screenshot-adamwathan.me-2020.11.16-11_07_28
https://adamwathan.me/composing-the-uncomposable-with-css-variables/
16.11.2020

SEPTEMBER 18, 2020

Composing the Uncomposable with CSS Variables

Many CSS properties are *shorthands* for a set of other properties, for example the `margin` property is a shorthand for setting `margin-top`, `margin-right`, `margin-bottom`, and `margin-left` all at once.

Because the `margin` property *decomposes* into those four separate properties, it translates well to a utility class system like Tailwind CSS, where we can create separate utility classes for each property, then compose them arbitrarily in HTML:

```
<style>
  .mt-2 {
    margin-top: 0.5rem;
  }
  /* ... */
  .mr-6 {
    margin-right: 1.5rem;
  }
  /* ... */
  .mb-8 {
    margin-bottom: 2rem;
  }
  /* ... */
  .ml-4 {
    margin-left: 1rem;
  }
</style>

<div class="mt-2 mr-6 mb-8 ml-4">
  <!-- ... -->
</div>
```

Without this composability, it would be essentially impossible to do all of the things you can do in pure CSS because of the combinatoric explosion of classes that would have to exist to support every combination of values.

For example, look at this absurdity:

```
.mt-2_mr-0_mb-0_ml-0 {
  margin: 0.5rem 0 0 0;
}
.mt-2_mr-0_mb-0_ml-1 {
  margin: 0.5rem 0 0 0.25rem;
}
.mt-2_mr-0_mb-0_ml-2 {
  margin: 0.5rem 0 0 0.5rem;
}
/* ... */
.mt-2_mr-0_mb-1_ml-0 {
  margin: 0.5rem 0 0.25rem 0rem;
}
.mt-2_mr-0_mb-1_ml-1 {
  margin: 0.5rem 0 0.25rem 0.25rem;
}
.mt-2_mr-0_mb-1_ml-2 {
  margin: 0.5rem 0 0.25rem 0.5rem;
}
/* ... */
.mt-48_mr-48_mb-48_ml-48 {
  margin: 12rem 12rem 12rem 12rem;
}
```

If we had to do this in Tailwind, the development build would be like 4gb *(instead of a lean and mean 3mb, har har)*.

Sadly, some CSS properties *do not* decompose into separate properties. One example is `transform` *(although there are proposals for splitting it up!)*

```
.awesomeify {
  transform: translateX(3rem) rotate(90deg) scale(1.5);
}
```

If we want to translate an element by *X*, rotate it by *Y*, and scale it by *Z*, there's no way to do it using three separate utility classes.

Or is there?

A brief introduction to CSS variables

CSS custom properties *(more commonly referred to as CSS variables)* allow you to create a single source of truth for a value and refer to it in other parts of your CSS:

```
:root {
  --color-brand: #0d84ff;
}

.btn-primary {
  background-color: var(--color-brand);
}

.link {
  color: var(--color-brand);
}
```

The cool things about CSS variables is that unlike in

The cool things about CSS variables is that unlike in Sass/Less/Stylus/potato, they are computed at *run-time* rather than at build-time, so they can change.

For example, you might set your headline color to black by default:

```
:root {
  --headline-color: #000;
}

.headline {
  color: var(--headline-color);
}
```

...then magically override it white any time it's within a parent with a class of `dark`:

```
.dark {
  --headline-color: #fff;
}
```

```
<h1 class="heading">My text is black</h1>

<div class="dark">
  <h1 class="heading">My text is white</h1>
</div>
```

Using CSS variables for partial values

This is cool on its own, but what's even cooler is that CSS variables don't have to represent a *complete* value — they can be used for partial values, too.

For example, you can store just the RGB channels of a color in a variable, and stick it into the `rgba()` function along with an opacity value:

```
:root {
  --rgb-brand: 13, 132, 255;
}

.btn-primary--faded {
  background-color: rgba(var(--rgb-brand), 0.75);
}
```

This is how Tailwind's **text opacity** utilities work.

Every text color utility sets a `--text-opacity` variable to `1`, then references it for the alpha channel of an `rgba` value:

```
.text-blue-300 {
  --text-opacity: 1;
  color: rgba(144, 205, 244, var(--text-opacity));
}
```

Then the actual text opacity utilities simply change that variable to another value:

```
.text-opacity-50 {
  --text-opacity: 0.5;
}
```

Since we define the text opacity utilities *after* the text color utilities, the variable value in the text opacity utility always takes precedence over the value set in the text color utility:

```
<h1 class="text-blue-300 text-opacity-50">
  I'm blue at 50% opacity.
</h1>
```

Without this technique, it would be impossible to control the color and opacity independently using classes, because all the work needs to be done in one CSS property.

Filling "slots" using CSS variables

So what about our original issue, trying to make the `transform` property composable?

The way we do it in Tailwind is by creating variables for the various transform features, and combining them together to create the entire transform value.

Here's a slightly simplified version *(we also let you scale X and Y independently, as well as skew)*:

```
.transform {
  --transform-translate-x: 0;
  --transform-translate-y: 0;
  --transform-rotate: 0;
  --transform-scale: 1;
  transform: translateX(var(--transform-translate-x)) translateY(var(--tra
}
```

You'll see that what we're doing here is creating "slots" in the transform value, one for each of: `translateX`, `translateY`, `rotate`, and `scale`.

We fill each slot with a "no-op" value by default, like rotating by 0 degrees, or scaling by a factor of 1.

Then we define additional classes that manipulate only the CSS variables:

```
.translate-x-2 {
  --transform-translate-x: 0.5rem
}
.translate-y-3 {
  --transform-translate-x: 1.5rem
}
.rotate-45 {
  --transform-rotate: 45deg
}
.scale-150 {
  --transform-scale: 1.5;
}
```

Again, since those utilities are defined *after* the `transform` utility, they override the variables, without actually blowing away the `transform` property itself:

```
<div class="transform translate-y-3 rotate-45 scale-150">
  <!-- ... -->
</div>
```

The `transform` class "enables" transforms, and the classes for each variable just manipulate one of individual transform "slots".

This takes a traditionally non-composable property, and makes it perfectly composable using multiple classes in your HTML.

Concatenating optional values using empty variables

This is where things get really wild.

Say you're trying to make something like the `font-variant-numeric` property composable.

It's made up of a bunch of different optional chunks:

```
font-variant-numeric: {ordinal?} {slashed-zero?} {figure-value?} {spacing-
```

So all of these are valid:

```
.my-class {
  font-variant-numeric: ordinal;
  font-variant-numeric: slashed-zero;
  font-variant-numeric: ordinal slashed-zero;
  font-variant-numeric: ordinal tabular-nums;
  font-variant-numeric: slashed-zero lining-nums;
  font-variant-numeric: slashed-zero diagonal-fractions;
  font-variant-numeric: ordinal tabular-nums diagonal-fractions;
  font-variant-numeric: ordinal slashed-zero oldstyle-nums tabular-nums di
  /* Etc. */
}
```

Taking what you've learned above, you might think to try something like this:

```
.variant-numeric {
  font-variant-numeric: var(--variant-ordinal) var(--variant-slashed-zero)
}
.ordinal {
  --variant-ordinal: ordinal;
}
.slashed-zero {
  --variant-slashed-zero: slashed-zero;
}
/* ... */
.stacked-fractions {
  --variant-fractions: stacked-fractions;
}
.diagonal-fractions {
  --variant-fractions: diagonal-fractions;
}
```

The problem is that doing things this way, our variables like `--variant-ordinal` have no default value, which means when `var(--variant-ordinal)` tries to resolve, it will be what the spec calls *the invalid value*, which actually invalidates the whole rule, so the CSS isn't applied at all.

So you might think, "well maybe we can default to an empty string?"

```
--variant-ordinal: '';
```

The problem is that's not actually an empty string at all — in CSS that's the literal value `''`. Remember, CSS values aren't quoted.

So what *can* we do? Well it turns out *whitespace* is a valid CSS value.

```
--variant-ordinal: ;
```

Yep, that's not a syntax error, that's perfectly valid CSS that sets `--variant-ordinal` to a literal space character.

Now when `var(--variant-ordinal)` resolves, we'll get a space, which is totally allowed in a list of values.

Here's what a working solution looks like:

```
.variant-numeric {
  --variant-ordinal: ;
  --variant-slashed-zero: ;
  --variant-figure: ;
  --variant-spacing: ;
```

```
  --variant-fractions: ;
  font-variant-numeric: var(--variant-ordinal) var(--variant-slashed-zero)
}
.ordinal {
  --variant-ordinal: ordinal;
}
.slashed-zero {
  --variant-slashed-zero: slashed-zero;
}
/* ... */
.stacked-fractions {
  --variant-fractions: stacked-fractions;
}
.diagonal-fractions {
  --variant-fractions: diagonal-fractions;
}
```

The only problem with this is that **it probably won't work in your production build**.

If you're using any kind of CSS minifier, it is almost guaranteed to strip these variables out in my testing. In fact, this is the only thing that actually worked:

```
.variant-numeric {
  --variant-ordinal: var(--not-a-real-variable,/*!*/ /*!*/);
  /* ... */
}
```

What the fuck? Yeah I agree. Basically what we're doing here is tricking the minifier into keeping that space that appears between the two comments. We have to put `!` in the comments to tell minifiers not the strip them, and we have to do all of that as the default value of a non-existent variable, otherwise the minifiers will see the value as empty and strip the whole declaration.

The nice part is this should **actually be fixed** when PostCSS 8 gains mass adoption, so we'll be able to go back to this slightly less weird looking syntax instead:

```
.variant-numeric {
  --variant-ordinal: ;
  /* ... */
}
```

CSS, what a riot.