JavaScript

Web Programming

Topics

Objects

Arrays

DOM/BOM

Javascript Events

Objects

Objects are used to store keyed collections of various data and complex entities

An **`object`** can be created with curly brackets `{...}` with an optional list of properties

A **`property`** is a "`key: value`" pair, where `key` is a `string` (also called a "property name"), and `value` can be `anything`

Objects

An empty object can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```

Literals and Properties

We can put some properties into `{...}` as “`key: value`” pairs:

```
let user = {
 name: "Aster",
 age: 22,
};
```

A property has a key (also known as “name” or “identifier”) before the colon "`:`" and a value to the right of it

Properties

We can add, remove and read properties

Property values are accessible using the dot notation

```
// get property values of the object
alert( user.name ); // Aster
alert( user.age ); // 22

// add new property
user.isAdmin = true;
```

Properties

To remove a property, we can use delete operator

```
delete user.age;
```

Properties

We can also use multiword property names, but then they must be quoted

```
let user = {
 name: "Aster",
 age: 22,
 "likes coffee": true,
};
```

Square Brackets

For multiword properties, the dot access doesn't work

```
// this would give a syntax error
User.likes coffee = true
```

The dot requires the key to be a valid variable identifier

contains no spaces, doesn't start with a digit and doesn't include special characters (`$` and `_` are allowed)

Square Brackets

Square Bracket notation works with multi-word properties

```
let user = {};
// set
user["likes coffee"] = true;
// get
alert(user["likes coffee"]); // true
// delete
delete user["likes coffee"];
```

Square Brackets

Square brackets also provide a way to obtain the property name as the result of any expression

```
let user = {
 name: "Aster",
 age: 22,
};

let key = prompt("What do you want to know about the user?");
// access by variable
alert(user[key]); // Aster (if enter "name")
```

Computed Properties

We can use square brackets in an object literal, when creating an object

That’s called computed properties

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// take property name from the fruit variable
bag[fruit] = 5;
```

Property Value Shorthand

This function can be re-written as shown in the next two slides

```
function makeUser(name, age) {
 return {
   name: name,
   age: age,
   // ...other properties
 };
}

let user = makeUser("Aster", 22);
alert(user.name); // Aster
```

Property Value Shorthand

```
function makeUser(name, age) {
 return {
   name: name,
   age,
   // ...other properties
 };
}

let user = makeUser("Aster", 22);
alert(user.name); // Aster
```

Property Value Shorthand

```
function makeUser(name, age) {
 return {
   name,
   age,
   // ...other properties
 };
}

let user = makeUser("Aster", 22);
alert(user.name); // Aster
```

Property Existence Test, "`in`" Operator

Reading a non-existing property just returns **`undefined`**

We can test whether the property exists using **`in`** operator

```
let user = { name: "Aster", age: 22 };

alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

The "for...in" loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`

```
let user = { name: "Aster", age: 22, isAdmin: true, };

for (let key in user) {
 // keys
 alert(key); // name, age, isAdmin
 // values for the keys
 alert(user[key]); // John, 30, true
}
```

Exercise

Write the code, one line for each action:

- Create an empty object user
- Add the property name with the value Abebe
- Add the property surname with the value Kebede
- Change the value of the name to Aster
- Remove the property name from the object

Exercise

Write the function `isEmpty(obj)` which returns true if the object has no properties, false otherwise.

It should work as follows

```
let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false
```

Exercise

Write the code to sum all salaries in the following object and store in the variable sum

If salaries is empty, then the result must be 0 otherwise it should be 390

```
let salaries = {
 Kebede: 100,
 Aster: 160,
 Kedir: 130,
};
```

Exercise

Create a function `multiplyNumeric(obj)` that multiplies all numeric property values of obj by 2

```
// before the call
let menu = {
 width: 200,
 height: 300,
 title: "My menu",
};
```

```
multiplyNumeric(menu);


// after the call
menu = {
 width: 400,
 height: 600,
 title: "My menu",
};
```

Object References and Copying

One of the fundamental differences of objects versus primitives is that

> `objects` are stored and copied "**by reference**", whereas
>
> primitive values: `strings`, `numbers`, `booleans`, etc – are always copied "**by value**"

Object References and Copying

Copying for primitive data type

```
let message = "Hello!";
let phrase = message;
```

Changing the value of phrase affect the value of message?

```
phrase = "Hello World!"
```

What is the output of the following

```
alert(message);
alert(phrase);
```

Object References and Copying

A variable assigned to an object stores not the object itself, but its “address in memory” – in other words “a reference” to it

What will be the output of `alert` in the last line

```
let user = { name: 'Aster' };
let admin = user;
admin.name = 'Marta';

alert(user.name);
```

Comparison by Reference

Two objects are equal only if they are the same object.

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true,
alert( a === b ); // true
```

Comparison by Reference

Two independent objects are not equal

```
let a = {};
let b = {}; // two independent objects


alert( a == b ); // false
```

Arrays

There are two syntaxes for creating an empty array:

```
let arr = new Array();
let arr = [];
```

Arrays

Array elements are numbered, starting with zero

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

Arrays

We can replace an element

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

Arrays

We can add a new one to the array

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Arrays

The total count of the elements in the array is its length:

```
let fruits = ["Apple", "Orange", "Plum"];


alert( fruits.length ); // 3
```

Arrays

An array can store elements of any type

```
// mix of values
let arr = [ 'Apple', { name: 'Aster' }, true, function() {
alert('hello'); } ];
// get the object at index 1 and then show its name
alert( arr[1].name ); // Aster
// get the function at index 3 and run it
arr[3](); // hello
```

Methods `pop/push`, `shift/unshift`

A **queue** is one of the most common uses of an array which supports two operations:

`push` appends an element to the end

`shift` get an element from the beginning

Methods `pop/push`, `shift/unshift`

There’s another use case for arrays – the data structure named **stack**

It supports two operations

- `push` add an element to the end
- `pop` takes an element from the end

Methods pop/push, shift/unshift

pop

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.pop() ); // remove "Pear" and alert it

alert( fruits ); // Apple, Orange
```

Methods pop/push, shift/unshift

push

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

Methods pop/push, shift/unshift

shift

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.shift() ); // remove Apple and alert it

alert( fruits ); // Orange, Pear
```

Methods pop/push, shift/unshift

unshift

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

Methods `pop/push`, `shift/unshift`

Methods `push` and `unshift` can add multiple elements at once

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

Array

An array is a special kind of object thus behaves like an object

```
let fruits = ["Banana"]
let arr = fruits; // copy by reference

alert( arr === fruits ); // true

arr.push("Pear"); // modify the array by reference

alert( fruits ); // Banana, Pear - 2 items now
```

Iterating Over Array Elements

One approach

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
 alert( arr[i] );
}
```

Iterating Over Array Elements

Using `for..of`

The `for..of` doesn't give access to the index of the current element, just its value

```
let arr = ["Apple", "Orange", "Pear"];

// iterates over array elements
for (let fruit of fruits) {
 alert(fruit);
}
```

Iterating Over Array Elements

Because arrays are objects, it is also possible to use `for..in` but not recommended as it is 10 to 100 times slower

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
 alert( arr[key] ); // Apple, Orange, Pear
}
```

A word about "length"

The `length` property automatically updates when we modify the array

It is actually not the count of values in the array, but the greatest numeric index plus one

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

A word about "`length`"

The `length` property is also writable

If we increase it manually, nothing interesting happens

But if we decrease it, the array is truncated

Exercise

What is this code going to show?

```
let fruits = ["Apples", "Pear", "Orange"];

// push a new value into the "copy"
let shoppingCart = fruits;
shoppingCart.push("Banana");

// what's in fruits?
alert( fruits.length ); // ?
```

Exercise

Create an array styles with items “Jazz” and “Blues”

Append “Rock-n-Roll” to the end

Replace the value in the middle by “Classics”

> Your code for finding the middle value should work for any arrays with odd length

Strip off the first value of the array and show it

Prepend Rap and Reggae to the array

Exercise

Write the function `sumInput()` that:

- Asks the user for values using prompt and stores the values in the array
- Finishes asking when the user enters a **non-numeric value,** an **empty string**, or presses "`Cancel`"
- Calculates and returns the sum of array items

Modules

As our application grows bigger, we want to split it into multiple files, so called “modules”

A module is just a file

One script is one module

Modules

Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

- `export` keyword labels variables and functions that should be accessible from outside the current module
- `import` allows the import of functionality from other modules

Modules: Export/Import a function

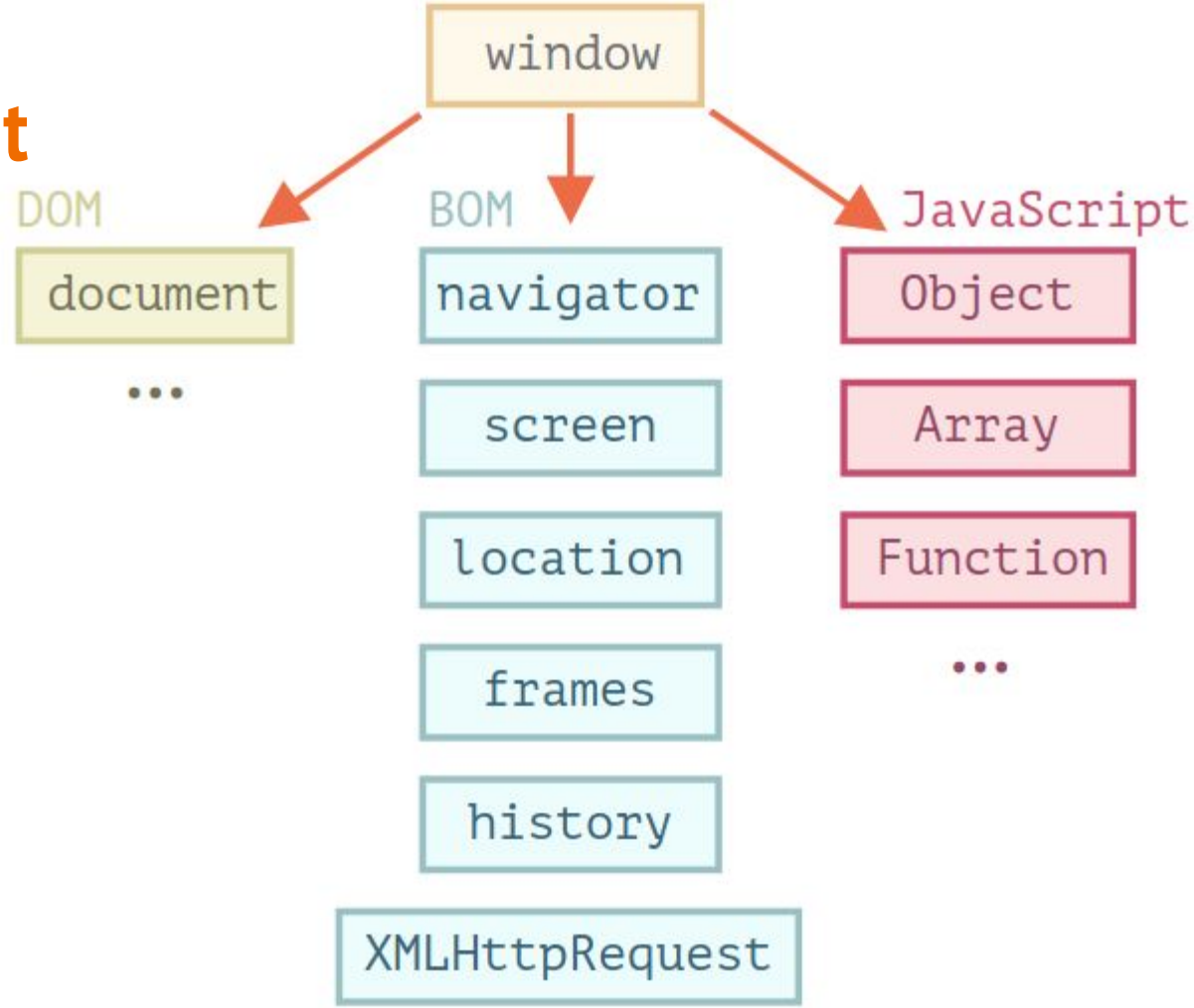
```
// 📁 sayHi.js
export function sayHi(user) {
 alert(`Hello, ${user}!`);
}
```

```
// 📁 main.js
import { sayHi } from "./sayHi.js";


alert(sayHi); // function...
sayHi("Aster"); // Hello, Aster!
```

Browser Environment

There's a `global` object calle
`window`

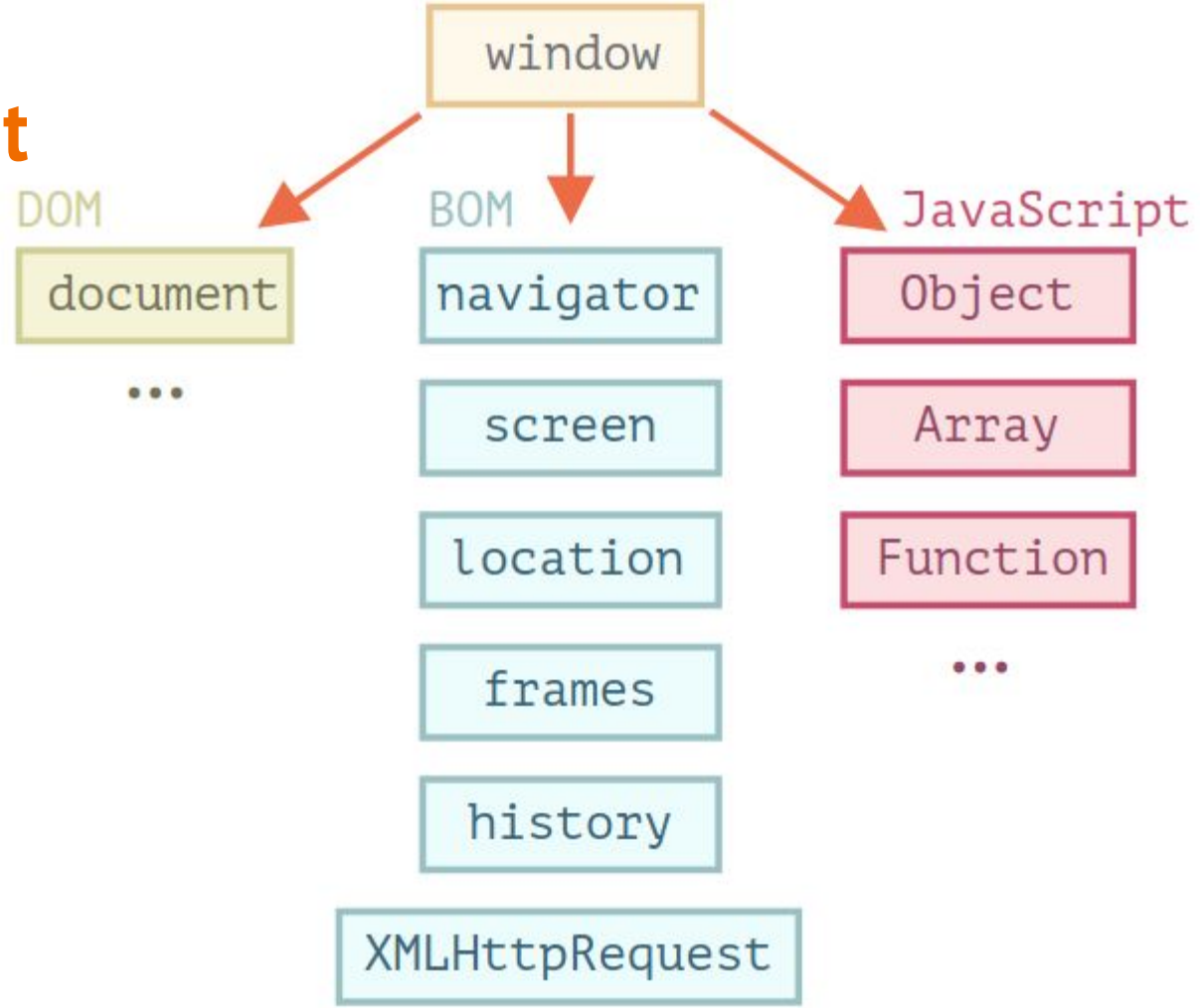


Browser Environment

There's a "**root**" object called `window`

It has two roles:

1. it is a **global object** for JavaScript code
2. it represents the "**browser window**" and **provides methods to control it**



window

As a global object

```
function sayHi() {
 alert("Hello");
}

// global functions are methods of the global object:
window.sayHi();
```

window

As a browser window, to see the window height

```
alert(window.innerHeight); // inner window height
```

The Global Object in Web Browsers

There is one `global object` per browser window or tab

All of the `JavaScript` code (except code running in worker threads) running in that window shares this single global object

The global object is where `JavaScript`'s standard library is defined

- the `parseInt()` function, the `Math` object, the `Set` class, and so on

The Global Object in Web Browsers

In web browsers, the global object also contains the main entry points of various web APIs

The Global Object in Web Browsers

In web browsers, the **global object** does double duty:

- defines built-in types and functions
- represents the current web browser window and defines properties like
  - `history`, which represent the **window's browsing history**,and
  - `innerWidth`, which holds the **window's width** in pixels

The Global Object in Web Browsers

One of the properties of the **global object** is named **window**, and **its value is the global object itself**

> This means that you can simply type window to refer to the global object in your client-side code

window

As a global object

```
function sayHi() {
 alert("Hello");
}

// global functions are methods of the global object
window.sayHi();
```

window

As a browser window

```
alert(window.innerHeight); // inner window height
```

Browser Object Model (BOM)

Represents additional objects provided by the browser (host environment)

For Example:

- The `navigator` object provides background information about the browser and the operating system such as `navigator.userAgent` and `navigator.platform`
- The `location` object allows us to read the current URL and can redirect the browser to a new one

Browser Object Model (BOM)

Represents additional objects provided by the browser (host environment)

```
alert(location.href); // shows current URL
if (confirm("Go to Wikipedia?")) {
 location.href = "https://wikipedia.org";
}
```

Document Object Model (DOM)

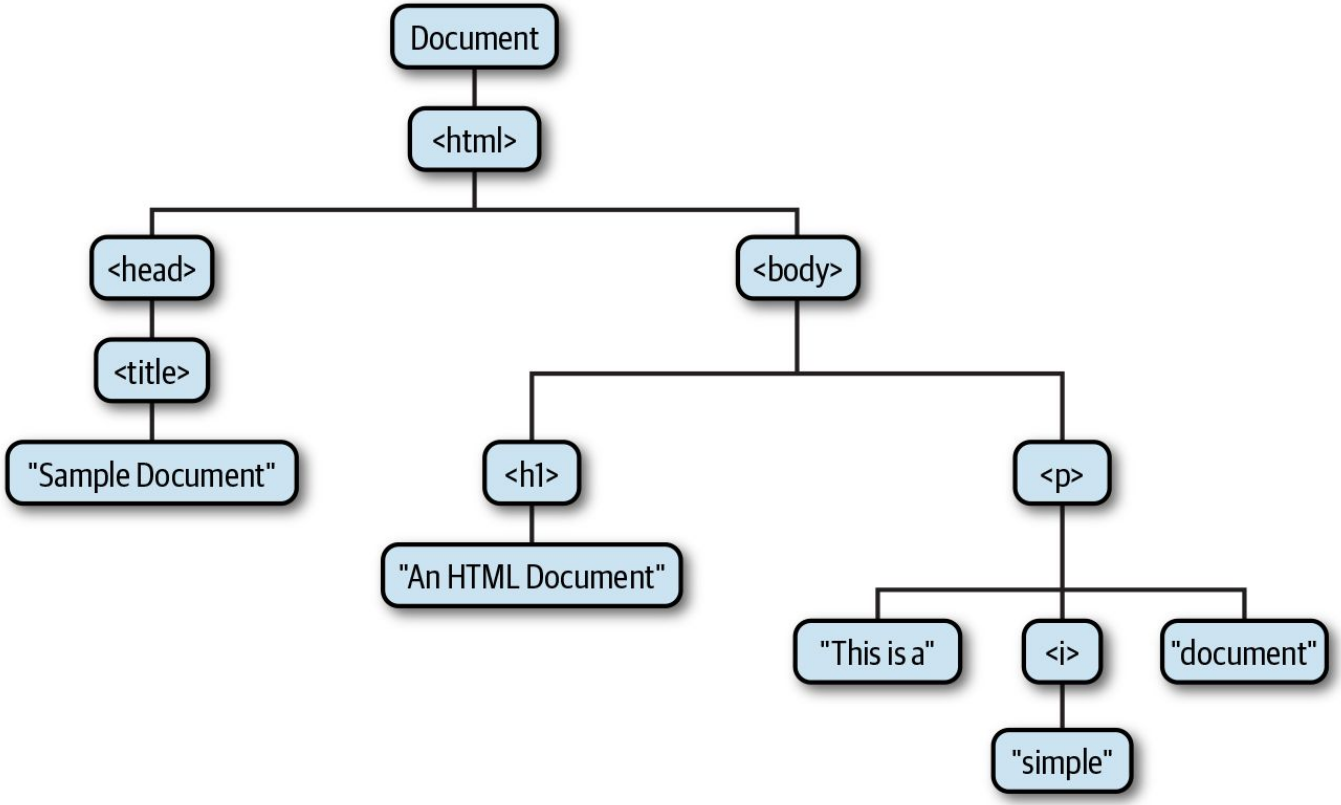
Document Object represents the `HTML` document that is displayed in a browser window or tab

The **`API`** for working with **`HTML`** documents is known as the **`Document Object Model`**, or **`DOM`**

Document Object Model (DOM)

```
<html>
 <head>
   <title>Sample Document</title>
 </head>
 <body>
   <h1>An HTML Document</h1>
   <p>This is a <i>simple</i> document.</p>
 </body>
</html>
```

Document Object Model (DOM)



Document Object Model (DOM)

The `DOM API` mirrors the tree structure of an `HTML` document

> **for each `HTML` tag in the document**, there is a corresponding `JavaScript Element` object, and
>
> **for each run of text in the document**, there is a corresponding `Text` object
>
> there are also **methods for moving elements** within the document and **for removing them**

Document Object Model (DOM)

There is a `JavaScript` class corresponding to each `HTML` tag type

Each occurrence of `HTML` tag in a document is represented by an instance of the class

> The `<body>` tag, for example, is represented by an instance of `HTMLBodyElement`
>
> A `<table>` tag is represented by an instance of `HTMLTableElement`

Document Object Model (DOM)

Most of the `JavaScript` element classes just mirror the attributes of an `HTML` tag, but some define additional methods

> The `HTMLAudioElement` and `HTMLVideoElement` classes, for example, define methods like `play()` and `pause()` for controlling playback of `audio` and `video` files

Document Object Model (DOM)

The `JavaScript` element objects have properties that correspond to the `HTML` attributes of the tags

- instances of `HTMLImageElement`, which represent `<img>` tags, have a `src` property that corresponds to the src attribute of the tag
- The initial value of the `src` property is the attribute value that appears in the `HTML` tag, and setting this property with JavaScript changes the value of the `HTML` attribute (and causes the browser to load and display a new image)

Document Object Model (DOM)

The document object is the main “entry point” to the page

We can change or create anything on the page using it

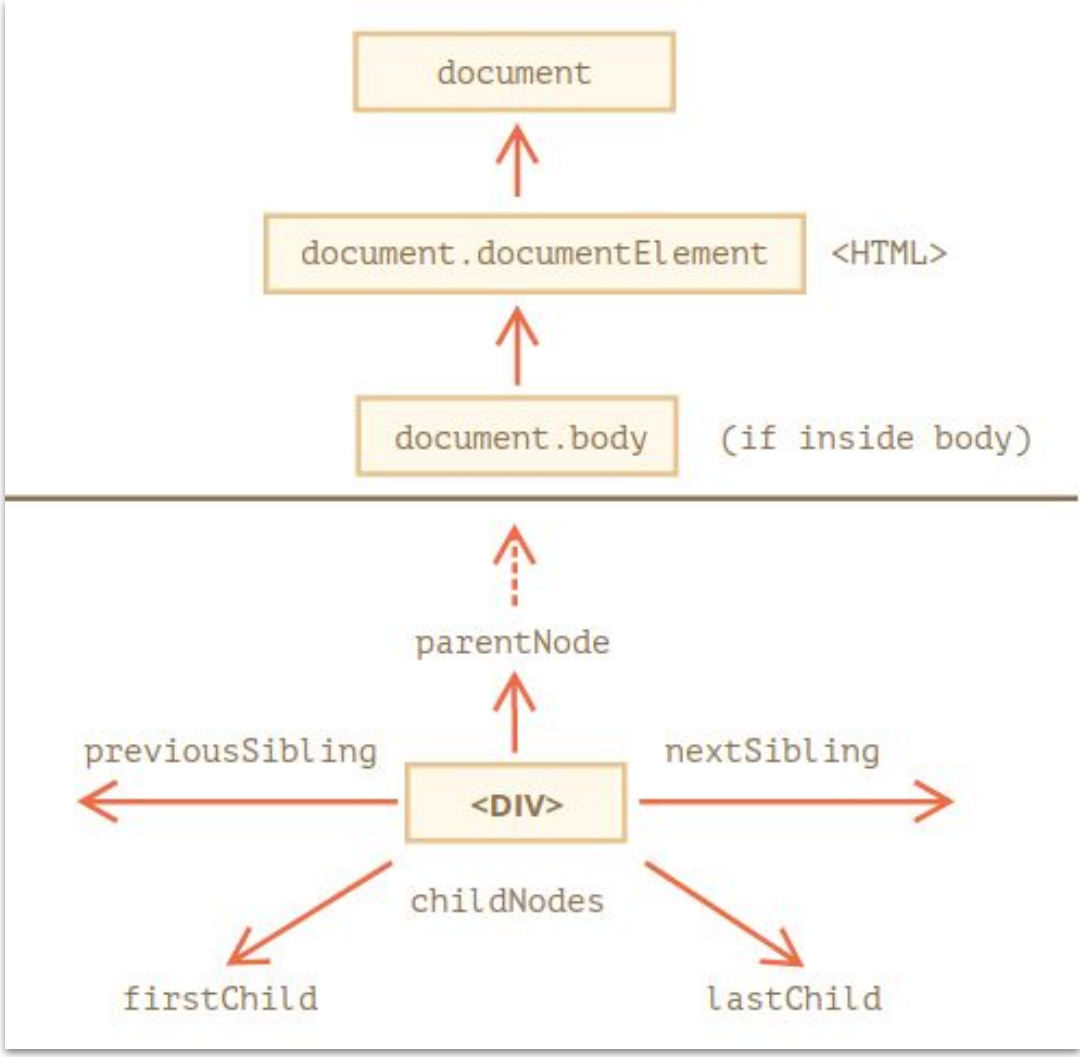
```
// change the background color to red
document.body.style.background = "red";


// change it back after 1 second
setTimeout(() => document.body.style.background = "", 1000);
```

Walking the DOM

On top

<html> = document.documentElement

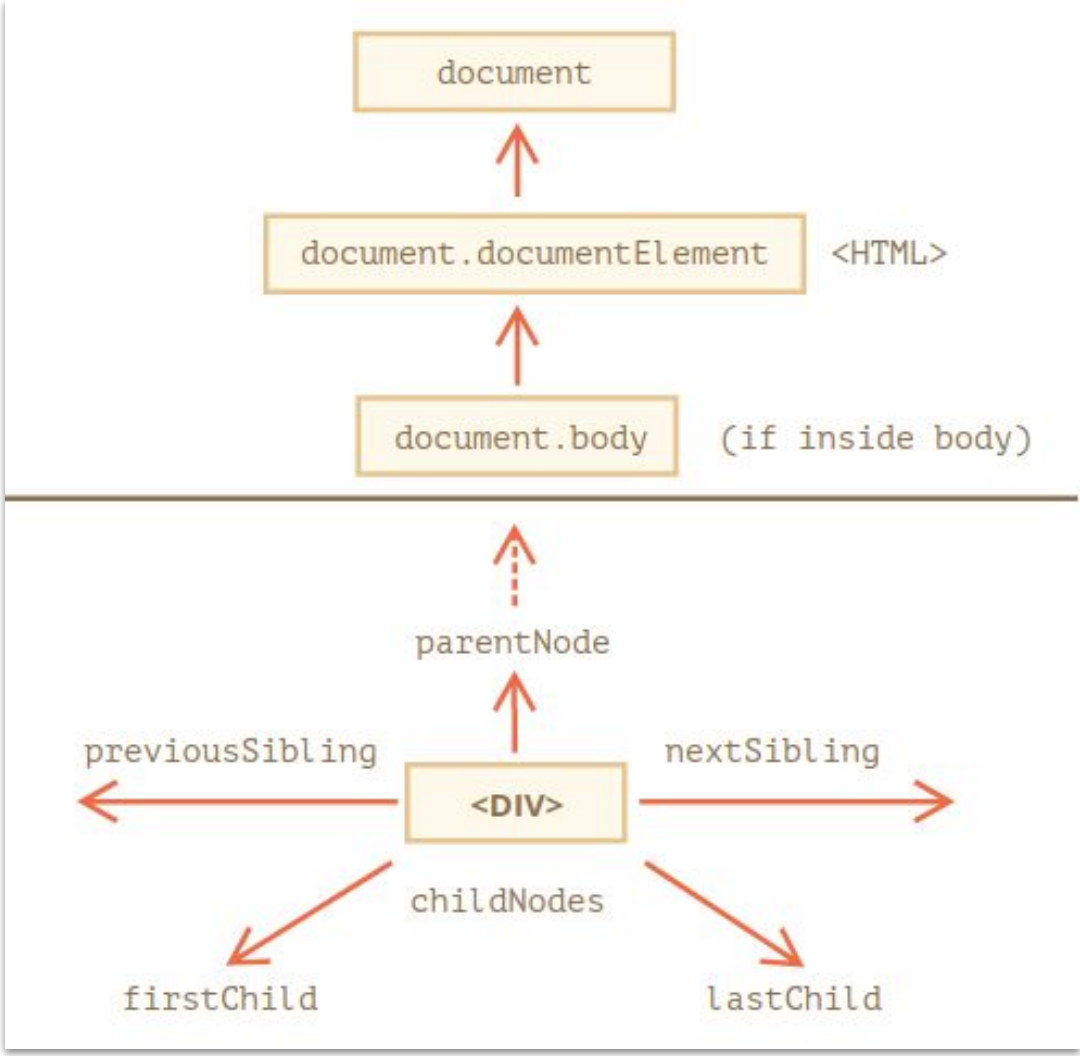


Walking the DOM

All operations on the DOM start with the **`document`** object

That's the main "entry point" to DOM

From it we can access any node



Walking the DOM

```
TOP: documentElement, body, head

    <html> = document.documentElement

    <body> = document.body

    <head> = document.head
```

Walking the DOM

`Children:` `childNodes, firstChild, lastChild`

Child nodes (or children) – elements that are direct children

Descendants – all elements that are nested in the given one, including children, their children and so on

```html
<html>
 <body>
   <div>Begin</div>
   <ul>
     <li>Information</li>
   </ul>
   <div>End</div>
   ...more stuff...
   <script src="example.js"></script>
 </body>
</html>
```

example.js

```
let children = document.body.childNodes;
 for (let i = 0; i < children.length; i++) {
   alert(children[i]);
 }
```

Walking the DOM

Children: childNodes, firstChild, lastChild

Properties **firstChild** and **lastChild** give direct access to the first and last children

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

There's also a special function **elem.hasChildNodes()** to check whether there are any child nodes

Walking the DOM

Siblings and the Parent

Siblings are nodes that are children of the same parent

```
// parent of <body> is <html>
alert( document.body.parentNode === document.documentElement );
// true
// after <head> goes <body>
alert( document.head.nextSibling ); // HTMLBodyElement
// before <body> goes <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Walking the DOM

Element-Only Navigation

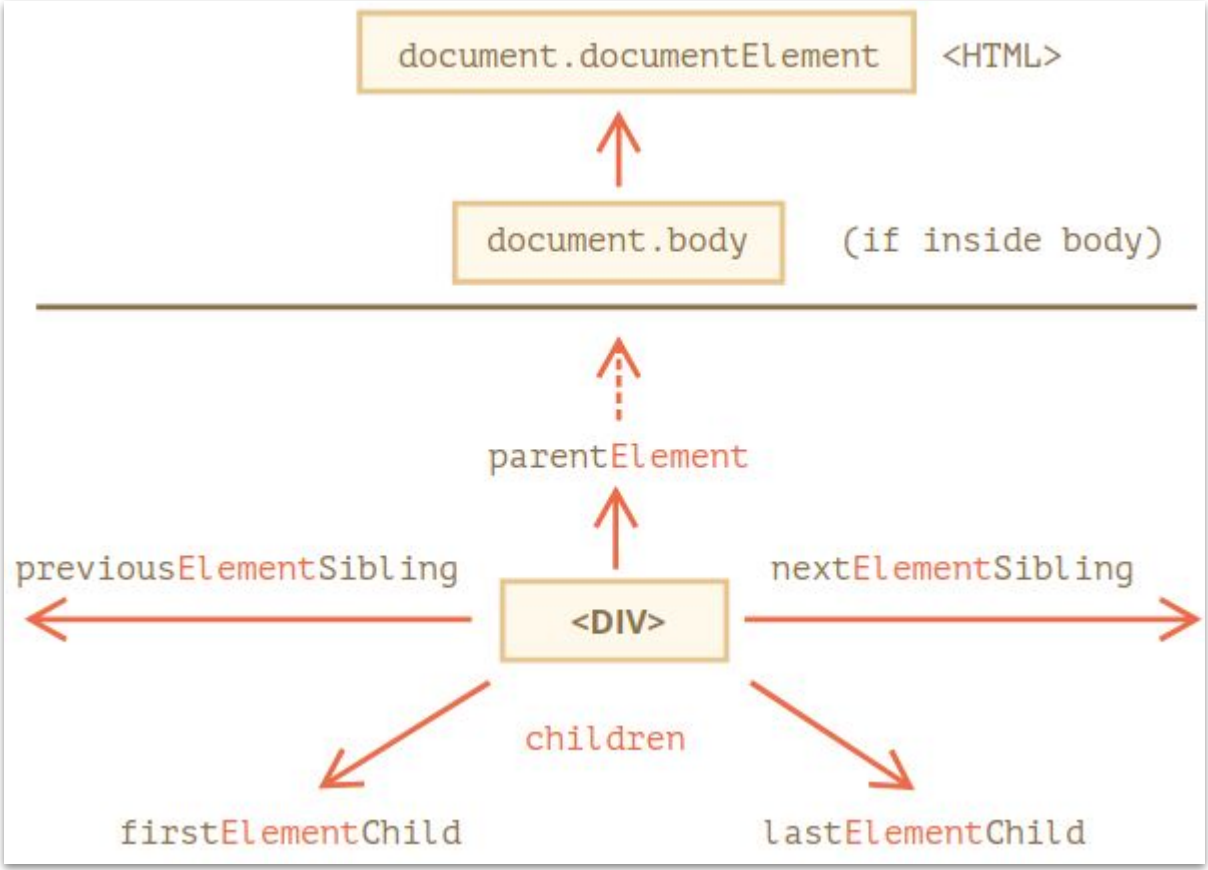
Navigation properties listed above refer to all nodes

For instance, in childNodes we can see both text nodes, element nodes, and even comment nodes if they exist

If you want to manipulate element nodes that represent tags and form the structure of the page, you can use element-only navigation

Walking the DOM

Element-Only Navigation



Walking the DOM

Element-Only Navigation

`children` – only those children that are element nodes.

`firstElementChild`, `lastElementChild` – first and last element children

`previousElementSibling`, `nextElementSibling` – neighbor elements

`parentElement` – parent element

Walking the DOM Tables

The **<table>** element supports the following

`table.rows` – the collection of `<tr>` elements of the table.

`table.caption/tHead/tFoot` – references to elements `<caption>`, `<thead>`, `<tfoot>`.

`table.tBodies` – the collection of `<tbody>` elements

Walking the DOM Tables

`<thead>`, **`<tfoot>`**, **`<tbody>`** elements provide the rows property:

`tbody.rows` – the collection of `<tr>` inside

Walking the DOM Tables

<tr>

`tr.cells` – the collection of <td> and <th> cells inside the given <tr>

`tr.sectionRowIndex` – the position (index) of the given <tr> inside the enclosing <thead>/<tbody>/<tfoot>

`tr.rowIndex` – the number of the <tr> in the table as a whole

Walking the DOM Tables

<td> and **<th>**

`td.cellIndex` – the index of the cell inside the enclosing `<tr>`

Walking the DOM Tables

```
<table id="table">
 <tr>
   <td>one</td>
   <td>two</td>
 </tr>
 <tr>
   <td>three</td>
   <td>four</td>
 </tr>
</table>
```

```
// get td with "two"
let td = table.rows[0].cells[1];
td.style.backgroundColor = "red";
```

Exercise

For each of the following, give at least one way of access them:

- The `<div>` DOM node
- The `<ul>` DOM node
- The second `<li>` (with Pete)

```
<html>
 <body>
   <div>Users:</div>
   <ul>
     <li>John</li>
     <li>Pete</li>
   </ul>
 </body>
</html>
```

Searching: `getElement*`, `querySelector*`

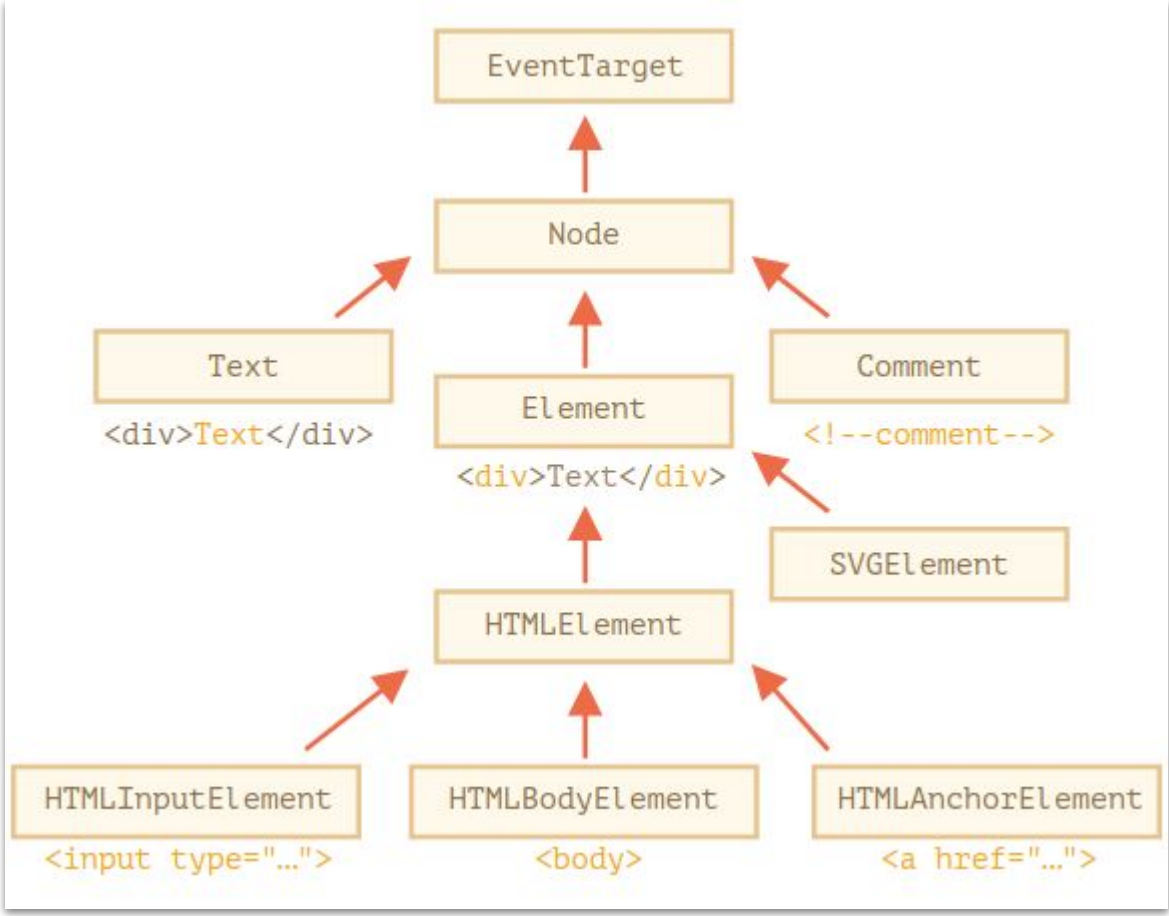
There are 6 main methods to search for nodes in DOM:

Method	Search By
`querySelector()`	`CSS-selector`
`querySelectorAll()`	`CSS-selector`
`getElementById()`	`id`
`getElementsByName()`	`name`
`getElementsByTagName()`	`tag or *`
`getElementsByClassName()`	`class`

Node Properties

Each DOM node belongs to a certain class

The classes form a hierarchy



Main Node Properties

`nodeType`

We can use it to see if a node is a text or an element node

It has a numeric value:

1 for elements,

3 for text nodes, and a few others for other node types

It is read-only

Main Node Properties

`nodeName/tagName`

- For elements, tag name
- For non-element nodes `nodeName` describes what it is
- It is read-only

Main Node Properties

`innerHTML`

The HTML content of the element

Can be modified

Main Node Properties

`outerHTML`

The full HTML of the element

A write operation into `elem.outerHTML` does not touch elem itself, instead it gets replaced with the new HTML in the outer context

Main Node Properties

`nodeValue/data`

The content of a non-element node (text, comment)

These two are almost the same, usually we use data

Can be modified

Main Node Properties

`textContent`

- The text inside the element: `HTML` minus all `<tags>`
- Writing into it puts the text inside the element, with all special characters and tags treated exactly as text
- Can safely insert user-generated text and protect from unwanted HTML insertions

Main Node Properties

`hidden`

When set to true, does the same as CSS display:none.

Modifying the Document

Methods to create new nodes

`document.createElement(tag)` – creates an element with the given tag,

`document.createTextNode(value)` – creates a text node (rarely used),

`elem.cloneNode(deep)` – clones the element, if deep==true then with all descendants

Modifying the Document

Insertion and removal

`node.append(...nodes or strings)` – insert into node, at the end

`node.prepend(...nodes or strings)` – insert into node, at the beginning

`node.before(...nodes or strings)` –- insert right before node

Modifying the Document

Insertion and removal

`node.after(...nodes or strings)` –- insert right after node

`node.replaceWith(...nodes or strings)` –- replace node

`node.remove()` –- remove the node

Text strings are inserted “as text”

Modifying the Document

Insertion and removal

`node.after(...nodes or strings)` –- insert right after node

`node.replaceWith(...nodes or strings)` –- replace node

`node.remove()` –- remove the node

Text strings are inserted “as text”

Browser Events

An event is a signal that something has happened

All DOM nodes generate such signals (but events are not limited to DOM)

DOM Events

Mouse Events

`click` – when the mouse clicks on an element (touchscreen devices generate it on a tap).

`contextmenu` – when the mouse right-clicks on an element

DOM Events

Mouse Events

`mouseover / mouseout` – when the mouse cursor comes over / leaves an element.

`mousedown / mouseup` – when the mouse button is pressed / released over an element.

`mousemove` – when the mouse is moved

DOM Events

Keyboard Events

`keydown` and `keyup` – when a keyboard key is pressed and released

DOM Events

Form Element Events

`submit` – when the visitor submits a **`<form>`**

`focus` – when the visitor focuses on an element, e.g. on an **`<input>`**

DOM Events

Document Events:

- `DOMContentLoaded` – when the HTML is loaded and processed, DOM is fully built

DOM Events

CSS Events

- `transitionend` – when a CSS-animation finishes

Event Handlers

To react on events we can assign a handler – a function that runs in case of an event

Event Handlers

HTML-Attribute

A handler can be set in HTML with an attribute named **`on<event>`**.

For instance, to assign a click handler for an input, we can use **`onclick`**, like here

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

Event Handlers

DOM Property

We can assign a handler using a DOM property `on<event>`

```
<input id="elem" type="button" value="Click me">
```

```
elem.onclick = function () {
 alert("Thank you");
};
```

To remove a handler – assign `elem.onclick = null`

Accessing the element: `this`

The value of **`this`** inside a handler is the element

In the code below the button shows its contents using `this.innerHTML`

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

Add Event Listener

`addEventListener`

The fundamental problem of the aforementioned ways to assign handlers – we can't assign multiple handlers to one event

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // replaces the previous handler
```

Add Event Listener

`addEventListener`

The syntax to add a handler

```
element.addEventListener(event, handler, [options]);
```

To remove the handler, use **`removeEventListener`**

Add Event Listener

`removeEventListener`

to remove the handler

```
element.removeEventListener(event, handler, [options]);
```

Add Event Listener

```
<input id="elem" type="button" value="Click me"/>
```

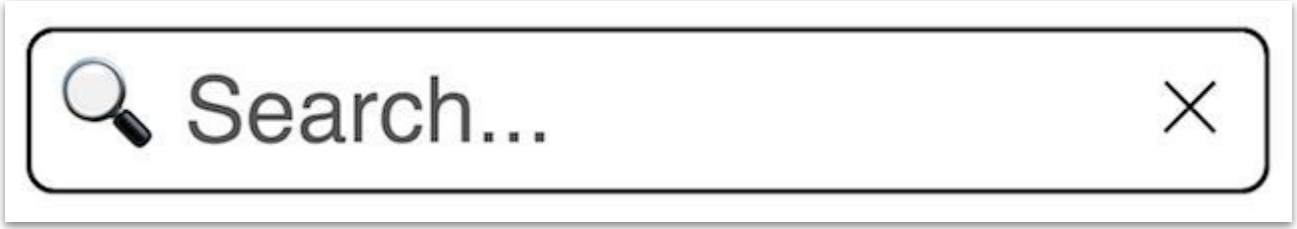
```
function handler1() {
 alert("Thanks!");
}
```

```
function handler2() {
 alert("Thanks again!");
}
```

```
elem.onclick = () => alert("Hello");
elem.addEventListener("click", handler1); // Thanks!
elem.addEventListener("click", handler2); // Thanks again!
```

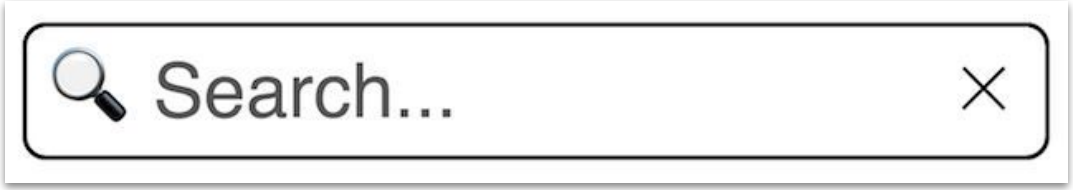
Web Components

How many HTML tags do you need to create the following search box?



Web Components

How many HTML tags do you need to create the following search box?



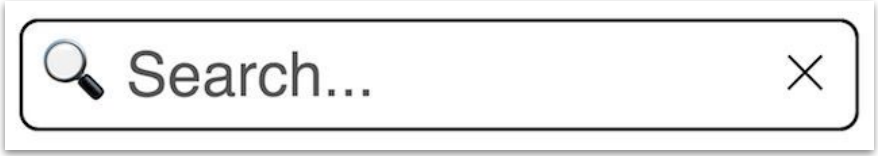
- an `<input>` element to accept and display the user's input
- two `<span>` elements for displaying the magnifying glass and cancel Unicode Glyphs
- a container `<div>` element to hold those three children

Web Components

What else do you need to do?

Web Components

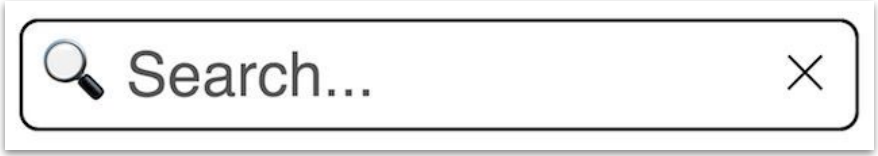
What else do you need to do?



- use CSS to remove the default border around the input box and to add a new border around the container div

Web Components

What else do you need to do to make the search box functional



- use JavaScript to register event handler to handle the click events for the magnifying and cancel icons

Web Components

Most frontend frameworks such as React and Angular support the creation of reusable user interface components like the search box

Web Component is a browser-native alternative that allow JavaScript to **extend HTML with new tags** that work as self-contained, reusable UI components

Using Web Components

As Web components are defined in JavaScript, you need to include the JavaScript file that defines the component

```
<script type="module" src="components/search-box.js"></script>
```

Using Web Components

Web components define **their own `HTML` tag names**, with the important restriction that **those tag names must include a hyphen**

To use a web component, just use its tag in your HTML file

```
<search-box>
 <img src="images/search-icon.png" slot="left" />
 <img src="images/cancel-icon.png" slot="right" />
</search-box>
```

Browser Features of Web Components

The three web browser features that allow us to implement web components

- **HTML Templates**
- **Custom Elements**
- **Shadow DOM**

Networking

Every time you load a web page, the browser makes network requests—using the HTTP and HTTPS protocols—for an HTML file as well as the images, fonts, scripts, and stylesheets that the file depends on

In addition to being able to make network requests in response to user actions, web browsers also expose JavaScript APIs for networking

Networking Javascript APIs

The **`fetch()`** method defines a Promise-based API for making HTTP and HTTPS requests

```
async function isServiceReady() {
 let response = await fetch("/api/service/status");
 let body = await response.text();
 return body === "ready";
}
```

Networking Javascript APIs

The **`Server-Sent Events (or SSE)`** API is event-based interface to HTTP “long polling” techniques where the web server holds the network connection open so that it can send data to the client whenever it wants

Networking Javascript APIs

`WebSockets` allow JavaScript code in the browser to easily exchange **text** and **binary** messages with a server

As with Server-Sent Events, the client must establish the connection, but once the connection is established, the server can asynchronously send messages to the client

Unlike SSE, binary messages are supported, and messages can be sent in both directions, not just from server to client

References

The Modern JavaScript Tutorial

JavaScript: The Definitive Guide, 7th Edition