

# **Client-side (Browser) JavaScript (JS)**

- Code runs IN THE BROWSER
- Code is unaware of anything not on the page
- Code IS aware of the page
  - And can change it

# How do we get JS onto a page?

We don't type it into the console

Console is great for:

- Checking the current state of the page
- Testing commands
- Testing syntax

# JS can be inline (don't)

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

## AVOID INLINE JS

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

Also, don't use `alert()`

- it "blocks", more on that later

# JS can be inside a script tag

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

## AVOID JS IN A SCRIPT ELEMENT

- Harder to edit
- Harder to reuse between files

Also, don't use `alert()`

- Some students miss the first message

# JS can load from a separate file

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

The preferred way

You often want your JS to load after the HTML

- Put `<script>` tag at the bottom of the `<body>`
  - just before the `</body>`
  - Could also use `defer` attribute

Why does the location of the script tag matter?

- Browser loads and runs the JS as it renders
- In `<head>` most HTML elements aren't known

# Try it!

- Create an HTML file
- Add `<script src="test.js"></script>`
- Create a `test.js` file

```
function greet() {  
  const message = 'Hello';  
  console.log(`${message}, all you cool cats`);  
}  
  
greet();
```

- View the HTML file in browser
  - Either via `File->Open`, or using a static server
  - Get used to running servers!

# Check global greet in console

In console:

- `greet` has a function value
- We polluted the global scope
  - Because we didn't use an IIFE
  - Always use an IIFE
  - Eventually we'll have tools do this for us

# IIFE - Immediately Invoked Function Expression

`var` + `function` variables created outside a function

- will be created in the GLOBAL scope

That's bad

```
(function() {  
  function greet() {  
    console.log(`this is in the function scope,  
      not in the global scope`);  
  }  
  greet();  
})();
```

IIFE (Immediately Invoked Function Expression)

- Put all Browser-based JS code in an IIFE



## **Does that feel "hacky"?**

Later we'll use tools that create IIFE for us

- Good to know why now

# Interacting with the page

Change the HTML to include:

```
<div class="demo">
  <p class="greet">Hello World</p>
</div>
```

Change `test.js` to be:

```
const divEl = document.querySelector(`.demo`);
let count = 0;
const render = () => divEl.innerHTML = `
  <p>You have clicked ${count} times</p>
  <button type="button">Click Me!</button>
`;
divEl.addEventListener('click', () => {
  count++;
  render();
});
render();
```

# Debugging

Devtools -> Sources

- On left menu, find `test.js`
- Click on line 8 *number* (the `count++` line)
  - sets "breakpoint"
- Click the button on the page
  - code runs and pauses at breakpoint
  - see "Scope" on right
    - See `count` value

# More Debugging

- Click "Step Over" button on right-top
  - arrow over dot
  - `count` has changed
  - rendered page has not
- Click "Step Over" again
  - rendered HTML updates
- Click "Resume" (Blue Arrow)
- Reload Page
  - Click button
  - Breakpoint still here!

# Debugging Notes

- Click Button again
  - Hold Down "Resume"
  - Select Play button (no stops for .5 secs)
- Click Button again
  - Go to Console
  - type `count`, see value
    - Even for local scope!

# What is the DOM

- D - Document
- O - Object
- M - Model
- hierarchical tree structure of JS nodes (objects)
- ...represent the rendered page
- allow 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 browser-side storage, navigation, and utilities

# 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()`

a `NodeList` or `HTMLCollection`

- "array-like" (indexed)
- NOT an array (may not have same methods)

`Array.from(arrayLike)` gives an actual array

- with array methods

# Selectors

We already have a way to select one or more elements:

- **CSS selectors**
- `document.querySelector()` - First matching node
- `document.querySelectorAll()` - NodeList (all)

# Reading from a node

- DOM Nodes have methods and properties
- Check MDN for more detail

Common ones:

- `.innerHTML`
- `.innerText`
- `.classList.contains()`
- `.id`
- `.getAttribute()`
- `.dataset`
- `.value`

## **.innerHTML**

Reading `.innerHTML` gives the HTML contents as string

- DO NOT TRY TO PARSE THE HTML!
  - The browser is a much better parser than you
- Rare to read innerHTML
  - Usually only to save, add, and replace

## **.innerText**

`.innerText` gives the TEXT contents of the node

- HTML is stripped out
- You rarely want to read this
  - Poor idea to read the DOM
    - Other ways to know your app state
    - Doesn't scale well

## **.classList**

`.classList` has methods

- `.contains()` to see if class is on element
- Better than modifying/parsing `class` attribute
  - `class` is space-separated attribute
  - Common mistakes avoided
- classes very often used to manage UI changes

## **.dataset**

`.dataset` is a special object

- properties match up to special attributes
- values will be strings
- attributes begin with "data-"
- attribute names converted to camelCase

```
<div data-name="Jorts" data-age="3" data-has-napped="true" />
```

- `.dataset.name`
- `.dataset.age`
- `.dataset.hasNapped` // Notice camelCase!

## **.value**

- for input-related elements
- holds the *current* value
  - such as something typed/selected
    - even if not sent!
- not the hardcoded `value=` in the original HTML
  - DOM is the rendered page
  - not the original HTML



# Creating a new node

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

- Creates element
- Updates element
- Adds element to page
  - Triggers browser rendering

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

- Creates element
- Adds element to page (render)
- Updates element (render!)

# innerHTML implicitly creates/configures elements

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

Using `innerHTML`

- Implicitly creates elements
- Implicitly configures attributes
- Adds Nodes to tree if parent in tree

# **innerHTML**

- "easier" than:
  - creating each element
  - setting values on each element
  - attaching child elements
- "riskier" however
  - unsanitized user input can inject JS/CSS
- "more maintainable"
  - change is easier to manage
  - output is more clear

# Modifying a node

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

- `classList` to interact with classes
  - Don't overwrite `class` attribute
    - May be other classes
- Don't style an element via properties
  - **add/remove classes** instead
- Don't use `style` attribute
  - People miss the first time I say it

# Example: Light/Dark theme

Imagine you:

- have a page
- want a button
  - to change between light and dark theme

# Do not do: Direct styling

Do NOT try to change the `style` of each element

- complex
- easy to mess up
- hard to keep up with changes

Instead, have CSS for both

- based off of a class on a top-level element
- button changes that class
- CSS will or will not match!

# Demonstration

```
<div class="content">
  <p>Maru</p>
  <p>Grumpy Cat</p>
  <p>Lilbub</p>
</div>
<button class="theme" type="button">Toggle Theme</button>
<script src="theme.js"></script>
```

```
.content {
  color: black;
  background-color: #C0FFEE;
}

.content.dark {
  color: white;
  background-color: darkgray;
}
```

# Demonstration JS

```
const button = document.querySelector('.theme');  
button.addEventListener('click', () => {  
  const content = document.querySelector('.content');  
  content.classList.toggle('dark');  
});
```

Changing one class rather than changing specific styles



# 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 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!");  
});
```

# Handler vs Listener

These terms are often used interchangeably

- **listener** says "when this event happens, call this function"
- **handler** is the function that gets called

Technically, the listener puts the handler on the queue

- doesn't call the function directly

# Event objects

Each event handler is called

- Passed an event object
- We may ignore event object
  - It is still passed

```
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 on that node for that event run
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**

- Have the same event and the same reaction to it
- Are added/removed to/from the list
  - 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

```
<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');
  }
});
```



# Dataset to connect w/data

You might need to associate a node with some data

- an identifier
- related data

Example:

- A visible username and related userid

```
<span class="username">Huang</span>
```

# HTML class gets complex

You might use the `class`

```
<span class="username userid-1234">Huang</span>
```

But this can get complex or unwieldy quickly

# HTML dataset

"Dataset" is a particular kind of HTML property

- starts with `data-`
- after `data-` is the name of the real key

```
<span class="username" data-userid="1234">Huang</span>
```

JS can easily access the data, as an object

```
const el = document.querySelector('.username');  
console.log(el.dataset.userid); // "1234"
```

# Multiple properties

You can have multiple properties

- Every value will be a string
- `kebab-case` is translated to `camelCase`

```
<span data-userid="1234" data-dog-lover="no">Huang</span>
```

JS can easily access the data, as an object

```
console.log(el.dataset);  
// { userid: "1234", dogLover: "no" }
```