WHAT IS THE DOM

- D - Document
- O - Object
- M - Model
- hierarchical tree structure of JS nodes (objects)
- …that represent the rendered page
- allows you to read/modify the rendered page
- …via the API calls it exposes.

Browser-side only (No document/page, no DOM!)

BROWSER SIDE JS

- Search the DOM for nodes
- Read details of a node (element)
- Write details to an existing node
- Create new nodes
- Listen for events

Also includes some browser-side storage, navigation, and utilities.

JS CAN BE INLINE (DON'T)

```
<div onload="alert('hello')">Hi</div>
```

DO NOT DO THIS

- A mess to edit
- A mess to maintain
- only allows one handler per event

Also, don't use `alert()`.

JS CAN BE INSIDE A `script` TAG

```
<script>
  alert('hi');
</script>
```

ALMOST NEVER DO THIS

- Harder to edit
- Harder to reuse between files

Also, don't use `alert()`.

JS CAN LOAD FROM A SEPARATE FILE

```
<script src="chat.js"></script>
```

The preferred way.

Generally you want your JS to load after the HTML, so at the bottom of the `<body>`

Why does at the bottom of the `<body>` make a difference?

FINDING A NODE

To interact with elements, first get the nodes.

DOM tree is

- tree-based set of nodes
- matches the page structure
  - Ex: node for `<html>` contains the nodes for `<head>` and `<body>`

`window` is the top-level global of the browser. (`window.foo` and the global `foo` are the same thing)

Top-level of DOM tree: `document` (`window.document`)

GETTING AN ELEMENT

A number of methods exist to find certain nodes:

- `document.getElementById()` (note: singular!)
- `document.getElementsByTagName()`
- `document.getElementsByClassName()`

Most of these return a NodeList, which is LIKE an array, but is NOT an array.

`Array.from(nodeList)` will give you a real array, with the right methods

SELECTORS

We already know a way to select one or more elements though: **CSS selectors**

- `document.querySelector()` - Returns first matching element
- `document.querySelectorAll()` - returns all matching elements (NodeList)

READING FROM A NODE

A Node is an object like any other

- has predefined methods and properties

Common ones:

- `.innerHTML`
- `.innerText`
- `.classList.contains()`
- `.id`
- `.getAttribute()`
- `.dataset` - A little special, check MDN
- `.value`

CREATING A NEW NODE

```
const el = document.createElement('div');
el.innerText = 'Hello World';
document.querySelector('body').appendChild(el);
```

```
const el = document.createElement('div');
document.querySelector('body').appendChild(el);
el.innerHTML = '<p>Hello</p><p>World</p>';
```

Second one will

- cause two renders, not one
  - Because it was appended, then updated
- the innerHTML implicitly creates new nodes
- Tip: Prefer to set `innerHTML`/`innerText` only. Only use `appendChild` when you must.

MODIFYING A NODE

```
const el = document.querySelector('.to-send');
el.value = 'boring conversation anyway';
el.classList.add('some-class-name');
el.disabled = true;
```

- `classList` is the best way to interact with classes
  - Don't overwrite `class` attribute - there may be other classes
- Don't style an element via properties - add/remove classes

EVENTS

When any running JS is done

- JS enters the 'Event Loop' - waiting for events

If an event occurs (click, keypress, mousemove, etc)

- the system looks to for any assigned "handlers".

If so, that code is run

When any running JS is done

- See the top and start again

ADDING AN EVENT LISTENER

Assign a callback function to the event ON A NODE.

```
const el = document.querySelector('.outgoing button');
// Passing named function
el.addEventListener('click', doSomething);
```

Can pass a named function, or a function directly

```
// Passing a function defined inline
el.addEventListener('click', function() {
  console.log("I can't handle the pressure!");
});
```

EVENT OBJECTS

Each event handler is called and passed an event object (in many cases we ignore it, but it still happens).

```
const el = document.querySelector('.to-send');
el.addEventListener('keydown', function( event ) {
  // event.target is the node that the event happened to
  console.log(event.target.value);
});
```

DEFAULT ACTIONS

Some events have "default" handlers, like clicking a link causing navigation.

These occur after custom actions, and the custom actions can decide to stop them.

```
const el = document.querySelector('.outgoing button');
el.addEventListener('click', function( event ) {
  event.preventDefault(); // button will not submit form
});
```

EVENT PROPAGATION

Propagation, or "bubbling", is where an event on a node, after the listeners on that node are finished, will trigger the listeners on the parent node, then the grandparent, and so forth up to the document.

1. Event triggered on a node
2. Listeners for that node for that event happen
3. That event is triggered on parent node
4. Repeat until there is no parent node

PROPAGATION IS USEFUL

Useful when you have **a list of nodes that**

- Will have the same event and the same reaction to it
- Are added/removed to (meaning you would have to remove/add listeners)

Put a single listener on an ancestor instead of on each of the many nodes

`event.target` still points to the original node that got the event, not the one with the listener

`event.stopPropagation()` does what it says

PROPAGATION EXAMPLE

```html
<ul class="todos">
  <li><span class="todo complete">Sleep</span></li>
  <li><span class="todo">Eat</span></li>
  <li><span class="todo">Knock things off shelves</span></li>
</ul>
```

```css
.todo.complete {
  text-decoration: line-through;
}
```

```javascript
const list = document.querySelector('.todos');
list.addEventListener('click', (e) => {
  if(e.target.classList.contains('todo')) {
    e.target.classList.toggle('complete');
  }
});
```

IIFE

Any variable or function-keyword function created in your JS file that isn't inside a function/block will be created in the GLOBAL scope.

That's bad.

```
(function() {
  const foo = `this is in the function scope,
    not in the global scope`;
})();
```

This is an IIFE (Immediately Invoked Function Expression). Put all your Browser-based JS code in one.