WHAT IS THE DOM

- D - Document
- O - Object
- M - Model

The DOM is a set of JS objects that represent the rendered page AND allow you to read/modify it via the API calls it exposes.

Read the page. Modify it. via JS. Only browser-side.

Also includes some browser 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

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 the page is done loading, the JS enters the 'Event Loop' - waiting for signals (events).

When an event occurs (a click, a keypress, a mousemove, etc) the systems looks to see if there are any assigned "handlers".

If so, that code is run. Once it is done, the event loop continues.

ADDING AN EVENT LISTENER

Assign a callback function to the event ON A NODE.

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

Can pass a function, or define one inline:

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

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 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 ) {
  console.log(event.target.value);
});
```

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.

PROPAGATION IS USEFUL

Useful:

- when you have a long list of nodes with similar treatments
- if you have a list of nodes that nodes are regularly added/removed from

Put a single listener on an ancestor instead of adding/removing from 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

```
<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>
```

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

```
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.