Working with the new CSS Typed Object Model

By Eric Bidelman

Engineer @ Google working on web tooling: Headless Chrome, Puppeteer, Lighthouse

TL;DR

CSS now has a proper object-based API for working with values in JavaScript.

```
el.attributeStyleMap.set('padding', CSS.px(42));
const padding = el.attributeStyleMap.get('padding');
console.log(padding.value, padding.unit); // 42, 'px'
```

The days of concatenating strings and subtle bugs are over!

Heads up: Chrome 66 adds support for the CSS Typed Object Model for a subset of CSS properties.

Introduction

Old CSSOM

CSS has had an object model (CSSOM) for many years. In fact, any time you read/set `.style` in JavaScript you're using it:

```
// Element styles.
el.style.opacity = 0.3;
typeof el.style.opacity === 'string' // Ugh. A string!?

// Stylesheet rules.
document.styleSheets[0].cssRules[0].style.opacity = 0.3;
```

New CSS Typed OM

The new CSS Typed Object Model (Typed OM), part of the Houdini effort, expands this worldview by adding types, methods, and a proper object model to CSS values. Instead of

strings, values are exposed as JavaScript objects to facilitate performant (and sane) manipulation of CSS.

Instead of using **`element.style`**, you'll be accessing styles through a new **`.attributeStyleMap`** property for elements and a **`.styleMap`** property for stylesheet rules. Both return a **`StylePropertyMap`** object.

```
// Element styles.
el.attributeStyleMap.set('opacity', 0.3);
typeof el.attributeStyleMap.get('opacity').value === 'number' // Yay, a number!

// Stylesheet rules.
const stylesheet = document.styleSheets[0];
stylesheet.cssRules[0].styleMap.set('background', 'blue');
```

Because **`StylePropertyMap`**s are Map-like objects, they support all the usual suspects (get/set/keys/values/entries), making them flexible to work with:

```
// All 3 of these are equivalent:
el.attributeStyleMap.set('opacity', 0.3);
el.attributeStyleMap.set('opacity', '0.3');
el.attributeStyleMap.set('opacity', CSS.number(0.3)); // see next section
// el.attributeStyleMap.get('opacity').value === 0.3

// StylePropertyMaps are iterable.
for (const [prop, val] of el.attributeStyleMap) {
  console.log(prop, val.value);
}
// → opacity, 0.3

el.attributeStyleMap.has('opacity') // true

el.attributeStyleMap.delete('opacity') // remove opacity.

el.attributeStyleMap.clear(); // remove all styles.
```

Note that in the second example, **`opacity`** is set to string (**`'0.3'`**) but a number comes back out when the property is read back later.

If a given CSS property supports numbers, Typed OM will accept a strings as input, but always returns a number! The analogy between the old CSSOM and the new Typed OM is similar to how **`.className`** grew up and got its own API, **`.classList`**.

Benefits

So what problems is CSS Typed OM trying to solve? Looking at the examples above (and throughout the rest of this article), you might argue that CSS Typed OM is far more verbose than the old object model. I would agree!

Before you write off Typed OM, consider some of the key features it brings to the table:

- **Fewer bugs**. e.g. numerical values are always returned as numbers, not strings.

```
el.style.opacity += 0.1;
el.style.opacity === '0.30.1' // dragons!
```

- **Arithmetic operations & unit conversion**. convert between absolute length units (e.g. `px` -> `cm`) and do basic math.
- **Value clamping & rounding**. Typed OM rounds and/or clamps values so they're within the acceptable ranges for a property.
- **Better performance**. The browser has to do less work serializing and deserializing string values. Now, the engine uses a similar understanding of CSS values across JS and C++. Tab Akins has shown some early perf benchmarks that put Typed OM at **~30% faster** in operations/sec when compared to using the old CSSOM and strings. This can be significant for rapid CSS animations using `requestionAnimationFrame()`. crbug.com/808933 tracks additional performance work in Blink.
- **Error handling**. New parsing methods brings error handling in the world of CSS.
- "Should I use camel-cased CSS names or strings?" There's no more guessing if names are camel-cased or strings (e.g. `el.style.backgroundColor` vs `el.style['background-color']`). CSS property names in Typed OM are always strings, matching what you actually write in CSS :)

Browser support & feature detection

Typed OM landed in Chrome 66 and is being implemented in Firefox. Edge has shown signs of support, but has yet to add it to their platform dashboard.

Note: Only a subset of CSS properties are supported in Chrome 66+ for now.

For feature detection, you can check if one of the `CSS.*` numeric factories is defined:

```
if (window.CSS && CSS.number) {
  // Supports CSS Typed OM.
}
```

API Basics

Accessing styles

Values are separate from units in CSS Typed OM. Getting a style returns a `CSSUnitValue` containing a `value` and `unit`:

```
el.attributeStyleMap.set('margin-top', CSS.px(10));
// el.attributeStyleMap.set('margin-top', '10px'); // string arg also works.
el.attributeStyleMap.get('margin-top').value  // 10
el.attributeStyleMap.get('margin-top').unit // 'px'

// Use CSSKeyWorldValue for plain text values:
el.attributeStyleMap.set('display', new CSSKeywordValue('initial'));
el.attributeStyleMap.get('display').value // 'initial'
el.attributeStyleMap.get('display').unit // undefined
```

Computed styles

Computed styles have moved from an API on `window` to a new method on `HTMLElement`, `computedStyleMap()`:

Old CSSOM

```
el.style.opacity = 0.5;
window.getComputedStyle(el).opacity === "0.5" // Ugh, more strings!
```

New Typed OM

```
el.attributeStyleMap.set('opacity', 0.5);
el.computedStyleMap().get('opacity').value // 0.5
```

Note: One gotcha between `window.getComputedStyle()` and `element.computedStyleMap()` is that the former returns resolved values whereas the latter returns computed values. For example, Typed OM retains percentage values (`width: 50%`), while CSSOM resolves them to lengths (e.g. `width: 200px`).

Value clamping / rounding

One of the nice features of the new object model is **automatic clamping and/or rounding of computed style values**. As an example, let's say you try to set `opacity` to a value outside of the acceptable range, [0, 1]. Typed OM clamps the value to `1` when computing the style:

```
el.attributeStyleMap.set('opacity', 3);
el.attributeStyleMap.get('opacity').value === 3  // val not clamped.
el.computedStyleMap().get('opacity').value === 1 // computed style clamps value.
```

Similarly, setting `z-index:15.4` rounds to `15` so the value remains an integer.

```
el.attributeStyleMap.set('z-index', CSS.number(15.4));
el.attributeStyleMap.get('z-index').value  === 15.4 // val not rounded.
el.computedStyleMap().get('z-index').value === 15   // computed style is rounded.
```

CSS numerical values

Numbers are represented by two types of `CSSNumericValue` objects in Typed OM:

1. `CSSUnitValue` - values that contain a single unit type (e.g. "`42px`").
2. `CSSMathValue` - values that contain more than one value/unit such as mathematical expression (e.g. "`calc(56em + 10%)`").

Unit values

Simple numerical values ("`50%`") are represented by `CSSUnitValue` objects. While you *could* create these objects directly (`new CSSUnitValue(10, 'px')`) , most of the time you'll be using the `CSS.*` factory methods:

```
const {value, unit} = CSS.number('10');
// value === 10, unit === 'number'

const {value, unit} = CSS.px(42);
// value === 42, unit === 'px'

const {value, unit} = CSS.vw('100');
// value === 100, unit === 'vw'

const {value, unit} = CSS.percent('10');
// value === 10, unit === 'percent'

const {value, unit} = CSS.deg(45);
// value === 45, unit === 'deg'

const {value, unit} = CSS.ms(300);
// value === 300, unit === 'ms'
```

Note: as shown in the examples, these methods can be passed a `Number` or `String` representing a number.

See the spec for the full list of `CSS.*` methods.

Math values

`CSSMathValue` objects represent mathematical expressions and typically contain more than one value/unit. The common example is creating a CSS `calc()` expression, but there are methods for all the CSS functions: `calc()`, `min()`, `max()`.

```
new CSSMathSum(CSS.vw(100), CSS.px(-10)).toString(); // "calc(100vw + -10px)"

new CSSMathNegate(CSS.px(42)).toString() // "calc(-42px)"

new CSSMathInvert(CSS.s(10)).toString() // "calc(1 / 10s)"

new CSSMathProduct(CSS.deg(90), CSS.number(Math.PI/180)).toString();
// "calc(90deg * 0.0174533)"

new CSSMathMin(CSS.percent(80), CSS.px(12)).toString(); // "min(80%, 12px)"

new CSSMathMax(CSS.percent(80), CSS.px(12)).toString(); // "max(80%, 12px)"
```

Nested expressions

Using the math functions to create more complex values gets a bit confusing. Below are a few examples to get you started. I've added extra indentation to make them easier to read.

`calc(1px - 2 * 3em)` would be constructed as:

```
new CSSMathSum(
  CSS.px(1),
  new CSSMathNegate(
    new CSSMathProduct(2, CSS.em(3))
  )
);
```

`calc(1px + 2px + 3px)` would be constructed as:

```
new CSSMathSum(CSS.px(1), CSS.px(2), CSS.px(3));
```

`calc(calc(1px + 2px) + 3px)` would be constructed as:

```
new CSSMathSum(
  new CSSMathSum(CSS.px(1), CSS.px(2)),
  CSS.px(3)
);
```

Arithmetic operations

One of the most useful features of The CSS Typed OM is that you can perform mathematical operations on `CSSUnitValue` objects.

Basic operations

Basic operations (`add`/`sub`/`mul`/`div`/`min`/`max`) are supported:

```
CSS.deg(45).mul(2) // {value: 90, unit: "deg"}

CSS.percent(50).max(CSS.vw(50)).toString() // "max(50%, 50vw)"

// Can Pass CSSUnitValue:
CSS.px(1).add(CSS.px(2)) // {value: 3, unit: "px"}

// multiple values:
CSS.s(1).sub(CSS.ms(200), CSS.ms(300)).toString() // "calc(1s + -200ms + -300ms)"

// or pass a `CSSMathSum`:
const sum = new CSSMathSum(CSS.percent(100), CSS.px(20));
CSS.vw(100).add(sum).toString() // "calc(100vw + (100% + 20px))"
```

Conversion

Absolute length units can be converted to other unit lengths:

```
// Convert px to other absolute/physical lengths.
el.attributeStyleMap.set('width', '500px');
const width = el.attributeStyleMap.get('width');
width.to('mm'); // CSSUnitValue {value: 132.29166666666669, unit: "mm"}
width.to('cm'); // CSSUnitValue {value: 13.229166666666668, unit: "cm"}
width.to('in'); // CSSUnitValue {value: 5.208333333333333, unit: "in"}

CSS.deg(200).to('rad').value // 3.49066...
CSS.s(2).to('ms').value // 2000
```

Equality

```
const width = CSS.px(200);
CSS.px(200).equals(width) // true

const rads = CSS.deg(180).to('rad');
CSS.deg(180).equals(rads.to('deg')) // true
```

CSS transform values

CSS transforms are created with a **`CSSTransformValue`** and passing an array of transform values (e.g. **`CSSRotate`**, **`CSScale`**, **`CSSSkew`**, **`CSSSkewX`**, **`CSSSkewY`**). As an example, say you want to re-create this CSS:

```
transform: rotateZ(45deg) scale(0.5) translate3d(10px,10px,10px);
```

Translated into Typed OM:

```
const transform =  new CSSTransformValue([
  new CSSRotate(CSS.deg(45)),
  new CSSScale(CSS.number(0.5), CSS.number(0.5)),
  new CSSTranslate(CSS.px(10), CSS.px(10), CSS.px(10))
]);
```

In addition to its verbosity (lolz!), **`CSSTransformValue`** has some cool features. It has a boolean property to differentiate 2D and 3D transforms and a **`.toMatrix()`** method to return the **`DOMMatrix`** representation of a transform:

```
new CSSTranslate(CSS.px(10), CSS.px(10)).is2D // true
new CSSTranslate(CSS.px(10), CSS.px(10), CSS.px(10)).is2D // false
new CSSTranslate(CSS.px(10), CSS.px(10)).toMatrix() // DOMMatrix
```

Example: animating a cube

Let's see a practical example of using transforms. We'll using JavaScript and CSS transforms to animate a cube.

```
const rotate = new CSSRotate(0, 0, 1, CSS.deg(0));
const transform = new CSSTransformValue([rotate]);

const box = document.querySelector('#box');
box.attributeStyleMap.set('transform', transform);

(function draw() {
  requestAnimationFrame(draw);
  transform[0].angle.value += 5; // Update the transform's angle.
  // rotate.angle.value += 5; // Or, update the CSSRotate object directly.
  box.attributeStyleMap.set('transform', transform); // commit it.
})();
```

Notice that:

1. Numerical values means we can increment the angle directly using math!
2. Rather than touching the DOM or reading back a value on every frame (e.g. no `box.style.transform=`rotate(0,0,1,${newAngle}deg)``), the animation is driven by **updating the underlying `CSSTransformValue` data object, improving performance**.

Demo

Below, you'll see a red cube if your browser supports Typed OM. The cube starts rotating when you mouse over it. The animation is powered by CSS Typed OM! 🤘

CSS custom properties values

CSS `var()` become a **`CSSVariableReferenceValue`** object in the Typed OM. Their values get parsed into **`CSSUnparsedValue`** because they can take any type (px, %, em, rgba(), etc).

```
const foo = new CSSVariableReferenceValue('--foo');
// foo.variable === '--foo'

// Fallback values:
const padding = new CSSVariableReferenceValue(
    '--default-padding', new CSSUnparsedValue(['8px']));
// padding.variable === '--default-padding'
// padding.fallback instanceof CSSUnparsedValue === true
// padding.fallback[0] === '8px'
```

If you want to get the value of a custom property, there's a bit of work to do:

```
<style>
  body {
    --foo: 10px;
  }
</style>
<script>
```

```
  const styles = document.querySelector('style');
  const foo = styles.sheet.cssRules[0].styleMap.get('--foo').trim();
  console.log(CSSNumericValue.parse(foo).value); // 10
</script>
```

Position values

CSS properties that take a space-separated x/y position such as `object-position` are represented by `CSSPositionValue` objects.

```
const position = new CSSPositionValue(CSS.px(5), CSS.px(10));
el.attributeStyleMap.set('object-position', position);

console.log(position.x.value, position.y.value);
// → 5, 10
```

Parsing values

The Typed OM introduces parsing methods to the web platform! This means you can finally **parse CSS values programmatically, *before* trying to use it**! This new capability is a potential life saver for catching early bugs and malformed CSS.

Parse a full style:

```
const css = CSSStyleValue.parse(
    'transform', 'translate3d(10px,10px,0) scale(0.5)');
// → css instanceof CSSTransformValue === true
// → css.toString() === 'translate3d(10px, 10px, 0) scale(0.5)'
```

Parse values into `CSSUnitValue`:

```
CSSNumericValue.parse('42.0px') // {value: 42, unit: 'px'}

// But it's easier to use the factory functions:
CSS.px(42.0) // '42px'
```

Error handling

Example - check if the CSS parser will be happy with this `transform` value:

```
try {
  const css = CSSStyleValue.parse('transform', 'translate4d(bogus value)');
  // use css
} catch (err) {
  console.err(err);
}
```

Conclusion

It's nice to finally have an updated object model for CSS. Working with strings never felt right to me. The CSS Typed OM API is a bit verbose, but hopefully it results in fewer bugs and more performant code down the line.

Last updated July 2, 2018.