Houdini Properties & Values

If you haven't read my prior articles about CSS Houdini, I'll point you to the video below by Tab Atkins where he talk about Houdini as a (far) future combination of CSS and Javascript (he should know, he edits the Houdini specs and a whole bunch of other CSS specs for W3C).

This is not CSS in JS but a way to use JS to hook into the browser's rendering lifecycle to add your own bits and pieces. With this we don't have to wait for browser vendors to implement a feature, we can create our version that will fail gracefully if the browser doesn't understand Houdini, calc or CSS variables.

Properties and values are CSS Variables in steroids. The spec is designed to address the shortcomings of variables as first implemented:

- Custom properties allow for validation because we know what kind of value they have without having to parse and check for errors on every use
    - It's not a string but a proper CSS Typed OM value
- We can assign default/initial values
    - No manual error handling every time the property is used
    - Provides a sensibble default when there is an error
- We can decide if the property inherits down the cascade or not
    - Variables as currently defined always inherit

- Allows for animating and transitioning of the custom properties
    - Values are known Typed OM values

Now that we know why these custom properties are important let's look at how they work.

Registering Properties

The firs step in using custom properties is to define them in JavaScript.

`CSS.registerProperty` (case sensitive) does this by taking

```
if ('registerProperty' in CSS) {
  CSS.registerProperty({
    name: '--my-custom-prop',
    syntax: '<color>',
    inherits: true,
    initialValue: 'black',
  });
}
```

There are 4 values that we need to pass to `registerProperty`:

The **name** is what we'll use to reference the property. The two dashes at the beginning should be familiar from CSS variables and are required. This is how we'll distinguish our custom variables and properties from what the CSS WG does and will do in the future.

Syntax indicates the possible syntaxes for the property. The following values are available in level 1 of the spec and matching corresponding units in CSS Values and Units Module Level 3

- length
- number
- percentage
- length-percentage (defined in the CSS Properties spec)
    - Any valid `<length>`
    - Any valid `<percentage>`
    - Any valid `<calc()>` expression combining `<length>` and

<percentage> components.
- color (defined in CSS Color spec)
- image
- url
- integer
- angle
- time
- resolution
- transform-function
- transform-list
    - One or more transform functions
- custom-ident
- Identifiers
    - Any sequence consisting of a name-start code point, followed by zero or more name code points, which matches the <custom-ident> production

In addition you can use the following modifiers or replacement tokens for the syntax:

- * any value
- | logical or (one or the other)
-     - one or more of the type specified
- # one or more of the type specified **separated by commas**

You can create farily complex syntax for your custom properties but until we become familiar with them, I advocate for the KISS (Keep It Simple Silly) principle.

`Inherit` tells the CSS parser if this custom rule should propagate down the cascade. Setting it to fallse gives us more power to style specific elements wituout being afraid to mess up elements further down the chain.

The final value is an `initialValue`. Use this to provide a sensible default for the property. We'll analyze why this is important later.

That's it... we now have a custom property.

Using custom properties

To demonstrate how to use Custom Properties we'll reuse the `--bg-color` Javascript example and use it in several different elements.

```
CSS.registerProperty({
  name: '--bg-color',
  syntax: '<color>',
  inherits: false,
  initialValue: 'red',
});
```

The CSS will not be any different than if we used variables. But the things it does for free are much more interresting.

First we define common parameters to create 200px by 200px squares using `div` elements.

```
div {
  border: 1px solid black;
  height: 200px;
  width: 200px;
}
```

`.smoosh1` and `.smoosh2` set up colors other than the initial value and each have a different color to change on hover.

```
.smoosh1 {
  --bg-color: rebeccapurple;
  background: var(--bg-color);
  transition: --bg-color 0.3s linear;
  position: absolute;
  top: 50vh;
  left: 15vw;

  &:hover {
    --bg-color: orange;
  }
}

.smoosh2 {
  --bg-color: teal;
```

```
  background: var(--bg-color);
  transition: --bg-color 0.3s linear;
  position: absolute;
  top: 20em;
  left: 45em;

  &:hover {
    --bg-color: pink;
  }
}
```

`.smoosh3` was set up with a wrong type of color (1 is not a valid CSS color). In normal CSS the rule would be ignored and there would be no background color. Because we added an `initialValue` to the property, it'll take this value instead of giving an error.

```
.smoosh3 {
  --bg-color: 1;
  background: var(--bg-color);
  transition: --bg-color 0.3s linear;
  position: absolute;
  top: 5em;
  left: 35em;

  &:hover {
    --bg-color: lightgrey;
  }
}
```

You can see the full working demo in Codepen

Why is this important?

There are a couple reasons that make custom properties particularly useful.

Validation

As we saw in the last section, custom properties allow for default initial values that give developers a way to avoid errors or unexpected behavior.

If we define the following custom property:

```
CSS.registerProperty({
  name: '--bg-color',
  syntax: '<color>',
  inherits: false,
  initialValue: 'red',
});
```

And mistakenly use the property with the following declaration below:

```
.dark-theme-section {
  --bg-color: 1;
}
```

We would expect to get an error or, in CSS, to ignore the rule altogether. But, as the demo showed, the CSS will use the initial value.

We can also test what the value is using Javascript.

```
const section = document.querySelector('.smoosh3');
const styles = getComputedStyle(section);
const themeColor = styles.getPropertyValue('--bg-color');

console.log(themeColor); '--bg-color'// "red"
```

Animations

CSS variables don't provide good support for animation because the browser doesn't know what type of value it has.

Because we assign a value to the property now the parser knows what to do

with the property when we ask to animate it.

The Codepen Demo shows how the animation works.

We could do further modularization in the animation itself by creating properties for the animation parameters or the positioning of each individual content... but, for now, baby steps are OK.

Links and Resources

- Spec
- CSS Custom Properties In Depth
- Houdini Quickstart
- CSS Houdini Experiments
- Houdini on Smashing Mag
- Working with the new CSS Typed Object Model
- Houdini is like Babel but for CSS