaScript:

   ```
   function createParagraph() {
     const para = document.createElement("p");
     para.textContent = "You clicked the button!";
     document.body.appendChild(para);
   }

   const buttons = document.querySelectorAll("button");

   for (const button of buttons) {
     button.addEventListener("click", createParagraph);
   }
   ```

4. Save and refresh your browser. You'll discover that clicking the button has no effect, and if you check your browser's console, you'll see an error along the lines of `Cross-origin`

`request blocked`. That's because like many external resources, JavaScript modules need to be loaded from the same origin as the HTML, and `file://` URLs don't qualify. There are two solutions to fix this problem:

- Our recommended solution is to set up a local testing server. With the server program running and serving the `apply-javascript-external.html` and `script.js` files on port `8000`, open your browser and go to `http://localhost:8000`.
- If you cannot run a local server, you can also use `<script defer src="script.js"></script>` instead of `<script type="module" src="script.js"></script>`. See Script loading strategies below for more information. But note that features we use in other parts of the tutorial may require a local HTTP server anyway.

The webpage works just the same as before, but now we've got our JavaScript in an external file:

```
<button>Click me</button>
```

External JavaScript is generally a good thing in terms of organizing your code and making it reusable across multiple HTML files. Plus, the HTML is easier to read without huge chunks of script dumped in it.

Inline JavaScript handlers

Note that sometimes you'll come across bits of actual JavaScript code living inside HTML. It might look something like this:

```
function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
  document.body.appendChild(para);
}
```

```
<button onclick="createParagraph()">Click me!</button>
```

This code has exactly the same functionality as the versions shown in the previous two sections, except that the button element includes an inline `onclick` handler to make the function run when the button is pressed.

Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the `onclick="createParagraph()"` attribute on every button you want the JavaScript to apply to.

Using addEventListener instead

Instead of including JavaScript in your HTML, use a pure JavaScript construct. The `querySelectorAll()` function allows you to select all the buttons on a page. You can then loop through the buttons, assigning a handler for each using `addEventListener()`. The code for this is shown below:

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

This might be a bit longer than the `onclick` attribute, but it will work for all buttons — no matter how many are on the page, nor how many are added or removed. The JavaScript does not need to be changed.

> **NOTE:** Try editing your version of `apply-javascript.html` and add a few more buttons into the file. When you reload, you should find that all of the buttons when clicked will create a paragraph. Neat, huh?

Script loading strategies

All the HTML on a page is loaded in the order in which it appears. If you are using JavaScript to manipulate elements on the page (or more accurately, the Document Object Model), your code won't work if the JavaScript is loaded and parsed before the HTML you are trying to do something to.

There are a few different strategies to make sure your JavaScript only runs after the HTML is parsed:

- In the internal JavaScript example above, the script element is placed at the bottom of the body of the document, and therefore only run after the rest of the HTML body is parsed.
- In the external JavaScript example above, the script element is placed in the head of the document, before the HTML body is parsed. But because we're using `<script type="module">`, the code is treated as a module and the browser waits for all HTML to be processed before executing JavaScript modules. (You could also place external scripts at the bottom of the body. But if there is a lot of HTML and the network is slow, it may take a lot of time before the browser can even start fetching and loading the script, so placing external scripts in the head is usually better.)
- If you still want to use non-module scripts in the document head, which could block the whole page from displaying, and could cause errors because it executes before the HTML

is parsed:

- For external scripts, you should add the `defer` (or if you don't need the HTML to be ready, the `async`) attribute on the script element.
- For internal scripts, you should wrap the code in a `DOMContentLoaded` event listener.

This is beyond the scope of the tutorial at this point, but unless you need to support very old browsers, you don't have to do this and can just use `<script type="module">` instead.

Comments

As with HTML and CSS, it is possible to write comments into your JavaScript code that will be ignored by the browser, and exist to provide instructions to your fellow developers on how the code works (and you, if you come back to your code after six months and can't remember what you did). Comments are very useful, and you should use them often, particularly for larger applications. There are two types:

- A single line comment is written after a double forward slash (`//`), e.g.

```
// I am a comment
```

- A multi-line comment is written between the strings `/*` and `*/`, e.g.

```
/*
  I am also
  a comment
*/
```

So for example, we could annotate our last demo's JavaScript with comments like so:

```
// Function: creates a new paragraph and appends it to the bottom of the HTML body.

function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
  document.body.appendChild(para);
}

/*
  1. Get references to all the buttons on the page in an array format.
  2. Loop through all the buttons and add a click event listener to each one.
```

```
  When any button is pressed, the createParagraph() function will be run.
*/

const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

> **NOTE:** In general more comments are usually better than less, but you should be careful if you find yourself adding lots of comments to explain what variables are (your variable names perhaps should be more intuitive), or to explain very simple operations (maybe your code is overcomplicated).

Summary

So there you go, your first step into the world of JavaScript. We've begun with just theory, to start getting you used to why you'd use JavaScript and what kind of things you can do with it. Along the way, you saw a few code examples and learned how JavaScript fits in with the rest of the code on your website, among other things.

JavaScript may seem a bit daunting right now, but don't worry — in this course, we will take you through it in simple steps that will make sense going forward. In the next article, we will plunge straight into the practical, getting you to jump straight in and build your own JavaScript examples.

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:32:46 UTC