JAVA SCRIPT

# WEEK1

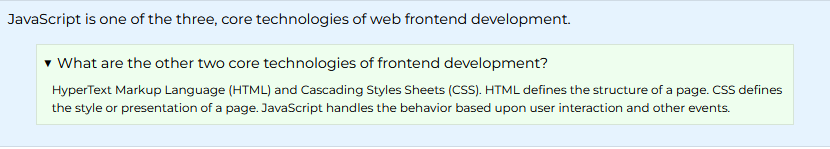
## JavaScript Introduction and Overview

### Overview

JavaScript is a popular, well-supported programming language that is used to create interactive and dynamic web pages and applications. It is used to create web, mobile, and desktop applications. It is a high-level, interpreted language that is designed to run in web browsers. JavaScript can also be used on the server-side with the help of Node.js.

A close-up of a message

Description automatically generated



### Prepare

**What is 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](https://developer.mozilla.org/en-US/docs/Learn/HTML) and [CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS)) we have covered in much more detail in other parts of the Learning Area.

A blue book with text on it

Description automatically generated

[HTML](https://developer.mozilla.org/en-US/docs/Glossary/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](https://developer.mozilla.org/en-US/docs/Glossary/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](https://developer.mozilla.org/en-US/docs/Glossary/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:

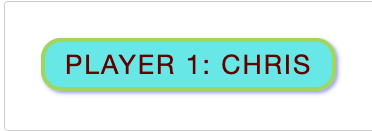
A screenshot of a computer program

Description automatically generated

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

A screenshot of a computer program

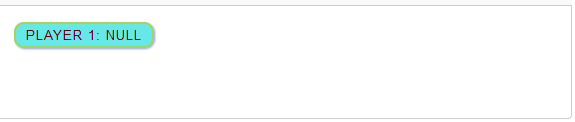
Description automatically generated



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

A screenshot of a computer program

Description automatically generated



Try clicking on this last version of the text label to see what happens (note also that you can find this demo on GitHub — see the [source code](https://github.com/mdn/learning-area/blob/main/javascript/introduction-to-js-1/what-is-js/javascript-label.html), or [run it live](https://mdn.github.io/learning-area/javascript/introduction-to-js-1/what-is-js/javascript-label.html))!

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

**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](https://developer.mozilla.org/en-US/docs/Web/API/Element/click_event) event in our example above to detect when the label 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.

A screenshot of a browser window

Description automatically generated

**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](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) 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](https://developer.mozilla.org/en-US/docs/Web/API/Geolocation) retrieves geographical information. This is how [Google Maps](https://www.google.com/maps) is able to find your location and plot it on a map.
* The [Canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API) and [WebGL](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API) APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies — see [Chrome Experiments](https://experiments.withgoogle.com/collection/chrome) and [webglsamples](https://webglsamples.org/).
* [Audio and Video APIs](https://developer.mozilla.org/en-US/docs/Web/Media/Audio_and_video_delivery) like [HTMLMediaElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement) and [WebRTC](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API) 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](https://chrisdavidmills.github.io/snapshot/) to get the idea).

A screenshot of a computer

Description automatically generated

**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 [Twitter API](https://developer.twitter.com/en/docs) allows you to do things like displaying your latest tweets on your website.
* The [Google Maps API](https://developers.google.com/maps/) and [OpenStreetMap API](https://wiki.openstreetmap.org/wiki/API) allows you to embed custom maps into your website, and other such functionality.

A close up of a message

Description automatically generated

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?**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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 [How CSS works](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#how_does_css_actually_work) 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).

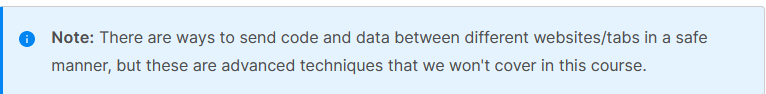
A blue and white computer screen

Description automatically generated

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). Note that the code in your web documents is generally loaded and executed in the order it appears on the page. Errors may occur if JavaScript is loaded and run before the HTML and CSS that it is intended to modify. You will learn ways around this later in the article, in the [Script loading strategies](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#script_loading_strategies) section.

[**Browser security**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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.

****

[**JavaScript running order**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#javascript_running_order)

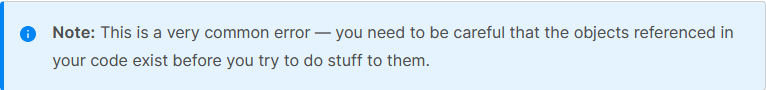
When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

**A screenshot of a computer code

Description automatically generated**

Here we are selecting a button (line 1), then attaching an event listener to it (line 3) so that when the button is clicked, the updateName() code block (lines 5–8) is run. The updateName() code block (these types of reusable code blocks are called "functions") asks the user for a new name, and then inserts that name into the button text to update the display.

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the [browser developer console](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Tools_and_setup/What_are_browser_developer_tools) — 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.



**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**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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, ASP.NET, 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](https://developer.mozilla.org/en-US/docs/Learn/Server-side) topic.

[**Dynamic versus static code**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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?**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) elements to apply external stylesheets and [<style>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style) elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) element. Let's learn how this works.

[**Internal JavaScript**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#internal_javascript)

1. First of all, make a local copy of our example file [apply-javascript.html](https://github.com/mdn/learning-area/blob/main/javascript/introduction-to-js-1/what-is-js/apply-javascript.html). Save it in a directory somewhere sensible.
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 in your head — just before your closing </head> tag:

**A white rectangular object with a white border

Description automatically generated**

1. Now we'll add some JavaScript inside our [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) element to make the page do something more interesting — add the following code just below the "// JavaScript goes here" line:

A screenshot of a computer program

Description automatically generated

1. 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.

A screenshot of a computer

Description automatically generated

**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 sample HTML file. Call it script.js — make sure it has that .js filename extension, as that's how it is recognized as JavaScript.
2. Replace your current [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) element with the following:

**A white rectangular object with a white border

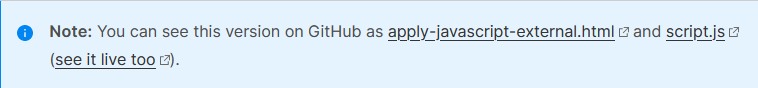
Description automatically generated**

1. Inside script.js, add the following script:

A screenshot of a computer program

Description automatically generated

1. Save and refresh your browser, and you should see the same thing! It works just the same, but now we've got our JavaScript in an external file. This 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.

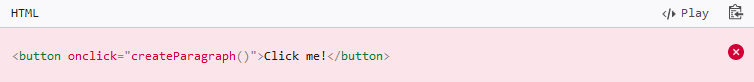


**Inline JavaScript handlers**

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

**A screen shot of a computer

Description automatically generated**

****

You can try this version of our demo below.

**A white background with black text

Description automatically generated**

This demo has exactly the same functionality as in the previous two sections, except that the [<button>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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:

**A screenshot of a computer program

Description automatically generated**

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.

**A screen shot of a computer screen

Description automatically generated**

**Script loading strategies**

There are a number of issues involved with getting scripts to load at the right time. Nothing is as simple as it seems! A common problem is that 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](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents#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.

In the above code examples, in the internal and external examples the JavaScript is loaded and run in the head of the document, before the HTML body is parsed. This could cause an error, so we've used some constructs to get around it.

In the internal example, you can see this structure around the code:

A screen shot of a computer

Description automatically generated

This is an event listener, which listens for the browser's DOMContentLoaded event, which signifies that the HTML body is completely loaded and parsed. The JavaScript inside this block will not run until after that event is fired, therefore the error is avoided (you'll [learn about events](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events) later in the course).

In the external example, we use a more modern JavaScript feature to solve the problem, the defer attribute, which tells the browser to continue downloading the HTML content once the <script> tag element has been reached.

A white rectangular object with a white border

Description automatically generated

In this case both the script and the HTML will load simultaneously and the code will work.

A close up of a message

Description automatically generated

An old-fashioned solution to this problem used to be to put your script element right at the bottom of the body (e.g. just before the </body> tag), so that it would load after all the HTML has been parsed. The problem with this solution is that loading/parsing of the script is completely blocked until the HTML DOM has been loaded. On larger sites with lots of JavaScript, this can cause a major performance issue, slowing down your site.

**async and defer**

There are actually two modern features we can use to bypass the problem of the blocking script — async and defer (which we saw above). Let's look at the difference between these two.

Scripts loaded using the async attribute will download the script without blocking the page while the script is being fetched. However, once the download is complete, the script will execute, which blocks the page from rendering. This means that the rest of the content on the web page is prevented from being processed and displayed to the user until the script finishes executing. You get no guarantee that scripts will run in any specific order. It is best to use async when the scripts in the page run independently from each other and depend on no other script on the page.

Scripts loaded with the defer attribute will load in the order they appear on the page. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

Here is a visual representation of the different script loading methods and what that means for your page:

A close-up of a screen

Description automatically generated

This image is from the [HTML spec](https://html.spec.whatwg.org/images/asyncdefer.svg), copied and cropped to a reduced version, under [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) license terms.

For example, if you have the following script elements:

**A screenshot of a computer program

Description automatically generated**

You can't rely on the order the scripts will load in. jquery.js may load before or after script2.js and script3.js and if this is the case, any functions in those scripts depending on jquery will produce an error because jquery will not be defined at the time the script runs.

async should be used when you have a bunch of background scripts to load in, and you just want to get them in place as soon as possible. For example, maybe you have some game data files to load, which will be needed when the game actually begins, but for now you just want to get on with showing the game intro, titles, and lobby, without them being blocked by script loading.

Scripts loaded using the defer attribute (see below) will run in the order they appear in the page and execute them as soon as the script and content are downloaded:

A screenshot of a computer program

Description automatically generated

In the second example, we can be sure that jquery.js will load before script2.js and script3.js and that script2.js will load before script3.js. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

To summarize:

* async and defer both instruct the browser to download the script(s) in a separate thread, while the rest of the page (the DOM, etc.) is downloading, so the page loading is not blocked during the fetch process.
* scripts with an async attribute will execute as soon as the download is complete. This blocks the page and does not guarantee any specific execution order.
* scripts with a defer attribute will load in the order they are in and will only execute once everything has finished loading.
* If your scripts should be run immediately and they don't have any dependencies, then use async.
* If your scripts need to wait for parsing and depend on other scripts and/or the DOM being in place, load them using defer and put their corresponding <script> elements in the order you want the browser to execute them.

**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 screenshot of a computer

Description automatically generated**

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

A screenshot of a computer program

Description automatically generated

A close-up of a text

Description automatically generated

[**Summary**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript#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, amongst 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](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash), getting you to jump straight in and build your own JavaScript examples.

How CSS Works

We have learned the basics of CSS, what it is for and how to write simple stylesheets. In this lesson we will take a look at how a browser takes CSS and HTML and turns that into a webpage.

[**How does CSS actually work?**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#how_does_css_actually_work)

When a browser displays a document, it must combine the document's content with its style information. It processes the document in a number of stages, which we've listed below. Bear in mind that this is a very simplified version of what happens when a browser loads a webpage, and that different browsers will handle the process in different ways. But this is roughly what happens.

1. The browser loads the HTML (e.g. receives it from the network).
2. It converts the [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) into a [DOM](https://developer.mozilla.org/en-US/docs/Glossary/DOM) (*Document Object Model*). The DOM represents the document in the computer's memory. The DOM is explained in a bit more detail in the next section.
3. The browser then fetches most of the resources that are linked to by the HTML document, such as embedded images, videos, and even linked CSS! JavaScript is handled a bit later on in the process, and we won't talk about it here to keep things simpler.
4. The browser parses the fetched CSS, and sorts the different rules by their selector types into different "buckets", e.g. element, class, ID, and so on. Based on the selectors it finds, it works out which rules should be applied to which nodes in the DOM, and attaches style to them as required (this intermediate step is called a render tree).
5. The render tree is laid out in the structure it should appear in after the rules have been applied to it.
6. The visual display of the page is shown on the screen (this stage is called painting).

The following diagram also offers a simple view of the process.

**A diagram of a software process

Description automatically generated**

[**About the DOM**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#about_the_dom)

A DOM has a tree-like structure. Each element, attribute, and piece of text in the markup language becomes a [DOM node](https://developer.mozilla.org/en-US/docs/Glossary/Node/DOM) in the tree structure. The nodes are defined by their relationship to other DOM nodes. Some elements are parents of child nodes, and child nodes have siblings.

Understanding the DOM helps you design, debug and maintain your CSS because the DOM is where your CSS and the document's content meet up. When you start working with browser DevTools you will be navigating the DOM as you select items in order to see which rules apply.

[**A real DOM representation**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#a_real_dom_representation)

Rather than a long, boring explanation, let's look at an example to see how a real HTML snippet is converted into a DOM.

Take the following HTML code:

A white rectangular object with a white border

Description automatically generated

In the DOM, the node corresponding to our <p> element is a parent. Its children are a text node and the three nodes corresponding to our <span> elements. The SPAN nodes are also parents, with text nodes as their children:

**A white background with a black border

Description automatically generated with medium confidence**

This is how a browser interprets the previous HTML snippet — it renders the above DOM tree and then outputs it in the browser like so:

A white rectangular object with a black border

Description automatically generated

[**Applying CSS to the DOM**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#applying_css_to_the_dom)

Let's say we add some CSS to our document, to style it. Again, the HTML is as follows:

A white rectangular object with a white border

Description automatically generated

Let's suppose we apply the following CSS to it:

A white rectangular object with a white border

Description automatically generated

The browser parses the HTML and creates a DOM from it. Next, it parses the CSS. Since the only rule available in the CSS has a span selector, the browser sorts the CSS very quickly! It applies that rule to each one of the three <span>s, then paints the final visual representation to the screen.

The updated output is as follows:

A white rectangular object with a black border

Description automatically generated

In our [Debugging CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Debugging_CSS) article in the next module we will be using browser DevTools to debug CSS problems, and will learn more about how the browser interprets CSS.

[**What happens if a browser encounters CSS it doesn't understand?**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#what_happens_if_a_browser_encounters_css_it_doesnt_understand)

The ["Browser support information" section in the "What is CSS" article](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#browser_support_information) mentioned that browsers do not necessarily implement new CSS features at the same time. In addition, many people are not using the latest version of a browser. Given that CSS is being developed all the time, and is therefore ahead of what browsers can recognize, you might wonder what happens if a browser encounters a CSS selector or declaration it doesn't recognize.

The answer is that it does nothing, and just moves on to the next bit of CSS!

If a browser is parsing your rules, and encounters a property or value that it doesn't understand, it ignores it and moves on to the next declaration. It will do this if you have made an error and misspelled a property or value, or if the property or value is just too new and the browser doesn't yet support it.

Similarly, if a browser encounters a selector that it doesn't understand, it will just ignore the whole rule and move on to the next one.

In the example below I have used the British English spelling for color, which makes that property invalid as it is not recognized. So my paragraph has not been colored blue. All of the other CSS have been applied however; only the invalid line is ignored.

A computer code on a white background

Description automatically generated

A screenshot of a computer

Description automatically generated

A close up of a text

Description automatically generated

This behavior is very useful. It means that you can use new CSS as an enhancement, knowing that no error will occur if it is not understood — the browser will either get the new feature or not. This enables basic fallback styling.

This works particularly well when you want to use a value that is quite new and not supported everywhere. For example, some older browsers do not support calc() as a value. I might give a fallback width for a box in pixels, then go on to give a width with a calc() value of 100% - 50px. Old browsers will use the pixel version, ignoring the line about calc() as they don't understand it. New browsers will interpret the line using pixels, but then override it with the line using calc() as that line appears later in the cascade.

A white rectangular object with a white border

Description automatically generated

### Common Misconceptions and Language Issues

A screenshot of a computer program

Description automatically generated