HTML

Hyper Text Markup Language
Describes structure of documents displayed on the web.
HTML is not case-sensitive, but some attribute values are.
HTML documents are basically a tree of nodes.

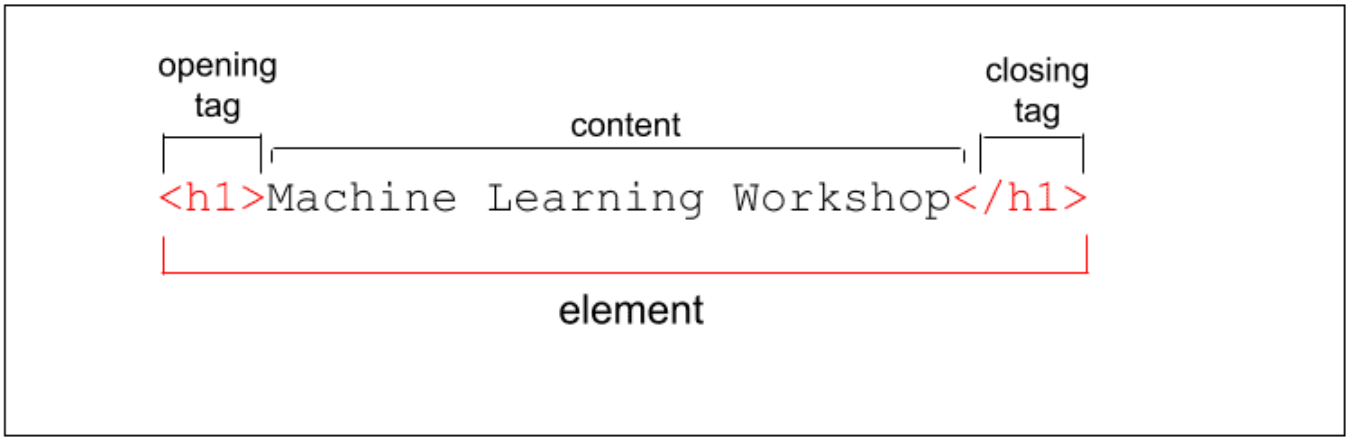
Overview

HTML consists of a series of

- Elements and attributes
- Arranged in a meaningful way

Elements

Provides semantic and formatting for documents.



Non-replaced elements

Have opening and closing tags and may include text and other sub-elements.
Ex: paragraph, header, lists, etc...

Replaced and void elements

Replaced by objects, comes with a default appearance.
Ex: form controls, raster image files, etc...

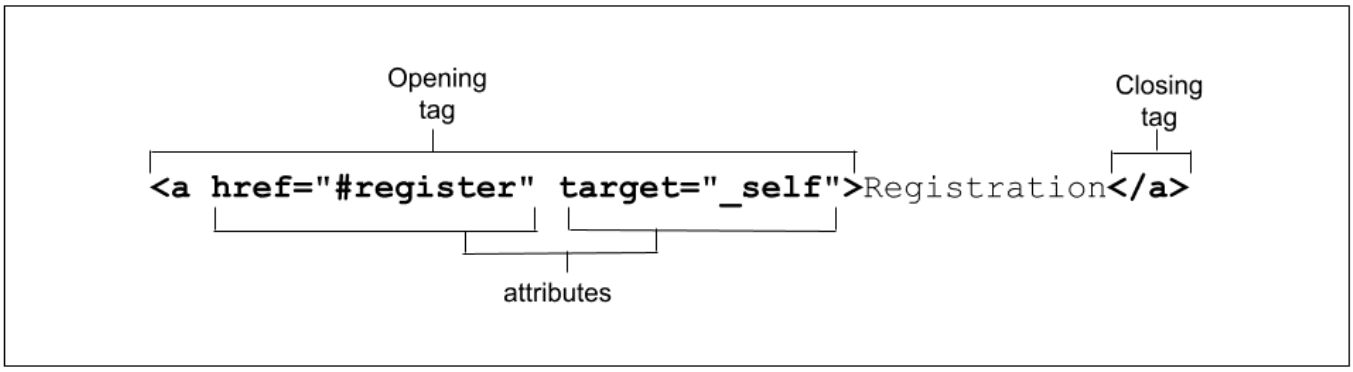
Node

Browser creates JavaScript object for every element and section of text encountered.

- Element nodes
- Text nodes
  HTMLElement interface represents the HTML element and all of its descendant nodes.

Attributes

Extra bits of space-separated name/value pairs.



Document structure

`<!DOCTYPE html>`

It is a doctype special node tells the browser to use standard mode.
If omitted, browsers will use quirks mode(old browsers supporting mode).

`<html>`

Root element for an HTML document.

Content language

The `lang` attribute added to the `<html>` tag defines the main language of the document.

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
  </head>
  <body>
  </body>
</html>
```

Note: `lang` is not just limited to `<html>` tag.

Required components in the `<head>`

The document metadata, including the document title, character set, viewport settings, description, base URL, stylesheet links, and icons, etc...

Character encoding

It is set to ensure the browser can render the characters in the document.

```
<!-- enables the one-to four-byte encoding of all characters -->
<meta charset="utf-8" />
```

Viewport metadata

Helps site responsiveness, enabling content to render well by default, no matter the viewport width.

```
<meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=1" />
```

CSS

Three ways to include CSS: `<link>`, `<style>` and `style` attribute.

`<link>`

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

`<style>`

```
<style>
        @import "styles.css"
</style>
```

Note: The `@import` statement must be the first statement in your `<style>` or linked stylesheet.

```
<style>
        :root {
                --theme-color: #226DAA;
        }
</style>
```

style

```
<style contenteditable style="display: block; font-family: monospace; white-space: pre;">
        head { display: block; }
        head * { display: none; }
        :root {
                --theme-color: #226DAA
        }
</style>
```

Other uses of the `<link>` element

The `link` element is used to create relationships between the HTML document and external resources.
The type of relationship is defined by the value of `rel` attribute. Ex: `icon`, `alternate`, `canonical` and `manifest`

Favicon

Very small icon that appears on the browser tab.
The `<link>` tag with the `rel="icon"` attribute to identify the favicon to be used for your document.

```
<!-- Says use the logo.png as the icon for scenarion where a 16px, 32px, or 48px makes sense. -->
<link rel="icon" sizes="16x16 32x32 48x48" type="image/png" href="/images/logo.png" />
<!-- For iOS devices applied only when the user adds a site to home screen -->
<link rel="apple-touch-icon" size="180x180" href="/images/logo.png" />
<!-- Only be used if the user pins the tab in desktop browser -->
<link rel="mask-icon" href="/images/logo.svg" color="#226DAA" />
```

Alternate versions of the site

The `alternate` value of `rel` attribute to identify translations, or alternate representations, of the site.

```
<link rel="alternate" href="https://www.example.com/fr" hreflang="fr-FR" />
<link rel="alternate" href="https://www.example.com/pt" hreflang="pt-BR" />
```

Note: When using `alternate` for a translation, the `hreflang` attribute must be set.

The `type` attribute can define the alternate URI for and RSS feed.

```
<link rel="alternate" type="application/x-pdf"
href="https://www.example.com/sample.pdf" />
```

If the `rel` value is `alternate stylesheet` the `title` attribute must be set giving that alternate style a name.

```
<link href="basic.css" rel="alternate stylesheet" title="Basic" />
```

Canonical

Use `rel="canonical"` to identify the preferred URL for the site or application.

```
<link rel="canonical" href="https://example.com" />
```

Scripts

The `<script>` tag is used to include, encapsulate or download scripts.

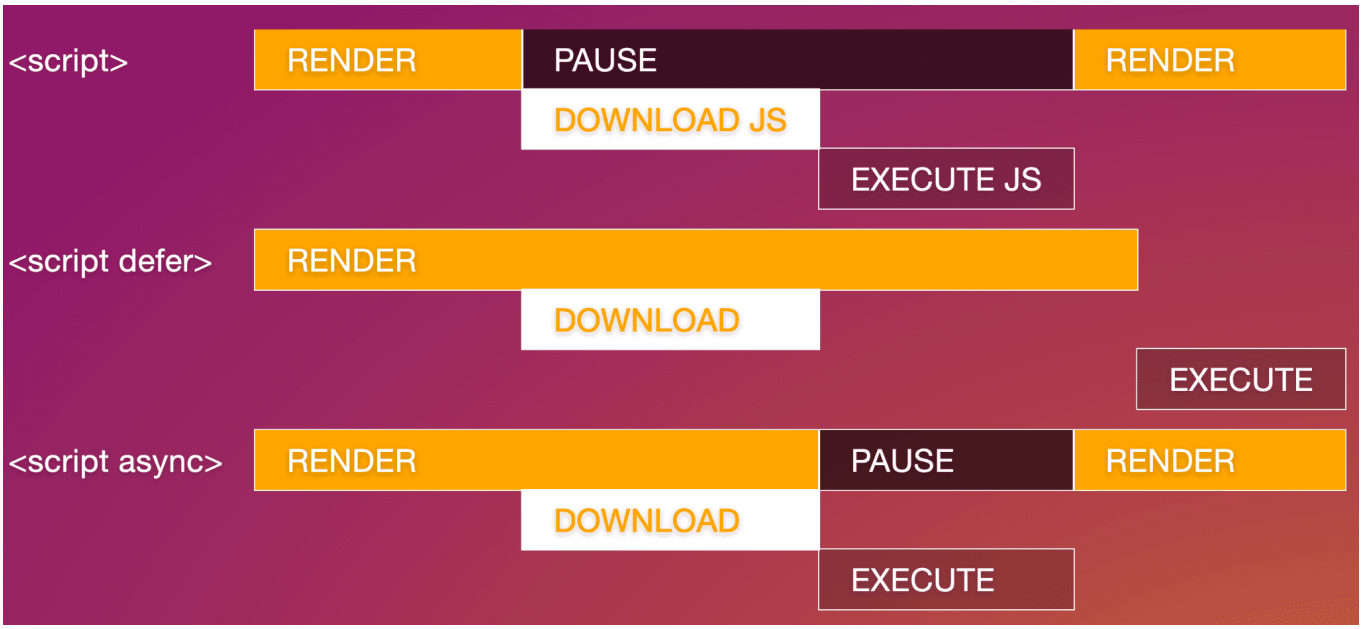
```
<script>
        document.getElementById('element_id').addEventListener('click',
function() {
                document.body.classList.toggle('black');
        });
</script>
```

Note: we add the above `<script>` at the bottom of the `<body>` rather than in the `<head>` to ensure elements exist before the script referencing them.

Parser-blocking JavaScript

It refers to JavaScript code that prevents the browser from parsing, rendering HTML document and firing `DOMContentLoaded` event until the script has been downloaded and executed. This can lead to performance issues.

There are two attributes that can reduce the blocking nature of JavaScript download and execution: `defer` and `async`



`defer` - HTML rendering is not blocked during download, and the JavaScript only executes after the document has otherwise finished rendering.
`async` - HTML rendering is not blocked during the download either, but once the script has finished downloading, the rendering is paused while the JavaScript is executed.

```
<script src="js/main.js" async type="module"></script>
```

Base

The `<base>` Used to specify a base URL for all relative URLs in a document. The `target` attribute of `<base>` tag specifies the default target for all hyperlinks in the document. The default value of `target` attribute is `_self` and other values are `_blank`, `_top` and `_parent`.

```
<!-- By default open in the same tab -->
<base href="https://www.example.com/" target="_self">

<!-- By default open in a new tab -->
<base href="https://www.example.com/" target="_blank">

<!-- By default open in the parent frame -->
<base href="https://www.example.com/" target="_parent">

<!-- By default open in the full window -->
<base href="https://www.example.com/" target="_top">
```

```
<!DOCTYPE html>
<html>
        <head>
```

```
                <meta charset="UTF-8" />
                <meta name="viewport" content="width=device-width, initial-
scale=1.0, user-scalable=1" />
                <title>Example</title>
                <!-- By default open in the same tab -->
                <base href="https://www.example.com/">
        </head>
        <body>
                <h1>Hi there!</h1>
                <!-- Resolves to https://www.example.com/about.html -->
                <a href="about.html">About</a>
                <!-- Resolves to https://www.example.com/images/logo.png -->
                <img src="images/logo.png" alt="Logo">
        </body>
</html>
```

Metadata

The `<meta>` tag represents "meta data" of the document that can't be represented by other elements.

Officially defined meta tags

There are two main types of meta tags: Pragma directives and Named meta types.

Pragma directives

These directives describe how the page should be parsed. Supported `http-equiv` values enable setting directives when you are unable to set HTTP headers directly.

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="refresh" content="60; https://example.com/signupTimeout"
/>
<meta http-equiv="content-security-policy" content="default-src https:" />
```

Named meta tags

Useful for Search Engine Optimization (SEO).

Description

Displayed under page title in search engine search results.

```
<meta name="description" content="Description of example.com" />
```

Robots

Tells search engine bots to whether to index the page or not.

```
<meta name="robots" content="noindex, nofollow" />
```

Theme color

Let the user define a color to customize the browser interface. The site color will be found immediately which may be faster on first load than waiting for manifest to load.

```
<meta name="theme-color" content="#226DAA" />
```

Open Graph

Used to control how social media sites display links to your content. If not included, social media sites will grab the page title and description.

```
<meta property="og:title" content="Website title" />
<meta property="og:description" content="You can find description here." />
<meta property="og:image" content="http://example.com/image/logo.png" />
<meta property="og:image:alt" content="Example logo" />
```

Other social media have their own similar syntaxes, like Twitter card markup.

```
<meta name="twitter:title" content="Website title" />
<meta name="twitter:description" content="You can find description here." />
<meta name="twitter:url" content="https://www.example.com/?src=twitter" />
<meta name="twitter:image:src"
content="http://www.example.com/image/logo.png" />
<meta name="twitter:image:alt" content="Example logo" />
<meta name="twitter:creator" content="@kumar" />
<meta name="twitter:site" content="@example" />
```

Other useful meta information

If your website has web-app capabilities you can use `<link>` and `<meta>` tags to provide native app like user experience.

Use `<link>` to provide icon and name for your web-app on the mobile device's home screen.

```
<!-- App icon -->
<link rel="apple-touch-startup-image" href="icons/ios-portrait.png"
media="orientation: portrait" />
<link rel="apple-touch-startup-image" href="icons/ios-landscape.png"
media="orientation: landscape" />

<!-- App name -->
<meta name="apple-mobile-web-app-title" content="Example" />
<meta name="mobile-web-app-title" content="Example" />
```

Use named `<meta>` to tell the browser that the web-app can stand on its own with minimal UI, such as no back button.

```
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="mobile-web-app-capable" content="yes" />
```

Note: Include this if your app is indeed app-capable.

Manifest

The manifest file can prevent an unwiedly header full of `<link>` and `<meta>` tags. Use the handy `<link>` tag with a `rel` attribute set to `manifest` and the `href` attribute set to the URL of the manifest file.

```
<link rel="manifest" href="/example.webmanifest">
```

Constructing the Object model

Before the browser can render the web page, it constructs the document object model (DOM) and CSS object model (CSSOM) trees. DOM and CSSOM are independent data structures.

Document object model (DOM)

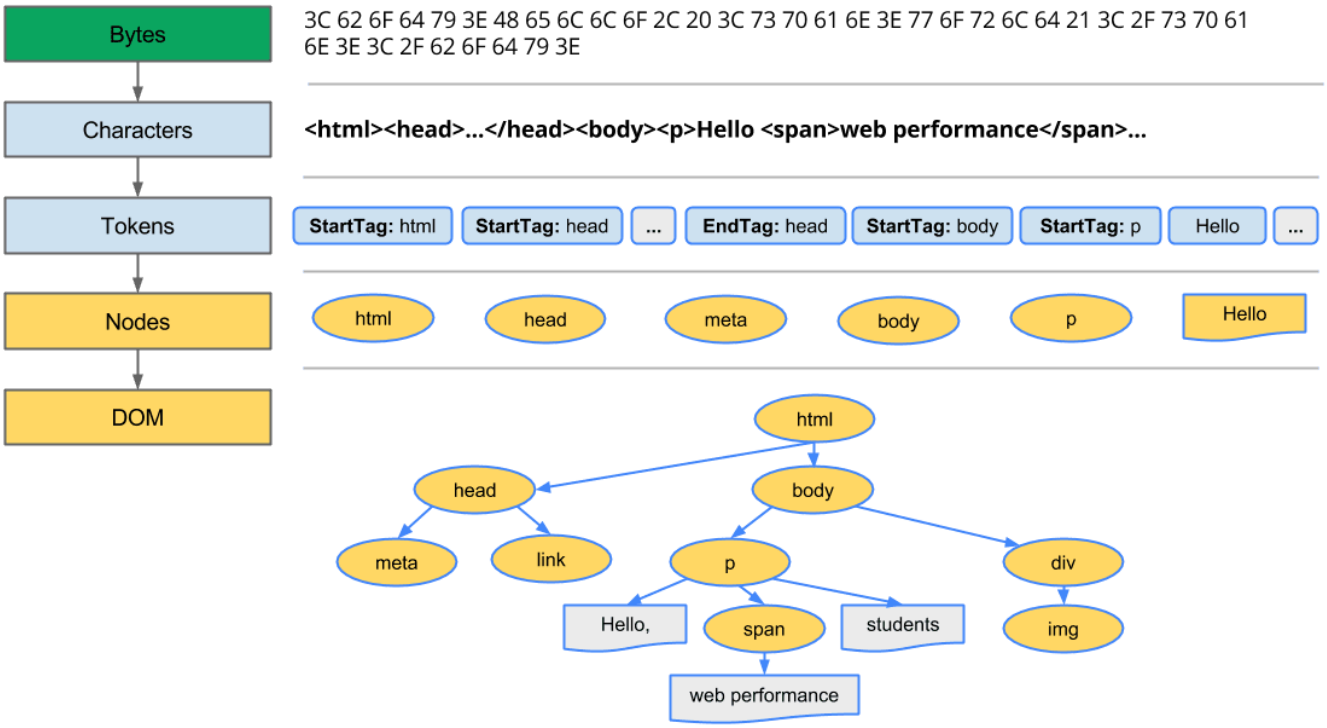
Let's start with the simplest possible case.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1,
```

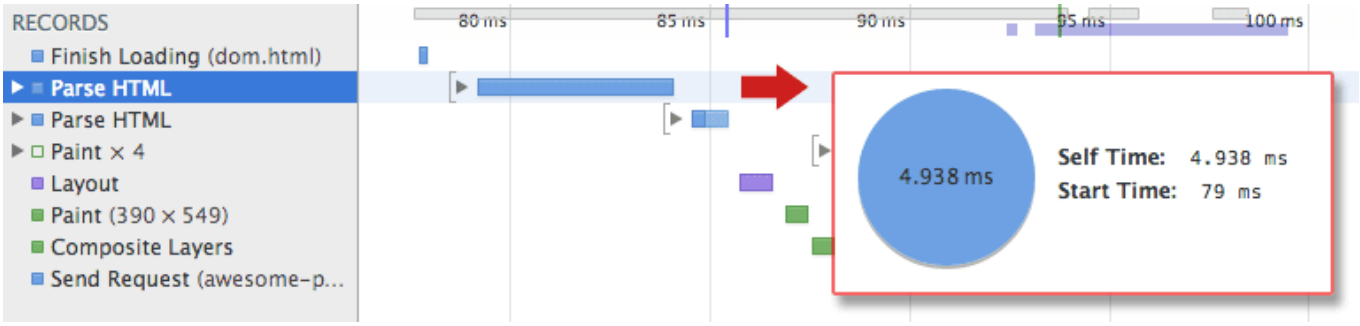
```
user-scalable=1" />
    <link href="style.css" rel="stylesheet" />
    <title>Example</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="photo.jpg" /></div>
  </body>
</html>
```

Every time the browser processes HTML markup, it goes through all of the previously defined steps: convert bytes to characters, identify tokens, convert tokens to nodes, and build the DOM tree.



1. **Conversion**: The browser reads the raw bytes of HTML and translates them into individual characters based on specific encoding of the file (for example, UTF-8).
2. **Tokenizing**: The browser converts strings of characters into distinct tokens—as specified by the W3C HTML5 standard for example, `<html>`, `<body>`—and other strings within angle brackets. Each token has a special meaning and its own set of rules.
3. **Lexing**: The emitted tokens are converted into "objects," which define their properties and rules.
4. **DOM construction**: Finally, because the HTML markup defines relationships between different tags (some tags are contained within other tags) the created objects are linked in a tree data structure that also captures the parent-child relationships defined in the original markup.

If you open up Chrome DevTools and record a timeline while the page is loaded, you can see the actual time taken to perform this step.



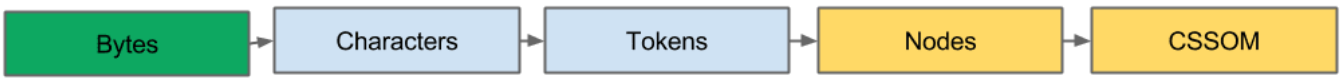
CSS object model (CSSOM)

While the browser was constructing the DOM of our basic page, it encountered a `<link>` element in the `<head>` of the document referencing an external CSS style sheet.

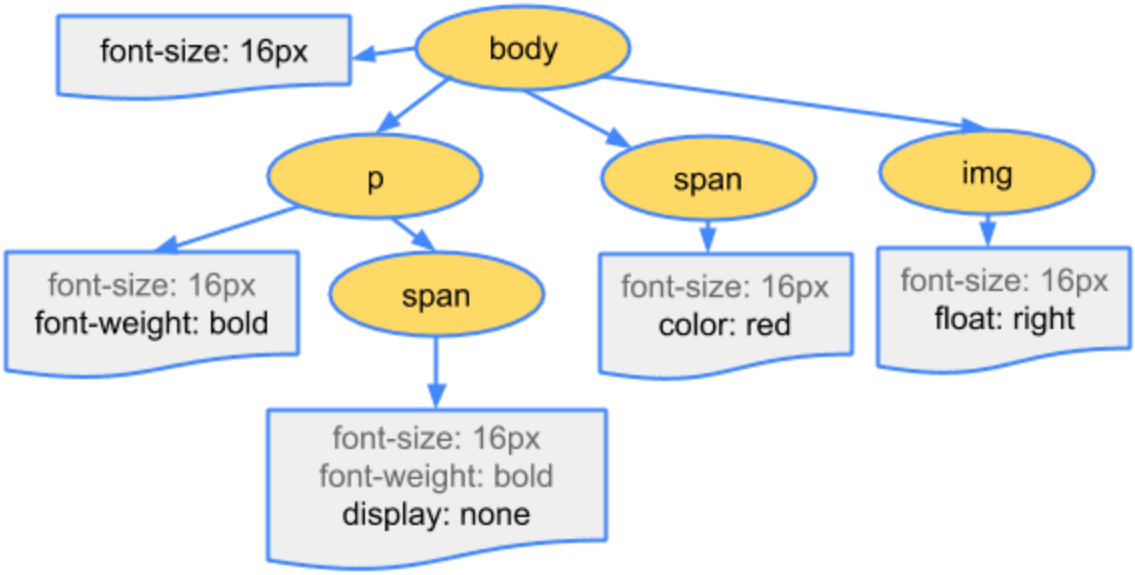
style.css

```
body {
  font-size: 16px;
}

p {
  font-weight: bold;
}

span {
  color: red;
}

p span {
  display: none;
}

img {
  float: right;
}
```

As with HTML, we need to convert the received CSS rules into something that the browser can understand and work with.

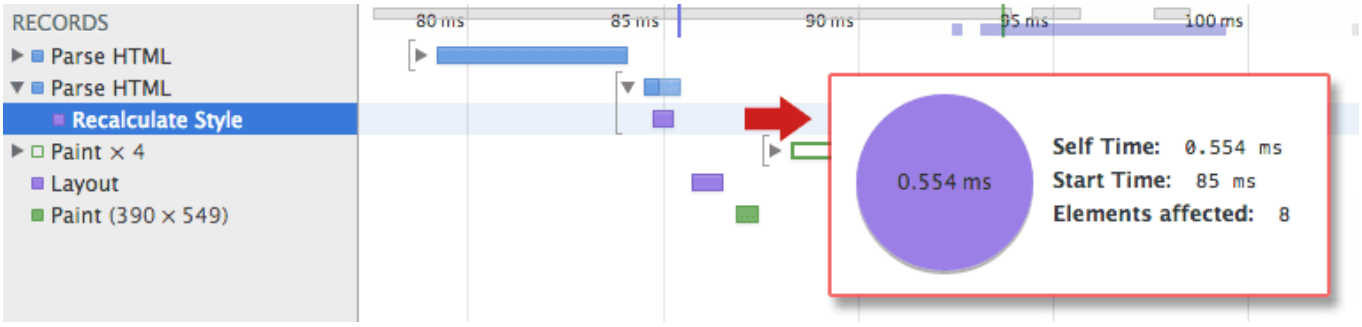


The CSS bytes are converted into characters, then tokens, then nodes, and finally they are linked into a tree structure known as the "CSS Object Model" (CSSOM):



Note: The tree described is not the complete CSSOM tree and only shows the styles we decided to override in our style sheet.

To find out how long the CSS processing takes you can record a timeline in DevTools and look for "Recalculate Style" event.



Semantic HTML

Semantic elements provide machine-readable structure to parse and interpret content. Writing semantic HTML elements to structure your content based on each element's meaning, not its appearance.

document1.html

```
<div>
  <span>Three words</span>
  <div>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
```

```
    <a>one word</a>
  </div>
</div>
<div>
  <div>
    <div>five words</div>
  </div>
  <div>
    <div>three words</div>
    <div>forty-six words</div>
    <div>forty-four words</div>
  </div>
  <div>
    <div>seven words</div>
    <div>sixty-eight words</div>
    <div>forty-four words</div>
  </div>
</div>
<div>
   <span>five words</span>
</div>
```

document2.html

```
<header>
  <h1>Three words</h1>
  <nav>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
  </nav>
</header>
<main>
  <header>
    <h1>five words</h1>
  </header>
  <section>
    <h2>three words</h2>
    <p>forty-six words</p>
    <p>forty-four words</p>
  </section>
  <section>
    <h2>seven words</h2>
    <p>sixty-eight words</p>
```

```
    <p>forty-four words</p>
  </section>
</main>
<footer>
  <p>five words</p>
</footer>
```

Accessibility object model (AOM)

Accessibility Object Model provides a way to expose the accessibility properites of web content in a structured way.

Let's compare how both of these document structures are rendered in the Firefox accessibility panel.

document1.html

document2.html

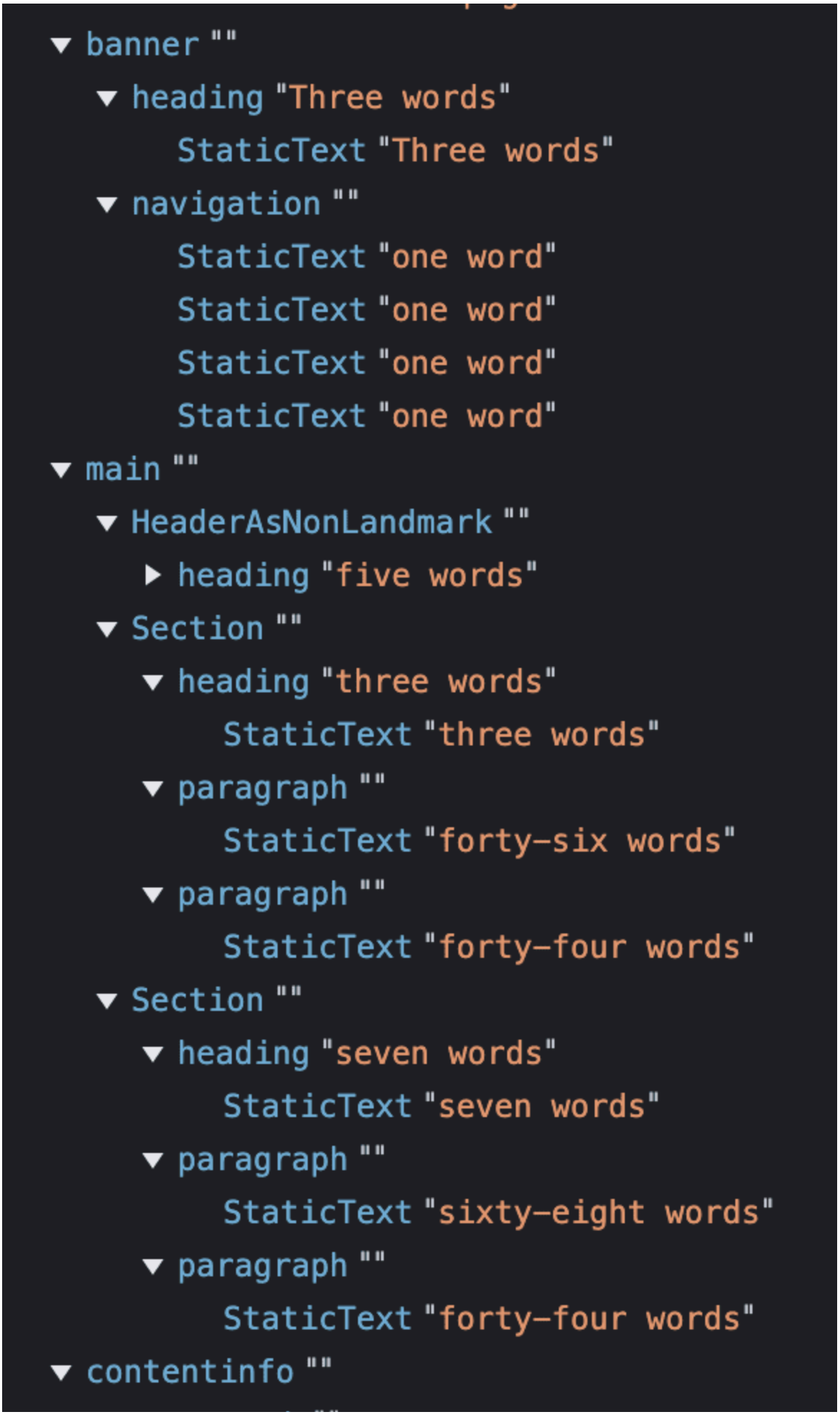
Landmarks provide structure to web content and ensure important sections of content are keyboard-navigable for screen reader users.

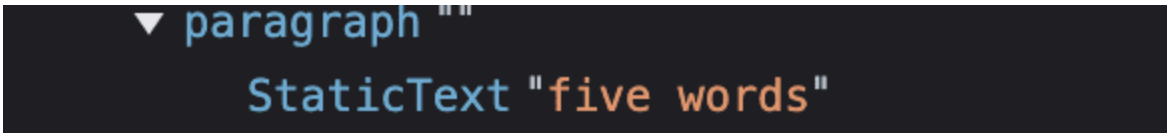
Chrome's AOM shows:

document1.html

```
StaticText "Three words"
StaticText "one word"
StaticText ""
StaticText "one word"
StaticText ""
StaticText "one word"
StaticText ""
StaticText "one word"
StaticText "five words"
StaticText "three words"
StaticText "forty-six words"
StaticText "forty-four words"
StaticText "seven words"
StaticText "sixty-eight words"
StaticText "forty-four words"
StaticText "five words"
```

document2.html


banner ""
heading "Three words"
StaticText "Three words"
navigation ""
StaticText "one word"
StaticText "one word"
StaticText "one word"
StaticText "one word"
main ""
HeaderAsNonLandmark ""
heading "five words"
Section ""
heading "three words"
StaticText "three words"
paragraph ""
StaticText "forty-six words"
paragraph ""
StaticText "forty-four words"
Section ""
heading "seven words"
StaticText "seven words"
paragraph ""
StaticText "sixty-eight words"
paragraph ""
StaticText "forty-four words"
contentinfo ""



It's pretty clear that semantic element usage helps accessibility, and using non-semantic elements reduces accessibility.

The role attribute

The `role` attribute describes the role an element has in the context of the document. The `role` attribute is a global attribute—meaning it is valid on all elements. Element role names are important in building the AOM.

in the Firefox accessibility dev tools screenshot, the top level `<header>`, `<main>`, `<footer>`, and `<nav>` were all landmarks, while the `<header>` nested in `<main>` was a section.

The Chrome lists these elements' `<main>` is `main`, `<nav>` is `navigation`, and `<footer>`, as it was the footer of the document, is `contentinfo`. When `<header>` is the header for the document, the default role is `banner`, which defines the section as the global site header. When a `<header>` or `<footer>` is nested within a sectioning element, it is not a landmark role.

Using the `role` attribute, you can give any element a role, including a different role than the tag implies.

```
<p role="button">Click Me</p>
```

While adding `role="button"` to an element will inform screen readers that the element is a button, it doesn't change the appearance or functionality of the element.

We can make the non-semantic version semantic by assigning each element a role:

```
<div role="banner">
  <span role="heading" aria-level="1">Three words</span>
  <div role="navigation">
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
  </div>
</div>
```

Semantic elements

The appropriate element which best represents the functionality is called semantic element. Choosing the right elements for the job as you code means you won't have to refactor or comment your HTML.

Attributes

Attributes are space-separated names and name/value pairs appearing in the opening tag, providing information about and functionality for the element.

While HTML is not case-sensitive, some attribute values are. Values that are part of the HTML specification are case-insensitive. Strings values that are defined, such as class and id names, are case-sensitive.

Boolean attributes

Boolean attributes include `autofocus`, `inert`, `checked`, `disabled`, `required`, `reversed`, `allowfullscreen`, `default,` `loop`, `autoplay`, `controls`, `muted`, `readonly`, `multiple,` and `selected`.

If the attribute value is false, omit the attribute. If the attribute is true, include the attribute but don't provide a value. For example, `required="required"` is not a valid value in HTML; but as `required` is boolean, invalid values resolve to true.

```
<!-- false -->
<input required>
<!-- true -->
<input required="">
<!-- true -->
<input required="required">
```

When toggling between true and false, add and remove the attribute altogether with JavaScript rather than toggling the value.

```
const myMedia = document.getElementById("mediaFile");
myMedia.removeAttribute("muted");
myMedia.setAttribute("muted");
```

Enumerated attributes

Enumerated attributes are HTML attributes that have a limited set of predefined valid values. For example, if you include `<style contenteditable>`, it defaults to `<style contenteditable="true">`.

Omitting the attribute doesn't mean it's false. example, `contenteditable` defaults to `inherit` if missing or invalid, and can be explicitly set to `false`. If the value is invalid, such as `<style contenteditable="contenteditable">`, the value is invalid and defaults to `inherit`.

Global attributes

Global attributes are attributes that can be set on any HTML element, including elements in the `<head>`. For example, setting `hidden` on a `<meta>` as meta content is not displayed.

id

The global attribute `id` is used to define a unique identifier for an element.

link fragment identifier

inks are not restricted to HTTP-based URLs; they can be fragment identifiers to sections of the page in the current document (or in other documents).

```
<nav>
  <a href="#reg">Register</a>
  <a href="#about">About</a>
  <a href="#teachers">Instructors</a>
  <a href="#feedback">Testimonials</a>
</nav>
```

scripting

```
<img src="svg/switch2.svg" id="switch" alt="light switch" class="light" />
```

```
const switchViaID = document.getElementById("switch");
const switchViaSelector = document.querySelector("#switch");
```

one JavaScript function makes use of this ability to target elements by their `id` attribute.

```
<script>
  /* switch is a reserved word in js, so we us onoff instead */
  const onoff = document.getElementById('switch');
  onoff.addEventListener('click', function(){
    document.body.classList.toggle('black');
  });
</script>
```

`<label>`

The HTML `<label>` element has a `for` attribute that takes as its value the `id` of the form control with which it is associated.

```
<label for="minutes">Send me a reminder</label>
<input type="number" name="min" id="minutes">
<label for="minutes">before the workshop resumes</label>
```

The association between `for` and `id` extends the control's click area, clicking anywhere on a label gives focus to the associated element.

other accessibility uses

`aria-*` attributes can be used to ensure accessibility. For example `aria-activedescendant`, which identifies the currently focused descendant element, takes as its value a single `id` reference. The `aria-labelledby` attribute identifies the element (or elements) that labels the element it is applied to.

```
<section id="about" aria-labelledby="about_heading">
<h2 id="about_heading">What you'll learn</h2>
```

class

The `class` attribute provides an additional way of targeting elements with CSS (and JavaScript), but serves no other purpose in HTML (though frameworks and component libraries may use them).

tabindex

The `tabindex` attribute can be added to any element to enable it to receive focus.
The `tabindex` attribute takes as its value an integer. A negative value (the convention is to use `-1`) makes an element capable of receiving focus, such as via JavaScript, but does not add the element to the tabbing sequence.

Custom attributes

You can create any custom attribute you want by adding the `data-` prefix.

```
<blockquote data-python-learning="workshop"
  data-first-name="Blendan" data-last-name="Smooth"
  data-formerly="Margarita Maker" data-aspiring="Load Balancer"
  data-year-graduated="2022">
```

```
  HAL and EVE could teach a fan to blow hot air.
</blockquote>
```

browsers won't implement default behaviors for any specific `data-`. Add custom attributes to elements in the form of `data-name` and access these through the DOM using `dataset[name]`.

```
<blockquote data-python-learning="workshop"
  data-first-name="Blendan" data-last-name="Smooth"
  data-formerly="Margarita Maker" data-aspiring="Load Balancer"
  data-year-graduated="2022">
  HAL and EVE could teach a fan to blow hot air.
</blockquote>
```

```
el.dataset["pythonLearning"]; // workshop
e.dataset.pythonLearning; // workshop

for (let key in el.dataset) {
  customObject[key] = el.dataset[key];
}
```

Forms

The HTML `<form>` element identifies a document landmark containing interactive controls for submitting information. Nested in a `<form>` you'll find all the interactive (and non-interactive) form controls that make up that form.

Submitting forms

```
<form method="GET">
        <input type="submit" name="student" value="Submit Form">
        <button type="submit">Submit Form</button>
</form>
```

The data sent is made up of name/value pairs of the form's various form controls. By default, this includes all the form controls nested within the form that have a `name`. However, with the `form` attribute, it is possible to include form controls outside the `<form>` and to omit form controls nested within the `<form>`. Supported on form controls and `<fieldset>`, the `form` attribute takes as its value the `id` of the form the control it is associated with, not necessarily the form it is nested in.

The `method` attribute defines the HTTP protocol of the request: generally `GET` or `POST`.

With `GET`, the form data is sent as a parameter string of `name=value` pairs, appended to the `action`'s URL.

With `POST`, the data is appended to the body of the HTTP request. When sending secure data, such as passwords and credit card information, always use `POST`.

If a `<form method="dialog">` is within a `<dialog>`, submitting the form will close the dialog.

After submitting the form

When the user submits a completed online form, the names and values of the relevant form controls get submitted. The name is the value of the `name` attribute. The values come from the contents of the `value` attribute or the value entered or picked by the user.

```
<form method="GET">
  <label for="student">Pick a student:</label>
  <select name="student" id="student">
    <option value="hoover">Student 1</option>
    <option>Student 2</option>
    <option value="toasty">Student 3</option>
  </select>
  <input type="submit" value="Submit Form">
</form>
```

Submitting the form will reload the page, setting the URL to:

```
https://web.dev/learn/html/forms?student=hoover
```

Because the second option doesn't have a `value` attribute, the inner text is submitted as the value.

```
https://web.dev/learn/html/forms?student=Blendan+Smooth
```

`<fieldset>`

The `<fieldset>` HTML element is used to group several controls as well as labels within a web form.

```
<fieldset>
  <legend>Your favorites:</legend>
  <ul start="6">
    <li>
```

```
      <fieldset>
        <legend>Who is your favorite student?</legend>
        <ul>
          <li>
            <!-- the rest of the code here -->
```

In addition to all the global attributes, `<fieldset>` also supports the `name`, `disabled`, and `form` attributes. When you disable a fieldset, all nested form controls are disabled.

Built-in validation

Without including any JavaScript, HTML can prevent forms with invalid values from being submitted. The browser supplied error messages stemming from errors based on the `required`, `pattern`, `min`, `max`, `type`, `minlength`, and `maxlength` attributes.

The **`ValidityState`** interface represents the *validity states* that an element can be in, with respect to constraint validation. Together, they help explain why an element's value fails to validate, if it's not valid. You can also set styles based on the validity states using CSS pseudo-class.

```
form:invalid [type="submit"] {
  opacity: 50%;
  pointer-events: none;
}
```

If any of the `validityState` properties return `true`, submission is blocked, and the browser displays an error message in the first incorrect form control, giving it focus.

Example:

```
<!-- index.html -->
<dialog open aria-labelledby="dialogid">
  <form action="thankyou.html">
    <button type="submit" aria-label="close" formmethod="dialog"
formnovalidate>X</button>
    <h2 id="dialogid">Application</h2>
    <p>All fields are required</p>
    <p>
       <label>Name:
         <input type="text" name="name" required />
      </label>
    </p>
    <p>
      <label>Warranty:
```

```
        <input type="number" min="0" max="10" name="warranty" required />
       </label>
    </p>
    <p>
      <label>Power source:
        <select name="powersoure">
          <option>AC/DC</option>
          <option>Battery</option>
          <option>Solar</option>
        </select>
      </label>
    </p>
    <p>
      <button type="submit" formmethod="post">Submit</button>
    </p>
  </form>
</dialog>
```

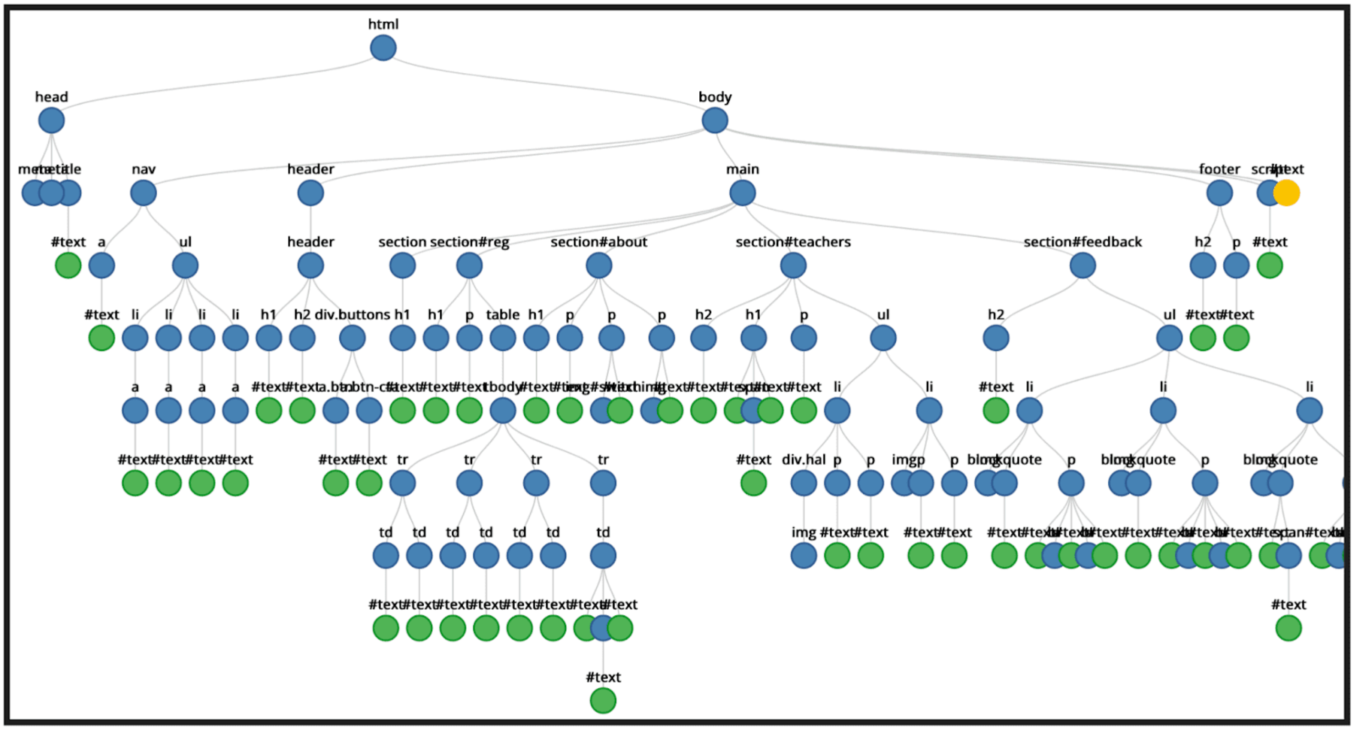
```
/* style.css */
body {
  background:lightpink;
  font-family: sans-serif;
}
[aria-label="close"] {
  float: right;
  appearance: none;
  border-radius: 5rem;
  margin: -1em;
}
p {
  margin-top: 1em;
}
h2 + p {
  margin-top: -1em;
}
label {
  display: flex;
  flex-direction: column;
}
```

HTML APIs

HTML elements are the nodes that make up the Document Object Model.

The DOM and AOM

The DOM is an API for accessing and manipulating documents. The DOM is the tree of all the nodes in the document. Some nodes can have children, others can't. The tree includes elements, along with their attributes, and text nodes.



Browser tools don't provide tree visualizations like the one above, but you can see the nodes in the element inspector.

```
<html lang="en-US"> scroll
▶ <head> ··· </head>
▼ <body>
  ▶ <nav> ··· </nav> flex
  ▼ <header>
    ▼ <header>
        <h1>Machine Learning Workshop</h1>
      ▶ <h2> ··· </h2>
      ▼ <div class="buttons"> flex
          <a class="btn" href="https://perfmattersconf.com">Teach a Machine*</a>
          <a class="btn-cta" href="#reg">Register Now</a>
        </div>
      </header>
    </header>
  ▼ <main>
    ▶ <section> ··· </section> flex
    ▶ <section id="reg" class="reg"> ··· </section> flex
    ▶ <section id="about"> ··· </section> flex
    ▶ <section id="teachers" class="teachers"> ··· </section> flex
    ▼ <section id="feedback" class="feedback"> flex
      ▶ <h2> ··· </h2> overflow
      ▼ <ul> flex
        ▼ <li> flex
            <img src="images/blender.svg" alt="Blender"> overflow
          ▶ <blockquote> ··· </blockquote> overflow
          ▶ <p> ··· </p> overflow
          </li>
        ▶ <li> ··· </li> flex
        ▶ <li> ··· </li> flex
        </ul>
      </section>
    </main>
  ▶ <footer> ··· </footer> flex
  ▶ <script> ··· </script>
  </body>
</html>
```

The AOM is based off the DOM; similarly, the accessibility tree contains objects representing all the markup elements, attributes, and text nodes.

HTML Elements APIs

Every node in the document tree is an object that can be manipulated with JavaScript. The browser provides numerous APIs providing natively supported methods, events, and property querying and updating. Element nodes contain information about all the attributes set on the element. You can use HTML interfaces to access information about an element's attributes.

```
let allImages = document.querySelectorAll('img');
allImages.forEach((imageInstance) => {
  console.log(imageInstance.alt);
});
```

HTML interfaces provide more than just access to an element's attributes; you can access a lot more information. We can find the read-only `HTMLElement.offsetHeight` to get the height of each section in our page, relative to the layout.

```
let allSections = document.querySelectorAll('section');
allSections.forEach((sectionInstance) => {
  console.log(sectionInstance.offsetHeight);
});
```

The HTML interface APIs is not limited to accessing attribute values. The DOM provides insight into the current state of the UI. HTML APIs can access all of that information. You can access the length of a video, where a view is in the current playback, and if the video (or audio) has finished playing with `HTMLMediaElement.duration`, `HTMLMediaElement.currentTime`, and `HTMLMediaElement.ended` respectively.

Available element interfaces

Most HTML element we've covered thus far in this series and have yet to cover, other than some sectioning elements, have an associated DOM interface. The base interface for all elements is aptly named `Element`. The `HTMLElement` inherits from Element and all HTML element-specific interfaces inherit from it.

The interfaces include:

- `HTMLAnchorElement` - `<a>`
- `HTMLButtonElement` - `<button>`
- `HTMLDialogElement` - `<dialog>`
- `HTMLFormElement` - `<form>`
- `HTMLHtmlElement` - `<html>`
- `HTMLImageElement` - `<img>`
- `HTMLInputElement` - `<input>`
- `HTMLLabelElement` - `<label>`
- `HTMLLinkElement` - `<link>`
- `HTMLMapElement` - `<map>`
- `HTMLMediaElement` - `<audio>`, `<video>`
- `HTMLScriptElement` - `<script>`
- `HTMLStyleElement` - `<style>`
- `HTMLTableElement` - `<table>`
- `HTMLTableRowElement` - `<tr>`
- `HTMLTableSectionElement` - `<thead>`, `<tbody>`, `<tfoot>`
- `HTMLTextAreaElement` - `<textarea>`
- `HTMLTimeElement` - `<time>`
- `HTMLTitleElement` - `<title>`
- `HTMLTrackElement` - `<track>`
- `HTMLVideoElement` - `<video>`

Other interfaces

There are additional interfaces that enable manipulating the branch locations of DOM nodes. The `EventTarget` interface, which provides us with `addEventListener()` and `removeEventListener()`, is inherited by the `Node` and `Window` interfaces. In turn, the Element, Document, and DocumentFragment interfaces inherit from Node, and the HTMLElement interface inherits from Element.

The node interface

Every type of DOM node is represented by an interface based on `Node`, which provides information and methods as elements relate to the DOM tree. The `Node` interface enables querying and adding nodes to the node tree.

The Document

The Window interface includes globally available items beyond the DOM that can be used to manipulate the DOM. Window provides functions, namespaces, objects, and constructors documented in MDN's JavaScript and DOM References.

The Window interface is the API for the object containing the document. The global `window` object is the window in which the script is running. Every browser tab contains its own Window object. The Window interface can query the contents of the tab as well as the overall window and device. For example, the `resizeTo()` method can be used to resize the browser window, the `devicePixelRatio` property provides access to the device display pixels. When accessing information about the tab the content is in rather than the DOM tree the tab displays, the window is likely the interface you're looking for.