The web stack: Introduction

Over the last 25 years, the web has evolved in ways that you couldn't have imagined.

I'm using the term "web stack" or "web platform" to refer to the set of technologies that we use to build web content.

There are a lot of different technologies that we can use to build web content but, no matter what they are, they all boil down to the same three things:

- HTML
- CSS
- Javascript

In recent years, most people have come to associate the three elements with HTML 5. A set of technologies first introduced as a coherent group in January of 2012.

In these posts, I will work with the assumption that, while these technologies are interrelated they can also be pushed individually in ways perhaps we didn't think it was possible.

I'm also going to assume that, unless I specifically say otherwise, we're working with the most recent versions of browsers and the version of a specification where a feature is first introduced.

HTML

The table below shows the evolution of the HTML standard and related technologies.

Name / Version	Date Released	Releasing Organization
HTML Tags	1991	Tim Berners-Lee
HTML+	1993	IETF
HTML 2.0 Released as a series of RFC documents	1995	IETF
HTML 3.2	1997	W3C
HTML 4.0	1997	W3C
HTML 4.01	1999	W3C
XHTML 1.0	2000	W3C
XHTML 1.1	2001	W3C
HTML. **This is a living standard where new features are added regularly and browser makers work against this `standard` document**.	First released in 2008, updated regularly since	WHATWG
XHTML 2.0.	Work started in 2002. Abandoned in 2009	W3C
HTML 5	2014	W3C
HTML 5.1	2016	W3C
HTML 5.2	2017	W3C
HTML 5.3 superseeded by the HTML Living Standard	2021	W3C → WHATWG

To Understand the evolution of HTML as the language of the Web we also have to understand the history and the politics of the web and its users (both developers

and end-users).

It's also interesting to see how little the tags changed over the years and how long it took for new features to be released and for browsers to adopt them.

The different versions of HTML

HTML 1.0

The initial HTML specification was strongly influenced by SGML (Standard Generalized Markup Language), a language used by publishers to describe the structure of their documents.

Looking at the list of tags in HTML Tags we can see the limited types of documents that we could create with the original HTML specification.

The initial version of HTML was designed for document exchange. There were no forms or styles elements... this will be addressed in later versions of HTML in different ways.

This specification (although qualifying it as a specification may be too generous) was released by Tim Berners-Lee as part of the original web development effort.

HTML 2.0

HTML version 2.0 was developed in 1995 with basic intention of improving HTML version 1.0. The IETF developed the HTML 2.0 spefication and released it as a series of RFC documents:

- RFC 1866 (initial capability)
- RFC 1867 (form-based file upload)
- RFC 1942 (tables)
- RFC 1980 (client-side image maps)
- RFC 2070 (internationalization)

At this time where we have more than one (graphical) browser. Mosaic had been released in 1993, Netscape was released by some of the same egineers in 1994 and the initial version of Internet Explorer appeared in 1995.

Mosaic is the grandfather of Internet Explorer. NSCA licensed the code from Spyglass and used it to build the original version of Internet Explorer although it wasn't long before all the code from Mosaic was removed from Internet Explorer and replaced with Microsoft written code.

Because the different browsers had the goal of attracting people they introduced tags as they felt they were needed, regardless of whether there was a standard or not. Most of these tags were introduced as messages in the `www-talk` mailing list.

Take, for example, this message to www-talk from Mark Andressen proposing the `img` tag and the different alternatives that were presented on that discussion thread.

There was no W3C or WHATWG at this time. The specification was released as an set of IETF RFC Draft documents following the IETF development process. We'll see the differences when the specifications come under the stewardship of the W3C and later the WHATWG.

HTML 3.2

It is arround this time that we get the two browsers that were part of the original browser wars: Netscape and Internet Explorer. This will shape this era of language development.

HTML 3.2 included styles as attributes of HTML elements. It was common to see things like this in HTML 3.2 documents, with attributes in the body element indicating different colors of background, text and links:

```
<body
  bgcolor=white
  text=black
  link=red
  vlink=maroon
  alink=fuchsia>

<!-- body content -->

</body>
```

All the attributes are optional, have default values defined in the browser's built-in stylesheet and have equivalents in CSS that will be fully introduced in future versions.

Another area that is important to note is that both IE and Netscape introduced propriertary tags and features that would only work on those browsers.

Here are some examples of proprietary netscape tags added to the browser. None of these were part of any HTML specification.

Netscape proprietary	Description / Use
<blink>	Causes text to blink on and off
<ilayer>	Inline layer; allows you to offset content from its natural position on the page
<keygen>	Facilitates generation of key material and submission of the public key as part of an HTML form (for privacy and encryption)
<layer>	Creates layers so that elements can be placed on top of each other (useful with DHTML)
<multicol>	Produces a multicolumn format
<nolayer>	Alternative text for browsers that do not support <layer> and <ilayer>
<server>	Specifies a server-side JavaScript application
<spacer>	Holds a specified amount of empty space (used for alignment of elements on the page and to hold table cells open to specific widths)

Microsoft didn't stay behind and developed their own proprietary tags for Internet Explorer. These tags wouldn't work in other browsers and wouldn't be added to the HTML specification.

Internet Explorer proprietary	Description / Use
<marquee>	Places scrolling marquee text on the page
<basefont color=color face=font face>	Sets the color and/or font of the entire document when placed in the <head> or for subsequent text when placed in the flow of the body text
<bgsound>	Inserts an audio file that plays in the background

Internet Explorer proprietary	Description / Use
<body bgproperties=value>	Determines whether background image scrolls with the background
<body leftmargin=n rightmargin=n>	Sets the margin between the browser window and the contents of the page
<caption valign=position>	Sets vertical alignment of table caption
<comment>	Inserts a comment in the HTML source that does not display in the browser (same as <! – and – >)
<form target=name>	Specifies a target window or frame for the output of a form
<frameset framespacing=n	Sets the amount of space between frames
<img dynsrc=url controls loop=n start=action>	Uses the image tag to place video or audio clips
<table bordercolor=color bordercolordark=color bordercolorlight=color>	Sets colors for 3-D table borders in the <table>, <td>, <th>, and <tr> tags
<table frame=value>	Controls the display of the outer borders of a table in the <table> tag

So most of the work of the W3C, created in 1994, was to shepard the new version of HTML forward while navigating the conflicting defacto tags introduced by Netscape and Microsoft.

HTML 4.0 and 4.01

HTML 4.0 is where the original HTML reached its peek in terms of expresiveness and power. Some of these changes are

- Introduction of new elements
- Separation of structure and presentation
- Accessibility
- Internationalization
- Stylesheets
- Client-side Scripting

HTML 4 introduced many elements that are still in use today like `iframe`, `fieldset`, `span`, `thead`, `tbody`, and `tfoot` among others.

It also began to separate presentation from style and layout by deprecating the style attributes for HTML elements and introducing better hooks to CSS.

Hooks for stylesheets, like the `class`, and `id` attributes provide hooks to apply CSS styles to elements.

- The ID will apply styles to the single element that matches the ID.

```
#my-id {
  color: red;
}
```

- Classes are used to apply styles to multiple elements, usually elements that share the same function in a document.

```
.narrow-paragraph {
  width: 50%;
}
```

The `style` tag was introduced to provide a way to incorporate CSS stylesheets in the HTML document.

```
<style>
p {
  color: red;
}

h1, h2, h3, h4, h5, h6 {
  color: blue;
}
</style>
```

There is also a way to load stylesheets from remote sources but, somewhat surprisingly, this is not done with a style tag and the source attribute like we do with the `<script>` tag. It uses the `<link>` tag instead:

```
<link   rel="stylesheet"
        href="path/to/my/styles.css">
```

Scripting also got a boost in this version of HTML. You can specify the type and language of the script you're using and can load script from remote servers using the `src` attribute.

XHTML 1.0, 1.1 and 2.0

We'll take a detour into XML-based languages with the XHTML family of specifications since they directly led to the creation of the WHATWG and HTML as it is today.

XHTML is basically HTML 4.0 converted to an XML-based language. That's where some of what you see in older web pages comes from:

Ideas like all tags being closed, even if they don't have a closing tag (like the `<img>` tag in the example below).

```
<p>This is an example paragraph.</p>

<img src="path/to/image.webp"/>
```

We also get concepts like all HTML documents being also well-formed XML documents, following these minimal rules:

- Content be defined
- Content be delimited with a beginning and end tag
- Content be properly nested (parents within roots, children within parents)

This is the exact opposite of tag soup markup where authors have taken advantage of browsers' unwillingness to break the web by refusing to parse markup that is not well-formed.

There were two major numbered versions of XHTML and several profiles adopting the core specification for specific purposes.

eXtensible HyperText Markup Language 1.0(XHTML 1.0) is part of the family of XML markup languages. It mirrors or extends versions (strict, transitional and

frameset) of HTML 4.0.

XHTML 1.1 extends XHTML 1.0 by providing a modular framework. Rather than have one standard for the entire family, XHTML is now split into modules for specific sections of HTML as listed below along with the elements that belong to each module.

Structure Module : body, head, html, title

Text Module : abbr, acronym, address, blockquote, br, cite, code, dfn, div, em, h1, h2, h3, h4, h5, h6, kbd, p, pre, q, samp, span, strong, var

Hypertext Module : a

List Module : dl, dt, dd, ol, ul, li

Object Module : object, param

Presentation Module : b, big, hr, i, small, sub, sup, tt

Edit Module : del, ins

Bidirectional Text Module : bdo

Forms Module : button, fieldset, form, input, label, legend, select, optgroup, option, textarea

Table Module : caption, col, colgroup, table, tbody, td, tfoot, th, thead, tr

Image Module : img

Client-side Image Map Module : area, map

Server-side Image Map Module : Attribute ismap on img

Intrinsic Events Module : Events attributes

Metainformation Module : meta

Scripting Module : noscript, script

Stylesheet Module : style element

Style Attribute Module (*Deprecated*) : style attribute

Link Module : link

Base Module : base

Ruby Annotation Module : ruby, rbc, rtc, rb, rt, rp

XHTML 2.0 is an abandoned version of XHTML that led to the end of this family of specifications. It had a lot of issues with the way it worked:

No browser implemented the technology and most browsers moved to HTML 5.

There was no backwards compatibility with HTML 4.0 and XHTML 1.0 and 1.1. This would lead to all sorts of compatibility issues.

Work on XHTML 2 was dropped and the charter for the W3C working group in charge of its development was not renewed.

The biggest winner from all these events was HTML 5.

HTML5

HTML 5 is the latest version of HTML but more than that, it is also a collection of Javascript APIs and related technologies used to create web applications.

HTML 5 was not developed by the W3C but by the Web Hypertext Application Technology Working Group (WHATWG). A group initially formed by Apple, Mozilla and Opera that sought to create a better standard for applications on the web than what the W3C offered with XHTML 2.

In 2004, The W3C organized a W3C Workshop on Web Applications and Compound Documents where Mozilla and Opera presented a paper that ultimately was rejected. This is the direct antecedent to the creation of HTML 5 and the WHATWG. For a more detailed explanation of the WHATWG and the HTML living standard, see History in the HTML living standard.

For the rest of this section we'll refer to the HTML portion of the HTML 5 family of specifications.

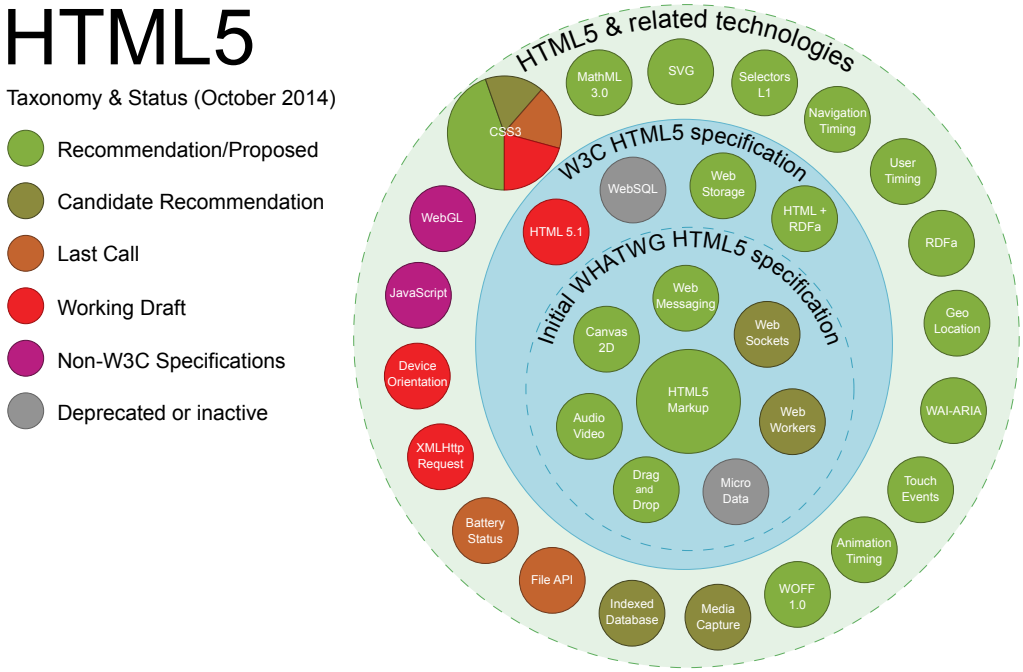

Figure 1: HTML5 APIs and related technologies taxonomy and status (October 2014).
Credit: Mercury999 via Wikimedia Commons

The HTML part of the HTML 5 specification, later renamed to the HTML Living Standard is different than previous families of HTML specifications in that it's evergreen, it's continually updated and browsers adopt this specification for ongoing development.

It's interesting that, while most tags introduced in HTML 5 work when creating web sites, their primary usage was for applications since this was the purpose of the WHATWG.

It introduced serveral new elements to the HTML vocabulary. Some of them are structural like the first table below:

Tags (Elements)	Description
<article>	Represents an independent piece of content like a blog entry or newspaper article
<section>	Represents a generic document or application section
<aside>	Represents a piece of content that is only slightly related to the rest of the page
<header>	Represents a group of introductory or navigational aids
<hgroup>	Represents the header of a section

Tags (Elements)	Description
<footer>	Represents a footer for a section
<nav>	Represents a section of the document intended for navigation

Other tags provide mulitmedia features without requiring plugins:

Tags (Elements)	Description
<audio>	Defines an audio file
<video>	Defines a video file
<canvas>	This is used for rendering dynamic bitmap graphics on the fly, such as graphs or games
<figure>	Represents a piece of self-contained flow content, typically referenced as a single unit from the main flow of the document

Other elements are presentational and deal with semantics for visual elements

Tags (Elements)	Description
<command>	Represents a command the user can invoke
<datalist>	Together with the a new list attribute for input can be used to make comboboxes
<details>	Represents additional information or controls which the user can obtain on demand
<keygen>	Represents control for key pair generation
<mark>	Represents a run of text in one document marked or highlighted for reference purposes, due to its relevance in another context
<meter>	Represents a measurement, such as disk usage
<output>	Represents some type of output, such as from a calculation done through scripting
<progress>	Represents a completion of a task, such as downloading or when performing a series of expensive operations

Tags (Elements)	Description
<ruby>	Together with Colorful typography on the web: get ready for multicolor fonts and **<rp>** allow for marking up ruby annotations used in Japanese and other East Asian languages
<time>	Represents a date and/or time
<wbr>	Represents a line break opportunity

The `<input>` tag got a lot of love with HTML 5. It gained the following new values:

attribute	Description
color	Color selector, which could be represented by a wheel or swatch picker
date	Selector for calendar date
datetime-local	Date and time display, with no setting or indication for time zones
datetime	Full date and time display, including a time zone
email	Input type should be an email
month	Selector for a month within a given year
number	A field containing a numeric value only
range	Numeric selector within a range of values, typically visualized as a slider
search	Term to supply to a search engine. For example, the search bar atop a browser
tel	Input type should be telephone number
time	Time indicator and selector, with no time zone information
url	Input type should be URL type
week	Selector for a week within a given year

While these tags and attributes were added to the HTML specification at the time of HTML 5 release, they may not have been implemented in any or all browsers.

The initial publication of HTML 5 (later the HTML living standard) wasn't the end. It has been continually updated and adopted by browsers.

Some of these newer additions include:

- The <picture> element and the srcset attribute for responsive images.
- Expanding the <source> to use in the <picture> element
- The <template> and <slot> elements to handle templates for custom elements

The HTML specification also includes a section on Microdata as a way to annotate HTML with structured data using attributes to the HTML we are annotating. If there was any Microdata implementation in browsers, it has been removed.

So now that we have covered the history of HTML in some detail, I want to dive deeper into some aspects of HTML that make it powerful enought that, even 30 years since the first tags were implemented/introduced, it hasn't been replaced.

Back to the beginning (again)

One thing that I find amusing when I hear all the talks about how Framework X is going to elimintate the need for HTML and CSS is that these frameworks produce an opinionated version of HTML and CSS along with the Javascript they use to create it.

Using HTML has evolved from the all-uppercase HTML tags developers used to create the first web pages to the convoluted build systems that we use today.

I know I'm old school but writing HTML shouldn't be harder than opening a text editor and typing out the HTML for the pages we need and link the necessary CSS stylesheets and scripts.

Creating an h1 elements in a page is as simple as typing the following inside the body of a document:

```
<h1>Hello World of HTML</h1>
```

But consider the following snippets of code that accomplish the same task:

React require the `render` method of the React.DOM module to be called to

render content. The `render` method takes two paramenters":

- The first parameter is the string of html that we want to render
- The second parameter is the DOM element that we want to render the html into

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

A similar Vue application will only render the data inside the element specified in the `el` attribute. The result will depend on how we structured the template containing the element

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Both libraries require you to build the page template separately, leaving components living in limbo. If I were to look at these components in isolation, I would have some idea of what the react component is used for but the Vue components gives no clue as to its intended use.

You get the idea. Now we're using Javascript to create the HTML we need to render our pages.

This would be good if people also learned how to build what is now called static content in some fashion. But we don't.

I learned to do web development in a way that is fully static and hand coded. What are now the fundamentals were the only things we needed to learn back in the day but they are no less important.

I'm not saying that everyone shoud take in-depth courses on CSS, HTML and Javascript but we should al have the same basic level of understanding regarding

these foundational technologies before we jump into specialized “lets Javascript all things” courses and frameworks. This is particularly important because new developers are coming directly to frameworks without any previous idea of how things work.

Web Components: Breaking the cycle

Perhaps the biggest drawback of HTML is how long it takes to get new tags and attributes adopted into the HTML specification and then implemented in browsers.

Web Components are a way to solve this problem. They are a family of specifications designed to allow the creaation of HTML elements and their use on a page independent of any library.

The technologies that make up web components are:

- **Custom elements**: A set of JavaScript APIs that allow you to define custom elements and their behavior
- **Shadow DOM**: A set of JavaScript APIs for attaching an encapsulated “shadow” DOM tree to an element
- **HTML templates**: The `template` and `slot` elements enable you to write inert markup templates that can be reused as the basis of a custom element’s structure.

These technologies may sound daunting but in practice it’s a fairly simple process:

1. Create a class in which you specify your web component functionality, using the ECMAScript 2015 class syntax
2. Register your new custom element using the `CustomElements.define()` method, passing it the element name to be defined, and the class or function that defines the custom element
3. If required, attach a shadow DOM to the custom element using Element.attachShadow() method. Add child elements, event listeners, etc., to the shadow DOM using regular DOM methods.
4. If required, define an HTML template using `template` and `slot` elements
5. Use your custom element wherever you like on your page, just like you would any regular HTML element

The following Javascript class handles steps 1, 2 and 3 of our web component:

```
class MyParagraph extends HTMLElement {
  constructor() {
    super();

    const template = document.getElementById('my-paragraph');
    const templateContent = template.content;

    this.attachShadow({mode: 'open'}).appendChild(
      templateContent.cloneNode(true)
    );
  }
}

customElements.define(
  'my-paragraph',
  MyParagraph
);
```

While the following teemplate is used to define the HTML structure of the custom element which we instantiate in the HTML class.

```
<template id="my-paragraph">
  <p name="my-content">Hello World</p>
</template>
```

With these two elements we can create as many copies of a web components as we need for our page. We can also have as many different web components as we want.

SO why is this important? It makes it easier for developers to create web components that are made for purpose and can integrate with other web components in a similar fashion to the atomic design philosophy: atom → molecule → organism.

When combined with the existing collection of HTML elements, we get the best of both worlds? We can leverage the power of existing elements in terms of accessibility and interactivity while creating new elements that are suited for our specific needs and purposes.

And perhaps this is the biggest drawbacks of web components: **they don't provide the same accessibility and interactivity as existing HTML elements**. If you want to add accessibility to your component, you need to impmlement it yourself and it's is not a trivial exercise.

Still, it's a good compromise between the two worlds. You get the benefits of HTML elements while you create the best elements to suit your needs.

The evolution of web components

Web components have an interesting story.

Alex Russell first introduced Web Components at the Fronteers Conference in 2011.

Polymer, a library based on Web Components was released by Google in 2013. Polymer also provides a set of ready-made components that you can drop into your project.

The x-tags library from Mozilla was one of the first libraries available for developers and was the basis of Brick

There are other web component libraries out there with varying levels of adoption. Browsers also have varying level of support for web components.

Over the years things have changed and other important libraries have come into the mix.

x-tags is now a Microsoft project since the lead developer left Mozilla and moved to Microsoft. The Bricks library is in maintenance mode and, as far as I can tell, it is no longer under development (the last tweet on the @mozbrick was in 2014).

Polymer had multiple versions that acommodated the different iterations of the web component specs (0.5, 0.8, 1, 2, and 3) before being replaced by the Lit element.

Salesforce introduced Lightning Web Components, the latest commercially supported web components library.

Generic Components are built against the WAI-ARIA specifications and make for a good starting point for your own web components since someone already to the heavy lifting regarding accessibility.

webcomponents.org has become both a documentation repository for web components and a search engine for existing component implementations.

For a more nuanced vision of where we're at with Web Components today, see Dave Rupert's HTML with Superpowers

Web Components break the slow cycle of HTML development. If there is something you need and you're willing to put in the effort to create it, you can have it.

Do web components work with (insert favorite framework)?

One of the biggest selling points of web components is interoperability. They can be used with any framework and in any browser that supports the specifications.

How can we tell?

Frameworks that have submitted data to Custom Elements Everywhere give you a measure of well they support web components. We can combine web components from different libraries and our own custom web components to create exactly what we need.

This sort of fulfill the promise of write once and use everywhere. It may also be possible to use multiple Polymer/Lit components alongside x-tags and Lightning Web Components. It will also require a lot of javascript to do so... whether you choose to or not is up to you.

Accessibility of web components

One of the biggest concerns about web components is accessibility. Because all custom elements must inherit from `HTMLElement` and not the more specific subclasses like `HTMLButtonElement`, `HTMLLinkElement`, `HTMLInputElement`, or similar. This means that custom elements will not inherit accessibility properties from their parent elements.

This means that whatever accessibility wins we get from built in elements are not available to custom elements... unless we put them back ourselves.

Explicit and Implicit Roles in Web Components

In HTML, you can define what an element is with a role (see: the ARIA Roles Model). This helps give users and Assistive Technologies (AT) tools like screen readers, context on how to interact with the element.

There are explicit roles that we can assign directly with the `role` attribute. In this example, the `<div>` has a `role="alert"` attribute so AT know to read it immediately when it appears on the page.

```
<div role="alert">
  This is an alert!
</div>
```

Some HTML elements have implicit roles (see: complete list of implicit roles). These implicit roles tell the browser and AT that an element is already understood to have a role; is implied to have role="list", and <li> has role="listitem".

Browsers and AT know how to announce these based the implicit roles and how they normally interpret the roles.

```
<ul>  // implied role="list"
  <li>content</li>  // implied role="listitem"
</ul>
```

For this to work the elements with implicit roles must be the direct children of the correct parent element,mixing web components with traditional elements will not work.

In this example the <li> element is not a direct child of <ul> but of <custom-list-item>, an element with an implicit role of presentation (meaning that there is no implicit role at all).

```
<ul>                  // implied role="list"
  <custom-list-item>  // implied role="presentation"
    <li>content</li>  // Implied role="listitem"
  </custom-list-item>
</ul>
```

Because of the nesting, browsers will no longer recognize this as a list with a number of items. Instead, it sees a list whose children have no implicit or explicit role.

To fix this, we need to add the explicit roles to the correct elements. In this example we assign the listitem role to the <custom-list-item> element.

Now the browser and AT will know that this is a list with a given number of

items.

```
<div role="list">
  <custom-list-item role="listitem">
    content
  </custom-list-item>
  ...
</div>
```

Global HTML attributes and properties on custom elements

Custom elements don’t implement property and attribute behavior by default.

Global HTML attributes or properties on custom element without a corresponding script will stay on the custom element unless otherwise specified.

If the attribute is meant to be on the custom element, this is fine, but if the attribute needs to be passed to a child of the custom element makes things more complicated.

While this is not the recommended way to pass information from parent to child, it is possible to do it with the following requirements:

- A setter function that takes the value from the custom element and puts it on the correct element.
- A getter function that returns the value from the correct element (not the one passed to the custom element)
- A function that checks if the value is on the custom element and removes it so that the browser isn’t confused.

This requires a lot of maintenance because we have to constantly monitor the web components to make sure there are no duplicate values between the web component and the child elements.

ID referencing + Shadow DOM

Many ARIA-related HTML attributes (aria-labelledby, aria-describedby, aria-controls, etc.) use the ID of another element as a connector between the two.

Developers also use DOM queries to set focus with document.getElementById.

A shadow boundary, even if it's open, makes referencing IDs from outside the component impossible because the web component's DOM is separate from that of the page.

Let's say we have a `custom-input` component with a unique ID and try to associate it with a label in that label's HTML for attribute, like this:

```
<label for="my-input">City</label>
<custom-input>
  #shadow-root (open)
    <input id="my-input" type="text" />
</custom-input>
```

This code will not work because when it is rendered, the input is in a separate DOM tree from the label, which means the browser can no longer link them together.

A first way to fix this is to place both elements that use the ID inside the component's shadow DOM. For this example, we move the label element inside the custom element so we can reference the input's ID from the label:

```
<custom-input label="City">
  #shadow-root (open)
    <label for="my-input">City</label>
    <input id="my-input" type="text" />
</custom-input>
```

Another most surprising and problematic way to do it is to use slots. If your component has a slot inside the element's shadow DOM you can reference it, even if it appears to cross shadow boundaries because the content of the slot is outside the component's shadow root, even if the slot is within the shadow boundary.

In the next component, the input element goes into a slot and the label outside stays outside the component.

```
<label for="another-input">Country</label>
```

```
<form-element-container>
  <input id="another-input" type="text" slot="input-slot"/>
</form-element-container>
```

The main drawback to the slots is that, since they are in the light DOM, all styles and JavaScript on the page will affect the content in the slots so we have to be careful on how we style our page and how we design our scripts for our pages and apps.

`tabindex="-1"` and Custom Elements

`tabindex="-1"` is a great tool! It is used to make something focusable and clickable, but not in the tab order.

Currently, if you put `tabindex="-1"` on a custom element with shadow DOM enabled, it makes so you can not tab through any of its children. This is unexpected, since tabindex usually does not propagate, and makes it so only people using a mouse can click the elements.

This limitation means we must be careful about what elements you put `tabindex="-1"` on. Ideally `tabindex="-1"` should never be on the custom element itself.

These are some examples of accessibility issues with web components. You should do detailed research on how will AT work with your custom elements and your page in general.

CSS

Unlike HTML, CSS has changed drastically since its inception. It was not part of the original World Wide Web set of specifications, the first version of the CSS specification was released in 1996.

Before the first formal specification of CSS there were many languages that could have taken the place of CSS. The Languages Which Almost Became CSS describes the alternative proposals for styling languages for the web.

There are three monolithic CSS specifications and a number of modules specified since the release of CSS 2.0 and 2.1.

CSS Version	Release Date
CSS 1	1996
CSS 2	1998
CSS 2.1	2011

Rather than one monolithic specification, work on what would have been the CSS 3 specification has been broken down into modules.

The full list of drafts currently under development at the CSS Working group is at drafts.csswg.org.

CSS (1.0)

The CSS 1 specification provides basic styling for HTML documents that, until this point had either no style or the Mosaic-influenced attribute styles.

Selectors were limited to class, id, tag names and contextual selectors. The following selectors were valid at the time CSS 1 was released:

```
H1 { color: red; }

.all-green { color: #00ff00; }
```

```
H1 EM { color: purple; }
```

Pseudo elements are also part of the initial specification. Links, first-line and first letter (surprisingly to me) are part of the specification:

```
A { color: #FF0000; }
A:link { color: #FF0000; }
A:visited { color: #0000FF; }

P:first-letter {
  font-size: 200%;
  font-colot: #FF00FF;
}

P:first-line {
  font-size: 150%;
  font-colot: #00FF00;
}
```

The rest of the specification is in the Cascading Style Sheets, level 1 specification.

CSS 2.0 and 2.1

Cascading Style Sheets Level 2 and the later Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) specification update expand on the original CSS 1 specification.

> CSS 2:
>
> - Introduced the concept of media types and aural style sheets
> - Added i10n (internationalization) features
> - Extended font selection
> - Added the context of automatic numbering and generated content
> - Added cursors
> - Added capabilities to control content overflow, and clipping
> - Added the ability to position content in the page (absolute, fixed and relative)
> - Extended the selector mechanism from CSS 1

Because CSS 2 expanded the capabilities of CSS, the specification is significantly larger both in page count... CSS 2 clocked at around 650 printed pages and in terms of capabilities.

CSS 3 and beyond?

As good as CSS 1 and 2 were at the time they presented a difficult dilemma for browser makers: Features mature at different rates but because they are all part of one standard document they can't be considered complete until the full specification is finished.

Rather than one monolithic specification, the CSS working group decided that work on what would have been the CSS 3 specification would be broken down into modules. This would allow the different features to mature at their own pace and be published as specifications when they were ready rather than having to wait for a full specification to be completed.

The following modules have completed the specification process and are now recommendations from the W3C

Specification	Status
CSS Color Level 3	REC
CSS Namespaces	REC
Selectors Level 3	REC
Media Queries	REC
CSS Style Attributes	REC
CSS Cascading and Inheritance Level 3	Rec
CSS Fonts Level 3	REC
CSS Writing Modes Level 3	REC
CSS Basic User Interface Level 3	REC
CSS Containment Level 1	REC

The following modules are not finalized but they are far enough into the standard track that they have multiple interoperable implementations

Specification	Status
CSS Backgrounds and Borders Level 3	Candidate Recommendation
CSS Conditional Rules Level 3	Candidate Recommendation
CSS Multi-column Layout Level 1	Working Draft
CSS Values and Units Level 3	Candidate Recommendation
CSS Flexible Box Layout Level 1	Candidate Recommendation
CSS Counter Styles Level 3	Candidate Recommendation

To see the full list of drafts currently at some stage of work by the CSS Working group, check drafts.csswg.org.

The advantage of this modular development process is that different features can mature independently of each other at their own pace. This makes it easier for browser makers to implement features and for developers to use them.

To see a list of the specifications the CSS Working considers complete, see the CSS Snapshot 2020. As the snapshot states: ***The primary audience for the CSS snapshot is CSS browser makers, not CSS authors, as this definition includes modules by specification stability, not Web browser adoption rate***.

The snapshots are updated annually.

Speed of adoption and adoption rate

In order to become a recommendation, a CSS specification must have two inteopeable implementations.

This requirement presents users with the problem of CSS speed of adoption.

This issue can be taken from either a browser maker's perspective (how long does it take to implement a feature once it reaches candidate recommendation stage? Will all browsers implement the feature?) and from a developer's perspective (how long will it take for me to implement a feature that is available in all browsers?).

The other question is what proportion of browsers support a given feature?

This is always a tricky balancing game where developers have to decide between adopting a feature that will make some users happy versus having to write large amounts of CSS and, maybe, Javascript to make it work for everyone.

CSS provides the `@supports` at-rule to make it easier to test if a feature is supported by a browser. We can then work around those browsers that don't support the feature.

CSS versus SASS versus PostCSS versus (put your pre-processor's name here)

I've used SASS for a long time and have seen it as a great way to write CSS with enhanced features that were not originally part of the any CSS specifications. But I've also seen CSS grow on its own as a parallel.

Take the following example of a mixin that changes the colors on the element it's attached to based on whether they are even or odds. I would normally use this code, or something similar to it, on tables.

```
@mixin stripes($length: 10) {
  @for $i from 1 through $length {
    @if $i % 2 == 0 {
      background-color: #ff00ff;
      color: #ffffff;
    } @else {
      background-color: #0000ff;
      color: #ffffff;
    }
  }
}
```

Initially CSS did not have the capability to do this so SASS was pretty much the only way to do it. It wasn't until the `nth-child` selector was added to CSS that it became possible to do it in a similar fashion.

This code will alternative colors between rown in the table body.

```
tbody tr:nth-child(odd) {
  background-color: #ff00ff;
  color: #000000;
}

tbody tr:nth-child(even) {
  background-color: #0000ff;
  color: #ffffff;
}
```

Another aspect of SASS that I find very useful is nesting. The idea is to make the code more readable and easier to maintain

```
.container{
  & h1{ font-size: 25px; color:#E45456; }
  & p{ font-size: 25px; color:#3C7949; }

  & .box{
    & h1 { font-size: 25px; color:#E45456;}
    & p{ font-size: 25px; color:#3C7949; }
  }
}
```

This will produce the following CSS

```
.container h1 { font-size: 25px; color: #E45456; }
.container p { font-size: 25px; color: #3C7949; }
.container .box h1 { font-size: 25px; color: #E45456; }
.container .box p { font-size: 25px; color: #3C7949; }
```

CSS will let you write nested rules in three different ways:

Write the nested selectors by hand as separate rules.

```
.foo { color: red; }
.foo > .bar { color: blue; }
```

Write the nested selectors using the & CSS selector defined in the CSS Nesting Module

```
.foo {
  color: blue;
  & > .bar { color: red; }
}
```

Or use the @nest at-rule, also defined in the CSS Nesting Module

```
.foo {
  color: red;
  @nest & > .bar {
    color: blue;
  }
}
```

All three methods produce the same CSS.

The final tool that I want to cover when talking about differences between SASS and CSS are variables. Variables in SASS are static and every change that we make means we have to recompile the stylesheets

```
$color-red: #ff0000;
$color-green: #00ff00;
$color-blue: #0000ff;

$color-white: #ffffff;
$color-black: #000000;
```

CSS variables, also known as CSS custom properties, are live, dynamic values that update the display of the page as soon as the values change.

```
:root {
  --color-red: #ff0000;
  --color-green: #00ff00;
```

```
  --color-blue: #0000ff;

  --color-white: #ffffff;
  --color-black: #000000;
}
```

There are ways to combine features from both languages to make a more powerful language.

This example combines CSS custom properties, SASS variables and SASS intrerpolation to create a color scheme.

```
$primary: #81899b;
$accent: #302e24;
$warn: #dfa612;

:root {
  --primary: #{$primary};
  --accent: #{$accent};
  --warn: #{$warn};

  // This is valid CSS
  --consumed-by-css: $primary;
}
```

These are some examples of where SASS and CSS have diverged over the years. It still holds true that SASS is a superset of CSS but there are SASS features that will never make it to CSS

PostCSS provides a way to test new CSS features even if they haven't been implemented in browsers yet, just like Babel does for Javascript. It also uses Browserslist to identify the browsers that support a given feature where this is necessary, not all plugins use version dependent code.

PostCSS requires an additional step when preparing your CSS. You need a basic configuration file that tells the PostCSS executable what plugins you want to use and how those plugins are configured or you can choose to incorporate the configuration into your build process.

The syntax for your CSS will change somewhat based on the plugins you have available. The idea is that the syntax will become part of the CSS standard and, because you've implemented the syntax ahead of time, your code won't have to change.

specificity and the cascade

A lot of times I hear about how hard it is to get CSS to work properly and for styles not to oveerride and bleed into each other. That's where two concepts come to mind, the cascade and specificity.

The cascade is the algorithm at the core of CSS that defines how to combine property values that affect thge same element but originate from different sources.

Specificity is the means by which browsers decide which CSS property values are the most relevant to an element and, will be applied to it.

Origin of CSS declarations

The CSS cascade algorithm's job is to select CSS declarations in order to determine the correct values for CSS properties. CSS declarations originate from different origins:

- The User-agent (browser) built-in stylesheet
- The Author stylesheets
- The User stylesheets.

Though style sheets come from these different origins, they overlap in scope; to make this work, the cascade algorithm defines how they interact.

User-agent (browser) stylesheet : The browser has a basic style sheet that gives a default style to any document. Some browsers use actual style sheets for this purpose, while others simulate them in code, but the end result is the same. : There are significant differences between browser's default stylesheets. This is why web developers often use a CSS reset style sheet, forcing common properties values to a known state before beginning to make alterations to suit their specific needs.

Author stylesheets : Author stylesheets are the most common type of style

sheet. These stylesheets are where the author of the page defines the styles for the document.

User stylesheets : The user (or reader) of the web site can choose to override styles in many browsers using a custom user stylesheet designed to tailor the experience to the user's wishes.

SO how do browsers decide which CSS declarations are the most relevant to an element?

1. It first filters all the rules from the different so that only the rules that apply to a given element
2. Then it sorts these rules according to their importance, that is, whether or not they are followed by !important, and by their origin

The cascade is in ascending order, which means that !important values from a user-defined style sheet have precedence over normal values originated from a user-agent style sheet

	Origin	Importance
1	user agent	normal
2	user	normal
3	author	normal
4	animations	
5	author	!important
6	user	!important
7	user agent	!important
8	transitions	

Specificity

The cascade deals with setting precendence and what elemenets are applied when multiple rules match a given selector.

Specificity deals with the other side of the coin. What rule will be applied when more than one rule matches a given element.

Stuff & Nons

a

1 x element selector

Sith power: 0,0,1

Figure 2: CSS specificity wars

If we follow the figure above we can see the difference of how the different element, classes, IDs and combinations thereof.

Once you add all the different components of the selector then the highest value “wins”.

```
p {
  color: #ff0000;
}

.container p {
  color: #00ff00;
}

#container2 {
  color: #663399;
}

a {
  color: #ff00ff !important;
}
```

So, why is this important?

A lot of of times we can use the cascade and the rules for specificity to improve our CSS code.

```
.bar { color: red; }
.foo > .bar { color: blue; }
```

New specifications like CSS Layers will help make working with CSS easier.

Houdini: the circuit breaker

As far as CSS has moved, there are still many things developers would want CSS to do that will not be incorporate to existing or future standards or lackk interest from browser makers to implement.

The Houdini task force consists of engineers from Mozilla, Apple, Opera, Microsoft, HP, Intel and Google working together to expose certain parts of the CSS engine to web developers.

Houdini has a list of APIs that are currently under development at drafts.css-houdini.org/.

Specification	Last Updated
Box Tree API 1	2020-04-07
CSS Animation Worklet 1	2020-08-18
CSS Layout API 1	2021-05-10
CSS Painting API 1	2020-12-15
CSS Parser API 1	2017-11-09
CSS Properties and Values API 1	2021-02-26
CSS Typed OM 1(Current Work)	2021-10-13
CSS Typed OM 2	2021-02-08
Font Metrics API 1	2021-05-12

While not strictly part of Houdini (it is part of the HTML specification), Worklets are important for Houdini to work.

Is Houdini ready yet?

	Google Chrome	Microsoft Edge	Opera	Samsung Internet	Mozilla Firefox	Apple Safari
Engine	Blink				Gecko	WebKit
Paint API (Explainer Demos Article)	Shipped (Chrome 65) Details	Shipped (Edge 79) Details	Shipped (Opera 52) Details	Shipped (Internet 9.2) Details	Under consideration Details	In Development Details
Properties & Values API (Demos Article)	Shipped (Chrome 78) Details	Shipped (Edge 79) Details	Shipped (Opera 65) Details	Shipped (Internet 12.0) Details	Under consideration Details	Partial support (Safari TP 67) Details
Typed OM (Explainer Article)	Shipped (Chrome 66) Details	Shipped (Edge 79) Details	Shipped (Opera 53) Details	Shipped (Internet 9.2) Details	Under consideration Details	In Development Details
Layout API (Explainer Demos)	Partial support (Canary) Details	Partial support (Canary) Details	Partial support (Developer) Details	No signal	Under consideration Details	Under consideration Details
Animation Worklet (Explainer Demos Article)	Partial support (Chrome 71) Details	Partial support (Edge 79)	Partial support (Opera 58)	Partial support (Internet 10.2)	No signal	Under consideration Details

I will concentrate on the following Houdini APIs:

- CSS Properties and Values API 1
- CSS Painting API 1
- CSS Typed OM 1

Properties and Values provides an enhanced API for working with CSS custom properties.

One of the problems with the current Custom Properties API is that everything is a string and it forces you to use calc to convert the strings to numbers when needed; because they are all treated as strings, you cannot use custom properties in animations or transitions, they also inherit by default, whether you want to or not.

Houdini Properties and values provides these additional tools. Using the `@property` at-rule, you can define a stronger custom property.

```
@property --my-color {
  syntax: '<color>';
  inherits: false;
  initial-value: #c0ffee;
}
```

You can also define custom properties from Javascript. This is the same property defined in Javascript.

```
window.CSS.registerProperty({
  name: '--my-color',
  syntax: '<color>',
  inherits: false,
  initialValue: 'c0ffee',
});
```

Both versions have the same requirements:

name (required) : The name of the property.

syntax (required) : The allowable syntax for the property We are no longer limited to strings. : The values are a subset of the values defined in the CSS Values and Units spefication. For the list of supported syntax values see Supported Names in the CSS Properties and Values API specification : You can also use + to allow for a space-separarted list of one or more values, # for a comma seprated list of values, and separate syntaxes with | to allow one syntax or another

inherits (required) : Boolean that controls whether the custom property inherits by default.

initial-value (optional) Sets the initial value for the property.

By having these fields, we can:

- Use the property's initial value (if one is defined) when the property has no value defined
- Animate the property
- Use the property in a transition
- Provide a clear explanation of the property and how we want to use it to people reading the code

While it's true that using the houdini versions of custom properties limits us to Chromium browsers, we can still code defensively in CSS by passing the regular custom property first and the passing the Houdini version, because of cascading order the houdini version will be used in browsers that support it and be ignored otherwise.

```
--my-color: #c0ffee;

@property --my-color {
  syntax: '<color>';
  inherits: false;
  initial-val'<color>'ee;
}
```

The Painting and Layout APIs use worklets, now part of the HTML specification, as their language.

Yes, this is another case where we use Javascript to write what we will use in CSS but, in this case, the ends justify the means.

This example, taken from the Chrome Teams Circle Paint API demo shows how to use the Paint API.

The code uses the following HTML

```
<div class="circle"></div>
```

In the main script load the worklet. We only load the paint worklet if the browser supports it; otherwise we notify the user.

```
<script>
  if ('paintWorklet' in CSS) {
    CSS.paintWorklet.addModule('paintworklet.js');
  } else {
    document.body.innerHTML = 'You need support for <a href="https://draft
  }
</script>
```

In the CSS portion of the example we define the custom property using the `@property` at rule and then we use the property and we use the `paint()` function to tell the browser that we want to use the paint worklet as the background image.

```
@property '--circle-color'{
    syntax: '<color>',
    inherits: t'<color>'initialValue: '#00ff00'
}

'#00ff00'.circle {
  --circle-color: green;
  background-image: paint(circle);
  height: 80vh;
  width: 100vw;
}
```

The worklet itself uses a subset of the Canvas API syntax to do the work.

```
registerPaint('circle', class {
  static get inputProperties() { return ['--circle-color']; }
 '--circle-color', properties) {
    // Get fill color from property
```

```
    const color = properties.get('--circle-color');

    '--circle-color'// Determine the center point and radius.le = size.wi
    const yCircle = size.height / 2;
    const radiusCircle = Math.min(xCircle, yCircle) - 2.5;

    // Draw the circle o/
    ctx.beginPa/ Draw the circle o/
    ctx.beginPath();
    ctx.arc(xCircle, yCircle, radiusCircle, 0, 2 * Math.PI);
    ctx.fillStyle = color;
    ctx.fill();
  }
});
```

The final bit of Houdini I want to talk about is the Typed OM API.

The Typed OM is an extension to the existing CSS Object Model (CSSOM) that exposes CSS values as typed JavaScript objects, instead of a simple strings like they are today. Trying to convert strings in to meaningful types and back today can have a big performance overhead, so this will allow us to work with CSS values in a much more performant way

With the Typed OM, CSS values are now members of a new base class, `CSSStyleValue`. The subclasses of `CSSStyleValue` more precisely describe a type of CSS Value:

CSSKeywordValue : CSS Keywords and other identifiers (like inherit or grid)

CSSPositionValue : Position (x and y) values

CSSImageValue : An object representing the value properties for an image

CSSUnitValue : Numeric values that can be expressed as a single value with single unit (like 50px) or a single value or percentage without a unit

CSSMathValue : Complicated numeric values, like you would find with calc, min, and max. This includes subclasses `CSSMathSum`, `CSSMathProduct`, `CSSMathMin`, `CSSMathMax`, `CSSMathNegate`, and `CSSMathInvert`

CSSTransformValue : A list of CSS transforms consisting of `CSSTransformComponents`, including `CSSTranslate`, `CSSRotate`, `CSSScale`, `CSSSkew`, `CSSSkewX`, `CSSSkewY`, `CSSPerspective`, and/or `CSSMatrixComponent`

There are methods for working with Typed OM:

You set the elements via on elements for working with the Typed OM:

- You set and get the attributes via `attributeStyleMap`.set and get
- You get the element's full typed OM styles via `computedStyleMap`

For the example below we use the typed OM to set and retrieve the value for font-size for the given attribute

```
const myElement = document.querySelector(".plum-crazy");

myElement.attributeStyleMap.set("font-size", CSS.em(1.2));

const fontSize = myElement"font-size"tyleMap.get("font-size");

console.log(fontSize);
"font-size"// CSSUnitValue { value: 1.2, unit: 'em' }
```

`computerStyleMap` returns a `CSSStyleDeclaration` object that can be used to get and set the values of the CSS properties. We can then get different values for different properties.

```
const myBox = document.querySelector(".color-box").computedStyleMap();

console.log(myBox.get("font-size"));
console.log(myBox.get("width"));
console.log(myBox.get("height"));
console.log(myBox.get("display"));
```

Note:

It is important to note that the browsers that implements Typed OM don't support all HTML elements.

Even if the support is not complete it still provides a powerful way to work with CSS properties via Javascript. This will make worklets and other APIs that work with CSS more powerful and performant.

Javascript

Out of the tree component technologies that make up the web, Javascript has gone through the worst growing pains of all the three.

We've probabley heard the story of how Javascript was created in ten days to coincide with the release of Navigator 2.0 and it was meant as a dynamic language that could be parsed by the browser but that's not the whole story. The history of Javascript is worth reviewing, if for no other reason than understanding the past may help us understand why we are where we are.

THe table below tracks three parallel strands of Javascript development:

- The original Javascript developed at Netscape and released as part of Netscape product line
- JScript, the Microsoft version of Javascript released with Internet Explorer
- Ecmascript, the standard version of Javascript that resulted from Netscape's Javascript submission to ECMA (then the European Computer Manufacturers Association)

Official Name	Year Released	Javascript Version	Ecmascript Version	Description
Javascript 1.0	1995	JavaScript 1.0	N.A	Released with Netscape Navigator 2.0
JScript 1.0	1996	JavaScript 1.0 (JScript 1.0)	N/A	Released with Internet Explorer 3.0
Javasacript 1.1	1996	JavaScript 1.1	N/A	Released with Netscape Navigator 3.0
Javascript 1.2	1997	JavaScript 1.2	N/A	Released with Netscape Navigator 4.0
JScript 3.0	1997	JavaScript 1.2 (JScript 3.0)	N/A	Released with Microsoft Internet Explorer 4.0
EcmaScript	1997	N/A	ES1	First edition of the stadardized

Official Name	Year Released	Javascript Version	Ecmascript Version	Description
1				language
Javascript 1.3	1998	JavaScript 1.3	N/A	Released with Netscape Navigator 4.5
ECMAScript 2	1998	N/A	ES2	Editorial changes
ECMAScript 3	1999	N/A	ES3	Added regular expressions and try/ catch
Javascript 1.4	1999	N/A	Javascript 1.4	
JScript 5.0	1999	JScript 5.0	N/A	Released with Microsoft Internet Explorer 5.0
Javascript 1.5	2000	N/A	Javascript 1.5	Released with Netscape 6.2 and Firefox 1.0
ECMAScript 4		N/A	ES4	Not released
Javasacript 1.6	2005	N/A	Javascript 1.6	Released in Firefox 1.5. Matches ECMASCript 3 and ECMAScript for XML
Javascript 1.7	2006	N/A	Javascript 1.7	Released in Firefox 2.0 Includes Features not part of the then current ECMAScript Standard
Javascript 1.8	2008	N/A	Javascript 1.8	Released in Firefox 3.0 Includes Features not part of the then current ECMAScript Standard
ECMAScript 5	2009	N/A	ES5	Added “strict mode”, JSON support, String.trim(), Array.isArray(), & Array iteration methods
Javascript 1.8.5	2011	N/A	Javascript 1.8.5	Released in Firefox 4.0 This was the last version of

Official Name	Year Released	Javascript Version	Ecmascript Version	Description
				Javascript released. In future versions, Mozilla would implement the ECMAScript specification
ECMAScript 2015	2015	N/A	ES6 / ES2015	Added let and const, default parameter values, Array.find(), & Array.findIndex()
ECMAScript 2016	2016	N/A	ES7 / ES2016	Added exponential operator & Array.prototype.includes
ECMAScript 2017	2017	N/A	ES8 / ES2017	Added string padding, Object.entries, Object.values, async functions, and shared memory
ECMAScript 2018	2018	N/A	ES9 / ES2018	Added rest / spread properties, asynchronous iteration, Promise.finally(), and RegExp

In the beginning

> In 1995, the Web and Web browsers were new technologies bursting onto the world, and Netscape Communications Corporation was leading Web browser development. JavaScript was initially designed and implemented in May 1995 at Netscape by Brendan Eich, one of the authors of this paper. It was intended to be a simple, easy to use, dynamic languageg that enabled snippets of code to be included in the definitions of Web pages. The code snippets were interpreted by a browser as it rendered the page, enabling the page to dynamically customize its presentation and respond to user interactions
>
> Javascript: The first 20 years.

As the authors of the paper cited above note the first version of Javascript was created to fill a very narrow niche on the web. If there was any heavy lifting programming to be done it would be done on the server using CGI scripts written in Perl or C or it would be written in Java and run through the Java plugin available for Netscape browsers.

This is CGI program written in Perl using the CGI.pm module.

```
#!/usr/bin/perl
use strict;
use warnings;

use CGI::Simple;
my $q = CGI::Simple->new;
print $q->header;

print "Hello World!";
```

And the same program written in C that requires you to compile the script and place it in the cgi-bin directory of your server:

```
#include <stdio.h>

int main(void)
{
  printf("Content-type: text/html\n\n");
  printf("<html><title>Hello</title><body>\n");
  printf("Hello World!\n");
  printf("</body></html>");

  return 1;
}
```

You would link to these CGI scripts from a form's action attribute or directly in a hyperlink. The browser would then execute the script and send the results back to the browser to render.

In contrast, one of the first uses of Javascript that I remember was to do image rollovers.

```
function move_image(img_name,img_src) {
  document[img_name].src=img_src;
}
```

```
function move_out(img_name,img_src) {
  document[img_name].src=img_src;
}
```

With the two functions in place we would use the onMouseOver and onMouseOut attributes to change the image source, creating the roll over.

```
<a href="link.html"
  OnMouseOver="move_in('image1','image_on.gif')"
  OnMouseOut="move_out('image1','image_out.gif')">

<img src="image_out.gif"
  alt="Move your mouse over here!" name="image1">

</a>
```

There was no notion of Javasacript as the language to write full appplications with, at least not yet. If anything there was a way to connect Javascript with Java on Netscape browsers through a proprietary technology called LiveConnect. The tecnology still survives but the plugin necessary to make it work does not so the technology itself is defunct.

There was also no standard to follow. Javascript (Netscape) and JScript (Microsoft) were not always compatible which led to problems for developers and the ugly “better viewed in Netscape” or “Best viewed in Internet Explorer” messages.

Through the late 1990s, around the time ESCript version 3 was released, there was a perceived stagnation in the development of ECMAScript. It wasn’t that there was no work being done but that the people who were woking on the specification couldn’t really agree on what the specification should be and do.

> As the end of the 1990s neared, it was clear that the Internet and in particular the World Wide Web was having a phenomenal impact upon the world [Miniwatts Marketing Group 2019]. The rapid growth of the Web had been enabled by the incremental pragmatic enhancement of browser

technologies by Netscape, Microsoft, and other browser developers. The success of the Web and the necessity of coordinating the ongoing evolution gave rise to standards groups such as Ecma TC39 and W3C working groups. Some of the participants in those groups were subject matter experts who were not directly involved with browser development. Their interest was focused on an idealized futureWeb. From that perspective, the existing pragmatically developedWeb technologies were viewed as an impediment to that future.

In May 1998, the W3C held a workshop titled: “Shaping the Future of HTML.” The conclusions in the record of the workshop say:

> In discussions, it was agreed that further extending HTML 4.0 would be difficult, as would converting 4.0 to be an XML application. The proposed way to break free of these restrictions is to make a fresh start with the next generation of HTML based upon a suite of XML tag-sets. The workshop expressed a need for a better match to database and workflow applications, and for the widely disparate capabilities of small/mobile devices. Modularizing HTML will provide the flexibility needed for this. [W3C 1998]

David Singer [1998], representing IBM, was more blunt in a workshop presentation: “The Future of HTML as we know it should be: Nasty, Brutish, and Short.”

As ES3 approached completion, TC39 found itself in a similar situation. With ES3, ECMAScript had caught up with the JavaScript features provided by the Netscape and Microsoft browsers and, at least initially, the browser vendors weren’t providing much guidance regarding what to do next. Unlike Netscape in 1995, TC39 was not constrained to avoid Java-like capabilities. Some TC39 participants saw a need for a second-generation browser scripting language that corrected mistakes made in the original JavaScript design and that offered

> features [Raggett 1999b; TC39 1999c; Appendix J] catering to the needs and sensibilities of professional software developers rather than non-professiional script writers. This new generation of ECMASript was targeted to be the 4th edition of of ECMA-262. Within TC39, it was initially called “E4” and later “ES4.”
>
> Javascript: The first 20 years.

In 1999 there were people working towards the next iteration of Java/Ecmascript after ES3. See in particular the work of Valdemar Horwat (then at Netscape) and his paper JavaScript 2.0: Evolving a Language for Evolving Systems and how it already reflects some of the thoughts and ideas discussed for ES4 and that we’d see in ES6 and subsequent drafts of the specification.

> The first major proposal came from Dave Raggett who was a W3C Fellow sponsored by Hewlett-Packard. At the W3C, Raggett was developing a proposal named “Spice” to improve the integration of HTML, CSS, and JavaScript. An early version of the proposal [Raggett 1998c] was submitted to TC39 in February 1998. In addition to HTML and CSS integration features, Raggett’s initial proposal included a construct for declaring prototype objects which was similar to the Borland class declaration proposal. It added the ability to declaratively associate event handlers with prototype objects.
>
> Javascript: The first 20 years.

See Javascript: The first 20 years for further discussion on Spice, the different proposals and their eventual rejection.

Another proposal for this early stage of work on ECMAScript 4 was Valdemar Horwat’s proposal for Javascript 2.0. This proposal was also not compatible with then current ECMAScript implementations. The solution was to provide multiple interpreters

> JavaScript 2.0 was not an attempt to be fully backward compatible with original JavaScript or even with the still-not-completed ECMAScript 3. When introducing JavaScript 2.0 to TC39, Waldemar Horwat said: “At a bare minimum you should

> be able to write code that works in ECMAScript 1.0 and 2.0 [ES4]. Full backwards [sic] compatibility would be rather painful." [Raggett 1999c] For example, the syntactic complications of the optional type annotations precluded supporting automatic semicolon insertions on line breaks. Horwat's solution to backward compatibility was for implementations to provide multiple compilers. He believed that switching compilers according to the language version was preferable to a single language with strict forward compatibility.
>
> Javascript: The first 20 years.

Even though we came to see Macromedia/Adobe Flash as the evil competitor against the web, we have to remember that when Macromedia first introduced the technology, the web was a far less capable platform than what it is today.

Eearlier versions of Actionscript tracked development of and implemented most of the features in ECMAScript 3. The problems started when applications started hitting performance bottlenecks.

> In 2003, the wide adoption of Flash for Web development was leading to the creation of large and complex ActionScript applications and some of them were encountering performance issues. Like most ECMAScript language designers and implementors at that time, the Macromedia team believed that dynamic typing (particularly of primitive types) was the main performance bottleneck, and were exploring ways to add static typing to the ActionScript runtime. Around this same time, Jeff Dyer, who had been a TC39 delegate since 1998, joined Macromedia. Dyer confirmed that TC39 shared that same perspective about static typing. This widely shared view of static typing in virtual-machine-based languages was strongly influenced by the design of the statically typed Java Virtual Machine (JVM). Jonathan Gay's and Lee Thornason's Maelstrom project was a Macromedia experiment to see if a JVM could be integrated into Flash and used as the runtime for a statically typed version of ActionScript. The experiment was successful enough that Macromedia approached Sun about licensing the Java 2 Micro Edition (J2ME) JVM for use in Flash. They wanted to use J2ME because the standard edition Java

> runtime was too large to embed within a Flash Web download. But Macromedia's proposed use of Java Micro Edition technologies did not align with Sun's Java licensing strategy. Edwin Smith, in a skunkworks effort, created a series of proof-of-concept virtual machines. Those VMs helped to convince Macromedia to build their own statically typed JVM-like virtual machine called AVM2 [Adobe 2007], and a new version of ActionScript to run on it. The new language was designed by Gary Grossman, Jeff Dyer, and Edwin Smith, and was heavily influenced by Horwat's draft $ES4_1$/JS2 specifications. However, like JScript.NET, ActionScript 3.0 was a simplification of the $ES4_1$ design. It was less dynamic than JS2 and, unlike JScript.NET, it was not constrained by the .NET type model. ActionScript 3.0 was also similar to JScript.NET in that it was not heavily constrained by legacy compatibility concerns. Flash would ship with both AVM2 to support ActionScript 3.0 and AVM1 to support ActionScript 1.0 and 2.0. This effort to create a new version of ActionScript and a new virtual machine took over three years to complete. It was announced in 2006 as a component of Flash Player 9, which ultimately shipped in 2007. By the time the effort was completed, Adobe had acquired Macromedia and Flash had become Adobe Flash.
>
> Javascript: The first 20 years.

Allen Wirfs-Brock from Microsoft and Douglas Crockford (from Yahoo) were not sold on ES4 as proposed and locked the committee into two separate releases for the specification:

- An incremental release (3.1) that improves the existing ECMAScript specification
- The continued work on the ECMAScript 4 specification

But in June 2008 Adobe anounced their withdrawal of support for ES4, mostly due to their perception that support for ES3.1 would make it impossible to incorporate things like the static type system from Actionscript 3 into the ECMAScript 4 specification.

The majority of the TC39 July 2008 meeting was spent explaining and socializing the concept of harmonization of TC39 around a common set of obtainable goals.

The overall plan was to focus the entire committee on completing the ES3.1 release during 2009 while simultaneously collaborating in planning a more significant follow-on edition, code named “Harmony”. This new release would not be constrained by the previous ten years of ES4 design decisions.

According to this email from Brendan Eich to es-discuss the objectives agreed upon at a meeting in Oslo on 2008 included:

1. Focus work on ES3.1 with full collaboration from all parties, and target two interoperable implementations by early next year.
2. Collaborate on the next step beyond ES3.1, which will include syntactic extensions but which will be more modest than ES4 in both semantic and syntactic innovation
3. Some ES4 proposals have been deemed unsound for the Web, and are off the table for good: packages, namespaces and early binding. This conclusion is key to Harmony
4. Other goals and ideas from ES4 are being rephrased to keep consensus in the committee; these include a notion of classes based on existing ES3 concepts combined with proposed ES3.1 extensions

For further reference see the white paper outlinining the agreements from the meeting.

During this time the work was first on getting ES3.1 off the ground and then ES5 ready for shipping as the next evolution of the language without all the major baggage of ES4.

Once this work was completed and the ECMAScript 5 specification was out the door, work could begin on ECMAScript Harmony.

There were multiple strawman proposals for what should be included in the specification. In July 2009, Brendan Eich published an updated version of the Harmony Goals Statement expanded from the one written for ES3.1.

Harmony is important because all the Harmony strawman proposals that were accepted became part of ES2015, accepted by the ECMA General Assembly in June 2015 and published as the *ECMAScript 2015 Language Specification*.

Even before the publication of ES2015 the comittee had to address the question of how to publish specifications without keeping features from being implemented in browsers until the whole specification was approved as a standard.

Unlike the CSS working group that decided to break features into their own specification or WHATWG that decided to keep HTML as a living standard, TC39 adopted an annual release cycle where new features (those that reach stage 4 of the TC39 process) ready for publication in December are added to the standard and the standard is presented to the general assembly in June of the following year for ratification.

The different stages for an ECMAScript proposals are shown below and further explained in the TC39 process document.

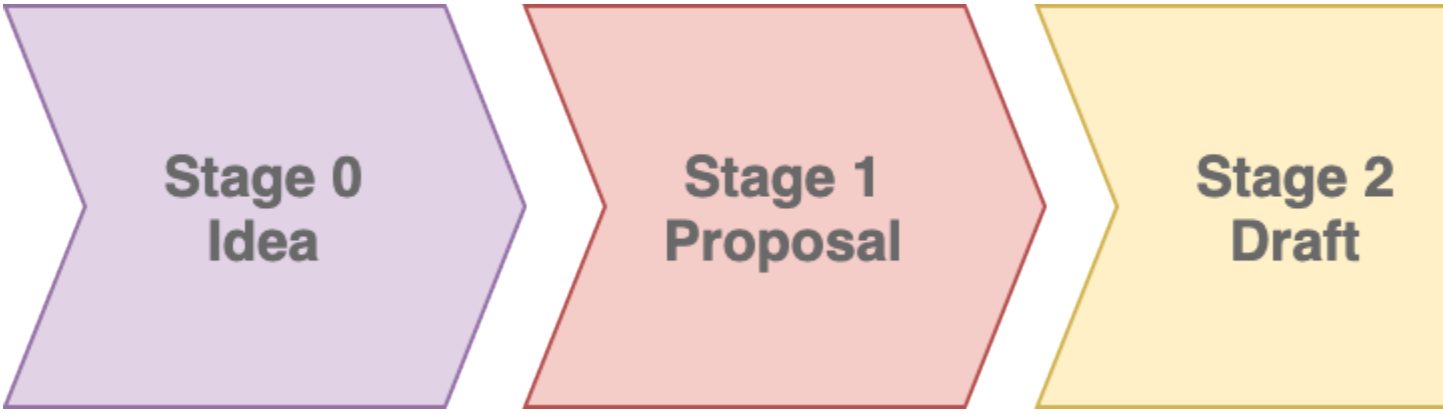

Figure 3: TC39 process divided into 5 stages

One language to rule them all?

It used to be that Javascript was the only way to write code that was guaranteed to run on every browser out there. This was a good thing because it was easy to write and easy to read. It also forced developers to work within the boundaries (good and bad) of the langauge.

But in the early 2000s the web was getting bigger and the need for languages that worked around the limitations of Javascript became necessary to work in these increasingly complex environments. Javasacript went from being the only language to write web applications to becoming the only target compilation languages for creating web applications.

Below is a selection of languages that can output Javascript code. I've listed versions, initial and latest release date along with the language's website for reference.

Language (source)	Version	Website	Initial Release Date	Last Update
CoffeeScript	2.6.1	https://coffeescript.org/	12/13/2009	10/04/2021
Elm	0.19.1	https://elm-lang.org/	03/30/2012	10/21/2019
TypeScript	4.5.2	https://www.typescriptlang.org/	10/01/2012	11/17/2021
Nim	1.6.0	https://nim-lang.org/	2008	
Kotlin	1.6.0	https://kotlinlang.org/	07/22/2011	11/16/2021
Haxe	4.2.4	https://haxe.org/	2005	10/22/2021
PureScript	0.14.5	https://www.purescript.org/	04/27/2014	10/22/2021
Dart	2.15	https://dart.dev	10/10/2011	
Ceylon	1.3.3	https://ceylon-lang.org/	2011	08/21/2017
Opal (Ruby)	1.3.2	https://opalrb.com/	7/7/2011	11/10/2021
Fable (F#)	3.6.3	https://fable.io/	03/31/2017	11/30/2021
GWT (Java)	2.9.0	http://www.gwtproject.org/	05/16/2006	05/02/2020
ClojureScript	1.10.891	https://clojurescript.org/	03/07/2015	12/03/2021
Reason ML (OCaml)	3.6.0	https://reasonml.github.io/	05/16/2016	03/05/2020
Scala.js (Scala)	1.8.0	https://www.scala-js.org/	11/29/2013	12/09/2021
Amber (Smalltalk)	0.29.8	https://amber-lang.net/		02/23/2021

Taking Opal as an example, we write code in Ruby that is then compiled to Javascript.

Install the Opal CLI Ruby Gem:

```
gem install opal
```

Then take the following Ruby code;

```
require 'ostruct'

greeting = OpenStruct.new(
  type: :Hello,
  target: :World,
  source: :Opal
)

puts "#{greeting.type}"#{greeting.type} #{greeting.target} from #{greeting
```

and compile it to Javascript:

```
opal -c hello_world.js.rb > hello_world.js
```

If you run the resulting `hello_world.js` file with Node, you will see the following output:

```
node hello_world.js
# &rarr; Hello World from Opal!
```

While it is possible to use other languages to create Javascript code, it is not always the best solution. Toolk like Opal will add a lot of overhead to the code with Opal-specific libraries necessary to duplicate Ruby code. I would assume this is the same for other languages that compile to Javacript.

WebAssembly all the things?

While it has been possible to write code that will transpile to Javascript for more than a decade it hasn't been possible to write fast code that will run on current

browsers.

A proprietary attempt at creating a sandbox for compiled code was the NaCl / PNaCl client.

> Native Client [NaCl] allows you to harness a client machine's computational power to a fuller extent than traditional web technologies. It does this by running compiled C and C++ code at near-native speeds, and exposing a CPU's full capabilities, including SIMD vectors and multiple-core processing with shared memory.
>
> While Native Client provides operating system independence, it requires you to generate architecture-specific executables (nexe) for each hardware platform. This is neither portable nor convenient, making it ill-suited for the open web.
>
> Source: NaCl and PNaCl

The successor to NaCl was the Portable Native Client (PNaCl).

> PNaCl solves the portability problem by splitting the compilation process into two parts:
>
> 1. compiling the source code to a bitcode executable (pexe), and
> 2. translating the bitcode to a host-specific executable as soon as the module loads in the browser but before any code execution.
>
> This portability aligns Native Client with existing open web technologies such as JavaScript. You can distribute a pexe as part of an application (along with HTML, CSS, and JavaScript), and the user's machine is simply able to run it.
>
> Source: NaCl and PNaCl

The PNaCl architecture was deprecated for operating systems other than CrhomeOS in 2017 in favor of WebAssembly. And as good as it was it locked you to a single group of browsers since Safari and Firefox would not support it.

The first attempt to create a crossplatform way to compile strongly typed languages like C/C++ to Javascript was Asm.js.

Asm.js started as a project by Mozilla and first implemented in Firefox in 2013. It is a strict subset of Javascript designed as a compilation target for C/C++ code using tools like Emscripten.

in 2015, the successor to asm.js, WebAssembly was first announced in 2015 with the MVP release happening in 2017.

Unlike asm.js, WebAseembly is not a subset of Javascript but a separate bytecode format that can be targeted from multiple languages (a list of languages can be found at Awesome WebAssembly Languages)

Also unlike ASM.js, WebAssembly will also run on runtimes outside the browser using the WASI standard when the final version of the specification is released.

Having the tools to run your favorite language in the browser, it is tempting to build full blown applications in the compiled language that will replace Javascript and all the web tooling, isn't it?

Not quite... at least not according to the people who created WebAssembly. According to the WebAssembly FAQ

> Is WebAssembly trying to replace JavaScript? No!
> WebAssembly is designed to be a complement to, not replacement of, JavaScript. While WebAssembly will, over time, allow many languages to be compiled to the Web, JavaScript has an incredible amount of momentum and will remain the single, privileged (as described above) dynamic language of the Web. Furthermore, it is expected that JavaScript and WebAssembly will be used together in a number of configurations:
>
> - Whole, compiled C++ apps that leverage JavaScript to glue things together.
> - HTML/CSS/JavaScript UI around a main WebAssembly-controlled center canvas, allowing developers to leverage the power of web frameworks to build accessible,
> - web-native-feeling experiences. Mostly HTML/CSS/

> JavaScript app with a few high-performance WebAssembly modules (e.g., graphing, simulation, image/sound/video processing, visualization, animation, compression, etc., examples which we can already see in asm.js today) allowing developers to reuse popular WebAssembly libraries just like JavaScript libraries today.
> - When WebAssembly gains the ability to access garbage-collected objects, those objects will be shared with JavaScript, and not live in a walled-off world of their own.

While you can do it, it is not always the best solution. Take, for example, Squoosh an image compression tool running on the web.

Rather than building the full application in C or Rust or any language, they built the core of the application using web technologies and only use WebAssembly to create web-accessible versions of the codec libraries written in C/C++ and Rust.

That's where I see the heaviest use of WebAssembly: writing bottlenecks in your application whether they are performance or functionality related. We'll be able to integrate your favorite libraries, regardless of the language they are written in into your web applications.

There are other examples of WebAssembly usage, like AutoCAD for the web, which is a part of the Autodesk software for the web and games ported from Unreal Engine and Unity... those are heavier examples of games written in C/C++ and C#.

The possibilities are expanded from just the Javascript on the web to doing a lot more with whatever language you want to use and Javascript to glue them together and show it to the user.