Hello and welcome, My name is Tommy Hodgins.

Have you all been having a good time here at Web Unleashed so far?

This is the first time I'm giving this talk to an audience, so as of right now this is exclusive content. There's some pretty wild stuff in today's presentation, so consider yourselves lucky.

In this talk I'm going to be introducing you to two different code patterns that you can apply to extend CSS to include *dozens* of new features. We're going to cover a lot of ground and so I'm going to move fast.

There will be a few sections of slides, broken up by three live coding explorations, where I demonstrate what we have learned. You're welcome to play around with your laptop if you want, but it might be easier if you follow along on the screen.

At the end there will be a section for Q&A, and anything we don't have time for in today's session you can ask me on Twitter!

Caffeinated Style Sheets

Today I'm going to talk to you about Caffeinated Style Sheets

What are Caffeinated Style Sheets?

Caffeinated
Style
Sheets

Well, It's CCS, but with JavaScript inside

{ CSS extended by JS }

We are going to be learning about extending CSS stylesheets with JavaScript

CSS R&D
2014 – 2018

Let me explain really briefly some of the big lessons I've learned from experimenting with CSS over the past few years, so you can understand where I'm coming from…

Big Takeaways

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Big Takeaways

- CSS can be extended in the browser by JavaScript

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Big Takeaways

- CSS can be extended in the browser by JavaScript
- Many desired CSS features are simple to implement with JavaScript

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Big Takeaways

- CSS can be extended in the browser by JavaScript
- Many desired CSS features are simple to implement with JavaScript
- There are 25–50 useful design techniques these features allow

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Big Takeaways

- CSS can be extended in the browser by JavaScript
- Many desired CSS features are simple to implement with JavaScript
- There are 25–50 useful design techniques these features allow
- Styling is event-driven in nature, and bigger than CSS

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Big Takeaways

- CSS can be extended in the browser by JavaScript
- Many desired CSS features are simple to implement with JavaScript
- There are 25–50 useful design techniques these features allow
- Styling is event-driven in nature, and bigger than CSS
- You don’t need a custom CSS syntax to extend CSS

My whole journey with extending CSS began with the question “Can CSS be extended at runtime in the browser” and discovering that, Yes, it can!

Then through exploring what was possible, I discovered that many different CSS features you might wish for, are actually pretty easy to implement with JavaScript.

After playing around with the features, and applying them in a variety of different situations, I discovered that there are a few dozen design techniques that these features make easy.

And through experimenting with all this, I began to see styling as something that was event-driven in nature, and bigger than CSS. CSS already has things like **:hover** and **:focus** that respond to events happening in the browser, and media queries which respond to changes in browser dimensions, so CSS itself is event-driven already, but not every event that happens in the browser is something CSS has knowledge of. We often rely on JavaScript to help us by toggling classes, or setting CSS variables to help CSS reach the things it doesn't have access to.

And the last big takeaway, and this one's fresh, is that you don't need a custom syntax to extend CSS.

Think of your stylesheet as a function of what's going on in the browser

In order to find the best way to extend your styles in the browser, it's important to think of your stylesheet at any given time as being a function of what's going on in the browser.

Think of your stylesheet as a function of what's going on in the browser

Or, to put it more clearly, to think of your stylesheet as a function.

Not a software framework,
a mental framework

Though I will be showing code samples today, I am not trying to convince you to use my plugins.

Not a software framework, a mental framework

I'm trying to teach you the patterns you need to go out and build the styling language of your dreams yourself.

I'm giving you a mental framework for thinking about how you can extend CSS.

The JS-in-CSS Recipe

For us to extend CSS, the first thing we need is a way to process and populate our event-driven styles. This is a simple pattern I call the JS-in-CSS pattern, and it's easy to implement. When you become aware of the pattern, and are intentional about applying it, you can use it to make your work simple, and your styles powerful.

There are three requirements for JS-in-CSS, the first is that the stylesheet is a regular JavaScript function, that returns a CSS stylesheet as a string of text.

The second requirement is that the stylesheet function can subscribe to different events happening in the browser.

And the last requirement is that somewhere a **<style>** tag is being populated with the resulting CSS, each time that an event that the stylesheet is listening to has been triggered, and the stylesheet function reprocesses.

As a bonus, by writing our stylesheets this way we can extend them with re-usable JavaScript functions that themselves return strings of CSS.

I'm going to show a series of slides quickly here where we will see this pattern grow out of one line of code, and I'll point out the distinctions in the pattern as they appear on the screen.

After this I'm going to code an example of the pattern that we can play around with.

The JS-in-CSS Recipe

- The stylesheet is a JS function that returns a string of CSS

For us to extend CSS, the first thing we need is a way to process and populate our event-driven styles. This is a simple pattern I call the JS-in-CSS pattern, and it's easy to implement. When you become aware of the pattern, and are intentional about applying it, you can use it to make your work simple, and your styles powerful.

There are three requirements for JS-in-CSS, the first is that the stylesheet is a regular JavaScript function, that returns a CSS stylesheet as a string of text.

The second requirement is that the stylesheet function can subscribe to different events happening in the browser.

And the last requirement is that somewhere a **<style>** tag is being populated with the resulting CSS, each time that an event that the stylesheet is listening to has been triggered, and the stylesheet function reprocesses.

As a bonus, by writing our stylesheets this way we can extend them with re-usable JavaScript functions that themselves return strings of CSS.

I'm going to show a series of slides quickly here where we will see this pattern grow out of one line of code, and I'll point out the distinctions in the pattern as they appear on the screen.

After this I'm going to code an example of the pattern that we can play around with.

The JS-in-CSS Recipe

- The stylesheet is a JS function that returns a string of CSS
- The stylesheet function can subscribe to events

For us to extend CSS, the first thing we need is a way to process and populate our event-driven styles. This is a simple pattern I call the JS-in-CSS pattern, and it's easy to implement. When you become aware of the pattern, and are intentional about applying it, you can use it to make your work simple, and your styles powerful.

There are three requirements for JS-in-CSS, the first is that the stylesheet is a regular JavaScript function, that returns a CSS stylesheet as a string of text.

The second requirement is that the stylesheet function can subscribe to different events happening in the browser.

And the last requirement is that somewhere a **<style>** tag is being populated with the resulting CSS, each time that an event that the stylesheet is listening to has been triggered, and the stylesheet function reprocesses.

As a bonus, by writing our stylesheets this way we can extend them with re-usable JavaScript functions that themselves return strings of CSS.

I'm going to show a series of slides quickly here where we will see this pattern grow out of one line of code, and I'll point out the distinctions in the pattern as they appear on the screen.

After this I'm going to code an example of the pattern that we can play around with.

The JS-in-CSS Recipe

- The stylesheet is a JS function that returns a string of CSS
- The stylesheet function can subscribe to events
- A **<style>** tag is populated with the resulting CSS

For us to extend CSS, the first thing we need is a way to process and populate our event-driven styles. This is a simple pattern I call the JS-in-CSS pattern, and it's easy to implement. When you become aware of the pattern, and are intentional about applying it, you can use it to make your work simple, and your styles powerful.

There are three requirements for JS-in-CSS, the first is that the stylesheet is a regular JavaScript function, that returns a CSS stylesheet as a string of text.

The second requirement is that the stylesheet function can subscribe to different events happening in the browser.

And the last requirement is that somewhere a **<style>** tag is being populated with the resulting CSS, each time that an event that the stylesheet is listening to has been triggered, and the stylesheet function reprocesses.

As a bonus, by writing our stylesheets this way we can extend them with re-usable JavaScript functions that themselves return strings of CSS.

I'm going to show a series of slides quickly here where we will see this pattern grow out of one line of code, and I'll point out the distinctions in the pattern as they appear on the screen.

After this I'm going to code an example of the pattern that we can play around with.

The JS-in-CSS Recipe

- The stylesheet is a JS function that returns a string of CSS
- The stylesheet function can subscribe to events
- A **<style>** tag is populated with the resulting CSS
- The stylesheet function can be extended with re-usable JS functions that return strings of CSS

For us to extend CSS, the first thing we need is a way to process and populate our event-driven styles. This is a simple pattern I call the JS-in-CSS pattern, and it's easy to implement. When you become aware of the pattern, and are intentional about applying it, you can use it to make your work simple, and your styles powerful.

There are three requirements for JS-in-CSS, the first is that the stylesheet is a regular JavaScript function, that returns a CSS stylesheet as a string of text.

The second requirement is that the stylesheet function can subscribe to different events happening in the browser.

And the last requirement is that somewhere a **<style>** tag is being populated with the resulting CSS, each time that an event that the stylesheet is listening to has been triggered, and the stylesheet function reprocesses.

As a bonus, by writing our stylesheets this way we can extend them with re-usable JavaScript functions that themselves return strings of CSS.

I'm going to show a series of slides quickly here where we will see this pattern grow out of one line of code, and I'll point out the distinctions in the pattern as they appear on the screen.

After this I'm going to code an example of the pattern that we can play around with.

```
onload = () ⇒ document.documentElement.style.background = 'lime'
```

So here we have the simplest possible event-driven style. An onload event, that runs a function, that sets one style. Here it sets a lime green background on the **<html>** tag.

```
<style></style>

<script>
  onload = () => document.querySelector('style').textContent = `
    html {
      background: lime;
    }
  `
</script>
```

To improve this, we can use a **<style>** tag instead, and return an entire stylesheet.

```
<style></style>

<script>
  window.addEventListener('load', loadStyles)

  function populate() {
    return document.querySelector('style').textContent = `
      html {
        background: lime;
      }
    `
  }
</script>
```

Let's make it a little more clear that we're dealing with event-driven virtual stylesheets here by separating the event listener from our stylesheet function.

Sometimes you'll be writing a stylesheet function and want to add it to a number of different listeners, or perhaps you'll want to add a listener for when a certain event happens on a number of different tags in the, document rather than global events.

```
window.addEventListener('load', loadStyles)

function populate() {
  let style = document.querySelector('#styles')
  if (!style) {
    style = document.createElement('style')
    style.id = 'styles'
    document.head.appendChild(style)
  }
  return style.textContent = `
    html {
      background: ${Math.random() > .5 ? 'lime' : 'hotpink'};
    }
  `
}
```

The next step here is optional, but makes it a little sturdier. It's to remove the **<style>** tag from HTML entirely, and have JavaScript either look for the **<style>** tag, or create it if it doesn't exist yet. This way our code is more resilient.

```
window.addEventListener('load', loadStyles)

function populate() {
  let style = document.querySelector('#styles')
  if (!style) {
    style = document.createElement('style')
    style.id = 'styles'
    document.head.appendChild(style)
  }
  return style.textContent = stylesheet()
}

function stylesheet() {
  return `
    html {
      background: ${Math.random() > .5 ? 'lime' : 'hotpink'};
    }
  `
}
```

Here we've separated the code for creating and populating the stylesheet from the stylesheet function even further. This makes it easy for us to focus on writing our stylesheet function, and our stylesheet function can even live in its own file.

See how our CSS stylesheet now includes JavaScript code inside the styles? The output of this stylesheet might be different each time the function is reprocessed.

```
function stylesheet() {
  return `
    html {
      color: ${coinToss('black', 'white')};
      background: ${coinToss('lime', 'hotpink')};
    }
  `
}

function coinToss(a, b) {
  return Math.random() > .5 ? a : b
}
```

Lastly, rather than repeating ourselves all the time, we can write extensions for our stylesheets as JavaScript functions that return strings of CSS, and re-use these plugins from stylesheet to stylesheet, and project to project.

Most of this presentation will be focused on showing you a pattern for building these plugins.

```
function jsincss(
  stylesheet = () ⇒ '',
  selector = window,
  events = ['load', 'resize', 'input', 'click', 'reprocess']
) {

  function registerEvent(target, event, id, stylesheet) {
    return target.addEventListener(
      event,
      e ⇒ populateStylesheet(id, stylesheet)
    )
  }

  function populateStylesheet(id, stylesheet) {
```

Just to show you the JS-in-CSS abstraction I usually work with, here is how it starts - I feed in a stylesheet function, a something to target (it could be the window object itself, or a CSS selector for tags I can match in the document), as well as an array of events I want to listen to. What you see here are my default settings.

Next we see the function that populates the **<style>** tag, as well as the function that subscribes the stylesheet function to different events in the browser. This part looks similar to the code in the previous slides.

Lastly we have this bit down here. For each new stylesheet we are registering, we generate a random ID for it and run our helper functions.

```
function registerEvent(target, event, id, stylesheet) {
  return target.addEventListener(
    event,
    e ⇒ populateStylesheet(id, stylesheet)
  )
}

function populateStylesheet(id, stylesheet) {
  let tag = document.querySelector(`#jsincss-${id}`)
  if (!tag) {
    tag = document.createElement('style')
    tag.id = `jsincss-${id}`
    document.head.appendChild(tag)
  }
  const currentStyles = tag.textContent
  const generatedStyles = stylesheet()
  if (!currentStyles || (generatedStyles !== currentStyles)) {
    return tag.textContent = generatedStyles
  }
}
```

Just to show you the JS-in-CSS abstraction I usually work with, here is how it starts - I feed in a stylesheet function, a something to target (it could be the window object itself, or a CSS selector for tags I can match in the document), as well as an array of events I want to listen to. What you see here are my default settings.

Next we see the function that populates the **<style>** tag, as well as the function that subscribes the stylesheet function to different events in the browser. This part looks similar to the code in the previous slides.

Lastly we have this bit down here. For each new stylesheet we are registering, we generate a random ID for it and run our helper functions.

```
      const generatedStyles = stylesheet()
      if (!currentStyles || (generatedStyles !== currentStyles)) {
        return tag.textContent = generatedStyles
      }
    }

    let id = Date.now() + Math.floor(Math.random() * 100)

    if (selector === window) {
      return events.forEach(event =>
        registerEvent(window, event, id, stylesheet)
      )
    } else {
      return document.querySelectorAll(selector).forEach(tag =>
        events.forEach(event =>
          registerEvent(tag, event, id, stylesheet)
        )
      )
    }
  }
```

Just to show you the JS-in-CSS abstraction I usually work with, here is how it starts - I feed in a stylesheet function, a something to target (it could be the window object itself, or a CSS selector for tags I can match in the document), as well as an array of events I want to listen to. What you see here are my default settings.

Next we see the function that populates the **<style>** tag, as well as the function that subscribes the stylesheet function to different events in the browser. This part looks similar to the code in the previous slides.

Lastly we have this bit down here. For each new stylesheet we are registering, we generate a random ID for it and run our helper functions.

```
function jsincss(
  stylesheet = () => '',
  selector = window,
  events = ['load', 'resize', 'input', 'click', 'reprocess']
) {

  function registerEvent(target, event, id, stylesheet) {
    return target.addEventListener(
      event,
      e => populateStylesheet(id, stylesheet)
    )
  }

  function populateStylesheet(id, stylesheet) {
    let tag = document.querySelector(`#jsincss-${id}`)
    if (!tag) {
      tag = document.createElement('style')
      tag.id = `jsincss-${id}`
      document.head.appendChild(tag)
    }
    const currentStyles = tag.textContent
    const generatedStyles = stylesheet()
    if (!currentStyles || (generatedStyles !== currentStyles)) {
      return tag.textContent = generatedStyles
    }
  }

  let id = Date.now() + Math.floor(Math.random() * 100)

  if (selector === window) {
    return events.forEach(event =>
      registerEvent(window, event, id, stylesheet)
    )
  } else {
    return document.querySelectorAll(selector).forEach(tag =>
      events.forEach(event =>
        registerEvent(tag, event, id, stylesheet)
      )
    )
  }
}
```

Just to show you the JS-in-CSS abstraction I usually work with, here is how it starts - I feed in a stylesheet function, a something to target (it could be the window object itself, or a CSS selector for tags I can match in the document), as well as an array of events I want to listen to. What you see here are my default settings.

Next we see the function that populates the **<style>** tag, as well as the function that subscribes the stylesheet function to different events in the browser. This part looks similar to the code in the previous slides.

Lastly we have this bit down here. For each new stylesheet we are registering, we generate a random ID for it and run our helper functions.

(Explore jsincss)

```
<style></style>

<script>
  // Event-Driven
  window.addEventListener('load', populate)

  // Virtual Stylesheet
  function populate() {
    return document.querySelector('style').textContent = stylesheet()
  }

  // Function
  function stylesheet() {
    return `
      html::before {
        font-size: 10vw;
        content: '${innerWidth} x ${innerHeight}';
      }
    `
  }
</script>
```

This is more or less what we should have ended up with, this is our JS-in-CSS pattern applied to just one **<style>** tag. This is all ready for us to extend with JavaScript functions that act like plugins.

But what is the name of this talk? Was it "Writing JavaScript Functions That Do Cool Things"? No! It's how to write Caffeinated Style sheets. So let's bring it home…

Extending a Selector

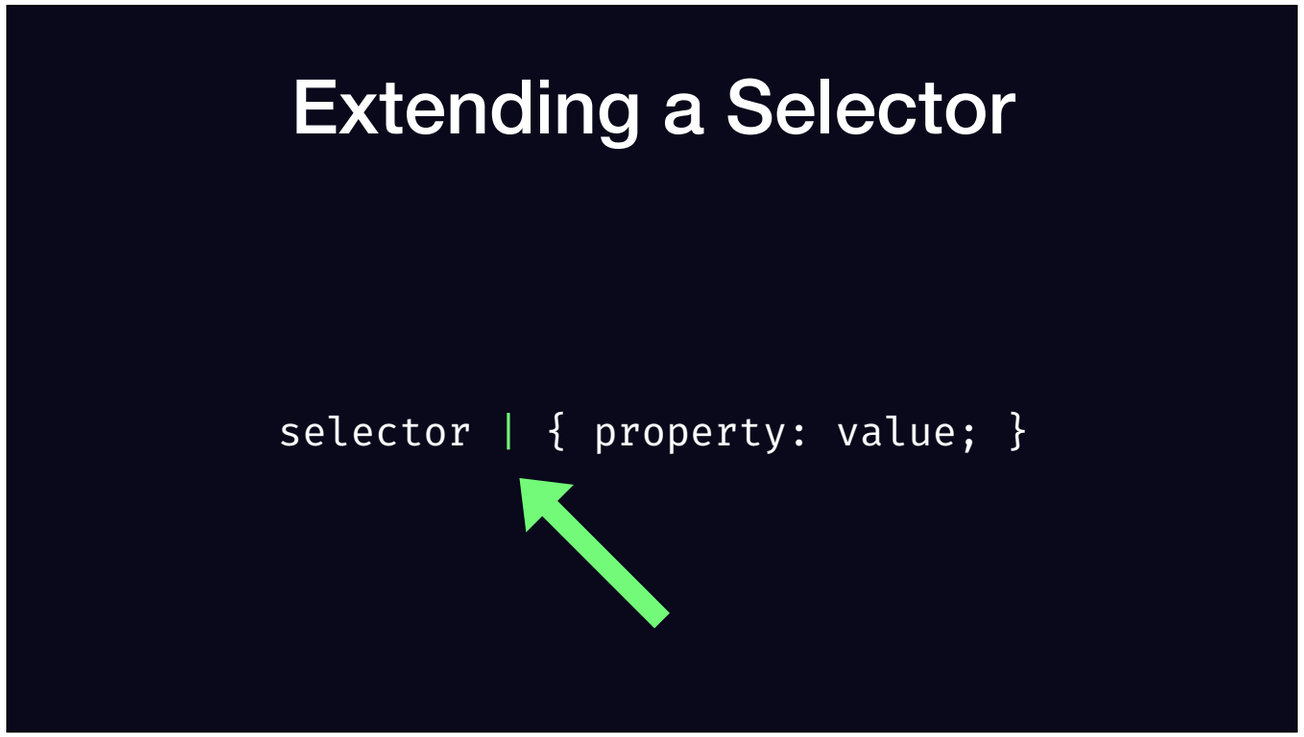
```
selector { property: value; }
```

How can we extend CSS itself? Well there are a few ways we can do this. CSS custom properties, also known as CSS variables, are a way that we can invent totally new properties, and set the value with information from CSS, HTML, or even JavaScript.

But what if you need to extend a selector? Can you do it? It turns out there's an easy way!

Here we have a CSS rule with a selector list that has a selector of 'selector', and a declaration list with one declaration of a property, with a property name of 'property', that has a value with a value of 'value'. Where do you think we can fit some extra information in here that will pass as valid CSS, but that can mark this rule as a JS-powered rule, and also encode whatever information we need to pass to JavaScript in order to process this rule?

Extending a Selector



If you guessed between the selector list and the declaration list, you're correct! This location effectively splits the CSS rule into two parts: everything before this point is the selector list, and everything that comes after is the declaration list.

So what is it we can add here?

Extending a Selector

```
selector [--custom] { property: value; }
```

An attribute selector that has an attribute that begins with a double dash. It's valid CSS, but we know it won't match anything in HTML, because HTML attributes don't start with a double dash.

Extending a Selector

```
selector [--custom="args"] { property: value; }
```

If we need to add any additional information, we can add it in the same format that a normal CSS attribute selector expects. Put it after an equals sign, quoted.

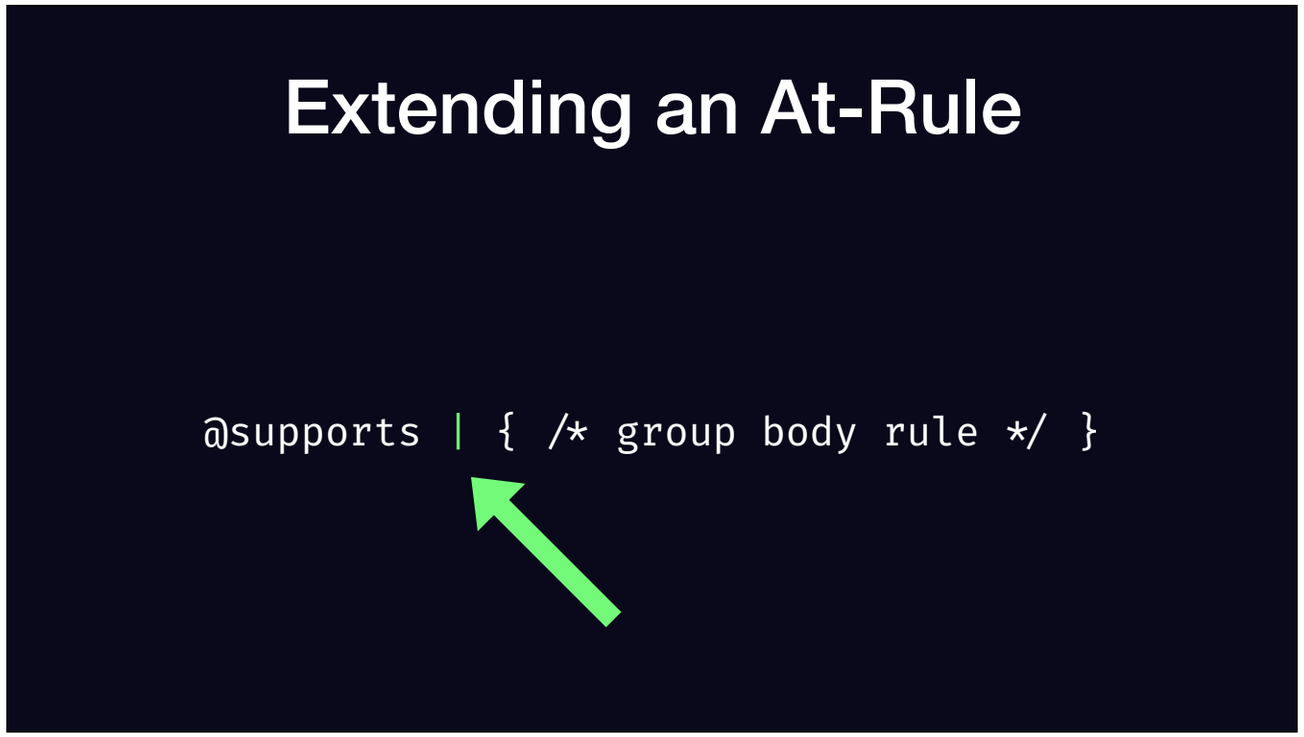
When I work with extended selectors like this I tend to treat this the same way you'd think of a JavaScript function's arguments: a list of comma-separated values

Extending an At-Rule

```
@supports { /* group body rule */ }
```

Similarly, if you want to extend an at-rule you can make use of **@supports**. If this is your **@supports** rule, and you have **@supports** written here, as well as your group body rule, where do you think you can extend this with JavaScript?

Extending an At-Rule



You guessed it, right between the **@supports** and the group body rule. Now there are a number of things that are officially supported here, but CSS syntax includes the flexibility for us to put just about anything we want in here as well, and the browser will still parse and retain that CSS, even if it doesn't know how to apply it.

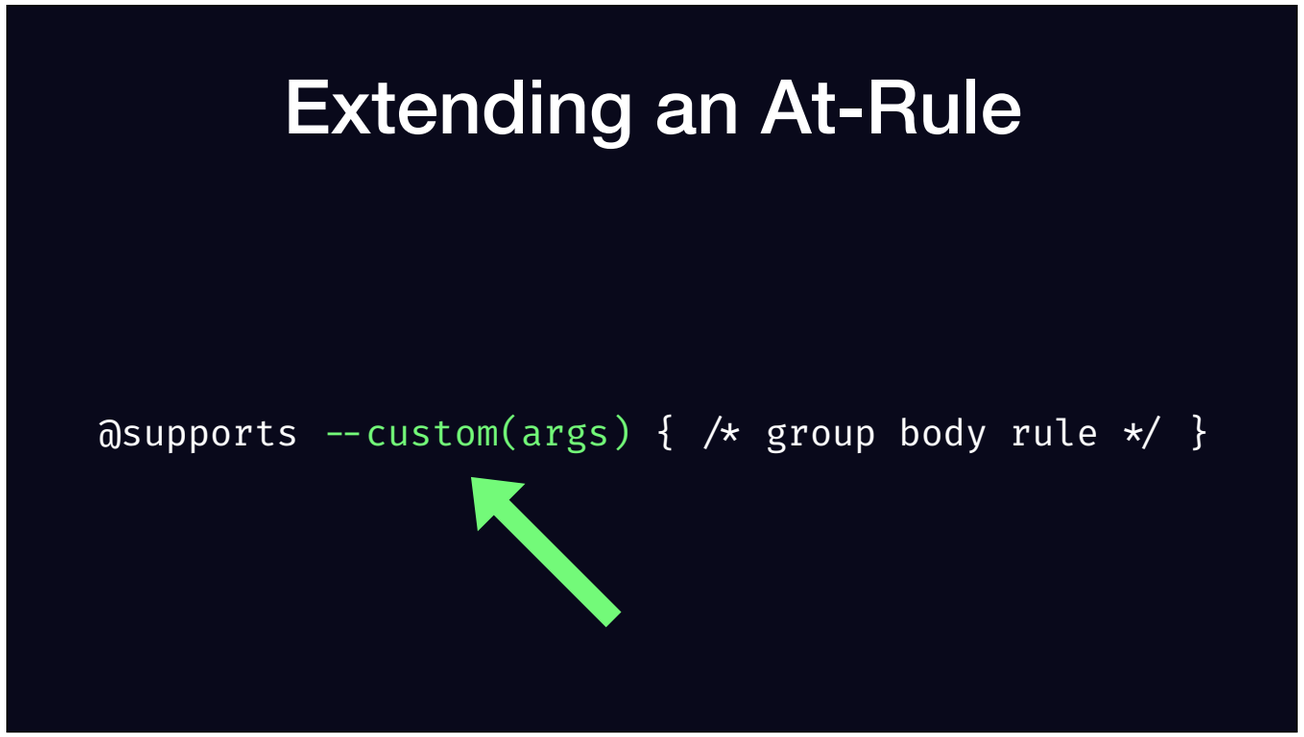
Extending an At-Rule

```
@supports --custom() { /* group body rule */ }
```

Just like before, we can start our custom extended at-rule with a double dash, then any name we want, followed by a pair of brackets.

This satisfies what CSS is looking for as far as **@supports** rules goes, and since we know no future features will begin with a double dash, this naming is safe for us to use.

Extending an At-Rule



Similar to before, if you want to add any additional information to this extended at-rule, you can add it inside the brackets.

This basically looks like a JavaScript function with arguments, the only difference is the name starts with a double dash, and you can picture the group body rule like the last argument being passed into the function.

If you can stay in double-dash land you'll have a good time!

Always use a
double-dash “--”

Finding JS-Powered Rules

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

Finding JS-Powered Rules

- Loop through the **document.styleSheets** object

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

Finding JS-Powered Rules

- Loop through the **document.styleSheets** object
- For each stylesheet, loop through it's **cssRules**

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

Finding JS-Powered Rules

- Loop through the **document.styleSheets** object
- For each stylesheet, loop through it's **cssRules**
- Look for **style** rules (type 1) or **@supports** rules (type 12)

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

Finding JS-Powered Rules

- Loop through the **document.styleSheets** object
- For each stylesheet, loop through it's **cssRules**
- Look for **style** rules (type 1) or **@supports** rules (type 12)
- Check **selectorText** or **conditionText** to see if it includes '**--**'

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

Finding JS-Powered Rules

- Loop through the **document.styleSheets** object
- For each stylesheet, loop through it's **cssRules**
- Look for **style** rules (type 1) or **@supports** rules (type 12)
- Check **selectorText** or **conditionText** to see if it includes '**--**'
- Parse selector (if a rule), arguments, and styles

Now that you know how to extend selectors and **@supports** rules, how do you get the information back out of CSS?

There's an object in the browser called **document.styleSheets** which contains a list of every stylesheet that is loaded in the browser. We can loop through each one of these stylesheets.

And for each stylesheet we can loop through its CSS rules.

And look for either regular style rules, which will show up with a type of 1, or **@supports** rules which have a type of 12.

Once we have found either of those, we can check either the selector text or the condition text to see if it includes a double dash.

And if it does, we can parse out the selector (if it's a rule), any arguments that might be there, and the styles.

```
Array.from(document.styleSheets).forEach(stylesheet ⇒
  Array.from(stylesheet.cssRules).forEach(rule ⇒ {
    if (rule.type === 1 && rule.selectorText.includes('--')) {
      console.log('found a js-powered style rule')
    } else if (rule.type === 12 && rule.conditionText.includes('--')) {
      console.log('found a js-powered @supports rule')
    }
  })
)
```

(Read code on screen)

```
body[--custom="1, 2, 3"] {
  background: lime;
}
```

So here's an example of a rule with an extended selector.

This is a selector list, that has one selector that targets the body tag, and here we have a custom selector for a plugin named 'custom', and we're feeding in the values one, two, and three as arguments. Inside the rule, there's a style.

```
if (rule.type === 1 && rule.selectorText.includes('--custom')) {

  const selector = rule.selectorText
    .split('[--custom')[0]
    .trim()

  const arguments = rule.selectorText.includes('--custom=')
    ? rule.selectorText.replace(/.*\[--custom=(.*)\]/, '$1').slice(1, -1)
    : ''

  const declarations = rule.cssText
    .split(rule.selectorText)[1]
    .trim()
    .slice(1, -1)
    .trim()

}
```

Where I had that first **console.log()** in the previous slide, this could be the code we would need there to extract the selector, arguments, and declarations from our example rule.

```
body

1, 2, 3

background: lime;
```

If we were to extract the information from our rule we'd end up with this.

```
custom('body', 1, 2, 3, `background: lime;`)
```

It would be easy for us to use this in JavaScript to call a function named custom, with the pieces of information that we extracted from CSS.

```
@supports --custom(1, 2, 3) {
  body {
    background: lime;
  }
}
```

Or for another example, here's an extended **@supports** rule. You can see it's passing in the values one, two, and three, and the group body rule contains one CSS rule.

```
if (rule.type === 12 && rule.conditionText.includes('--custom')) {

  const arguments = rule.conditionText
    .split('--custom')[1]
    .trim()
    .slice(1, -1)

  const stylesheet = rule.cssText
    .split(rule.conditionText)[1]
    .trim()
    .slice(1, -1)
    .trim()

}
```

This is the code we would need in the previous slide where the second **console.log()** was in order to parse out the arguments and stylesheet.

The reason I'm showing you this is not so you understand every line of it, but to see how simple the whole thing can be from scratch.

```
1, 2, 3

body { background: lime; }
```

Here's the information we extracted from our **@supports** rule

```
custom(1, 2, 3, `body {background: lime; }`)
```

And here's how we could run that through a custom plugin.

The “No More Tears” approach to CSS parsing,
Write only 100% valid CSS and let the browser parse it!

I call this the “No more tears” approach to CSS parsing.

That is, we always work with 100% valid CSS syntax, and let the browser do all the parsing for you!

Can be processed
server-side or client-slide

The best thing about this, is that as long as you have a CSS stylesheet, you can do this kind of parsing server side *or* client side. I use Chrome Puppeteer, a version of Chrome that is controlled from the command line, and I can have it read a CSS file as input and give me back 2 files: one with 100% clean CSS, and another with 100% clean JS that includes my JS-in-CSS pattern, plus any plugins I used, and any JS-powered styles I described in my input CSS file.

If you don't mind a build step you can parse these JS powered rules out in advance, it's pretty straightforward as well.

(Explore Qaffeine)

```
/* :parent */
.child[--parent] {
  border: 10px dashed red;
}

/* :has() */
ul[--has="'strong'"] {
  background: cyan;
}

/* @element {} */
@supports --element('input', {minCharacters: 5}) {
  [--self] {
    background: hotpink;
  }
}
```

We should have ended up with something like this: A Caffeinated Style Sheet.

Valid CSS that is extended for being read and run by JavaScript.

The Tag Reduction Pattern

After writing a couple dozen JS-in-CSS plugins I noticed a few patterns emerge. Today I'm going to share the most pattern I found. You can make endless variations off this one pattern. This is the Tag Reduction pattern.

Here are the requirements: first you need a selector to search for tags in the document.

The second thing you need is a test that JavaScript can run, that can be used to filter the tags we find.

Next we need to know which tag we should apply the styles to, it's *not* always the same as the tag we are testing.

And lastly we need either the rule or stylesheet that we wish to apply.

The Tag Reduction Pattern

- A selector to search for tags in the document

After writing a couple dozen JS-in-CSS plugins I noticed a few patterns emerge. Today I'm going to share the most pattern I found. You can make endless variations off this one pattern. This is the Tag Reduction pattern.

Here are the requirements: first you need a selector to search for tags in the document.

The second thing you need is a test that JavaScript can run, that can be used to filter the tags we find.

Next we need to know which tag we should apply the styles to, it's *not* always the same as the tag we are testing.

And lastly we need either the rule or stylesheet that we wish to apply.

The Tag Reduction Pattern

- A selector to search for tags in the document
- A test to filter matching tags

After writing a couple dozen JS-in-CSS plugins I noticed a few patterns emerge. Today I'm going to share the most pattern I found. You can make endless variations off this one pattern. This is the Tag Reduction pattern.

Here are the requirements: first you need a selector to search for tags in the document.

The second thing you need is a test that JavaScript can run, that can be used to filter the tags we find.

Next we need to know which tag we should apply the styles to, it's *not* always the same as the tag we are testing.

And lastly we need either the rule or stylesheet that we wish to apply.

The Tag Reduction Pattern

- A selector to search for tags in the document
- A test to filter matching tags
- To know which tag we want to apply styles to

After writing a couple dozen JS-in-CSS plugins I noticed a few patterns emerge. Today I'm going to share the most pattern I found. You can make endless variations off this one pattern. This is the Tag Reduction pattern.

Here are the requirements: first you need a selector to search for tags in the document.

The second thing you need is a test that JavaScript can run, that can be used to filter the tags we find.

Next we need to know which tag we should apply the styles to, it's *not* always the same as the tag we are testing.

And lastly we need either the rule or stylesheet that we wish to apply.

The Tag Reduction Pattern

- A selector to search for tags in the document
- A test to filter matching tags
- To know which tag we want to apply styles to
- A rule or stylesheet we want to apply

After writing a couple dozen JS-in-CSS plugins I noticed a few patterns emerge. Today I'm going to share the most pattern I found. You can make endless variations off this one pattern. This is the Tag Reduction pattern.

Here are the requirements: first you need a selector to search for tags in the document.

The second thing you need is a test that JavaScript can run, that can be used to filter the tags we find.

Next we need to know which tag we should apply the styles to, it's *not* always the same as the tag we are testing.

And lastly we need either the rule or stylesheet that we wish to apply.

The Tag Reduction Pattern

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector
2. Reduce the array to a string (our CSS stylesheet)

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector
2. Reduce the array to a string (our CSS stylesheet)
3. Create a unique identifier (from plugin name, selector, options)

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector
2. Reduce the array to a string (our CSS stylesheet)
3. Create a unique identifier (from plugin name, selector, options)
4. Test each matching tag

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector
2. Reduce the array to a string (our CSS stylesheet)
3. Create a unique identifier (from plugin name, selector, options)
4. Test each matching tag

   If tag passes: add unique identifier to tag, and add copy of CSS rule or stylesheet to output

To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.

The Tag Reduction Pattern

1. Get an array of all tags matching the selector

2. Reduce the array to a string (our CSS stylesheet)

3. Create a unique identifier (from plugin name, selector, options)

4. Test each matching tag

   If tag passes: add unique identifier to tag, and add copy of CSS rule or stylesheet to output

   If tag fails: remove any unique identifier that might exist

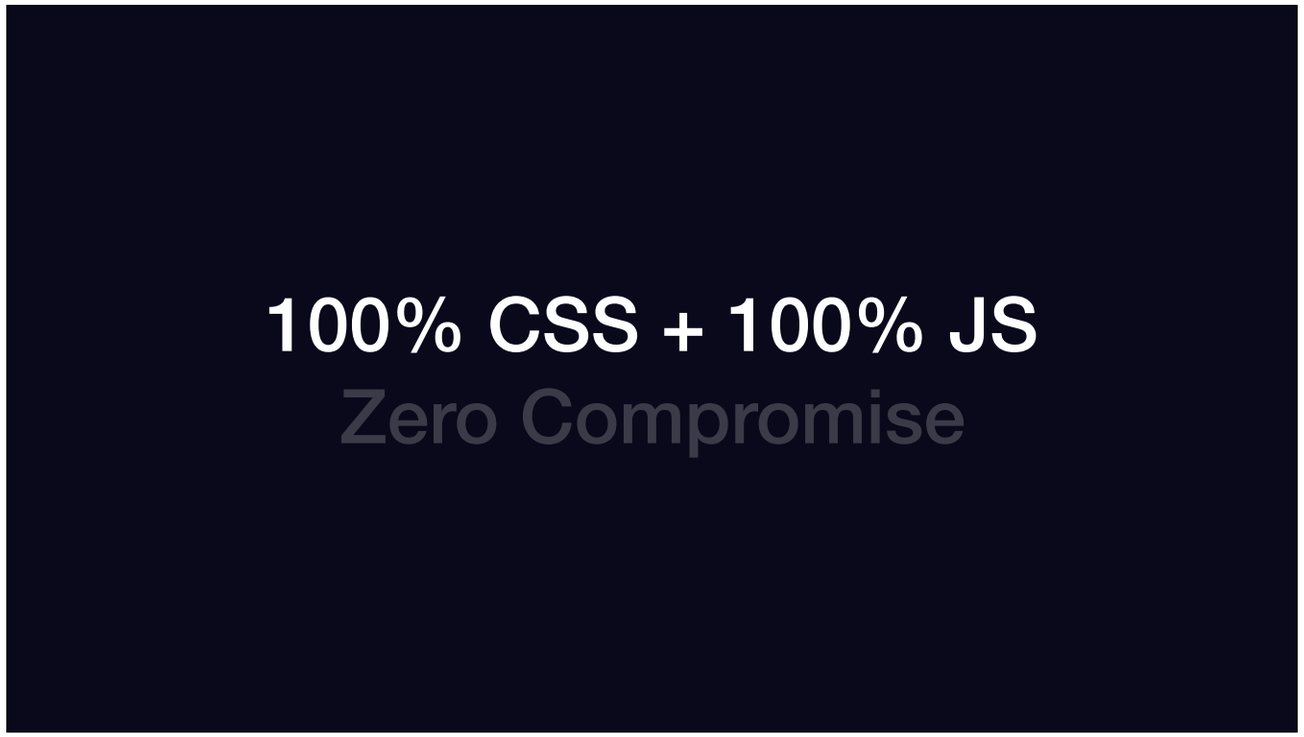
To implement this, we need to get an array of tags in the document matching the input selector.

Then we can reduce the array of tags (FYI that's where the name comes from ;)) and reduce that array of tags to a CSS stylesheet as a string of text.

We will need to create a unique identifier we can use to target the tags we want to apply styles to, and we will include the plugin name, selector, and any options that are present to make this identifier unique enough, but still useful for debugging.

Then we test each tag. And if the tag passes, we add our unique identifier to the tag, and add a copy of the CSS styles to the output

If the tag fails, we remove any unique identifier that might exist.



It's 100% of the flexibility of CSS, plus 100% of the power of JavaScript.

With zero compromise! We can marry both languages together without one limiting the other, and anything we know how to do in one of the two languages, can be leveraged when they are working together this way.

The great thing is that when any new CSS feature gets added, it will be immediately usable for you to use in any stylesheets written this way, and when any new JavaScript feature gets added you can use it straight away in writing your stylesheet functions and plugins as well.

Min-Width as a Selector

What we want in CSS:

```
.example:min-width(500px) { /* rule */ }
```

So let's look at a few examples of things we might want to add. I'm going to flip through these kind of fast so buckle up!

Here's something we might want, a min-width pseudo class selector that would apply a style when a tag matching that selector has a certain width.

Who wants something like this in CSS?

Unfortunately this is not in CSS, so what can we do?

Min-Width as a Selector

What we can test with JS:

```
el.offsetWidth ⩾ width
```

Well, here's what we can test in JavaScript: how an element's **offsetWidth** compares to a number

Min-Width as a Selector

What we can write in CSS:

```
.example[--min-width="500"] { /* rule */ }
```

Thinking back to our extended selectors, here's what we can express in CSS

Min-Width as a Selector

What we can run in JS:

```
minWidth('.example', 500, '/* rule */')
```

And after the browser parses that, we can easily transform that information into this.

```
function minWidth(selector, width, rule) {
  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {
      const attr = (selector + width).replace(/\W/g, '')
      if (tag.offsetWidth >= width) {
        tag.setAttribute(`data-minwidth-${attr}`, count)
        styles += `[data-minwidth-${attr}="${count}"] { ${rule} }\n`
     } else {
        tag.setAttribute(`data-minwidth-${attr}`, '')
      }
      return styles
    }, '')
}
```

Here's the function you'd need to make that feature work. It's a tag reduction written for a CSS rule. Let's walk through the code together.

Min-Width as an At-Rule

What we want in CSS:

```
@element .example and (min-width: 500px) {
  :self { background: lime; }
}
```

Have you ever wished you could write a rule like this? Here's something similar, but expressed as an at-rule so you could put an entire stylesheet in there to apply when the condition is true.

Min-Width as an At-Rule

What we can test with JS:

```
el.offsetWidth ⩾ width
```

What we can test in JavaScript is the same as before.

Min-Width as an At-Rule

What we can write in CSS:

```
@supports --minWidth('.example', 500) {
  [--self] { background: lime; }
}
```

But we can also express it as an **@supports** rule instead. Notice here I wrote the rule inside, to target a selector with a double dash 'self'? That's something I'll search and replace inside the stylesheet for the unique identifier we're creating for the tag.

Min-Width as an At-Rule

What we can run in JS:

```
minWidth('.example', 500, '/* stylesheet */')
```

Pretty much the exact same as before, except we're passing in a stylesheet rather than an individual CSS rule.

```
function minWidth(selector, width, stylesheet) {
  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {
      const attr = (selector + width).replace(/\W/g, '')
      if (tag.offsetWidth >= width) {
        tag.setAttribute(`data-minwidth-${attr}`, count)
        styles += stylesheet.replace(
          /\[--self\]/g,
          `[data-minwidth-${attr}="${count}"]`
        )
     } else {
        tag.setAttribute(`data-minwidth-${attr}`, '')
      }
      return styles
    }, '')
}
```

Here's min-width as a tag reduction, when written for a stylesheet. It's largely the same.

```
function minWidth(selector, width, rule) {
  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {
      const attr = (selector + width).replace(/\W/g, '')
      if (tag.offsetWidth >= width) {
        tag.setAttribute(`data-minwidth-${attr}`, count)
        styles += `[data-minwidth-${attr}="${count}"] { ${rule} }\n`
      } else {
        tag.setAttribute(`data-minwidth-${attr}`, '')
      }
      return styles
    }, '')
}
```

```
function minWidth(selector, width, stylesheet) {
  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {
      const attr = (selector + width).replace(/\W/g, '')
      if (tag.offsetWidth >= width) {
        tag.setAttribute(`data-minwidth-${attr}`, count)
        styles += stylesheet.replace(
          /\[--self\]/g,
          `[data-minwidth-${attr}="${count}"]`
        )
      } else {
        tag.setAttribute(`data-minwidth-${attr}`, '')
      }
      return styles
    }, '')
}
```

Let's compare the rule version of the pattern to the stylesheet version of the pattern. Now I don't expect you to be able to read the code here, but you can see a couple things: first these plugins aren't very long, there really isn't much code here. And secondly there are only two places where the code is different, all of the rest is verbatim.

```
A  1     function minWidth(selector, width, rule) {
   B  1  function minWidth(selector, width, stylesheet) {
2     2    return Array.from(document.querySelectorAll(selector))
3     3      .reduce((styles, tag, count) => {
4     4        const attr = (selector + width).replace(/\W/g, '')
5     5        if (tag.offsetWidth >= width) {
6     6          tag.setAttribute(`data-minwidth-${attr}`, count)
   B  7          styles += stylesheet.replace(
      8            /\[--self\]/g,
A  7             styles += `[data-minwidth-${attr}="${count}"] { ${rule} }\n`
   B  9            `[data-minwidth-${attr}="${count}"]`
   B 10           )
8     11       } else {
9     12         tag.setAttribute(`data-minwidth-${attr}`, '')
10    13       }
11    14       return styles
12    15     }, '')
13    16 }
```

Here we can zoom in on the two differences:

In one we're passing in a rule, and in the other we're passing in a stylesheet.

And here down below in the stylesheet version we replace the double dash self selector with our unique identifier, and in the rule version we output a new rule written for the unique identifier instead.

Parent Selector

What we want in CSS:

```
.example:parent { /* rule */ }
```

Have you ever wished you had a parent selector in CSS to target the tag directly containing another tag?

Parent Selector

What we can test with JS:

```
el.parentElement
```

JavaScript knows each element's parent element.

Parent Selector

What we can write in CSS:

```
.example[--parent] { /* rule */ }
```

So we should be able to express this in CSS with an extended selector using double dash parent.

Parent Selector

What we can run in JS:

```
parent('.example', '/* rule */')
```

As long as we have a tag reduction that takes a selector and a rule.

```
function parent(selector, rule) {

  return Array.from(document.querySelectorAll(selector))

    .filter(tag => tag.parentElement)

    .reduce((styles, tag, count) => {

      const attr = selector.replace(/\W/g, '')

      tag.parentElement.setAttribute(`data-parent-${attr}`, count)
      styles += `[data-parent-${attr}="${count}"] { ${rule} }\n`

      return styles

    }, '')

}
```

And applies the unique identifier to the tag's parent element, so we can target it with styles.

There's your parent selector, defined in JavaScript.

Do you see the power here?

Do you see the *POWER* here?

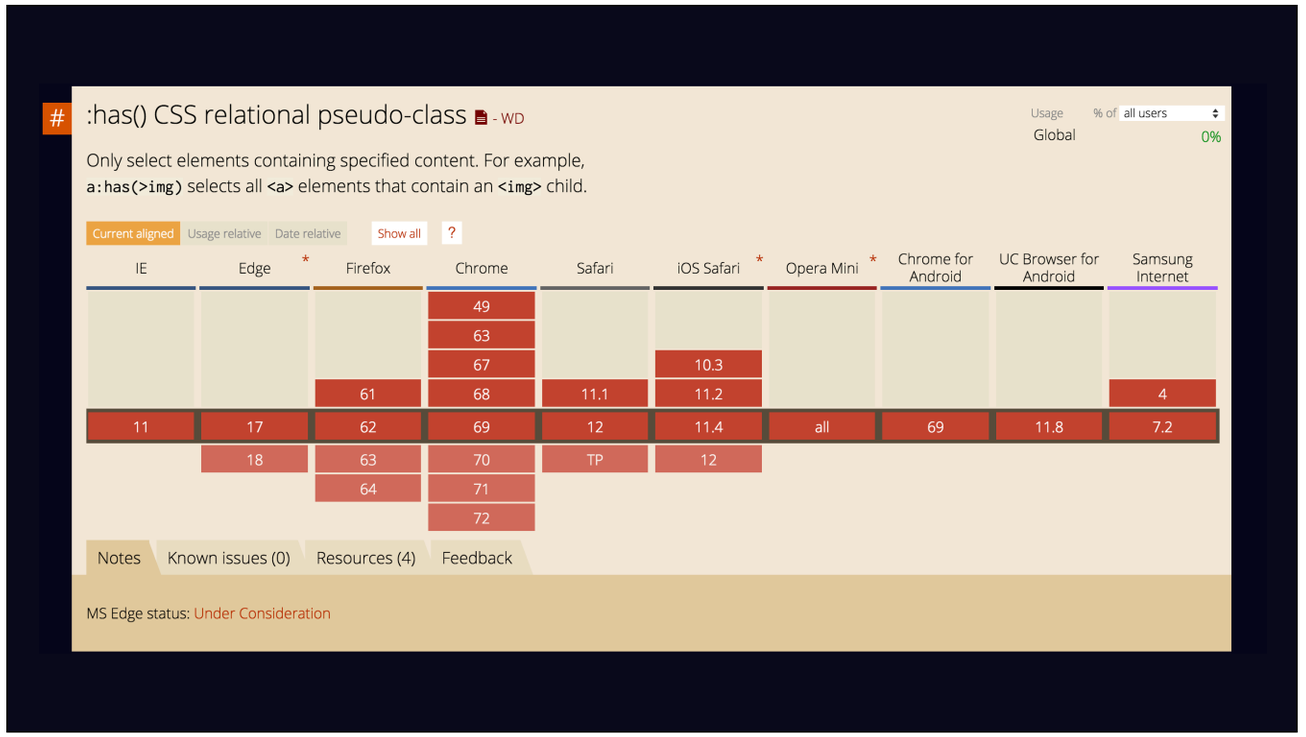
:has()

What we want in CSS:

```
.example:has(.demo) { /* rule */ }
```

Have you ever wanted this in CSS?

That was a trick question, it's actually been in CSS for *years*.



The problem is that it currently has zero browser support, and the outlook doesn't seem very hopeful. Who knows when we'll ever be able to use this selector in browsers.

:has()

What we can test with JS:

```
el.querySelector(child)
```

So what can we do in the meantime? It turns out querying to see if an element contains at least one tag matching another CSS selector is pretty simple with JavaScript, it's just **querySelector()**

:has()

What we can write in CSS:

```
.example[--has="'.demo'"] { /* rule */ }
```

What can we write in CSS? Not so different from what we want.

:has()

What we can run in JS:

```
has('.example', '.demo', '/* rule */')
```

And if we can make a tag reduction passing in the two selectors and the rule.

```
function has(selector, child, rule) {

  return Array.from(document.querySelectorAll(selector))

    .filter(tag => tag.querySelector(child))

    .reduce((styles, tag, count) => {

      const attr = (selector + child).replace(/\W/g, '')

      tag.setAttribute(`data-has-${attr}`, count)
      styles += `[data-has-${attr}="${count}"] { ${rule} }\n`

      return styles

    }, '')

}
```

We should be able to target those tags fairly easily.

Previous Selector

What we want in CSS:

```
.example:previous { /* rule */ }
```

Have you ever wished you could target the tag that came directly *before* another tag?

Previous Selector

What we can test with JS:

```
el.previousElementSibling
```

JavaScript knows a tag's previous element sibling.

Previous Selector

What we can write in CSS:

```
.example[--previous] { /* rule */ }
```

We can simply it in CSS as double dash previous

Previous Selector

What we can run in JS:

```
previous('.example', '/* rule */')
```

And call a function like this. Are you getting the idea now :D

Contains String

What we want in CSS:

```
.example:contains-text('demo') { /* rule */ }
```

Have you ever wished you could target a tag by the text it contained?

I get asked if this is possible at least once a week by people learning CSS. I don't think it's ever going to happen in CSS.

Contains String

What we can test with JS:

```
el.textContent.includes('string')
```

But JavaScript can test if an element's text content includes a string

Contains String

What we can write in CSS:

```
.example[--contains-text="'demo'"] { /* rule */ }
```

And we can express what we need in CSS

Contains String

What we can run in JS:

```
containsText('.example', 'demo', '/* rule */')
```

It should be pretty simple for us to write a tag reduction using this information.

If you wanted to search for a pattern in the text instead, testing a regular expression is just as easy!

Attribute Comparison

What we want in CSS:

```
.example[price > 50] { /* rule */ }
```

And one last example here, have you ever wished you could compare attribute values as numbers in CSS?

Attribute Comparison

What we can test with JS:

```
el.getAttribute(attr) ⩾ number
```

JavaScript can!

Attribute Comparison

What we can write in CSS:

```
.example[--attr-greater="'price', 50"] { /* rule */ }
```

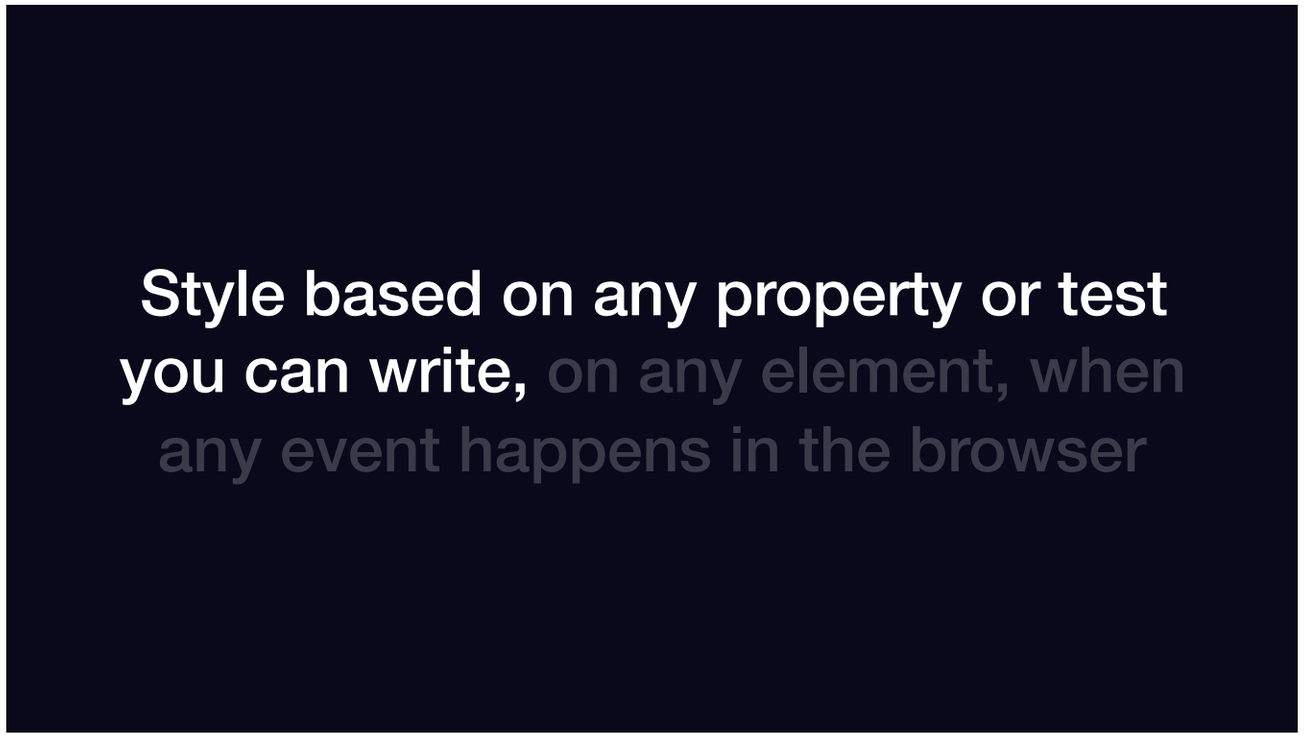
Which means we can in CSS too.

Attribute Comparison

What we can run in JS:

```
attrGreater('.example', 'price', 50, '/* rule */')
```

As long as we have a plugin like this.



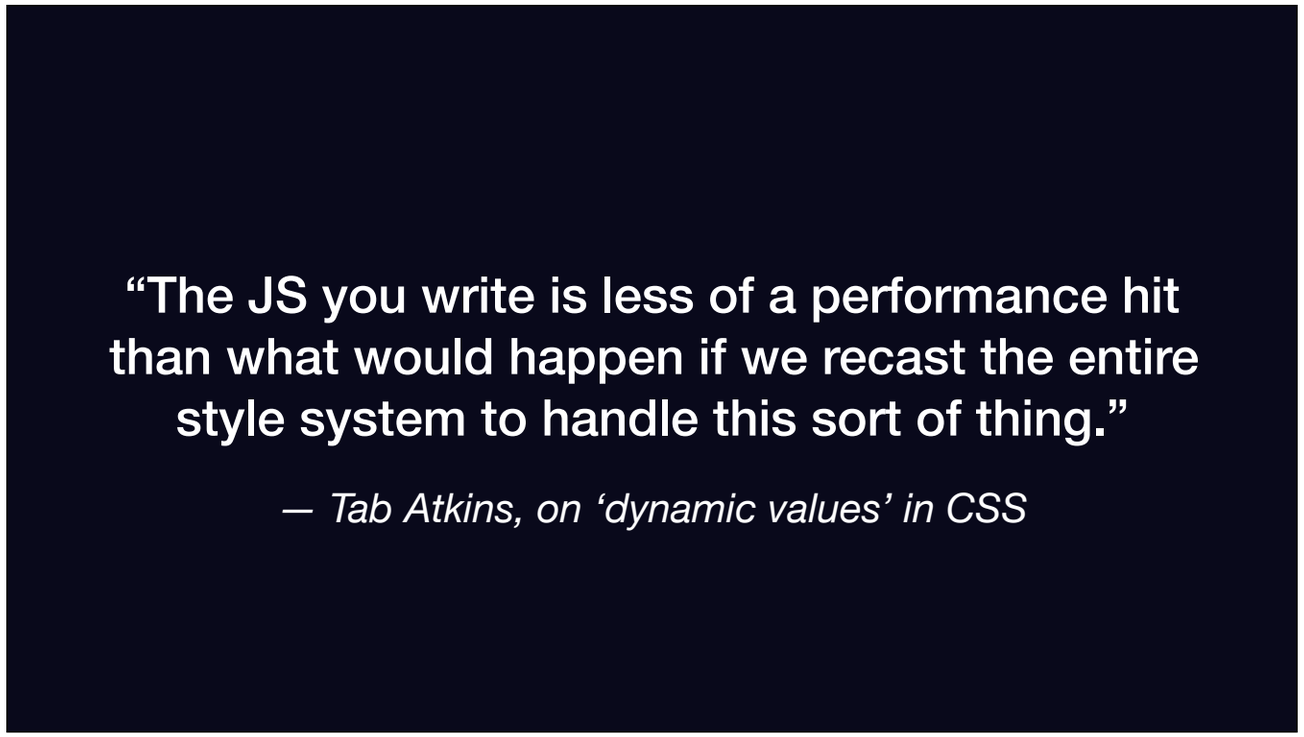
So to recap, we can style based on *any* property or test you can write…

Style based on any property or test you can write, on any element, when any event happens in the browser

…On *any* element…

Style based on any property or test you can write, on any element, when any event happens in the browser

…When *any* event happens in the browser.



Recently somebody asked the CSS Working Group if CSS would ever support dynamic values, like using the rendered size of one element as a measurement for sizing a different element. Here's what one of my CSS heroes, Tab Atkins, said:

“The JavaScript you write is *less of a performance hit* than what would happen if we recast the entire style system to handle this sort of thing.”

And the reality is that for a lot of these features, they will forever fall outside of the limitations of what CSS is designed to be.

(Explore Tag Reduction Pattern)

```
// Tag Reduction
function custom(selector, option, rule) {
  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {
      const attr = (selector + option).replace(/\W/g, '')
      if (option(tag)) {
        tag.setAttribute(`data-custom-${attr}`, count)
        styles += `[data-custom-${attr}="${count}"] { ${rule} }\n`
      } else {
        tag.setAttribute(`data-custom-${attr}`, '')
      }
      return styles
    }, '')
}
```

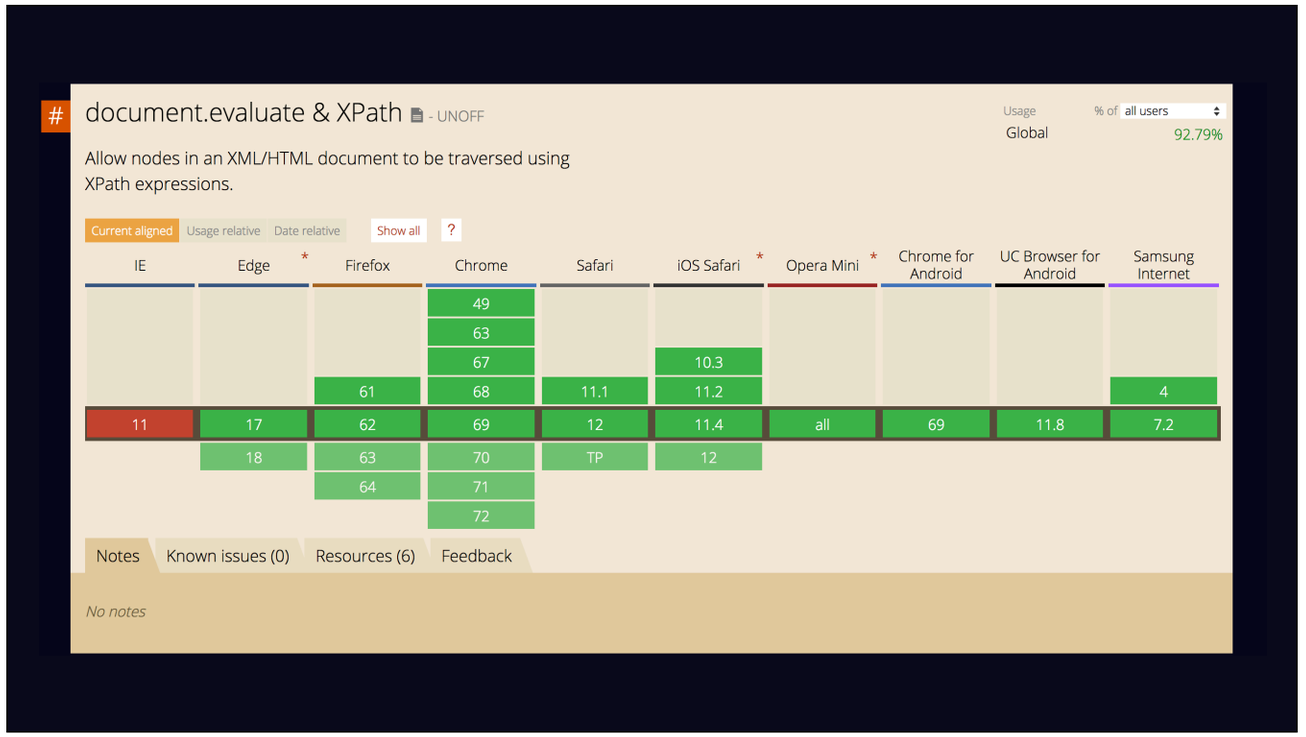
We should have ended up with something like this, a pretty standard tag reduction pattern.

Now I know some of you might ask: What about XPath?

Can't it do some of these same things? Selecting the parents of elements, selecting previous siblings, or targeting tags based on the tags or text they contain.

Even comparing attribute values as numbers.

And it's true, a lot of the things I wish I could add to CSS, are features that XPath supports.



XPath even has pretty good browser support, so why not use it instead?

Well at this point, anybody in the audience paying *any* attention should know what's going to happen next. Can you guess?

```
function xpath(selector, rule) {
  const tags = []
  const result = document.evaluate(
    selector,
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null
  )
  for (let i=0; i < result.snapshotLength; i++) {
    tags.push(result.snapshotItem(i))
  }
  return tags.reduce((styles, tag, count) => {
      const attr = selector.replace(/\W/g, '')
      tag.setAttribute(`data-xpath-${attr}`, count)
      styles += `[data-xpath-${attr}="${count}"] { ${rule} }\n`
     return styles
    }, '')
}
```

Yes, we can write a simple JavaScript function that lets us use XPath (and all of its functionality) to apply CSS styles to tags in a document.

```
/* :has(li) */
\/\/\*[li] [--xpath] {
  background: lime;
}

/* li:parent */
\/\/li\/parent\:\:\* [--xpath] {
  background: hotpink;
}

/* li:contains-text('hello') */
\/\/\*\[contains\(text\(\)\,\'hello\'\)\] [--xpath] {
  background: purple;
}
```

Which means, as long as we escape any characters CSS doesn't like with a backslash, we can even write XPath selectors directly inside CSS files, and add our custom extended selector here with a double dash XPath so JavaScript can find a process it.

We can treat everything before the custom selector as our XPath selector, and everything after it as our CSS rule.

```
xpath(

  rule.selectorText
    .split('[--xpath]')[0]
    .replace(/\\/g, '')
    .trim(),

  rule.cssText.split(rule.selectorText)[1]
    .trim()
    .slice(1, -1)

)
```

Here's how easy it is for us to get the XPath selector and CSS rule, and run them through our XPath tag reduction

```
import jsincss from 'https://unpkg.com/jsincss/index.vanilla.js'
import xpath from 'https://unpkg.com/jsincss-xpath-selector/index.vanilla.js'

Array.from(document.styleSheets).forEach(stylesheet ⇒
  Array.from(stylesheet.cssRules).forEach(rule ⇒ {
    if (rule.type === 1 && rule.selectorText.includes('[--xpath]')) {
      jsincss(() ⇒ xpath(

        rule.selectorText
          .split('[--xpath]')[0]
          .replace(/\\/g, '')
          .trim(),

        rule.cssText.split(rule.selectorText)[1]
          .trim()
          .slice(1, -1)

      ))
    }
  })
)
```

A complete solution to read the previous slide with the XPath selectors in CSS might look like this.

First we import the jsincss pattern, then we pull in our xpath tag reduction. Next we can loop through the CCSS O M and process the rules find.

That's *all the code we need* to make this whole thing work!

XPath in CSS

So there you have it, you can do anything, even XPath in CSS! There's just no limit.

So now I want to shift gears a little and inspire your creativity. Media queries took a period of eleven years to specify from the time people first proposed the idea to the time they were approved and able to be used. That's a long, long time to wait for a feature.

Media Queries

What we wanted in CSS:

```
@media (min-width: 500px) { /* stylesheet */ }
```

This is what we wanted, and eventually got in CSS, **@media** with a condition, a test value, and a group body rule containing more CSS rules.

Let's try looking at media queries through the lens of the event-driven styling we've been doing today.

Media Queries

What we could test with JS:

```
window.innerWidth ⩾ number
```

Here's what we can test with JavaScript, how the browser viewport's width or height compares to a number. This isn't the only thing media queries do, but the *vast majority* of media queries out there focus on testing the browser's width.

Media Queries

What we can run in JS:

```
media({minWidth: 500}, '/* stylesheet */')
```

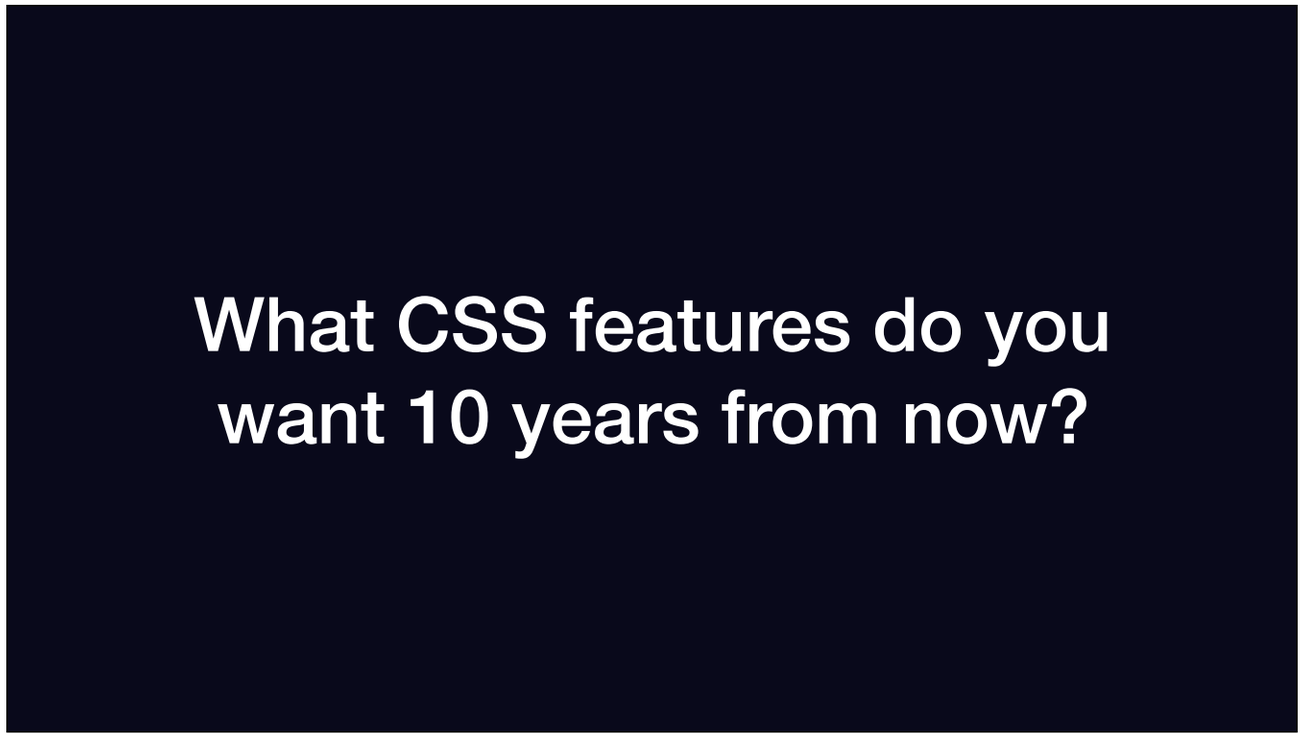
If we were teleported back 10 years in time, before media queries had landed, we would have been able to imitate them with a JavaScript function kind of like this. We supply the condition and test value, and the stylesheet.

```
function media(conditions, stylesheet) {

  return Object.keys(conditions).every(
    test ⇒ ({
      minWidth: number ⇒ number ⩽ innerWidth,
      maxWidth: number ⇒ number ⩾ innerWidth,
      minHeight: number ⇒ number ⩽ innerHeight,
      maxHeight: number ⇒ number ⩾ innerHeight,
      minAspectRatio: number ⇒ number ⩽ innerWidth / innerHeight,
      maxAspectRatio: number ⇒ number ⩾ innerWidth / innerHeight,
      orientation: string ⇒ {
        switch (string) {
          case 'portrait': return innerWidth < innerHeight
          case 'landscape': return innerWidth > innerHeight
        }
      }
    })[test](conditions[test])
  )
  ? stylesheet
  : ''

}
```

And here's a function that would support min-width and max-width, as well as min-height and max-height. And if you already know the *width* and *height*, things like aspect ratio are just a simple formula.

Having a function like this would let you build responsive websites, while thinking about responsive design the same way we do now, but you could have built websites this way ten or more years ago!

The best part is, it didn't take eleven *years* to write this function. Not even close! It was more like eleven *minutes*.



So my question to you is: what CSS features do you want 10 years from now? Just like how we could implement media query functionality with JavaScript, what functionality can we build with JavaScript *today* to be building sites, more like how we will in the future, rather than waiting 10 years to realize that we *could have been building this way* all along, in hindsight.

Element Queries?

Would you create element queries?

```
function element(selector, conditions, stylesheet) {

  const features = {
    minWidth: (el, number) => number <= el.offsetWidth,
    maxWidth: (el, number) => number >= el.offsetWidth,
    minHeight: (el, number) => number <= el.offsetHeight,
    maxHeight: (el, number) => number >= el.offsetHeight,
    minAspectRatio: (el, number) => number <= el.offsetWidth / el.offsetHeight,
    maxAspectRatio: (el, number) => number >= el.offsetWidth / el.offsetHeight,
    orientation: (el, string) => {
      switch (string) {
        case 'portrait': return el.offsetWidth < el.offsetHeight
        case 'square': return el.offsetWidth === el.offsetHeight
        case 'landscape': return el.offsetWidth > el.offsetHeight
      }
    },
    minChildren: (el, number) => number <= el.children.length,
    children: (el, number) => number === el.children.length,
    maxChildren: (el, number) => number >= el.children.length,
    minCharacters: (el, number) => number <= (
      (el.value && el.value.length) || el.textContent.length
    ),
    characters: (el, number) => number === (
```

Here's an element query function, the first part of it looks very similar to our media function, except that here we're passing in a selector for tags to target.

Inside the function we have some very similar tests, instead of looking at the window's **innerWidth**, we can look at the tag's **offsetWidth**, etc.

Since we're testing a tag, there are actually a few more conditions that make sense on tags, that don't make sense for a media query, and so we can add some of those extra conditions too.

Then, the bottom part of this function is just our basic tag reduction pattern. See how easy this can be?

```
    maxAspectRatio: (el, number) => number >= el.offsetWidth / el.offsetHeight,
    orientation: (el, string) => {
      switch (string) {
        case 'portrait': return el.offsetWidth < el.offsetHeight
        case 'square': return el.offsetWidth === el.offsetHeight
        case 'landscape': return el.offsetWidth > el.offsetHeight
      }
    },
    minChildren: (el, number) => number <= el.children.length,
    children: (el, number) => number === el.children.length,
    maxChildren: (el, number) => number >= el.children.length,
    minCharacters: (el, number) => number <= (
      (el.value && el.value.length) || el.textContent.length
    ),
    characters: (el, number) => number === (
      (el.value && el.value.length) || el.textContent.length
    ),
    maxCharacters: (el, number) => number >= (
      (el.value && el.value.length) || el.textContent.length
    ),
    minScrollX: (el, number) => number <= el.scrollLeft,
    maxScrollX: (el, number) => number >= el.scrollLeft,
    minScrollY: (el, number) => number <= el.scrollTop,
    maxScrollY: (el, number) => number >= el.scrollTop
  }
```

Here's an element query function, the first part of it looks very similar to our media function, except that here we're passing in a selector for tags to target.

Inside the function we have some very similar tests, instead of looking at the window's **innerWidth**, we can look at the tag's **offsetWidth**, etc.

Since we're testing a tag, there are actually a few more conditions that make sense on tags, that don't make sense for a media query, and so we can add some of those extra conditions too.

Then, the bottom part of this function is just our basic tag reduction pattern. See how easy this can be?

```
    }

    return Array.from(document.querySelectorAll(selector))
      .reduce((styles, tag, count) => {

        const attr = (
          selector + Object.keys(conditions) + Object.values(conditions)
        ).replace(/\W/g, '')

        if (Object.entries(conditions).every(test =>
          features[test[0]](tag, test[1])
        )) {
          tag.setAttribute(`data-element-${attr}`, count)
          styles += stylesheet.replace(
            /\[--self\]/g,
            `[data-element-${attr}="${count}"]`
          )
        } else {
          tag.setAttribute(`data-element-${attr}`, '')
        }
        return styles
      }, '')
  }
```

Here's an element query function, the first part of it looks very similar to our media function, except that here we're passing in a selector for tags to target.

Inside the function we have some very similar tests, instead of looking at the window's **innerWidth**, we can look at the tag's **offsetWidth**, etc.

Since we're testing a tag, there are actually a few more conditions that make sense on tags, that don't make sense for a media query, and so we can add some of those extra conditions too.

Then, the bottom part of this function is just our basic tag reduction pattern. See how easy this can be?

```
function element(selector, conditions, stylesheet) {

  const features = {
    minWidth: (el, number) => number ≤ el.offsetWidth,
    maxWidth: (el, number) => number ≥ el.offsetWidth,
    minHeight: (el, number) => number ≤ el.offsetHeight,
    maxHeight: (el, number) => number ≥ el.offsetHeight,
    minAspectRatio: (el, number) => number ≤ el.offsetWidth / el.offsetHeight,
    maxAspectRatio: (el, number) => number ≥ el.offsetWidth / el.offsetHeight,
    orientation: (el, string) => {
      switch (string) {
        case 'portrait': return el.offsetWidth < el.offsetHeight
        case 'square': return el.offsetWidth === el.offsetHeight
        case 'landscape': return el.offsetWidth > el.offsetHeight
      }
    },
    minChildren: (el, number) => number ≤ el.children.length,
    children: (el, number) => number === el.children.length,
    maxChildren: (el, number) => number ≥ el.children.length,
    minCharacters: (el, number) => number ≤ (
      (el.value && el.value.length) || el.textContent.length
    ),
    characters: (el, number) => number === (
      (el.value && el.value.length) || el.textContent.length
    ),
    maxCharacters: (el, number) => number ≥ (
      (el.value && el.value.length) || el.textContent.length
    ),
    minScrollX: (el, number) => number ≤ el.scrollLeft,
    maxScrollX: (el, number) => number ≥ el.scrollLeft,
    minScrollY: (el, number) => number ≤ el.scrollTop,
    maxScrollY: (el, number) => number ≥ el.scrollTop
  }

  return Array.from(document.querySelectorAll(selector))
    .reduce((styles, tag, count) => {

      const attr = (
        selector + Object.keys(conditions) + Object.values(conditions)
      ).replace(/\W/g, '')

      if (Object.entries(conditions).every(test =>
        features[test[0]](tag, test[1])
      )) {
        tag.setAttribute(`data-element-${attr}`, count)
        styles += stylesheet.replace(
          /\[--self\]/g,
          `[data-element-${attr}="${count}"]`
        )
      } else {
        tag.setAttribute(`data-element-${attr}`, '')
      }
      return styles
    }, '')
}
```

Here's an element query function, the first part of it looks very similar to our media function, except that here we're passing in a selector for tags to target.

Inside the function we have some very similar tests, instead of looking at the window's **innerWidth**, we can look at the tag's **offsetWidth**, etc.

Since we're testing a tag, there are actually a few more conditions that make sense on tags, that don't make sense for a media query, and so we can add some of those extra conditions too.

Then, the bottom part of this function is just our basic tag reduction pattern. See how easy this can be?

Scoped Styles?

Would you use something like this to create scoped styles for your components?

New Selectors?

Or how about new selectors based on relationships between tags, or new ways to test attributes?

New Pseudo-Classes?

Would you invent new pseudo classes based on any property a tag might have, or any property that can be tested with JavaScript?

New Properties?

How about entirely new properties?

New At-Rules?

Would you create custom at-rules that conditionally apply an entire stylesheet at once?

New Units?

Or maybe define new units to use in your styling, like how about element-based units which could be similar to viewport units, but based on the size of an element. That would be a huge win for responsive typography, right?

Look up ‘jsincss’ on npm for some ideas

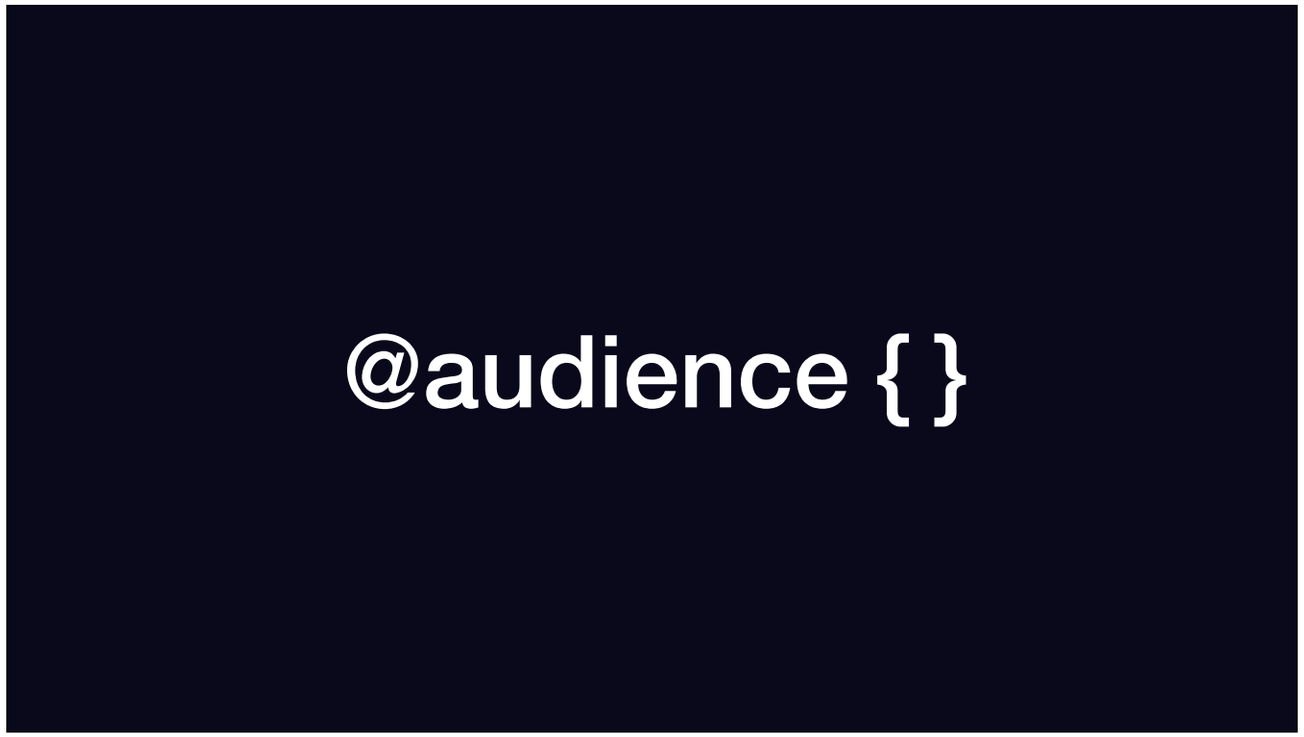
If you want to find examples of these patterns you can play around with and learn, look up JS-in-CSS on npm, you'll find over two dozen plugins there already, showing a wide variety of ideas to get you started.

Please feel free to publish your own ‘jsincss’ plugins!

And, one of the great things about these patterns is that both your JS-in-CSS pattern that processes your stylesheet functions, and the plugins that extend your stylesheet functions are largely interchangeable. So please if you are publishing code that uses these patterns, feel free to include the term JS-in-CSS so we can all find your work.

Thank you so much for coming to see this talk, I'm sure it challenged your idea of what CSS can be and gave you some feelings, and you probably saw some things that you didn't expect to see.

You can follow me on Twitter, @innovati to stay up to date with the latest experiments I'm working on!



Now I want to know if any audience queries exist. This is the question and answer period.