CSS Variables — No, really!

Dave Gash Follow
Apr 19, 2017 · 20 min read

Introduction

The ability to use variables in CSS is a useful and powerful feature that web developers have long been asking for. Well, it has finally arrived, and it's awesome! In this article we'll look at the history, importance, and use of CSS variables, and how you can leverage them to make your CSS development and maintenance faster and easier.

The history of CSS variables is a bit spotty. Originally spec'd out by the W3C in 2012, the feature was initially implemented by only Chrome and Firefox. When the specification was updated in 2014 with a significant syntax improvement, Firefox kept up and modified its implementation, while Chrome decided to back-burner implementation efforts until things settled down. For its part, Microsoft sort of whistled and looked the other way.

As late as September of 2015, caniuse.com reported browser support for CSS variables at less than 9%, as seen in the upper-right corner of this report.

Eek, that's a lot of red

However, in 2016, Chrome, Safari, Opera, and Android browser all jumped on the variables train, and suddenly support soared to 69%! (That's global support; US-only support is 77%.)

Ahh, much better

That's more like it. And as you can see from the more recent report, even Microsoft Edge is working on it. (I know, right?!?)

So CSS variables are now officially A Thing You Can Actually Use! Let's see how.

Variables Refresher

Variables, regardless of the language they're used in, are just named boxes that hold a value for later use. In the expression x = 3, x is the variable *name* and 3 is the variable *value*. The value can change, or vary (it's a "variable", duh), but the current value is

always available by simply referring to the name. Thus, if x = 3, then the expression x + 2 initially equals 5; but if the value of x changes to 4 at some point, then that same expression, x + 2, now equals 6. The name x always returns the variable's current value, no matter where or how often it is used.

But CSS is a declarative, pattern-matching language, not a procedural, step-wise language, so why use variables in CSS? For the same reasons we use them in math and programming: simplicity, consistency, and ease of maintenance. Think of the acronym DRY — Don't Repeat Yourself. By using variables in CSS, you can localize values and simplify initial development, iterative testing, and later maintenance all in one go. The value of variables (see what I did there?) is this: set it once and it's set everywhere; *change it once and it's changed everywhere*.

But Are They Variables or Properties?

Yes.

CSS variables are properly called "custom properties", and that's a fair description. You could almost call them "invalid properties", because the gist of a CSS variable is that it's a property whose name doesn't exist, one that has no meaning to a browser.

That is, while the property name `background-color` is known to browsers and has a specific meaning, the property name `--hey-this-is-my-cool-bg-color` is not known and has no meaning to browsers. It is, in fact, an invalid property name… unless it's a variable.

Custom properties have ordinary values just like regular properties but, unlike regular properties, their values can be reused later by referring to them by name. That ability alone makes them act exactly like variables. Also, as you'll see in a moment, they are even referenced using the keyword `var()`. Bottom line: although they are technically "custom properties", everyone just calls them CSS variables — even the W3C.

The Problem

Even a relatively straightforward chunk of CSS can quickly get messy and hard to maintain. Many property/value pairs will ultimately be duplicated, either accidentally or of necessity. For example, for every rule in which you want to set some element's left

margin to 20px, you would code `margin-left: 20px;`. If you later decide that a left margin of 15px is better, you must change the value *in each rule*. That's a maintenance problem.

Consider this small block of CSS that uses the standard color name `darkcyan`.

```
body { margin: 50px; padding: 20px; border: 5px solid darkcyan;
  font-family: tahoma, verdana; text-align: center; }
.legend { color: darkcyan; font-weight: bold; }
a { outline: none; color: Darkcyan; text-decoration: none; }
fieldset { border: 1px solid darkcyan; height: 300px; }
```

If I want to change that repeated darkcyan color to, say, orange, I have to carefully look through the CSS and manually change it three — no, wait — four times. And, realistically, what if those rules aren't conveniently adjacent in my real CSS as they are in the example above, but instead are several dozen, or several hundred, lines apart? Now the problem is compounded, and maintenance becomes a nightmare.

I know what you're thinking: just do a global search and replace! And sure, that works fine if you're 100% consistent with your coding. But while CSS property names are case-sensitive (more about that later), property values aren't. So both `darkcyan` and `Darkcyan` will work in your CSS, but both won't be found in a single search and replace operation. (In fact, with that in mind, have another look at the CSS block above. Really, go ahead.) Or what about color codes like `#ffffff`, `#FFFFFF`, `#FFF`, `#fff`, `rgb(255, 255, 255)`, or even `white`? Yes, they're all "white", they're all valid, but they're all *different* for search and replace purposes. Uh-oh, now what?

CSS variables, that's what.

Syntax

Declaration

Recall that CSS variables are just property/value pairs; you define them exactly as you would properties of any valid selector, except with property names you make up yourself. The only catch is that the browser has to know that your "invalid" property

name is deliberate and not a typo like “background-colour”. (Wow, if I had a nickel for every time I’ve seen that one!)

That differentiation is done by simply starting your variable names with two dashes, e.g., `--my-background-color`, `--standard-shadow`, `--bqindents`. That’s all there is to it; property names that begin with two dashes are automatically identified as variables instead of invalid names, and their property values become their variable values.

Retrieval

That’s half the battle; now we just have to retrieve the values by using the variable names as property values in later rules. This is done with the `var()` keyword. (It looks like a function call because, strictly speaking, it is, but why nitpick?) Use `var()` to tell the browser to retrieve the value of a previously defined variable and plug it into the actual property value, e.g., `var(--bqindents)`.

So… that’s it?

Yup. First, declare your variables as property/value pairs on a selector such as the `:root` pseudo-class.

```
:root {
  --bgcolour: #ffffd0;
  --bqindents: 40px;
  --warningtextsize: 125%;
}
```

Then, retrieve the variables’ contents by referencing their names as property values in later rules wherever they’re needed.

```
table {
  background-color: var(--bgcolour);
}
blockquote {
  margin-left: var(--bqindents);
  margin-right: var(--bqindents);
}
p.warning {
  color: red;
  background-color: var(--bgcolour);
```

```
    font-size: var(--warningtextsize);
  }
```

Let's review.

1. Declare
2. Retrieve
3. Enjoy

No, really!

PAQs

We're all familiar with FAQs — Frequently Asked Questions — but this section contains some things you'll *probably* ask. So… PAQs.

Q: Should I always use the `:root` pseudo-class as a selector to define variables?

A: Not necessarily, but usually, as doing so serves a couple of purposes. First, it collects all your variables in one place for easy maintenance (especially if that's the first rule in your CSS file), which is kind of the point. Second, `:root` matches the whole page (essentially everything in the <html> element), so it makes all your variables global in scope.

Q: Hmm, does that mean I can set identical variables with different values in different rules to get specifically-scoped, non-global variables?

A: Yes. Normally this is not necessary, but there are cases in which it makes sense. See **Advanced Stuff**, below.

Q: Do variable-based properties cascade?

A: Yes, they are inherited like any other property. If you set a 's background color with a variable, its children will all have that background color.

Q: Are variable names case-sensitive?

A: Yes, but don't get me started on case. The names `--h1color`, `--H1color`, and `--h1Color` are all different variables, and will gleefully cause you countless hours of debugging grief. Go ahead, use camelCase if you must, but don't come crying to me when your rules break.

Q: Can I use dashes or underscores in variable names?

A: Yes, but not spaces. The names `--h1-color`, `--h1_color`, and `--h-1_color` are all valid variables, while `--h1 color` is, of course, not. In fact, because CSS is a dash-friendly language and uses dashes in many standard property names, dashes are preferred over underscores for consistency and variable name readability.

Q: Can I use one variable for multiple — even completely different — property values?

A: Yes, as long as the value is valid. For example, you might define a variable whose value is 20px and then use it for all manner of properties later: margins, padding, font size, whatever. See the double use of `--bqindents` in the example above.

Q: Okay, can I set a variable just to a numeric value and then use it for different kinds of properties?

A: No, you can't "build up" a value with separate numbers and units; a variable has to have a single, valid, unambiguous value. For example, this won't work:

```
:root { --my-value: 20; }
. . .
blockquote { margin-left: var(--my-value)px; }
```

Q: Well then, can I use a variable for a property name instead of a value?

A: *Definitely* no; variables can contain values only. For example, this absolutely will not work.

```
:root { --my-property: font-size; }
. . .
p { var(--my-property): 24px; }
```

Q: Can I use a variable value to set another variable value?

A: Um, yes. I can't really think of a good use case for it, but it does work.

```
:root { --mybasecolor: red; }
. . .
p.wow { --myparacolor: var(--mybasecolor); }
```

Q: Why not just use SASS, LESS, or some other CSS pre-processor for variables?

A: Why not just use a twelve-pound sledge to swat a gnat? :-|

What's the Deal with Colors?

Many CSS variables articles — and this one is no exception — lean heavily on color-based examples, especially at the outset. This is done for two good reasons. First, color replacement is a straightforward, easily-grasped use case; second, it is a very pragmatic and common application of the feature. Although setting colors is by no means the only way to use variables, it is the way most CSS authors first experiment with them.

Let's look at some practical, real-world examples using variables, both color-related and not.

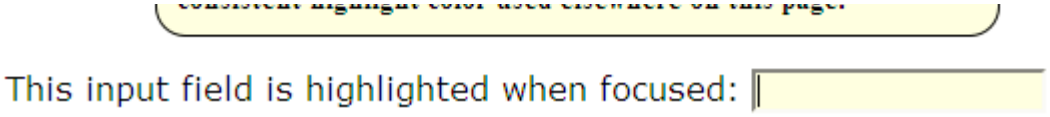
Consistent Highlight Colors

Imagine you have a web page in which you want to use a single color for different element highlighting effects: say, zebra tables, span highlight, blockquote background, and input field focus. You could code the highlight color in each rule that affects the elements, put the rules where they belong, and test your site. It might look something like this.

Row 1 Col 1	Row 1 Col 2
Row 2 Col 1	Row 2 Col 2
Row 3 Col 1	Row 3 Col 2

This is a paragraph with some text hightlighted and some not.

> This a blockquote that uses a variety of properties, including the consistent highlight color used elsewhere on this page.



But, looking at it, you're not as thrilled with the pastel yellow as you thought, and decide to use a different highlight color. Of course, now you have to find the color code in every instance where it is used in your CSS — and again, the rules may be nowhere near each other — and change it, hoping you don't miss one or make a typo. Change, break, repeat. Ugh.

Or, you could just set a variable to the original color value and use the variable in the actual rules, like this.

```
:root { --hilitecolor: #ffffe0; } /* pastel yellow */
. . .
tbody tr:nth-child(odd) {
  background-color: var(--hilitecolor); }
span.hilite { background-color: var(--hilitecolor); }
blockquote { font-family: times; font-weight: bold;
  font-size: 90%; margin-left: 75px; margin-right: 75px;
  background-color: var(--hilitecolor); padding: 10px;
  border: 1px solid black; border-radius: 15px; }
input:focus { background-color: var(--hilitecolor); }
```

Now, when you want to change the highlight color everywhere it's used in the page, you need only change *one variable declaration*, without ever touching the rules themselves, and all the highlights change with it. Still not happy with the color? Change it again, *in just one place*, and reload. Easy-peasy.

Zebra Tables

Zebra tables and other typically single-use items raise an interesting issue: If you're only going to define one or two rules that use a variable, then why use a variable at all? There's really no multiple-change benefit, so why not just code the property value in the actual rule in the first place?

That's a great question, and the great answer is: ease of maintenance. The rules that define the zebra table's alternating background colors might be buried way down in your CSS, and thus hard to find and maintain, but if you declare your color variables on

the `:root` selector and put that declaration at the top of the stylesheet, you can find it, change it, and be done with it literally in seconds. Here's an example using both even and odd row highlighting.

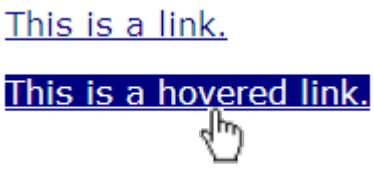
Row 1 Col 1	Row 1 Col 2
Row 2 Col 1	Row 2 Col 2
Row 3 Col 1	Row 3 Col 2

```
:root { --zebraevencolor: lightgreen;
        --zebraoddcolor: lightblue; }
. . .
tbody tr:nth-child(even) {
  background-color: var(--zebraevencolor); }
tbody tr:nth-child(odd) {
  background-color: var(--zebraoddcolor); }
```

Now, no matter where the two `:nth-child()` rules are, you don't have to go looking for them. Just change the color values defined in the two variables at the top of the CSS and you're done.

Swapping Link Colors on Hover

Here's a single-use case that was just made for variables. Some developers (ahem) like to swap foreground and background link colors when the user hovers over a text link. This simple technique makes the links "pop" visually, but because it uses the existing colors (in reverse) it doesn't detract from the page's color scheme. Or, you can use a color combo that contrasts with the page scheme for added effect. Here's how it looks in blue and white.



As simple as this sounds, it is a complete bear to code correctly, both initially and — oh please, no! — if you ever want to change the colors later. In this scheme, the `:link` and `:visited` pseudo-class rules use foreground for foreground and background for

background, but the `:hover` and `:active` pseudo-class rules use background for foreground and foreground for background. Got that?

That is, there are *eight* color properties and values in those "simple" rules, and if you ever get one wrong — and you will — they're no fun to find and fix. However, if you set the colors with variables, then it's a piece of cake to change the colors later without even looking at the rules. All it takes is two variables.

```
:root { --linkfg: #000080;
        --linkbg: #ffffff; }
 . . .
 a:link {color:var(--linkfg); background-color:var(--linkbg);}
 a:visited {color:var(--linkfg); background-color:var(--linkbg);}
 a:hover {color:var(--linkbg); background-color:var(--linkfg);}
 a:active {color:var(--linkbg); background-color:var(--linkfg);}
```

Once the four rules are coded with the `--linkfg` and `--linkbg` variables, they're good forever. As with the examples above, this tactic greatly simplifies tweaking the colors multiple times until you get a combination you like.

> *And you may already know to code the link pseudo-classes in the correct order: link, visited, hover, and active. Many CSS coders remember the order via the mnemonic "**L**o**V**e over **HA**te".*

Multi-part Values

Recall we mentioned earlier how variables can take any valid value; that's true whether the value is a singleton like `yellow` or a multiple like `10px 20px 15px 20px` where the values are just strung together and not comma-delimited. Let's try an example.

Shadows are a great feature, but they usually require a bit of trial and error to get right. There are, after all, four components to a shadow: horizontal offset, vertical offset, blur radius, and color. Let's say you have a page where you want various textual and graphical elements to have identical shadows, but you don't really want to spend the next eighteen hours tweaking one component at a time in multiple rules scattered about your CSS to make sure that they all look right — and that they all look the same. CSS variables to the rescue.

Just set up one variable containing your “standard shadow” multi-part value and code all the shadow rules to use that variable. Want to tweak it, test it, change it around later? No worries, just change the single variable declaration and reload.

```
:root { --stdshadow: 3px 3px 5px #A0A0A0; }
. . .
h1 { text-shadow: var(--stdshadow); }
p.shadow { text-shadow: var(--stdshadow); }
img.boxed { box-shadow: var(--stdshadow); }
```

Et voilà, consistent shadows minus the frustration.

Themes (AKA Keeping Marketing Happy)

The logical extension of all this is to use variables not just for the occasional color or indent setting, but to use them as a way to set — and, more importantly, maintain — substantial collections of values that work together, such as colors, margins, spacing, line height, list style, shadows, justification, font stacks, and so on; in other words, themes.

Variables-based themes can be useful not only for your own sanity, but for keeping interdepartmental peace, like when the marketing rep helpfully says, “No, not *that* blue, *this* blue!” Regardless of who requests a change or why, you can quickly and easily test it without archiving multiple “just in case” copies of your entire CSS. Just keep one

variables block that represents your test theme and another that's your fallback theme, comment out the fallback, and tweak away.

For example, without even seeing the page to which these theme blocks apply, you can easily imagine how different it will look when one or the other variable set is applied.

```
/* Variable set 1 — test theme (currently in use) */

:root {
--transition-type: padding-left 250ms ease-out;
--transition-padding: 20px;
--transition-font-size: 100%;
--navbar-bgcolor: lightgray;
--navbar-h2-color: brown;
--navbar-h2-border: thin brown solid;
--navbar-font-size: 12px;
--font-stack: Cambria, Georgia, "Times New Roman", serif;
--font-size: 16px;
--font-color: darkblue;
--max-width: 1000px;
--header-fontvariant: normal;
--header-color: white;
--header-bg: teal;
--header-shadow: 2px 3px 4px red;
--header-curve: 20px;
--header-border: thin darkblue solid;
--teaser-color: yellow;
--byline-color: white;
--h3-color: teal;
--h3-fontvariant: small-caps;
--dropcap-size: 200%;
--dropcap-font: Times;
--pullquote-border: thick black solid;
--pullquote-start: url(quotestart.png);
--pullquote-end: url(quoteend.png);
--zebraoddcolor: white;
--zebraevencolor: lightyellow;
--linkfg: darkblue;
--linkbg: white;
}

/* Variable set 2 -- fallback theme (currently commented out)

:root {
--transition-type: font-size 250ms ease-out;
--transition-padding: 0px;
--transition-font-size: 200%;
--navbar-bgcolor: lightblue;
--navbar-h2-color: darkgreen;
```

```
--navbar-h2-border: thin darkgreen dashed;
--navbar-font-size: 14px;
--font-stack: Verdana, Arial, sans;
--font-size: 18px;
--font-color: darkgreen;
--max-width: 1500px;
--header-fontvariant: small-caps;
--header-color: black;
--header-bg: lightyellow;
--header-shadow: 2px 3px 4px orange;
--header-curve: 100px;
--header-border: thick red solid;
--teaser-color: red;
--byline-color: orange;
--h3-color: darkred;
--h3-fontvariant: normal;
--dropcap-size: 400%;
--dropcap-font: Courier;
--pullquote-border: thick #ff6a00 dotted;
--pullquote-start: url(orangequotestart.png);
--pullquote-end: url(orangequoteend.png);
--zebraoddcolor: yellow;
--zebraevencolor: lightgray;
--linkfg: red;
--linkbg: white;
}

*/
```

If the test theme goes south on you, you can comment it out and reinstate the fallback theme until it's fixed. Again, you do not have to find and modify individual rules, ever. That's the power of variables-based themes.

> *If you'd like to see the theme idea in action, visit my CSS variables folder. There are two pages, apocalypsetoday1.htm and apocalypsetoday2.htm, that are styled by apocalypsetoday1.css and apocalypsetoday2.css, respectively. Both the HTML pages and the CSS stylesheets are absolutely identical except for their variables-based themes. Open the HTML pages side by side to see the striking (but not necessarily attractive ;-) differences achieved by simply swapping out a set of CSS variables.*

Just One More Thing

Speaking of fallbacks, there's one more thing you should know about: fallback values. The `var()` function has a second, optional parameter that takes over if the variable resolution fails for any reason (like, oh, I don't know, a camelCase error?!?).

Similar to font stacks that let the browser recover from a bad font call, fallback values let your rules recover from a bad variable. They're simple to code and can totally save your bacon when things go wrong. After the variable reference in the property value, just add a comma and a value to be used as the fallback if the variable doesn't resolve correctly, like this.

```
:root { --custom-width: 80%;
        --myTextColour: darkblue; }
. . .
div.special { width: var(--custom-width, 80%);
  color: var(--mytextcolour, red); }
```

In the `class="special"` divs, the reference to the `--custom-width` variable succeeds and, even if it doesn't, the fallback value matches the intended variable value, so it will always look right. However, the reference to the `--mytextcolour` variable will fail — I'm pretty sure you know why — so the fallback value will take the place of the unresolved variable value.

In fact, that second example demonstrates a creative and quite useful approach to variable fallbacks: using outrageous values as a debugging tool! In this example, you'll expect the div text to be dark blue, but it will jump out at you in red instead — a clear indication that the variable failed to resolve. Find that pesky camelCase, kill it with fire, and retest. Ah, dark blue text.

Advanced Stuff

Scope

Let's revisit the concept of scope. The most common scope of CSS variables is global, meaning that they are available to any CSS rule and thus to any HTML element. But there are cases where you might want to limit the scope in order to simplify page maintenance and updating.

Consider an HTML page with multiple content blocks, each containing a series of short articles, where the articles belong to different logical groups or sections. The basic HTML might look something like this.

```
<section class="thisweek">
<article>
. . .
</article>
<article>
. . .
</article>
<article>
. . .
</article>
</section>

<section class="lastweek">
<article>
. . .
</article>
<article>
. . .
</article>
<article>
. . .
</article>
</section>

<section class="oldstuff">
<article>
. . .
</article>
<article>
. . .
</article>
<article>
. . .
</article>
</section>
```

The section classes would be used by CSS to set various properties, so the articles in a section look similar to each other, but different from those in other sections. That is, each section has its own look, or mini-theme.

You can infer from the structure that the articles would likely be moved from section to section as they get older and new content is added. This would be a fairly easy maintenance task; just cut and paste an entire article from one section to another. But how will they look once moved? If the elements in the articles have their own CSS rules — and they will — how can you ensure they'll take on the mini-themes of their new sections? (I expect you're ahead of me on this one; if you aren't, I'm doing it wrong.)

To keep things simple, let's just play with the color and size of the <h1>s in the articles, but bear in mind that we might modify many other properties of the various article elements as well — text size, font, images, shadows, margins, backgrounds, link appearance, etc. — in order to visually distinguish among the sections. Here's some CSS.

```
section.thisweek {
  --h1-color: red;
  --h1-size: 110%;
}

section.lastweek {
  --h1-color: green;
  --h1-size: 100%;
}

section.oldstuff {
  --h1-color: blue;
  --h1-size: 90%;
}

article h1 {
  color: var(--h1-color);
  font-size: var(--h1-size);
}
```

Note that we first set the two properties in question using identically named variables, but on the `section` rules, not on a `:root` rule. This establishes different values for the same variables in different, non-global scopes. The single follow-on rule, using descendant (also called contextual) selectors to match any h1s inside the articles, sets the heading color and font size for all articles in every section, using the resolved-at-run-time values of the variables *in their current scope*.

The upshot of this technique is that you can easily move articles from section to section, knowing that they will look like the other articles in the target section, without having to modify any article-specific rules. And, of course, none of these rules affect the h1s — or anything else — outside of those three sections.

Calculations

Earlier, we said that you can't build up a property value using a variable value and a hard-coded unit of measurement, and that's true. You can, however, use the (somewhat

esoteric but completely useful) CSS `calc()` function to calculate an on-the-spot value that gets the job done.

Say you have a centered banner that should stretch most of the way, but not all the way, across a page. You could code its width as a percentage of the page width, e.g., `width: 85%;` , but that's less than optimal; every time the page width changes, so does the banner width and the width of the clear space at its left and right edges. If you wanted to maintain a constant 40px on either end, regardless of page width, you could do it like this using CSS math: `width: calc(100% – 80px);` , splitting the leftover 80px into 40px per side.

So how can we combine `calc()` and `var()` to achieve a useful result? Let's take a different example. Say we want blockquote page text to be at 1.5em and warning page text to be at 2.0em, but we recognize that those sizes may not be set in stone; user testing, marketing requirements, or other outside influences might force them to change from time to time. Here's some CSS that puts a resizing factor into an easily-maintained variable and then uses `calc()` to set the actual size at run-time.

```
:root {
  --bq-resize-by: 1.5;
  --warning-resize-by: 2.0;
}
. . .
blockquote {
  font-size: calc(var(--bq-resize-by) * 1em);
}
.warning {
  font-size: calc(var(--warning-resize-by) * 1em);
}
```

By multiplying the variable values against the constant font size of 1em, we get the desired sizes for the two text types. And, as always, when the multiplication factors do need to change, they are easily found in the `:root` rule at the top of the CSS. If blockquotes need to be at, say, 0.9em and warnings need to be at 1.75em, leave the rules alone; they're fine. Just change the variables and you're done.

JavaScript

Okay, just one more “just one more thing” thing. This is another capability I can’t think of a great use case for offhand, but I’m sure there is one. Also, the devs in the audience will probably beat me with an anonymous function if I fail to mention it.

Simply put, you can access CSS variables — both get and set — from JavaScript. To get a specific variable’s value, say, `--my-bg-colour`, retrieve the document’s computed style object, get the property value from that, and put it into a JavaScript variable.

```
var allstyles = getComputedStyle(document.documentElement);
var varval = string(allstyles.getPropertyValue("--my-bg-colour"));
var varval = varval.trim();
```

The `trim()` function isn’t required, but does snip off any whitespace from the value.

Conversely, to set a variable’s value, just specify it directly in the document’s style object.

```
document.documentElement.style.setProperty("--my-bg-colour",
"#008080");
```

Finally, just as you can set one variable’s value from another variable in CSS, you can do the same thing in JavaScript by specifying the other variable as the second parameter. And yes, the `var()` and the quotes are required!

```
document.documentElement.style.setProperty("--my-bg-colour", "var(--
my-fg-colour)");
```

Okay, I think we’re done here.

Takeaways

If you’ve been paying attention, you’ve probably come to these conclusions already, but let’s recap the main points.

1. CSS variables are a BIG deal. Developers have asked for this feature for years, and genuine native browser support has been a long time coming, but it's finally here. No more third-party utilities, no more extra build steps, no more (eek!) command-line shenanigans just to make your CSS work like it should.

2. CSS variables are not just an esoteric programmer's trick, some oddball feature that only hardcore bit-flippers will appreciate. Everyone who uses CSS can and should use CSS variables; they are a huge time-saver in both development and maintenance modes.

3. CSS variables are an easy-to-learn and (dare we say it?) fun-to-use feature, and incredibly useful. Whether you're new to the concept of variables in general or an old hand at using them in procedural languages, you'll catch on to the CSS flavor right away — and the more you use them, the more uses you will find for them.

Resources

Of course, a quick search for "css variables" returns thousands of hits — not all of them with current, or even correct, information. Here are a few links that I found useful.

· A short and useful basic CSS variables tutorial

· A comparison of native and preprocessor CSS variables

· A very good "What You Should Know" tutorial

· A CSS variables guide that covers calc() and JavaScript access

· The W3C Editor's Draft specification

Thanks!

Thank you for reading this article; I hope you had fun and learned something along the way! Comments or questions? Contact the author, Dave Gash, at dave@davegash.com.

CSS Web Development Css Variables