CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

Today's schedule

Wednesday

- DOM: How to interact with your web page
- HW1 due tonight
- HW2 is out!
- Victoria's Office Hours → moved to Friday again
- Amy and Cindy have office hours at 4pm like usual

Friday

- More DOM
- data attributes
- Browser extensions
- Victoria's Office Hours from 2:30 to 4pm

var, let, const

Declare a variable in JS with one of three keywords:

- Function scope variable:

```
var x = 15;
```

- Block scope variable:

```
let fruit = 'banana';
```

- Block scope constant (cannot be reassigned):

```
const isHungry = true;
```

What's a "block"?

In the context of programming languages, a **block** is a group of 0 or more statements, usually surrounded by curly braces. (wiki / mdn)

- Also known as a **compound statement**
- Not JavaScript-specific; exists in most languages (C++, Java, Python, etc)
- Has **absolutely nothing** to do with the HTML/CSS notion of "block", i.e. block elements

What's a "block"?

For example, the precise definition of an if-statement might look like:

`if (`***expression***`)` ***statement***

And a block might look like

```
{
  console.log('Hello, world!');
  console.log('Today is a good day.');
}
```

A "block" or compound statement is a type of ***statement***, which is why we can execute multiple statements when the condition is true.

Blocks and scope

Most languages that include blocks also tie scoping rules to blocks, i.e. via "block scope":

```
// C++ code, not JS:
if (…) {
    int x = 5;
    ...
}
// can't access x here
```

This is the behavior of Java, C++, C, etc.

Blocks and scope

This is also the behavior of JavaScript variables so long as you use `const` and `let`:

```
if (…) {
    let x = 5;
    ...
}
// can't access x here
```

Blocks and scope

But if you use `var`, the variable exists for the entirety of the function, completely independent of blocks:

```
if (…) {
    var x = 5;
    ...
}
// x is 5 here
```

This is the same behavior as Python, which also has function scope.

* Note that variable hoisting and function scope are not the same thing, either.

Blocks and scope

For more details, come to office hours!

In 193X we encourage you
to always use `let` and `const`,
so you don't need to understand
`var` very deeply anyway.

JavaScript language tour

Arrays

Arrays are Object types used to create lists of data.

```
// Creates an empty list
let list = [];
let groceries = ['milk', 'cocoa puffs'];
groceries[1] = 'kix';
```

- 0-based indexing
- Mutable
- Can check size via `length` property (not function)

Looping through an array

You can use the familiar for-loop to iterate through a list:

```
let groceries = ['milk', 'cocoa puffs', 'tea'];
for (let i = 0; i < groceries.length; i++) {
  console.log(groceries[i]);
}
```

Or use a for-each loop via `for...of` (mdn):

(intuition: for each item of the groceries list)

```
for (let item of groceries) {
    console.log(item);
}
```

Maps through Objects

- Every JavaScript object is a collection of property-value pairs. (We'll talk about this more later.)
- Therefore you can define maps by creating Objects:

```
// Creates an empty object
const prices = {};
const scores = {
  'peach': 100,
  'mario': 88,
  'luigi': 91
};
console.log(scores['peach']);   // 100
```

Maps through Objects

FYI, string keys do not need quotes around them.

Without the quotes, the keys are still of type string.

```
// This is the same as the previous slide.
const scores = {
  peach: 100,
  mario: 88,
  luigi: 91
};
console.log(scores['peach']);   // 100
```

Maps through Objects

There are two ways to access the value of a property:

1. ***objectName*[*property*]**
2. ***objectName.property***

(2 only works for string keys.)

```
const scores = {
  peach: 100,
  mario: 88,
  luigi: 91
};
console.log(scores['peach']);   // 100
console.log(scores.luigi);      // 91
```

Maps through Objects

There are two ways to access the value of a property:

1. ***objectName*[*property*]**
2. ***objectName*.*property***

(2 only works for string keys.)

Generally prefer style (2), unless the property is stored in a variable, or if the property is not a string.

```
const scores = {
  peach: 100,
  mario: 88,
  luigi: 91
};
console.log(scores['peach']);   // 100
scores.luigi = 87;
console.log(scores.luigi);      // 91
```

Maps through Objects

To add a property to an object, name the property and give it a value:

```
const scores = {
  peach: 100,
  mario: 88,
  luigi: 91
};
scores.toad = 72;
let name = 'wario';
scores[name] = 102;
console.log(scores);
```

```
▶ Object {peach: 100, mario: 88, luigi: 91, toad: 72, wario: 102}
```

Maps through Objects

To remove a property to an object, use **delete**:

```
const scores = {
  peach: 100,
  mario: 88,
  luigi: 91
};
scores.toad = 72;
let name = 'wario';
scores[name] = 102;
delete scores.peach;
console.log(scores);
```

```
▶ Object {mario: 88, luigi: 91, toad: 72, wario: 102}
```

Iterating through Map

Iterate through a map using a for...in loop (mdn):

(intuition: for each key in the object)

```
for (key in object) {
   // … do something with object[key]
}
```

```
for (let name in scores) {
   console.log(name + ' got ' + scores[name]);
}
```

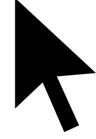
- You can't use for...in on lists; only on object types
- You can't use for...of on objects; only on list types

Events

Event-driven programming

Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.

Click Me!



Example:
Here is a UI element that the user can interact with.

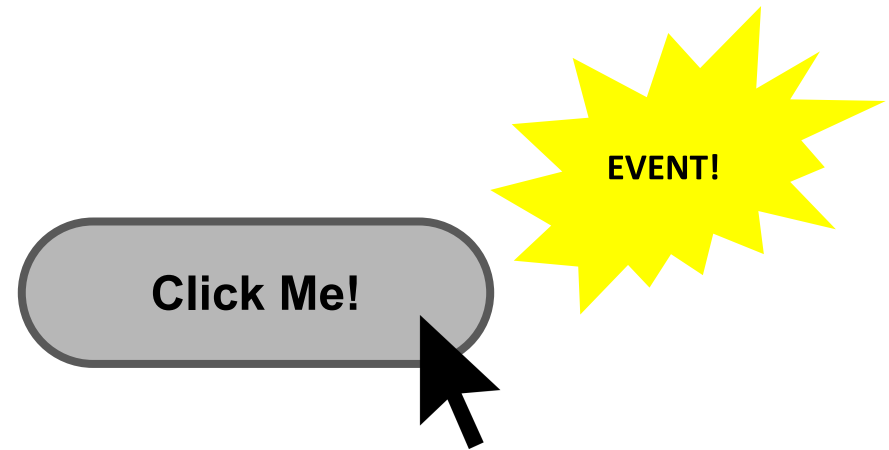
Event-driven programming

Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.

When the user clicks the button...

Event-driven programming

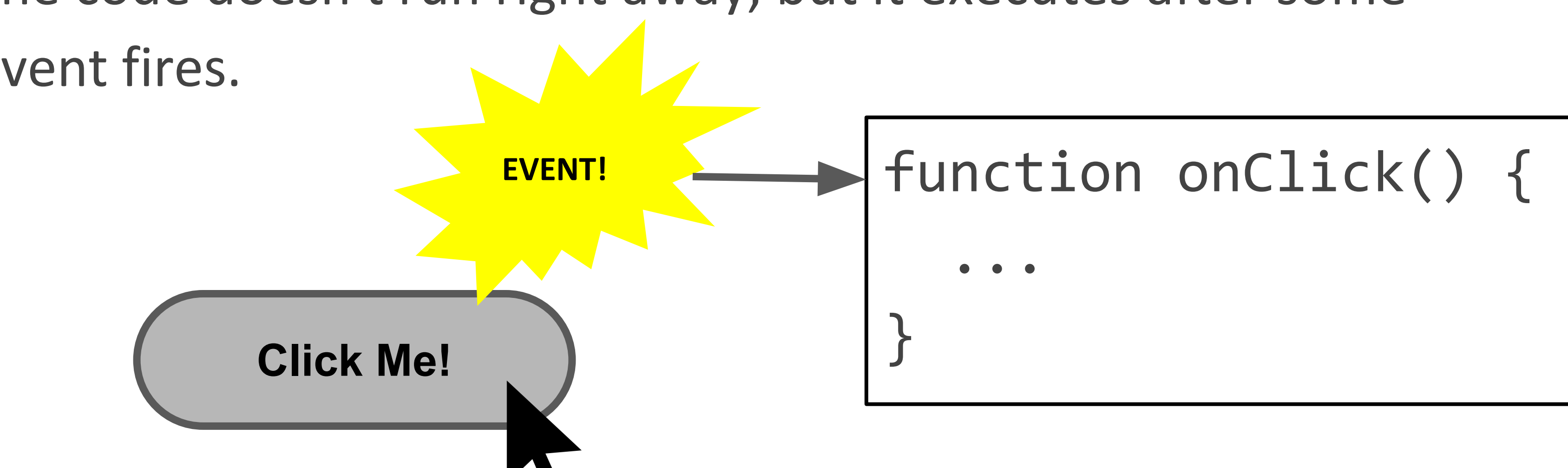
Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.



...the button emits an "**event**," which is like an announcement that some interesting thing has occurred.

Event-driven programming

Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.



Any function listening to that event now executes. This function is called an "**event handler**."

Quick aside...

Let's learn some input-related HTML elements

A few more HTML elements

Buttons:

Single-line text input:

```
<input type="text" />
```

hello

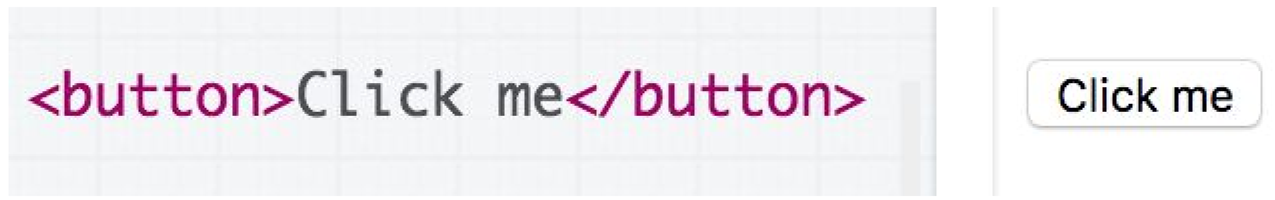
Multi-line text input:

```
<textarea></textarea>
```

I can add
multiple lines of text!

Using event listeners

Let's print "Clicked" to the Web Console when the user clicks the given button:



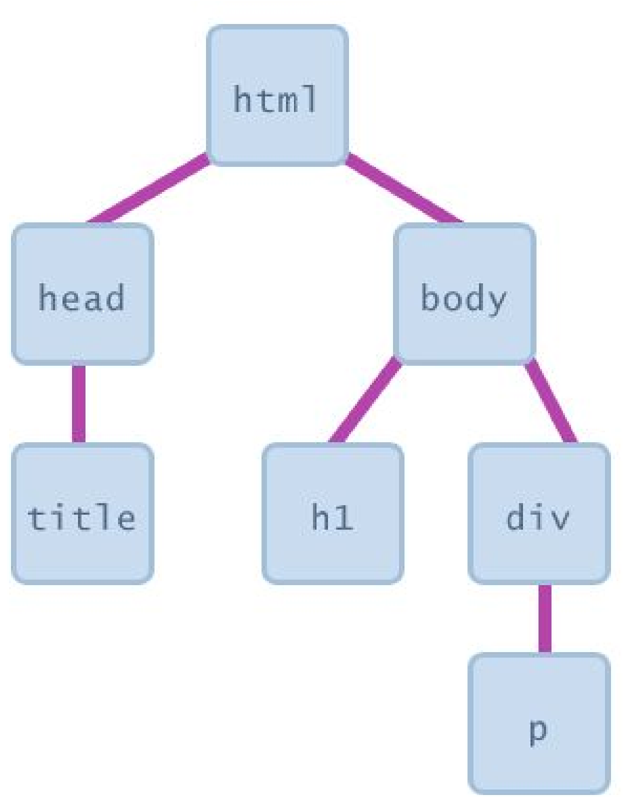
We need to add an event listener to the button...

Q: How do we access an element in HTML from JavaScript?

The DOM

Every element on a page is accessible in JavaScript through the **DOM**: **Document Object Model**

```
<html>
    <head>
        <title></title>
    </head>
    <body>
        <h1></h1>
        <div>
            <p></p>
        </div>
    </body>
</html>
```



The DOM

The DOM is a tree of node objects corresponding to the HTML elements on a page.

- JS code can **examine** these nodes to see the state of an element
    - (e.g. to get what the user typed in a text box)
- JS code can **edit** the attributes of these nodes to change the attributes of an element
    - (e.g. to toggle a style or to change the contents of an <h1> tag)
- JS code can **add elements** to and **remove elements** from a web page by adding and removing nodes from the DOM

How do we access a DOM object from JavaScript?

Getting DOM objects

We can access an HTML element's corresponding DOM node in JavaScript via the `querySelector` function:

```
document.querySelector('css selector');
```

- Returns the **first** element that matches the given CSS selector.

And via the `querySelectorAll` function:

```
document.querySelectorAll('css selector');
```

- Returns **all** elements that match the given CSS selector.

Getting DOM objects

```
// Returns the DOM object for the HTML element
// with id="button", or null if none exists.
let element = document.querySelector('#button');


// Returns a list of DOM objects containing all
// elements that have a "quote" class AND all
// elements that have a "comment" class.
let elementList =
    document.querySelectorAll('.quote, .comment');
```

Adding event listeners

Each DOM object has the following method defined:

addEventListener(***event name, function name***);

- ***event name*** is the string name of the JavaScript event you want to listen to
  - Common ones: `click`, `focus`, `blur`, etc
- ***function name*** is the name of the JavaScript function you want to execute when the event fires

Removing event listeners

To stop listening to an event, use removeEventListener:

removeEventListener(***event name, function name***);

- ***event name*** is the string name of the JavaScript event to stop listening to
- ***function name*** is the name of the JavaScript function you no longer want to execute when the event fires

```html
<html>
▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```javascript
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

script.js ×

```
1 function onClick() {
2   console.log('clicked');
3 }
4
5 const button = document.querySelector('button');
6 button.addEventListener('click', onClick);
7
```

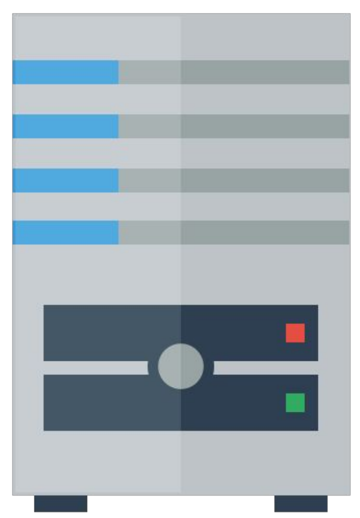
Elements | Console | Sources | Network | Timeline | Profiles | »

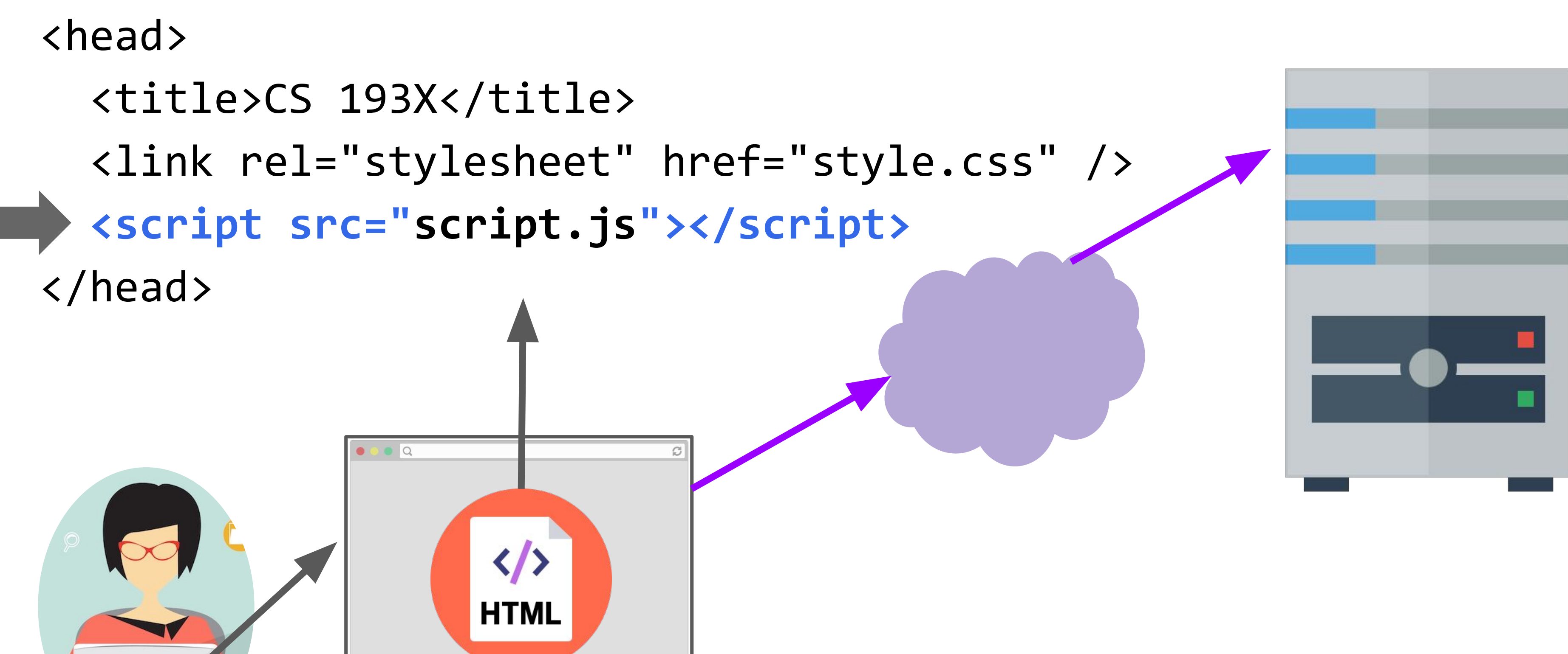
top ▼ | Preserve log

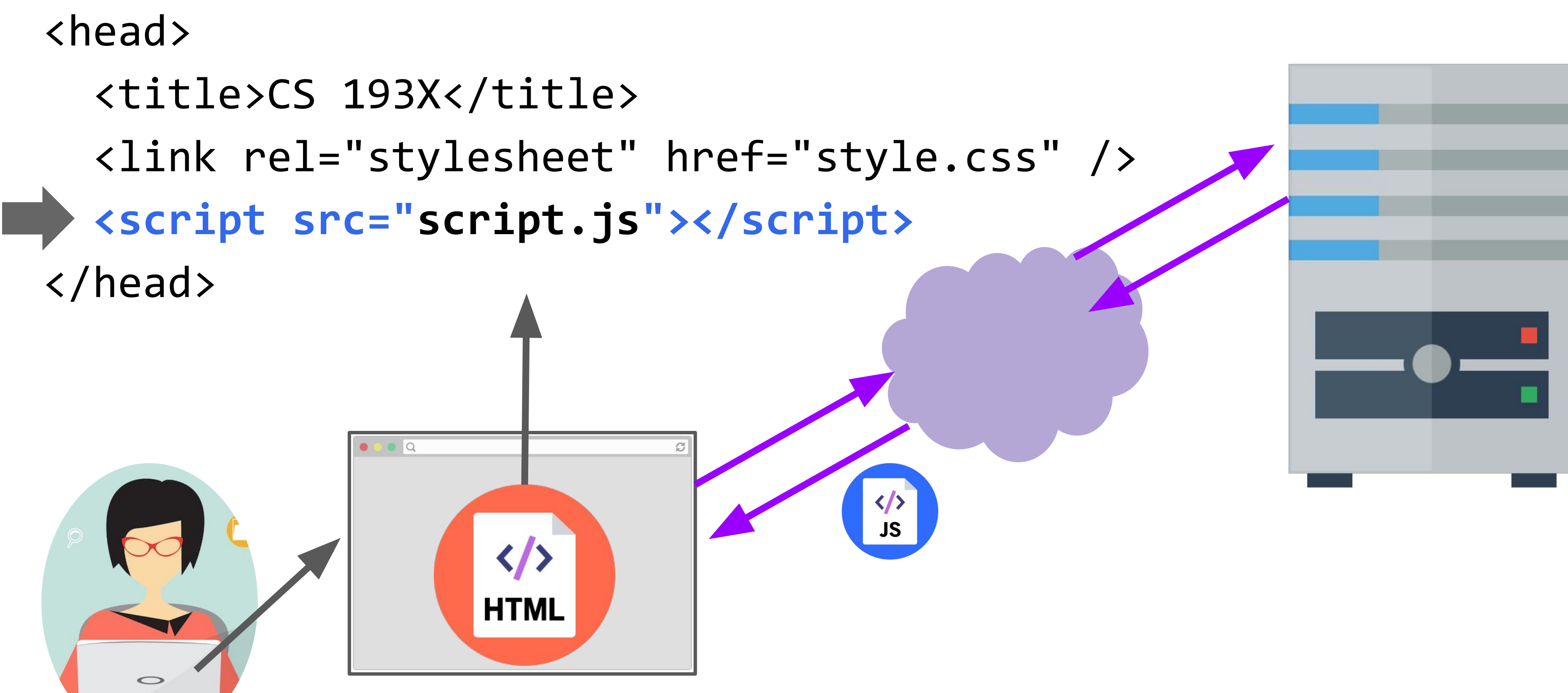
```
Uncaught TypeError: Cannot read property 'addEventListener' of null
    at script.js:6
>
```

Error! Why?

```
<head>
  <title>CS 193X</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```




<head>
<title>CS 193X</title>
<link rel="stylesheet" href="style.css" />
<script src="script.js"></script>
</head>
HTML


<head>
<title>CS 193X</title>
<link rel="stylesheet" href="style.css" />
<script src="script.js"></script>
</head>
HTML
JS

```
<head>
  <title>CS 193X</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```

HTML

JS

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelec
button.addEventListener('click', o
```

```
<head>
  <title>CS 193X</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```

HTML

JS

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelec
button.addEventListener('click', o
```

```
<head>
  <title>CS 193X</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelec
button.addEventListener('click', o
```

We are only at the <script> tag, which is at the top of the document… so the <button> isn't available yet.

```
<head>
  <title>CS 193X</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```

HTML

JS

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelec
button.addEventListener('click', o
```

Therefore `querySelector` returns `null`, and we can't call `addEventListener` on `null`.

Use `defer`

You can add the `defer` attribute onto the script tag so that the JavaScript doesn't execute until after the DOM is loaded (mdn):

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

Use defer

You can add the `defer` attribute onto the script tag so that the JavaScript doesn't execute until after the DOM is loaded (mdn):

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

Other old-school ways of doing this (**don't do these**):

- Put the `<script>` tag at the bottom of the page
- Listen for the `"load"` event on the window object

You will see tons of examples on the internet that do this. They are out of date. defer is widely supported and better.

```html
<html>
▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer></script>
  </head>
▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```javascript
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

Log messages aren't so interesting...

How do we interact with the page?

A few technical details

The DOM objects that we retrieve from `querySelector` and `querySelectorAll` have types:

- Every DOM node is of general type [Node](#) (an interface)
- [`Element`](#) implements the [Node](#) interface
  (FYI: This has nothing to do with NodeJS, if you've heard of that)
- Each HTML element has a specific [`Element`](#) derived class, like [`HTMLImageElement`](#)

Attributes and DOM properties

Roughly every **attribute** on an HTML element is a **property** on its respective DOM object...

HTML

```
<img src="puppy.png" />
```

JavaScript

```
const element = document.querySelector('img');
element.src = 'bear.png';
```

(But you should always check the JavaScript spec to be sure. In this case, check the `HTMLImageElement`.)

Adding and removing classes

You can control **classes** applied to an HTML element via `classList.add` and `classList.remove`:

```
const image = document.querySelector('img');
// Adds a CSS class called "active".
image.classList.add('active');
// Removes a CSS class called "hidden".
image.classList.remove('hidden');
```

(More on classList)

Example: Present

See the CodePen - much more exciting!

```
function openPresent() {
  const image = document.querySelector('img');
  image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

Finding the element twice...

```
function openPresent() {
  const image = document.querySelector('img');
  image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

This redundancy is unfortunate.

Q: Is there a way to fix it?

Finding the element twice...

```
function openPresent() {
  const image = document.querySelector('img');
  image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

This redundancy is unfortunate.

Q: Is there a way to fix it?

CodePen

Event.currentTarget

An Event element is passed to the listener as a parameter:

```
function openPresent(event) {
  const image = event.currentTarget;
  image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
  image.removeEventListener('click', openPresent);
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

The event's currentTarget property is a reference to the object that we attached to the event, in this case the <img>'s Element to which we added the listener.

Psst.. Not to be confused with `Event.target`

(Note: Event has both:

- ***event*.**[`target`]: the element that was clicked / "dispatched the event" (might be a child of the target)
- ***event*.**[`currentTarget`]: the element that the original event handler was attached to)

(Programming note: I got these mixed up in lecture and used `target` when I meant `currentTarget`, so I'm correctly the slides retroactively. Whoops, sorry!)

Example: Present

Click for a present:

It would be nice to change the text after the present is "opened"...

Some properties of `Element` objects

Property	Description
`id`	The value of the id attribute of the element, as a string
`innerHTML`	The raw HTML between the starting and ending tags of an element, as a string
`textContent`	The text content of a node and its descendants. (This property is inherited from Node)
`classList`	An object containing the classes applied to the element

Maybe we can adjust the **`textContent`**!

CodePen

```
function openPresent(event) {
  const image = event.currentTarget;
  image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';

  const title = document.querySelector('h1');
  title.textContent = 'Hooray!';

  image.removeEventListener('click', openPresent);
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

We can select the h1 element then set its `textContent` to change what is displayed in the h1. (CodePen)

Another approach:
Changing the elements

Add elements via DOM

We can create elements dynamically and add them to the web page via [`createElement`] and [`appendChild`]:

`document.createElement(`***tag string***`)`

element`.appendChild(`***element***`);`

Technically you can also add elements to the webpage via `innerHTML`, but it poses a [security risk].

```
// Try not to use innerHTML like this:
element.innerHTML = '<h1>Hooray!</h1>';
```

Remove elements via DOM

We can also call remove elements from the DOM by calling the `remove`() method on the DOM object:

element`.remove();`

And actually setting the `innerHTML` of an element to an **empty string** is a fine way of removing all children from a parent node:

```
// This is fine and poses no security risk.
element.innerHTML = '';
```

```
function openPresent(event) {
  const newHeader = document.createElement('h1');
  newHeader.textContent = 'Hooray!';
  const newImage = document.createElement('img');
  newImage.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';

  const container = document.querySelector('#container');
  container.innerHTML = '';
  container.appendChild(newHeader);
  container.appendChild(newImage);
}

const image = document.querySelector('img');
image.addEventListener('click', openPresent);
```

[CodePen]

Click for a present:

Hmm, the effect is slightly janky though:
The text changes faster than the image loads.

Q: How do we fix this issue?

`display: none;`

There is yet another super helpful value for `display`:

```
display: block;
display: inline;
display: inline-block;
display: flex;
display: none;
```

`display: none;` turns off rendering for the element and all its children. It's treated as if the element were not in the document at all...

`display: none;`

There is yet another super helpful value for `display`:

```
display: block;
display: inline;
display: inline-block;
display: flex;
display: none;
```

`display: none;` turns off rendering for the element and all its children. It's treated as if the element were not in the document at all…

...but the content (such as the images) is still loaded.

```html
<div id="gift-outside">
  <h1>Click for a present:</h1>
  <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png" />
</div>
<div id="gift-inside" class="hidden">
  <h1>Hooray!</h1>
  <img src="https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif" />
</div>
```

```css
.hidden {
  display: none;
}
```

We can add both views to the HTML, with one view hidden by default…

(CodePen)

```
function openPresent(event) {
  const image = event.currentTarget;
  image.removeEventListener('click', openPresent);

  const giftOutside = document.querySelector('#gift-outside');
  const giftInside = document.querySelector('#gift-inside');
  giftOutside.classList.add('hidden');
  giftInside.classList.remove('hidden');
}

const image = document.querySelector('#gift-outside img');
image.addEventListener('click', openPresent);
```

Then we toggle the display state of the containers by adding/removing the `hidden` class.

(CodePen)

Recap

Several strategies for updating HTML elements in JS:

1. Change content of existing HTML elements in page:

- Good for simple text updates

2. Add elements via `createElement` and `appendChild`

- Needed if you're adding a variable number of elements

3. Put all "views" in the HTML but set inactive ones to hidden, then update `display` state as necessary.

- Good when you know ahead of time what element(s) you want to display
- Can be used in conjunction with (1) and/or (2)