Document Object Model (DOM)

The Document Object Model (DOM) **connects web pages to scripts or programming languages by representing the structure of a document** — such as the HTML representing a web page — in memory. Usually, it refers to JavaScript, even though modeling HTML, SVG, or XML documents as objects are not part of the core JavaScript language.

The DOM represents a document with a logical tree. **Each branch of the tree ends in a node, and each node contains objects**. **DOM methods allow programmatic access to the tree**. With them, you can change the document's structure, style, or content.

Nodes can also have event handlers attached to them. Once an event is triggered, the event handlers get executed.

DOM interfaces

DOM interfaces, also known as DOM APIs or DOM objects, are a **standardized set of programming tools provided by web browsers to allow developers to interact with web documents, such as HTML, XML, XHTML, or SVG files**, in a structured way.

The DOM is organized into different levels, each representing a specific version of the DOM specification:

1. DOM Level 1: The first level provided basic functionality for accessing and modifying the elements in an HTML or XML document. It laid the foundation for the subsequent DOM levels.
2. DOM Level 2: Building upon the foundation of Level 1, this level introduced additional interfaces and features. It added support for event handling, allowing developers to respond to user interactions like clicks and keyboard input. DOM Level 2 also included enhanced navigation and manipulation capabilities.
3. DOM Level 3: This level further extended the DOM by adding support for advanced features such as the ability to use XPath expressions to navigate and select elements in a document. Additionally, it included improvements to event handling and introduced more precise control over the document.

These DOM interfaces represent the elements of a web page as a tree-like structure, where each element is a node in the tree. Developers can use methods and properties provided by these interfaces to traverse the tree, read and modify content, create new elements, and respond to user actions and events. This enables developers to build dynamic and interactive web

applications by manipulating the content and appearance of web pages in real-time using JavaScript or other scripting languages. The different DOM levels ensure that browsers adhere to standardized APIs, allowing developers to write consistent and cross-browser compatible code.

HTML DOM

HTML DOM (Document Object Model) is a programming interface provided by web browsers that represent the structure and content of an HTML document as a tree-like structure of objects. It allows developers to interact with web pages dynamically using JavaScript or other scripting languages.

Key points about HTML DOM:

1. **Tree-like Structure**: The DOM represents an HTML document as a tree of objects, where each HTML element is a node in the tree. The "document" object represents the entire HTML page and serves as the root of the tree. All other elements (e.g., headings, paragraphs, images) are its children, forming a hierarchical structure.
2. **Access and Manipulate Elements**: With the HTML DOM, developers can access specific elements in the HTML document using methods like **getElementById**, **getElementsByTagName**, **getElementsByClassName**, **querySelector**, etc. Once accessed, they can modify the elements' attributes, content, or even add and remove elements dynamically.
3. **Dynamic Interaction**: The HTML DOM enables developers to create interactive web pages by adding event listeners to elements. This way, they can respond to user actions (e.g., clicks, keypresses) and trigger appropriate actions or functions.
4. **Cross-Browser Compatibility**: The HTML DOM provides a standardized way to interact with HTML documents, ensuring that code works consistently across different web browsers.

By utilizing the HTML DOM, developers can build powerful and dynamic web applications, enhance user experience, and create responsive and interactive content on web pages.

Introduction to the DOM

What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content.

The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

For example, the DOM specifies that the `querySelectorAll` method in this code snippet must return a list of all the `<p>` elements in the document:

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

All of the properties, methods, and events available for manipulating and creating web pages are organized into objects. For example, the document object that represents the document itself, any table objects that implement the `HTMLTableElement` DOM interface for accessing HTML tables, and so forth, are all objects.

The DOM is built using multiple APIs that work together. The core DOM defines the entities describing any document and the objects within it. This is expanded upon as needed by other APIs that add new features and capabilities to the DOM. For example, the HTML DOM API adds support for representing HTML documents to the core DOM, and the SVG API adds support for representing SVG documents.

DOM and JavaScript

The previous short example, like nearly all examples, is JavaScript. That is to say, it is written in JavaScript, but uses the DOM to access the document and its elements. The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of web pages, HTML documents, SVG documents, and their component parts. The document as a whole, the head, tables within the document, table headers, text within the table cells, and all other elements in a document are parts of the document object model for that document. They can all be accessed and manipulated using the DOM and a scripting language like JavaScript.

The DOM is not part of the JavaScript language, but is instead a Web API used to build websites. JavaScript can also be used in other contexts. For example, Node.js runs JavaScript programs on a computer but provides a different set of APIs, and the DOM API is not a core part of the Node.js runtime.

The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API. Even if most web developers will only use the DOM through JavaScript, implementations of the DOM can be built for any language, as this Python example demonstrates:

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

Accessing the DOM

You don't have to do anything special to begin using the DOM. You use the API directly in JavaScript from within what is called a script, a program run by a browser.

When you create a script, whether inline in a <script> element or included in the web page, you can immediately begin using the API for the document or window objects to manipulate the document itself, or any of the various elements in the web page (the descendant elements of the document). Your DOM programming may be something as simple as the following example, which displays a message on the console by using the `console.log()` function:

```
<body onload="console.log('Welcome to my home page!');">
  …
</body>
```

As it is generally not recommended to mix the structure of the page (written in HTML) and manipulation of the DOM (written in JavaScript), the JavaScript parts will be grouped together here, and separated from the HTML.

For example, the following function creates a new `h1` element, adds text to that element, and then adds it to the tree for the document:

```
<html lang="en">
  <head>
    <script>
      // run this function when the document is loaded
      window.onload = () => {
        // create a couple of elements in an otherwise empty HTML page
        const heading = document.createElement("h1");
        const headingText = document.createTextNode("Big Head!");
        heading.appendChild(headingText);
        document.body.appendChild(heading);
      };
    </script>
  </head>
  <body></body>
</html>
```

Fundamental data types

This page tries to describe the various objects and types in simple terms. But there are a number of different data types being passed around the API that you should be aware of.

Note: Because the vast majority of code that uses the DOM revolves around manipulating HTML documents, it's common to refer to the nodes in the DOM as elements, although strictly speaking **not every node is an element**.

The following table briefly describes these data types.

Data type (Interface)	Description
`Document`	When a member returns an object of type document (e.g., the `ownerDocument` property of an element returns the document to which it belongs), this object is the root `document` object itself. The DOM document Reference chapter describes the `document` object.
`Node`	Every object located in document is node of some kind. In an HTML document, an object can be an element node but also a text node or attribute node.
`Element`	The `element` type is based on `node`. It refers to an element or a node of type `element` returned by a member of the DOM API. Rather than

	saying, for example, that the `document.createElement()` method returns an object reference to a `node`, we just say that this method returns the `element` that has just been created in the DOM. `element` objects implement the DOM Element interface and also the more basic Node interface, both of which are included together in this reference. In an HTML document, elements are further enhanced by the HTML DOM API's `HTMLElement` interface as well as other interfaces describing capabilities of specific kinds of elements (for instance, `HTMLTableElement` for `<table>` elements).
`NodeList`	A `nodeList` is an array of elements, like the kind that is returned by the method `document.querySelectorAll()`. Items in a `nodeList` are accessed by index in either of two ways: • list.item(1) • list[1] These two are equivalent. In the first, `item()` is the single method on the nodeList object. The latter uses the typical array syntax to fetch the second item in the list.
`Attr`	When an `attribute` is returned by a member (e.g., by the `createAttribute()` method), it is an object reference that exposes a special interface for attributes. Attributes are nodes in the DOM just like elements are, though you may rarely use them as such.
`NamedNodeMap`	A `namedNodeMap` is like an array, but the items are accessed by name or index, though this latter case is merely a convenience for enumeration, as they are in no particular order in the list. A `namedNodeMap` has an `item()` method for this purpose, and you can also add and remove items from a `namedNodeMap`.

There are also some common terminology considerations to keep in mind. It's common to refer to any `Attr` node as an `attribute`, for example, and to refer to an array of DOM nodes as a `nodeList`. You'll find these terms and others to be introduced and used throughout the documentation.

DOM interfaces

This guide is about the objects and the actual things you can use to manipulate the DOM hierarchy. There are many points where understanding how these work can be confusing. For example, the object representing the HTML `form` element gets its name property from the `HTMLFormElement` interface but its `className` property from the `HTMLElement` interface. In both cases, the property you want is in that form object.

But the relationship between objects and the interfaces that they implement in the DOM can be confusing, and so this section attempts to say a little something about the actual interfaces in the DOM specification and how they are made available.

Interfaces and objects

Many objects implement several different interfaces. The table object, for example, implements a specialized `HTMLTableElement` interface, which includes such methods as `createCaption` and `insertRow`. But since it's also an HTML element, `table` implements the `Element` interface. And finally, since an HTML element is also, as far as the DOM is concerned, a node in the tree of nodes that make up the object model for an HTML or XML page, the table object also implements the more basic `Node` interface, from which `Element` derives.

When you get a reference to a `table` object, as in the following example, you routinely use all three of these interfaces interchangeably on the object, perhaps without knowing it.

```
const table = document.getElementById("table");
const tableAttrs = table.attributes; // Node/Element interface
for (let i = 0; i < tableAttrs.length; i++) {
  // HTMLTableElement interface: border attribute
  if (tableAttrs[i].nodeName.toLowerCase() === "border") {
    table.border = "1";
  }
}
// HTMLTableElement interface: summary attribute
table.summary = "note: increased border";
```

Core interfaces in the DOM

This section lists some of the most commonly-used interfaces in the DOM. The idea is not to describe what these APIs do here but to give you an idea of the sorts of methods and properties you will see very often as you use the DOM.

The `document` and `window` objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, **the `window` object represents something like the browser, and the `document` object is the root of the document itself**.

`Element` inherits from the generic `Node` interface, and together these two interfaces provide many of the methods and properties you use on individual elements. These elements may also have specific interfaces for dealing with the kind of data those elements hold, as in the `table` object example in the previous section.

The following is a brief list of common APIs in web and XML page scripting using the DOM.

- `document.querySelector()`
- `document.querySelectorAll()`
- `document.createElement()`
- `Element.innerHTML`
- `Element.setAttribute()`
- `Element.getAttribute()`
- `EventTarget.addEventListener()`
- `HTMLElement.style`
- `Node.appendChild()`
- `window.onload`
- `window.scrollTo()`

Using the Document Object Model

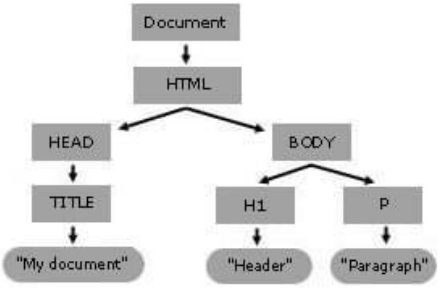
The Document Object Model (DOM) is an API for manipulating DOM trees of HTML and XML documents (among other tree-like documents). This API is at the root of the description of a page and serves as a base for scripting on the web.

What is a DOM tree?

A DOM tree is a tree structure whose nodes represent an HTML or XML document's contents. Each HTML or XML document has a DOM tree representation. For example, consider the following document:

```
<html lang="en">
  <head>
    <title>My Document</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>Paragraph</p>
  </body>
</html>
```

It has a DOM tree that looks like this:



The tree-like structure shown in the example may look similar to the original HTML or XML document, but it might not be exactly identical due to whitespace differences.

When a browser parsing an HTML or XML document to create a DOM tree, the DOM representation is designed to capture the structural hierarchy and relationships between elements in the document. However, the actual DOM tree might not preserve all the details of the original document, especially when it comes to whitespace.

Whitespace in this context refers to spaces, tabs, line breaks, and other characters used for formatting and indentation in the source code of the document.

So, while the visual representation of the DOM tree in the example might look similar to the original HTML or XML document, it might not be identical in terms of preserving all the whitespace details from the source code. The DOM tree focuses on the structural content of the document, not its visual appearance.

What does the Document API do?

The Document API, also known as the DOM API (Document Object Model API), is a powerful tool that allows you to manipulate the structure and content of a web page represented by a

DOM tree. The DOM tree is a tree-like representation of the HTML or XML document that a web page consists of.

The Document API gives web developers the ability to create new HTML or XML documents from scratch or modify existing ones dynamically. This is done by using JavaScript to access the `document` property of the global object, which represents the current web page. The `document` object implements the Document interface and provides various methods and properties to interact with the DOM tree.

In the provided example, let's consider a simple HTML document with a header (`<h2>Header</h2>`) and a paragraph (`<p>Paragraph</p>`). The author wants to use JavaScript to change the header text and add a second paragraph dynamically when a button is clicked.

```
<html lang="en">
  <head>
    <title>My Document</title>
  </head>
  <body>
    <input type="button" value="Change this document." onclick="change()" />
    <h2>Header</h2>
    <p>Paragraph</p>
  </body>
</html>
```

```
function change() {
  // document.getElementsByTagName("h2") returns a NodeList of the <h2>
  // elements in the document, and the first is number 0:
  const header = document.getElementsByTagName("h2").item(0);

  // The firstChild of the header is a Text node:
  header.firstChild.data = "A dynamic document";

  // Now header is "A dynamic document".

  // Access the first paragraph
  const para = document.getElementsByTagName("p").item(0);
  para.firstChild.data = "This is the first paragraph.";

  // Create a new Text node for the second paragraph
  const newText = document.createTextNode("This is the second paragraph.");

  // Create a new Element to be the second paragraph
  const newElement = document.createElement("p");
```

```
    // Put the text in the paragraph
    newElement.appendChild(newText);

    // Put the paragraph on the end of the document by appending it to
    // the body (which is the parent of para)
    para.parentNode.appendChild(newElement);
  }
```

DOM Traversing

DOM traversing refers to the process of navigating or moving through the Document Object Model (DOM) tree to access, find, or manipulate specific elements or nodes within a web page. The DOM tree is a hierarchical representation of the HTML or XML document's structure, and traversing allows developers to interact with and manipulate its nodes dynamically.

DOM traversing is commonly performed using various DOM methods and properties that allow you to move between parent and child nodes, sibling nodes, or find elements based on specific criteria. Some of the most frequently used DOM traversing methods and properties include:

1. `getElementById`: Retrieves an element by its unique ID attribute.
2. `querySelector:` Finds the first element that matches a specified CSS selector.
3. `querySelectorAll`: Retrieves all elements that match a specified CSS selector.
4. `parentNode`: Accesses the parent node of a given element.
5. `childNodes`: Returns a collection of all child nodes of an element (including text nodes, element nodes, and comment nodes).
6. `firstChild`: Gets the first child node of an element.
7. `lastChild:` Gets the last child node of an element.
8. `nextSibling`: Accesses the next sibling node (i.e., the next node at the same level within the parent).
9. `previousSibling`: Accesses the previous sibling node (i.e., the previous node at the same level within the parent).

DOM traversing is often used in JavaScript to perform various tasks, such as:

1. Modifying the content of specific elements.
2. Adding or removing elements dynamically.
3. Traversing the DOM to find and interact with elements based on user interactions (e.g., clicking a button).
4. Finding and selecting elements for specific actions, like form validation.

By using DOM traversing methods and properties effectively, developers can create dynamic and interactive web pages, as well as perform various manipulations on the document's structure and content to provide a rich user experience.

Locating DOM elements using selectors

The Selectors API provides methods that make it quick and easy to retrieve `Element` nodes from the DOM by matching against a set of selectors. This is much faster than past techniques, wherein it was necessary to, for example, use a loop in JavaScript code to locate the specific items you needed to find.

The NodeSelector interface

This specification adds two new methods to any objects implementing the `Document`, `DocumentFragment`, or `Element` interfaces:

`querySelector()`
Returns the first matching `Element` node within the node's subtree. If no matching node is found, `null` is returned.

`querySelectorAll()`
Returns a `NodeList` containing all matching `Element` nodes within the node's subtree, or an empty `NodeList` if no matches are found.

Note: The `NodeList` returned by `querySelectorAll()` is not live, which means that changes in the DOM are not reflected in the collection. This is different from other DOM querying methods that return live node lists. To return live `NodeList` we need to use methods like `document.getElementsByTagName()`, `document.getElementsByClassName()`, `element.getElementsByTagName()`, `element.getElementsByClassName()`.

Selectors

The selector methods accept selectors to determine what element or elements should be returned. This includes selector lists so you can group multiple selectors in a single query.

To protect the user's privacy, some pseudo-classes are not supported or behave differently. For example `:visited` will return no matches and `:link` is treated as `:any-link`.

Only elements can be selected, so pseudo-classes are not supported.

Examples

To select all paragraph (`p`) elements in a document whose classes include `warning` or `note`, you can do the following:

```
const special = document.querySelectorAll("p.warning, p.note");
```

You can also query by ID. For example:

```
const el = document.querySelector("#main, #basic, #exclamation");
```

After executing the above code, `el` contains the first element in the document whose ID is one of `main`, `basic`, or `exclamation`.

How to create a DOM tree (XML example)

This page describes how to use the DOM API in JavaScript to create XML documents.

Consider the following XML document:

```
<?xml version="1.0"?>
<people>
  <person first-name="eric" middle-initial="H" last-name="jung">
    <address street="321 south st" city="denver" state="co" country="usa"/>
    <address street="123 main st" city="arlington" state="ma" country="usa"/>
  </person>

  <person first-name="jed" last-name="brown">
    <address street="321 north st" city="atlanta" state="ga" country="usa"/>
  </person>
</people>
```

You can use the DOM API to create an in-memory representation of this document:

```
const doc = document.implementation.createDocument("", "", null);
const peopleElem = doc.createElement("people");

const personElem1 = doc.createElement("person");
personElem1.setAttribute("first-name", "eric");
personElem1.setAttribute("middle-initial", "h");
personElem1.setAttribute("last-name", "jung");

const addressElem1 = doc.createElement("address");
addressElem1.setAttribute("street", "321 south st");
addressElem1.setAttribute("city", "denver");
addressElem1.setAttribute("state", "co");
addressElem1.setAttribute("country", "usa");
personElem1.appendChild(addressElem1);

const addressElem2 = doc.createElement("address");
addressElem2.setAttribute("street", "123 main st");
addressElem2.setAttribute("city", "arlington");
addressElem2.setAttribute("state", "ma");
addressElem2.setAttribute("country", "usa");
personElem1.appendChild(addressElem2);

const personElem2 = doc.createElement("person");
personElem2.setAttribute("first-name", "jed");
personElem2.setAttribute("last-name", "brown");

const addressElem3 = doc.createElement("address");
addressElem3.setAttribute("street", "321 north st");
addressElem3.setAttribute("city", "atlanta");
addressElem3.setAttribute("state", "ga");
addressElem3.setAttribute("country", "usa");
personElem2.appendChild(addressElem3);

peopleElem.appendChild(personElem1);
peopleElem.appendChild(personElem2);
doc.appendChild(peopleElem);
```

DOM trees can be:

- queried using XPath expressions
- converted to strings using XMLSerializer
- posted to a web server using `XMLHttpRequest`
- transformed using XSLT or XLink.

How whitespace is handled by HTML, CSS, and in the DOM

The presence of whitespace in the DOM can cause layout problems and make manipulation of the content tree difficult in unexpected ways, depending on where it is located. This article explores when difficulties can occur, and looks at what can be done to mitigate resulting problems.

What is whitespace?

Whitespace is any **string of text composed only of spaces, tabs or line breaks** (to be precise, CRLF sequences, carriage returns or line feeds). These characters allow you to format your code in a way that will make it easily readable by yourself and other people. In fact, much of our source code is full of these whitespace characters, and we only tend to get rid of it in a production build step to reduce code download sizes.

HTML largely ignores whitespace?

In the case of HTML, whitespace is largely ignored — whitespace in between words is treated as a single character, and whitespace at the start and end of elements and outside elements is ignored. Take the following minimal example:

```
<!DOCTYPE html>

  <h1>      Hello      World!     </h1>
```

This source code contains a couple of line feeds after the `DOCTYPE` and a bunch of space characters before, after, and inside the `<h1>` element, but the browser doesn't seem to care at all and just shows the words "Hello World!" as if these characters didn't exist at all:

Hello World!

This is so that whitespace characters don't impact the layout of your page. Creating space around and inside elements is the job of CSS.

What happens to whitespace?

They don't just disappear, however.

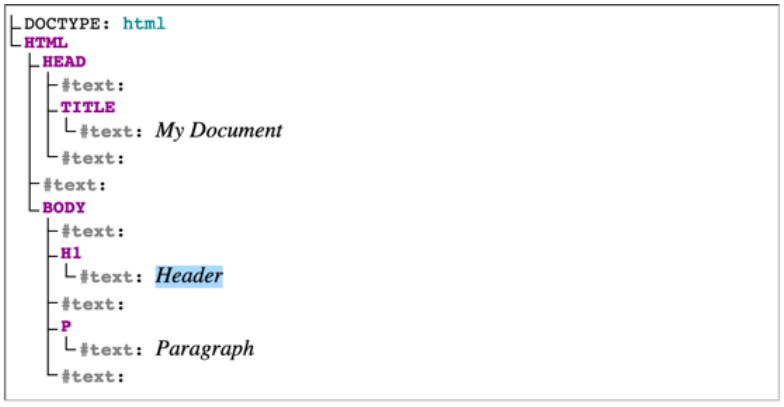
Any whitespace characters that are outside of HTML elements in the original document are represented in the DOM. This is needed internally so that the editor can preserve the formatting of documents. This means that:

- There will be some text nodes that contain only whitespace, and
- Some text nodes will have whitespace at the beginning or end.

Take the following document, for example:

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title>My Document</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>Paragraph</p>
  </body>
</html>
```

The DOM tree for this looks like so:



Conserving whitespace characters in the DOM is useful in many ways, but there are certain places where this makes certain layouts more difficult to implement and causes problems for developers who want to iterate through nodes in the DOM.

How does CSS process whitespace?

Most whitespace characters are ignored, not all of them are. In the earlier example one of the spaces between "Hello" and "World!" still exists when the page is rendered in a browser. There

are rules in the browser engine that decide which whitespace characters are useful and which aren't — these are specified at least in part in CSS Text Module Level 3, and especially the parts about the CSS white-space property and whitespace processing details, but we also offer an easier explanation below.

Example

Let's take another example. To make it easier, we've added a comment that shows all spaces with ◦, all tabs with ⇥, and all line breaks with ⏎:

This example:

```
<h1>   Hello
     <span> World!</span>   </h1>

<!--
<h1>◦◦◦Hello◦⏎
⇥⇥⇥⇥<span>◦World!</span>⇥◦◦</h1>
-->
```

is rendered in the browser like so:

Hello World!

Explanation

The `<h1>` element contains only inline elements. In fact it contains:

- A text node (consisting of some spaces, the word "Hello" and some tabs).
- An inline element (the , which contains a space, and the word "World!").
- Another text node (consisting only of tabs and spaces).

Because of this, it establishes what is called an inline formatting context. This is one of the possible layout rendering contexts that browser engines work with.

Inside this context, whitespace character processing can be summarized as follows:

1. First, all spaces and tabs immediately before and after a line break are ignored so, if we take our example markup from before:

```
<h1>◦◦◦Hello◦⏎
→→→→<span>◦World!</span>→◦◦</h1>
```

...and apply this first rule, we get:

```
<h1>◦◦◦Hello⏎
<span>◦World!</span>→◦◦</h1>
```

2. Next, all tab characters are handled as space characters, so the example becomes:

```
<h1>◦◦◦Hello⏎
<span>◦World!</span>◦◦◦</h1>
```

3. Next, line breaks are converted to spaces:

```
<h1>◦◦◦Hello◦<span>◦World!</span>◦◦◦</h1>
```

4. After that, any space immediately following another space (even across two separate inline elements) is ignored, so we end up with:

```
<h1>◦Hello◦<span>World!</span>◦</h1>
```

5. And finally, sequences of spaces at the beginning and end of an element are removed, so we finally get this:

```
<h1>Hello◦<span>World!</span></h1>
```

This is why people visiting the web page will see the phrase "Hello World!" nicely written at the top of the page, rather than a weirdly indented "Hello" followed but an even more weirdly indented "World!" on the line below that.

Whitespace in block formatting contexts

Above we just looked at elements that contain inline elements, and inline formatting contexts. If an element contains at least one block element, then it instead establishes what is called a block formatting context.

Within this context, whitespace is treated very differently.

Example

Let's take a look at an example to explain how. We've marked the whitespace characters as before.

We have 3 text nodes that contain only whitespace, one before the first `<div>`, one between the 2 `<div>`s, and one after the second `<div>`.

```
<body>
  <div>  Hello  </div>

   <div>  World!   </div>
</body>

<!--
<body>⏎
⇥<div>◦◦Hello◦◦</div>⏎
⏎
◦◦◦<div>◦◦World!◦◦</div>◦◦⏎
</body>
-->
```

This renders like so:

Hello
World!

Explanation

We can summarize how the whitespace here is handled as follows (there may be some slight differences in exact behavior between browsers, but this basically works):

1. Because we're inside a block formatting context, everything must be a block, so our 3 text nodes also become blocks, just like the 2 s. Blocks occupy the full width

available and are stacked on top of each other, which means that, starting from the example above:

```
<body>⏎
⇥<div>◦◦Hello◦◦</div>⏎
⏎
◦◦◦<div>◦◦World!◦◦</div>◦◦⏎
</body>
```

...we end up with a layout composed of this list of blocks:

```
<block>⏎⇥</block>
<block>◦◦Hello◦◦</block>
<block>⏎◦◦◦</block>
<block>◦◦World!◦◦</block>
<block>◦◦⏎</block>
```

2. This is then simplified further by applying the processing rules for whitespace in inline formatting contexts to these blocks:

```
<block></block>
<block>Hello</block>
<block></block>
<block>World!</block>
<block></block>
```

3. The 3 empty blocks we now have are not going to occupy any space in the final layout, because they don't contain anything, so we'll end up with only 2 blocks taking up space in the page. People viewing the web page see the words "Hello" and "World!" on 2 separate lines as you'd expect 2 `<div>`s to be laid out. The browser engine has essentially ignored all of the whitespace that was added in the source code.

Spaces in between inline and inline-block elements

Let's move on to look at a few issues that can arise due to whitespace, and what can be done about them. First of all, we'll look at what happens with spaces in between inline and inline-block elements. In fact, we saw this already in our very first example, when we described how whitespace is processed inside inline formatting contexts.

We said that there were rules to ignore most characters but that word-separating characters remain. When you're only dealing with block-level elements such as <p> that only contain inline elements such as , , , etc., you don't normally care about this because the extra whitespace that does make it to the layout is helpful to separate the words in the sentence.

It gets more interesting however when you start using inline-block elements. These elements behave like inline elements on the outside, and blocks on the inside, and are often used to display more complex pieces of UI than just text, side-by-side on the same line, for example navigation menu items.

Because they are blocks, many people expect that they will behave as such, but really they don't. If there is formatting whitespace between adjacent inline elements, this will result in space in the layout, just like the spaces between words in text.

Example

Consider this example (again, we've included an HTML comment that shows the whitespace characters in the HTML):

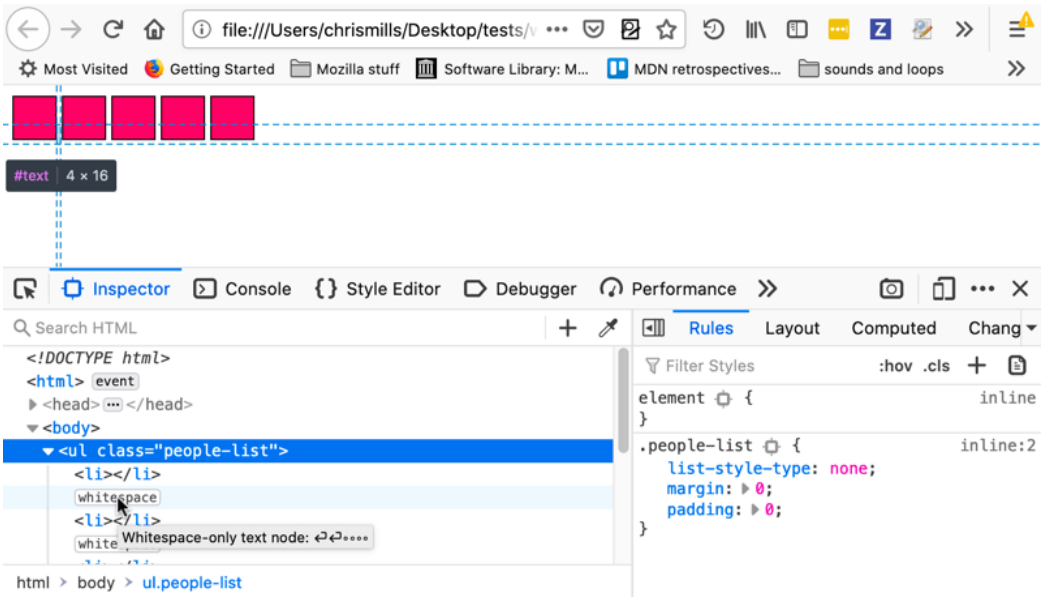
```
.people-list {
  list-style-type: none;
  margin: 0;
  padding: 0;
}

.people-list li {
  display: inline-block;
  width: 2em;
  height: 2em;
  background: #f06;
  border: 1px solid;
}
```

```
<ul class="people-list">
  <li></li>

  <li></li>

  <li></li>

  <li></li>

  <li></li>
```

```
</ul>

<!--
<ul class="people-list">⏎
◦◦<li></li>⏎
⏎
◦◦<li></li>⏎
⏎
◦◦<li></li>⏎
⏎
◦◦<li></li>⏎
⏎
◦◦<li></li>⏎
</ul>
-->
```

You probably don't want the gaps in between the blocks — depending on the use case (is this a list of avatars, or horizontal nav buttons?), you probably want the element sides flush with each other, and to be able to control any spacing yourself.

The Firefox DevTools HTML Inspector will highlight text nodes, and also show you exactly what area the elements are taking up — useful if you are wondering what is causing the problem.



Solutions

There are a few ways of getting around this problem:

Use Flexbox to create the horizontal list of items instead of trying an inline-block solution. This handles everything for you, and is definitely the preferred solution:

```
ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
  display: flex;
}
```

If you need to rely on `inline-block`, you could set the `font-size` of the list to 0. This only works if your blocks are not sized with ems (based on the font-size, so the block size would also end up being 0). rems would be a good choice here:

```
ul {
  font-size: 0;
  /* … */
}

li {
  display: inline-block;
  width: 2rem;
  height: 2rem;
  /* … */
}
```

Or you could set negative margin on the list items:

```
li {
  display: inline-block;
  width: 2rem;
  height: 2rem;
  margin-right: -0.25rem;
}
```

You can also solve this problem by putting your list items all on the same line in the source, which causes the whitespace nodes to not be created in the first place:

```
<li></li><li></li><li></li><li></li><li></li>
```

DOM traversal and whitespace

When trying to do DOM manipulation in JavaScript, you can also encounter problems because of whitespace nodes. For example, if you have a reference to a parent node and want to affect its first element child using `Node.firstChild`, if there is a rogue whitespace node just after the opening parent tag you will not get the result you are expecting. The text node would be selected instead of the element you want to affect.

As another example, if you have a certain subset of elements that you want to do something to based on whether they are empty (have no child nodes) or not, you could check whether each element is empty using something like `Node.hasChildNodes()`, but again, if any target elements contain text nodes, you could end up with false results.

Whitespace helper functions

The JavaScript code below defines several functions that make it easier to deal with whitespace in the DOM:

```
/**
 * Throughout, whitespace is defined as one of the characters
 *  "\t" TAB \u0009
 *  "\n" LF  \u000A
 *  "\r" CR  \u000D
 *  " "  SPC \u0020
 *
 * This does not use JavaScript's "\s" because that includes non-breaking
 * spaces (and also some other characters).
 */

/**
 * Determine whether a node's text content is entirely whitespace.
 *
 * @param nod  A node implementing the |CharacterData| interface (i.e.,
 *             a |Text|, |Comment|, or |CDATASection| node
 * @return     True if all of the text content of |nod| is whitespace,
 *             otherwise false.
```

```
 */
function is_all_ws(nod) {
  return !/[^\t\n\r ]/.test(nod.textContent);
}

/**
 * Determine if a node should be ignored by the iterator functions.
 *
 * @param nod  An object implementing the DOM1 |Node| interface.
 * @return     true if the node is:
 *                1) A |Text| node that is all whitespace
 *                2) A |Comment| node
 *             and otherwise false.
 */

function is_ignorable(nod) {
  return (
    nod.nodeType === 8 || // A comment node
    (nod.nodeType === 3 && is_all_ws(nod))
  ); // a text node, all ws
}

/**
 * Version of |previousSibling| that skips nodes that are entirely
 * whitespace or comments. (Normally |previousSibling| is a property
 * of all DOM nodes that gives the sibling node, the node that is
 * a child of the same parent, that occurs immediately before the
 * reference node.)
 *
 * @param sib  The reference node.
 * @return     Either:
 *               1) The closest previous sibling to |sib| that is not
 *                  ignorable according to |is_ignorable|, or
 *               2) null if no such node exists.
 */
function node_before(sib) {
  while ((sib = sib.previousSibling)) {
    if (!is_ignorable(sib)) {
      return sib;
    }
  }
  return null;
}

/**
 * Version of |nextSibling| that skips nodes that are entirely
 * whitespace or comments.
 *
 * @param sib  The reference node.
 * @return     Either:
 *               1) The closest next sibling to |sib| that is not
```

```
 *                   ignorable according to |is_ignorable|, or
 *                2) null if no such node exists.
 */
function node_after(sib) {
  while ((sib = sib.nextSibling)) {
    if (!is_ignorable(sib)) {
      return sib;
    }
  }
  return null;
}

/**
 * Version of |lastChild| that skips nodes that are entirely
 * whitespace or comments. (Normally |lastChild| is a property
 * of all DOM nodes that gives the last of the nodes contained
 * directly in the reference node.)
 *
 * @param sib  The reference node.
 * @return     Either:
 *               1) The last child of |sib| that is not
 *                  ignorable according to |is_ignorable|, or
 *               2) null if no such node exists.
 */
function last_child(par) {
  let res = par.lastChild;
  while (res) {
    if (!is_ignorable(res)) {
      return res;
    }
    res = res.previousSibling;
  }
  return null;
}

/**
 * Version of |firstChild| that skips nodes that are entirely
 * whitespace and comments.
 *
 * @param sib  The reference node.
 * @return     Either:
 *               1) The first child of |sib| that is not
 *                  ignorable according to |is_ignorable|, or
 *               2) null if no such node exists.
 */
function first_child(par) {
  let res = par.firstChild;
  while (res) {
    if (!is_ignorable(res)) {
      return res;
    }
```

```
    res = res.nextSibling;
  }
  return null;
}

/**
 * Version of |data| that doesn't include whitespace at the beginning
 * and end and normalizes all whitespace to a single space. (Normally
 * |data| is a property of text nodes that gives the text of the node.)
 *
 * @param txt  The text node whose data should be returned
 * @return     A string giving the contents of the text node with
 *             whitespace collapsed.
 */
function data_of(txt) {
  let data = txt.textContent;
  data = data.replace(/[\t\n\r ]+/g, " ");
  if (data[0] === " ") {
    data = data.substring(1, data.length);
  }
  if (data[data.length - 1] === " ") {
    data = data.substring(0, data.length - 1);
  }
  return data;
}
```

Example

The following code demonstrates the use of the functions above. It iterates over the children of an element (whose children are all elements) to find the one whose text is "`This is the third paragraph`", and then changes the class attribute and the contents of that paragraph.

```
let cur = first_child(document.getElementById("test"));
while (cur) {
  if (data_of(cur.firstChild) === "This is the third paragraph.") {
    cur.className = "magic";
    cur.firstChild.textContent = "This is the magic paragraph.";
  }
  cur = node_after(cur);
}
```