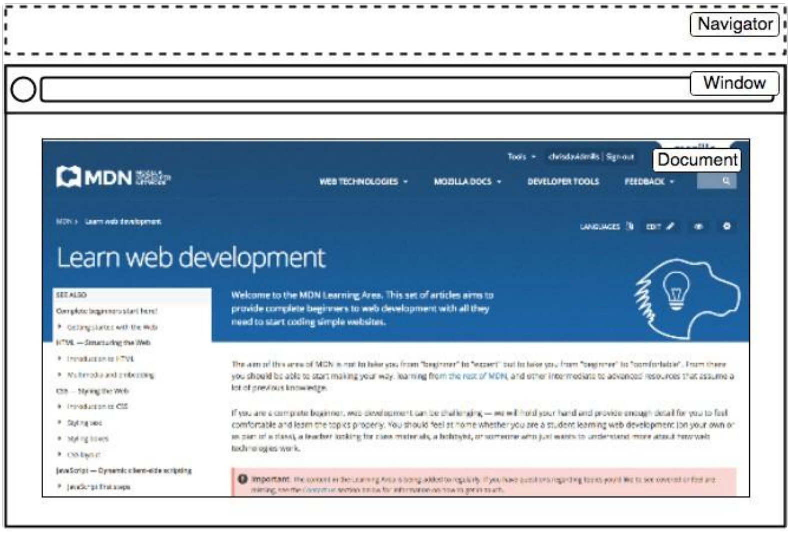
The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript. You might think that such limitations are a bad thing, but browsers are locked down for good reasons, mostly centering around security. Imagine if a website could get access to your stored passwords or other sensitive information, and log into websites as if it were you?

Despite the limitations, Web APIs still give us access to a lot of functionality that enable us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:



- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the Window object. Using methods available on this object you can do things like return the window's size (Window.innerWidth and Window.innerHeight), manipulate the document loaded into that window, store data specific to that document on the client-side (for example using a local database or other storage mechanism), attach an event handler to the current window, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the Navigator object. You can use this object to retrieve things like the user's preferred language, a media stream from the user's webcam, etc.
- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the Document object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.

The document object model

Let's provide a brief recap on the Document Object Model (DOM. The document currently loaded in each one of your browser tabs is represented by a DOM. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages — for example the browser itself uses it to apply styling and other information to the correct elements as it renders a page, and developers like you can manipulate the DOM with JavaScript after the page has been rendered.

We have created an example page containing a <section> element inside which you can find an image, and a paragraph with a link inside. The HTML source code looks like this:

HTML

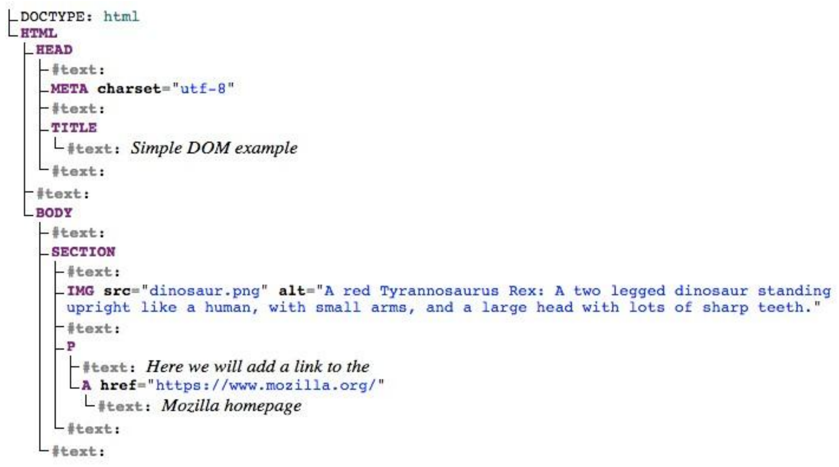
```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>Simple DOM example</title>
  </head>
  <body>


<section>

<img

        src="dinosaur.png"
        alt="A red Tyrannosaurus Rex: A two legged dinosaur standing upright like a human, with small arms,
and a large head with lots of sharp teeth." />
      <p>
        Here we will add a link to the
        <a href="https://www.mozilla.org/">Mozilla homepage</a>
      </p>
    </section>
  </body>
</html>
```

The DOM on the other hand looks like this:



```
├DOCTYPE: html
└HTML
  └HEAD
  │ ├#text:
  │ ├META charset="utf-8"
  │ ├#text:
  │ ├TITLE
  │ │ └#text: Simple DOM example
  │ └#text:
  ├#text:
  └BODY
    ├#text:
    ├SECTION
    │ ├#text:
    │ ├IMG src="dinosaur.png" alt="A red Tyrannosaurus Rex: A two legged dinosaur standing
    │ │ upright like a human, with small arms, and a large head with lots of sharp teeth."
    │ ├#text:
    │ ├P
    │ │ ├#text: Here we will add a link to the
    │ │ └A href="https://www.mozilla.org/"
    │ │   └#text: Mozilla homepage
    │ └#text:
    └#text:
```

Each entry in the tree is called a **node**. You can see in the diagram above that some nodes represent elements (identified as HTML , HEAD , META and so on) and others represent text (identified as #text). There are other types of nodes as well, but these are the main ones you'll encounter.

Nodes are also referred to by their position in the tree relative to other nodes:

- **Root node**: The top node in the tree, which in the case of HTML is always the HTML node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node**: A node directly inside another node. For example, IMG is a child of SECTION in the above example.
- **Descendant node**: A node anywhere inside another node. For example, IMG is a child of SECTION in the above example, and it is also a descendant.

IMG is not a child of BODY , as it is two levels below it in the tree, but it is a descendant of BODY .

Parent node: A node which has another node inside it. For example, BODY is the parent node of SECTION in the above example.

Sibling nodes: Nodes that sit on the same level under the same parent node in the DOM tree. For example, IMG and P are siblings in the above example.

It is useful to familiarize yourself with this terminology before working with the DOM, as a number of the code terms you'll come across make use of them. You'll also come across them in CSS (e.g. descendant selector, child selector).

Basic DOM manipulation

To start learning about DOM manipulation, let's begin with a practical example.

- Add a <script></script> element just above the closing </body> tag.
- To manipulate an element inside the DOM, you first need to select it and store a reference to it inside a variable. Inside your script element, add the following line:

JS

```js
const link = document.querySelector("a");
```

- Now we have the element reference stored in a variable, we can start to manipulate it using properties and methods available to it (these are defined on interfaces like HTMLAnchorElement in the case of <a> element, its more general parent interface HTMLElement , and Node — which represents all nodes in a DOM). First of all, let's **change the text** inside the link by updating the value of the Node.textContent property. Add the following line below the previous one:

JS

```js
link.textContent = "Mozilla Developer Network";
```

We should also change the URL the link is pointing to, so that it doesn't go to the wrong place when it is clicked on. Add the following line, again at the bottom:

JS

```js
link.href = "https://developer.mozilla.org";
```

Note that, as with many things in JavaScript, there are many ways to select an element and store a reference to it in a variable. Document.querySelector() is the recommended modern approach. It is convenient because it allows you to select elements using CSS selectors. The above querySelector() call will match the first <a> element that appears in the document. If you wanted to match and do things to multiple elements, you could use Document.querySelectorAll() , which matches every element in the document that matches the selector, and stores references to them in an array-like object called a NodeList .

There are older methods available for grabbing element references, such as:

Document.getElementById() , which selects an element with a given id attribute value, e.g. <p id="myId">My paragraph</p> . The ID is passed to the function as a parameter, **i.e. const elementRef = document.getElementById('myId') .**

Document.getElementsByTagName() , which returns an array-like object containing all the elements on the page of a given type, for example <p> s, <a> s, etc. The element type is passed to the function as a parameter, **i.e. const elementRefArray = document.getElementsByTagName('p') .**

These two work better in older browsers than the modern methods like querySelector() , but are not as convenient. Have a look and see what others you can find!

Creating and placing new nodes

The above has given you a little taste of what you can do, but let's go further and look at how we can create new elements.

- Going back to the current example, let's start by grabbing a reference to our <section> element — add the following code at the bottom of your existing script (do the same with the other lines too):

JS

```
const sect = document.querySelector("section");
```

- Now let's create a new paragraph using Document.createElement() and give it some text content in the same way as before:

JS

```
const para = document.createElement("p");
para.textContent = "We hope you enjoyed the ride.";
```

- You can now append the new paragraph at the end of the section using Node.appendChild() :

JS

```
sect.appendChild(para);
```

- Finally for this part, let's add a text node to the paragraph the link sits inside, to round off the sentence nicely. First we will create the text node using Document.createTextNode() :

JS

```
const text = document.createTextNode(
  " — the premier source for web development knowledge.", );
```

 Now we'll grab a reference to the paragraph the link is inside, and append the text node to it:

JS

```
const linkPara = document.querySelector("p");
linkPara.appendChild(text);
```

Moving and removing elements

There may be times when you want to move nodes, or delete them from the DOM altogether. This is perfectly possible. If we wanted to move the paragraph with the link inside it to the bottom of the section, we could do this:

JS

```
sect.appendChild(linkPara);
```

This moves the paragraph down to the bottom of the section. You might have thought it would make a second copy of it, but this is not the case linkPara is a reference to the one and only copy of that paragraph. If you wanted to make a copy and add that as well, you'd need to use Node.cloneNode() instead.

Removing a node is pretty simple as well, at least when you have a reference to the node to be removed and its parent. In our current case, we just use Node.removeChild() , like this:

JS

```
sect.removeChild(linkPara);
```

When you want to remove a node based only on a reference to itself, which is fairly common, you can use Element.remove() :

JS

```
linkPara.remove();
```

This method is not supported in older browsers. They have no method to tell a node to remove itself, so you'd have to do the following:

JS

```
linkPara.parentNode.removeChild(linkPara);
```

Have a go at adding the above lines to your code.

Manipulating styles

It is possible to manipulate CSS styles via JavaScript in a variety of ways. To start with, you can get a list of all the stylesheets attached to a document using Document.stylesheets , which returns an array-like object with

CSSStyleSheet objects. You can then add/remove styles as wished. However, we're not going to expand on those features because they are a somewhat archaic and difficult way to manipulate style. There are much easier ways.

The first way is to add inline styles directly onto elements you want to dynamically style. This is done with the HTMLElement.style property, which contains inline styling information for each element in the document. You can set properties of this object to directly update element styles.

As an example, try adding these lines to our ongoing example:

JS

```
para.style.color = "white"; para.style.backgroundColor = "black"; para.style.padding = "10px";
para.style.width = "250px"; para.style.textAlign = "center";
```

 Reload the page and you'll see that the styles have been applied to the paragraph. If you look at that paragraph in your browser's Page Inspector/DOM inspector, you'll see that these lines are indeed adding inline styles to the document:

HTML

```
<p
 style="color: white; background-color: black; padding: 10px;
width: 250px; text-align: center;"> We hope you enjoyed the ride.
</p>
```

Note: Notice how the JavaScript property versions of the CSS styles are written in lower camel case (**Camel case** is a way of writing phrases without spaces, where the first letter of each word is capitalized, except for the first letter of the entire compound word, which may be either upper or lower case. The name comes from the similarity of the capital letters to the humps of a camel's back. It's often stylized as "camelCase" to remind the reader of its appearance.) whereas the CSS versions are hyphenated (kebab-case) (e.g. backgroundColor versus background-color). (**Kebab case** is a way of writing phrases without spaces, where spaces are replaced with hyphens -, and the words are typically all lower case. The name comes from the similarity of the words to meat on a kebab skewer. It's often stylized as "kebab-case" to remind the reader of its appearance.) Make sure you don't get these mixed up, otherwise it won't work.

There is another common way to dynamically manipulate styles on your document, which we'll look at now.

 Delete the previous five lines you added to the JavaScript.

 Add the following inside your HTML <head> :

HTML

```
<style> .highlight {   color: white;   background-color: black;   padding: 10px;   width: 250px;
text-align: center;
 }
</style>
```

 Now we'll turn to a very useful method for general HTML manipulation — Element.setAttribute() — this takes two arguments, the attribute you want to set on the element, and the value you want to set it to. In this case we will set a class name of highlight on our paragraph:

JS

```
para.setAttribute("class", "highlight");
```

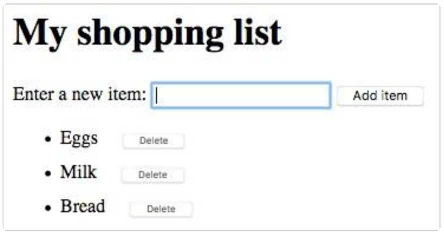
 Refresh your page, and you'll see no change — the CSS is still applied to the paragraph, but this time by giving it a class that is selected by our CSS rule, not as inline CSS styles.

Which method you choose is up to you; both have their advantages and disadvantages. The first method takes less setup and is good for simple uses, whereas the second method is more purist (no mixing CSS and JavaScript, no inline styles, which are seen as a bad practice). As you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.
At this point, we haven't really done anything useful! There is no point using JavaScript to create static content — you might as well just write it into your HTML and not use JavaScript. It is more complex than HTML, and creating your content with JavaScript also has other issues attached to it (such as not being readable by search engines).

A dynamic shopping list
In this challenge we want to make a simple shopping list example that allows you to dynamically add items to the list using a form input and button. When you add an item to the input and press the button:

- The item should appear in the list.
- Each item should be given a button that can be pressed to delete that item off the list.
- The input should be emptied and focused ready for you to enter another item.

The finished demo will look something like this:



follow the steps below, and make sure that the list behaves as described above.

□□ Create three variables that hold references to the list (<ul>), <input> , and <button> elements.

□□ Create a function that will run in response to the button being clicked.

□□ Inside the function body, start off by storing the current value of the input element in a variable.

□□ Next, empty the input element by setting its value to an empty string '' .

□□ Create three new elements — a list item (<li>), <span> , and <button> , and store them in variables.

□□ Append the span and the button as children of the list item.

□□ Set the text content of the span to the input element value you saved earlier, and the text content of the button to 'Delete'.

□□ Append the list item as a child of the list.

□□Attach an event handler to the delete button so that, when clicked, it will delete the entire list item (<li>...</li>).

□□ Finally, use the focus() method to focus the input element ready for entering the next shopping list item.

What is the DOM?
The **Document Object Model** (DOM) is the data representation of the objects that comprise the structure and content of a document on the web. The DOM represents an HTML document in memory and use APIs to create web content and applications.

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:

JS

```
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 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

```
# 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:

HTML

```
<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

```
<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>
```

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.
Node	Every object located within a document is a 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 (albeit small) 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.

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?

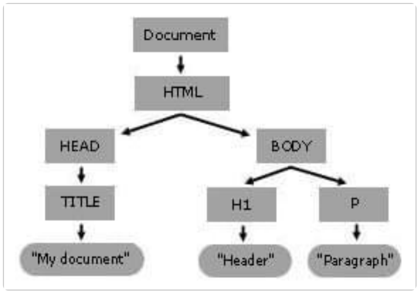
A **DOM tree** is a tree structure whose nodes represent an HTML or XML document's contents. Each HTML or XML document has a DOM tree representation. For example, consider the following document:

HTML

```
<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:



Although the above tree is similar to the above document's DOM tree, it's not identical, as the actual DOM tree preserves whitespace.

When a web browser parses an HTML document, it builds a DOM tree and then uses it to display the document.

What does the Document API do?

The Document API, also sometimes called the DOM API, allows you to modify a DOM tree in any way you want. It enables you to create any HTML or XML document from scratch or to change any contents of a given HTML or XML document. Web page authors can edit the DOM of a document using JavaScript to access the document property of the global object. This document object implements the Document interface.

Reading and modifying the tree

Suppose the author wants to change the header of the above document and write two paragraphs instead of one. The following script would do the job:

HTML

HTML Play

```
<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>
```

JavaScript

JS Play

```
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);
}
```

Play

Change this document.

A dynamic document

This is the first paragraph.

This is the second paragraph.

This is the second paragraph.

This is the second paragraph.

This is the second paragraph.

Creating a tree

You can create the above tree entirely in JavaScript too.

JS

```
const root = document.createElement("html");
root.lang = "en";
```

```
const head = document.createElement("head");
const title = document.createElement("title");
 title.appendChild(document.createTextNode("My Document"));
head.appendChild(title);
const body = document.createElement("body");
const header = document.createElement("h1");
header.appendChild(document.createTextNode("Header"));
const paragraph = document.createElement("p");
paragraph.appendChild(document.createTextNode("Paragraph"));
 body.appendChild(header); body.appendChild(paragraph);
root.appendChild(head); root.appendChild(body);
```

DOM interfaces

There are many points where understanding how this 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 described in the DOM Element Reference chapter. 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.

JS

```
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";
```

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()

Examples

Setting text content

This example uses a <div> element containing a <textarea> and two <button> elements. When the user clicks the first button we set some text in the <textarea> . When the user clicks the second button we clear the text. We use:

- Document.querySelector() to access the <textarea> and the button
- EventTarget.addEventListener() to listen for button clicks
- Node.textContent to set and clear the text.

HTML Play

```
<div class="container">
  <textarea class="story"></textarea>
  <button id="set-text" type="button">Set text content</button>
  <button   id="clear-text"   type="button">Clear

text content</button> </div>
```

CSS Play

```
.container  {
display:
flex;     gap:
0.5rem;
  flex-direction: column;
}
button  {
width:
200px;
}
```

JavaScript

JS Play

```
const story = document.body.querySelector(".story");
const setText = document.body.querySelector("#set-text");
setText.addEventListener("click", () => {
  story.textContent = "It was a dark and stormy night...";
});
const clearText = document.body.querySelector("#clear-text");
clearText.addEventListener("click", () => {   story.textContent = "";
});
```

Result

Play

Adding a child element

This example uses a <div> element containing a <div> and two <button> elements. When the user clicks the first button we create a new element and add it as a child of the <div> . When the user clicks the second button we remove the child element. We use:

- Document.querySelector() to access the <div> and the buttons
- EventTarget.addEventListener() to listen for button clicks
- Document.createElement to create the element
- Node.appendChild() to add the child
- Node.removeChild() to remove the child.

HTML

HTML Play

```
<div class="container">
  <div class="parent">parent</div>
  <button id="add-child" type="button">Add a
child</button>
  <button                              id="remove-child"
type="button">Remove child</button> </div>
```

CSS

CSS Play

```
.container        {
display:      flex;
gap:       0.5rem;
flex-direction:
column;
}
button    {
width:
100px;
}
div.parent {
  border:        1px
solid          black;
padding:       5px;
width:      100px;
height: 100px;
}
div.child {
  border:       1px
solid             red;
margin:      10px;
padding:      5px;
width:       80px;
height:      60px;
box-sizing:
border-box;
}
```

JavaScript

JS Play

```
const parent = document.body.querySelector(".parent");
const addChild = document.body.querySelector("#add-child");
 addChild.addEventListener("click", () => {  // Only add a child if we don't already have one
 // in addition to the text node "parent"
if (parent.childNodes.length > 1) {     return;   }
  const child = document.createElement("div");
child.classList.add("child");
child.textContent = "child";
 parent.appendChild(child);
});
const removeChild = document.body.querySelector("#remove-child");
removeChild.addEventListener("click", () => {
 const child = document.body.querySelector(".child");
 parent.removeChild(child);
});
```