

Interactive Web Programming

1st semester of 2021

Murilo Camargos
(**murilo.filho@fgv.br**)

Heavily based on [Victoria Kirst](#) slides

Today's schedule

Today

- Intro to Amateur JavaScript
 - What is JavaScript?
 - Tour of language features
 - Basic event handling

Thursday

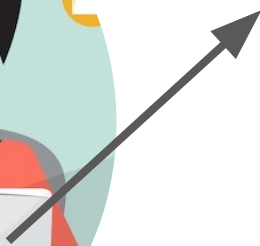
- DOM: How to interact with your web page
- HW1 due
- HW2 goes out

How do web pages
work again?

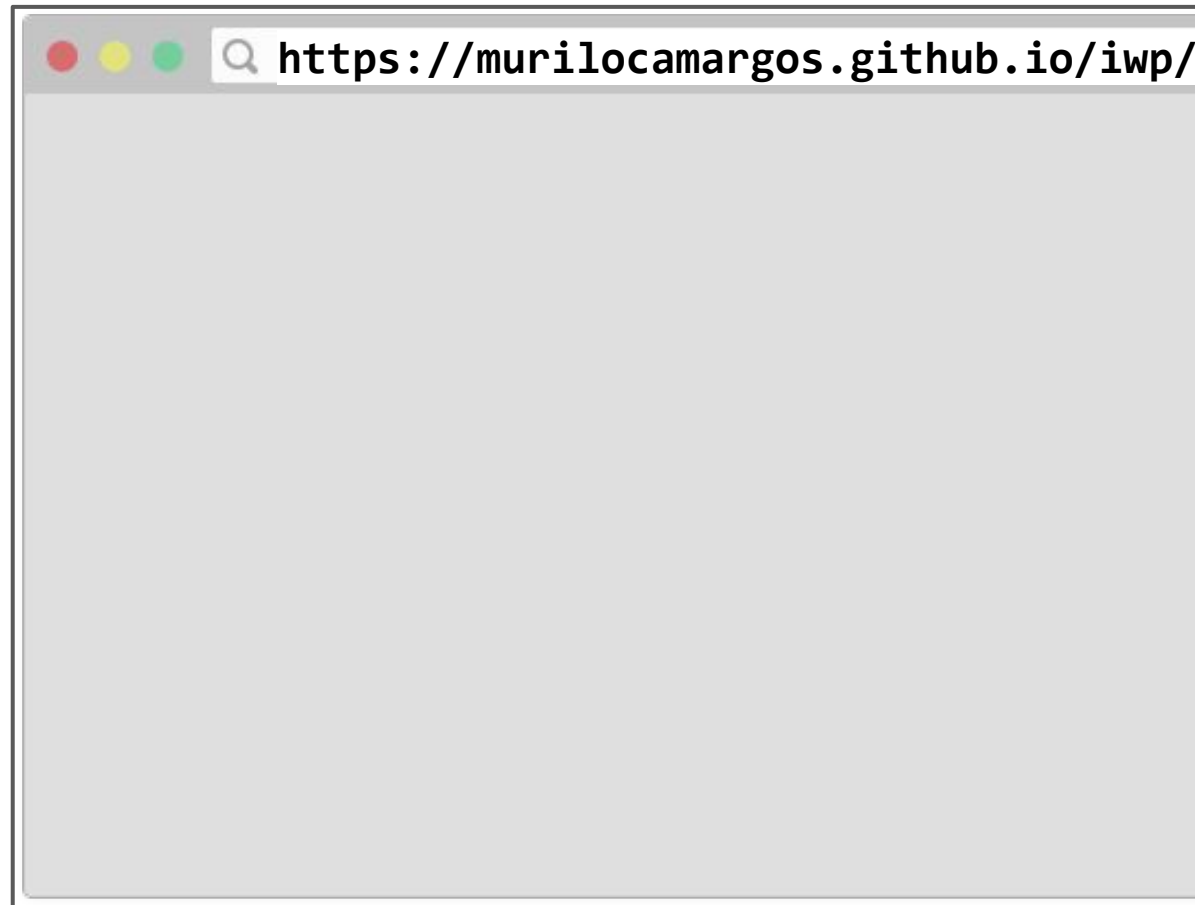
You are on
your laptop

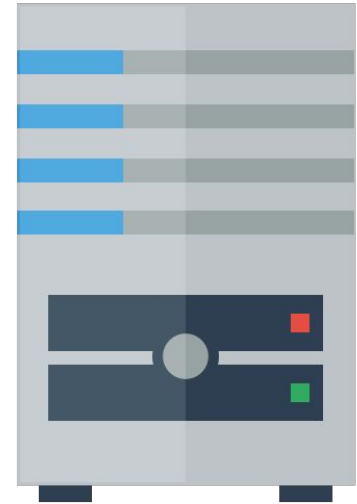
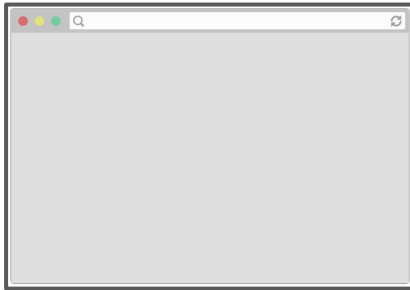


Your laptop is
running a web
browser, e.g.
Chrome



You type a URL in
the address bar and
hit "enter"





(Warning: Somewhat inaccurate,
massive hand-waving begins now.

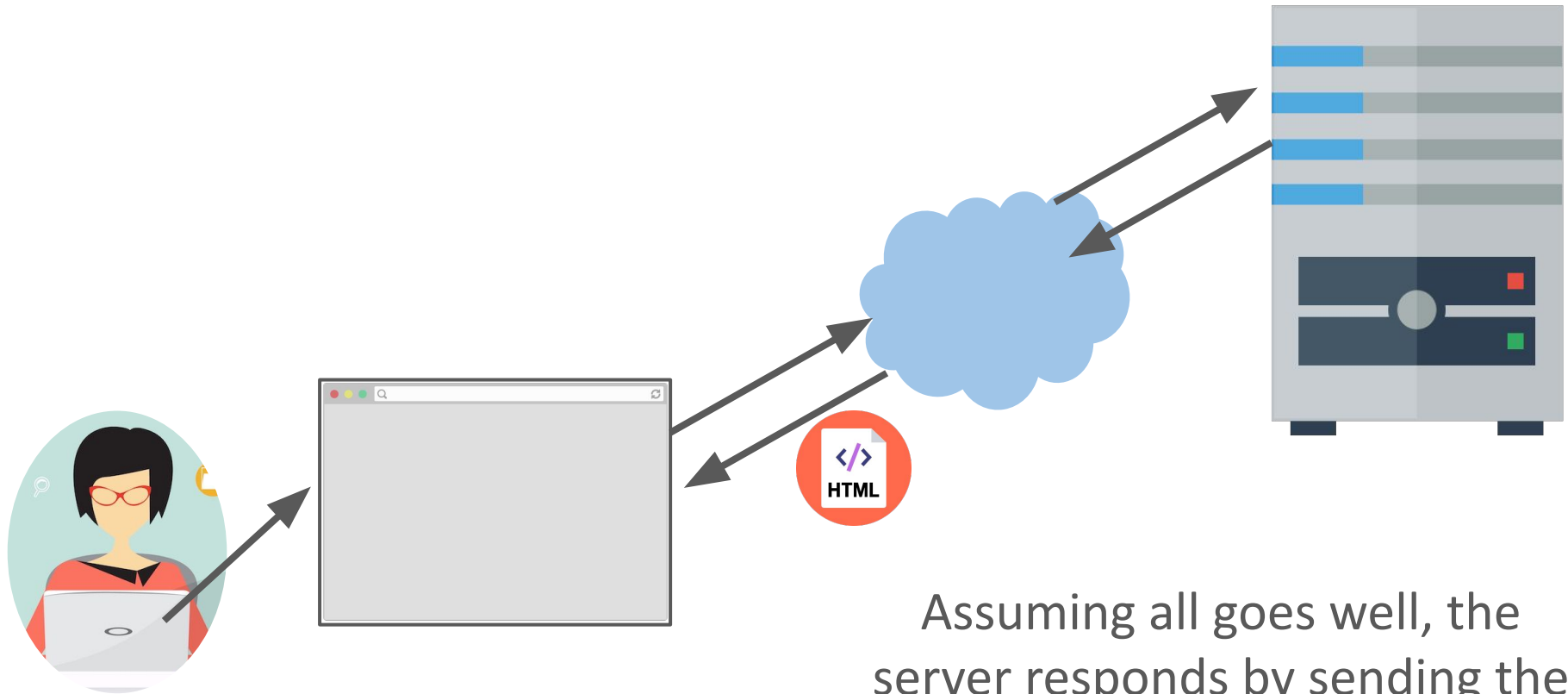
See [this Quora answer](#) for slightly more detailed/accurate handwaving)

Server at
<https://murilocamargos.github.io/iwp/>

Browser sends an HTTP request saying
"Please GET me the index.html file at
<https://murilocamargos.github.io/iwp/>"



Server at
<https://murilocamargos.github.io/iwp/>



Assuming all goes well, the server responds by sending the HTML file through the internet back to the browser to display.

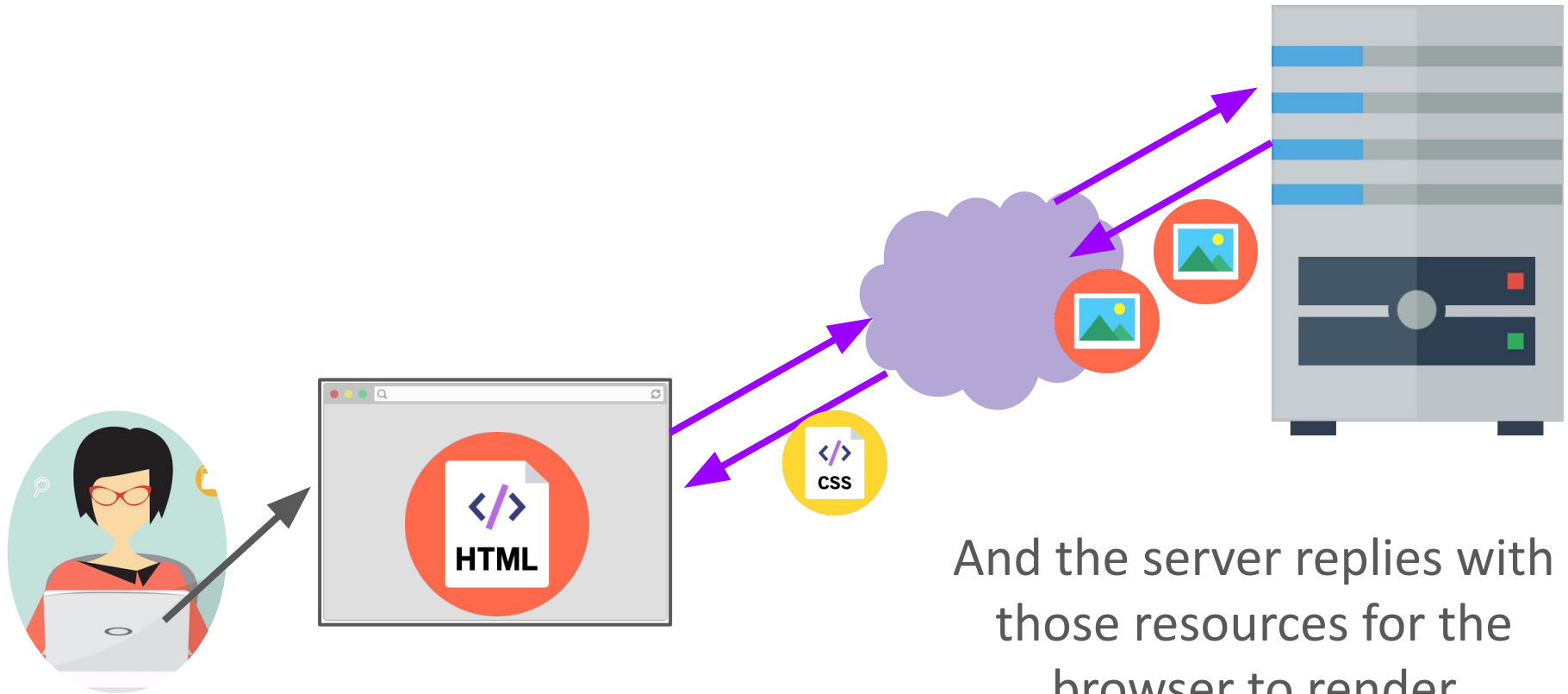
Server at

<https://murilocamargos.github.io/iwp/>

The HTML will include things like
`` and
`<link src="style.css" .../>`
which generate more requests for
those resources



Server at
<https://murilocamargos.github.io/iwp/>



Finally, when all resources are loaded,
we see the loaded web page

A screenshot of a web browser window. The address bar shows the URL `https://murilocamargos.github.io/iwp/`. The page has a dark blue header with navigation links: SYLLABUS, INFORMATION, LECTURES, and HOMEWORK. The main content area is white and contains the following text:

FGV EMAp
IWP: 1/2021

Welcome to IWP: Interactive Web Programming! In this class, you will learn modern full-stack web development techniques without use of a frontend framework.

- **Prereq** [Programming Languages](#)
- **Lectures** Tue-Thu, 14h00-15h30 online
- **Exams** No exams at all.

Announcements

- [02/03] [Homework 0](#) is released and is due **Tue, Mar 9**.

Contact

Any questions regarding content, homework or personal matters, you can contact-me through e-mail:

- Murilo Camargos murilo.filho@fgv.br

At the bottom of the dark blue header, it says "BASED ON STANFORD'S CS193X".



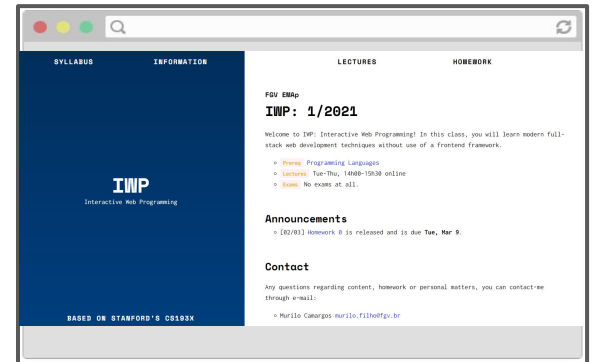
Describes the content and structure of the page

+



Describes the appearance and style of the page

produces



A web page...
that doesn't do
anything

What we've learned so far

We've learned how to build web pages that:

- Look the way we want them to
- Can link to other web pages
- Display differently on different screen sizes



But we don't know how build web pages that **do** anything:

- Get user input
- Save user input
- Show and hide elements when the user interacts with the page
- etc.



What we've learned so far

We've learned how to build web pages that:

- Look the way we want them to
- Can link to other web pages
- Display differently on different screen sizes



But we don't know how build web pages that *do* anything:

- Get user input
- Save user input
- Show and hide elements when the user interacts with the page
- etc.



Enter JavaScript!

JavaScript

JavaScript

JavaScript is a programming language.

It is currently the only programming language that your browser can execute natively. (There are [efforts](#) to change that.)

Therefore if you want to make your web pages do stuff, you must use JavaScript: There are no other options.



JavaScript

- Created in 1995 by Brendan Eich
- JavaScript has nothing to do with Java
 - Literally named that way for [marketing reasons](#)
- The first version was written in 10 days
- Several fundamental language decisions were made because of company politics and not technical reasons

"I was under marketing orders to make it look like Java but not make it too big for its britches ... [it] needed to be a silly little brother language." ([source](#))

JavaScript

- Created in 1995 by Brendan Eich
- JavaScript has nothing to do with Java
 - Literally named that way for [marketing reasons](#)
- The first version was written in 10 days
- Several fundamental language decisions were made because of company politics and not technical reasons

In other words:

**JavaScript is messy and full of drama...
and our only option.**

(though it's gotten much, much better in the last few years)

Our JavaScript Strategy

This week: "**old-school JavaScript**"

- Mostly **not** best practice
 - Everything in global scope
 - No classes / modules
 - Will result in a big mess if you code this way for anything but very small projects
- (But easy to get started)

Next week(?): Modern JavaScript

- More disciplined and based on best practices

JavaScript in the browser

Code in web pages

HTML can embed JavaScript files into the web page via the `<script>` tag.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Interactive Web Programming</title>
    <link rel="stylesheet" href="style.css" />
    <script src="filename.js"></script>
  </head>
  <body>
    ... contents of the page...
  </body>
</html>
```

console.log

You can print log messages in JavaScript by calling `console.log()`:

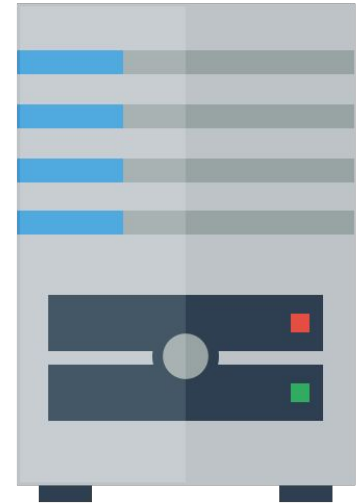
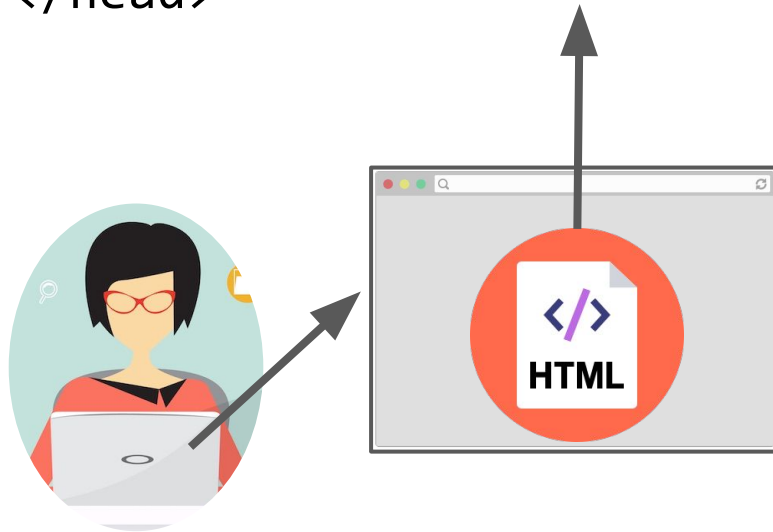
script.js

```
console.log('Hello, world!');
```

This JavaScript's equivalent of Java's `System.out.println`, `print`, `printf`, etc.

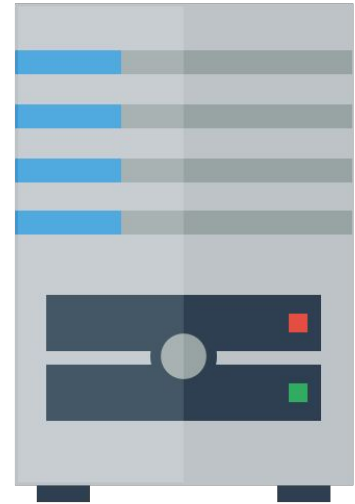
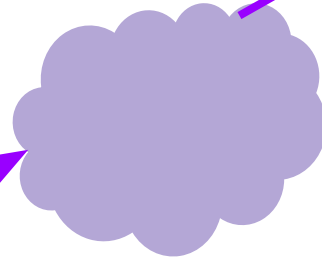
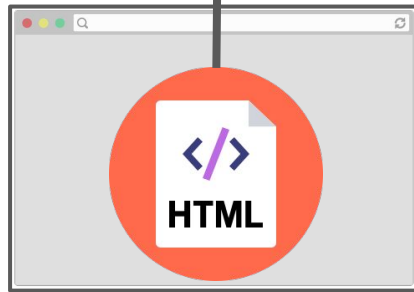
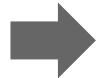
How does JavaScript get loaded?


```
<head>  
  <title>Interactive Web Programming</title>  
  <link rel="stylesheet" href="style.css" />  
  <script src="script.js"></script>  
</head>
```



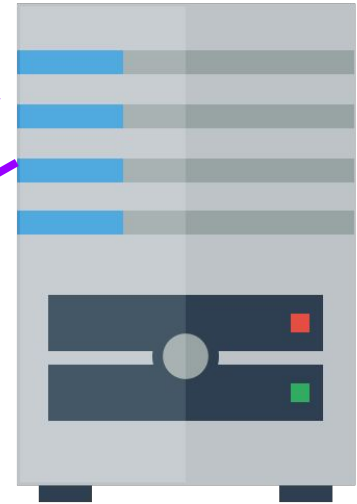
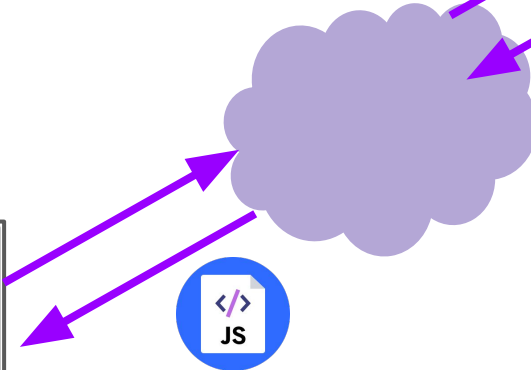
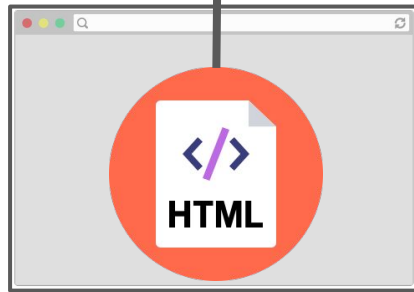
The browser is parsing the HTML file, and gets to a script tag, so it knows it needs to get the script file as well.

```
<head>
  <title>Interactive Web Programming</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```



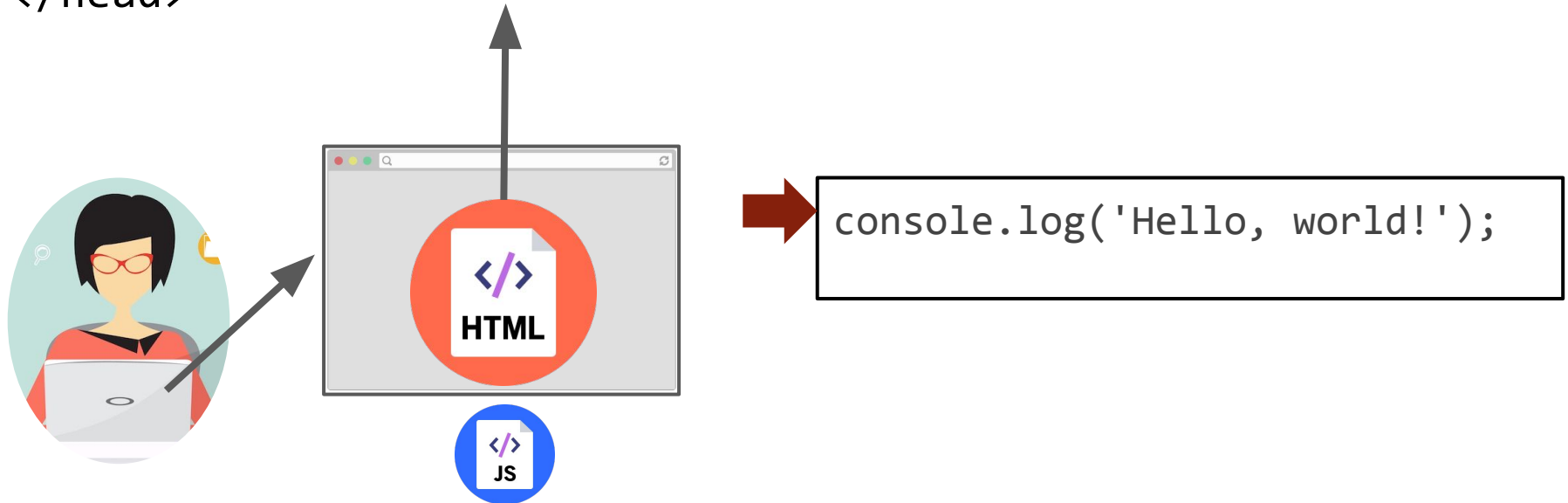
The browser makes a request to the server for the script.js file, just like it would for a CSS file or an image...

```
<head>  
  <title>Interactive Web Programming</title>  
  <link rel="stylesheet" href="style.css" />  
  <script src="script.js"></script>  
</head>
```



And the server responds with the JavaScript file, just like it would with a CSS file or an image...

```
<head>
  <title>Interactive Web Programming</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```



Now at this point, the JavaScript file will execute "**client-side**", or in the browser on the user's computer.

JavaScript execution

There is **no "main method"**

- The script file is executed from top to bottom.

There's **no compilation** by the developer

- JavaScript is compiled and executed on the fly by the browser

(Note that this is slightly different than being "interpreted": see

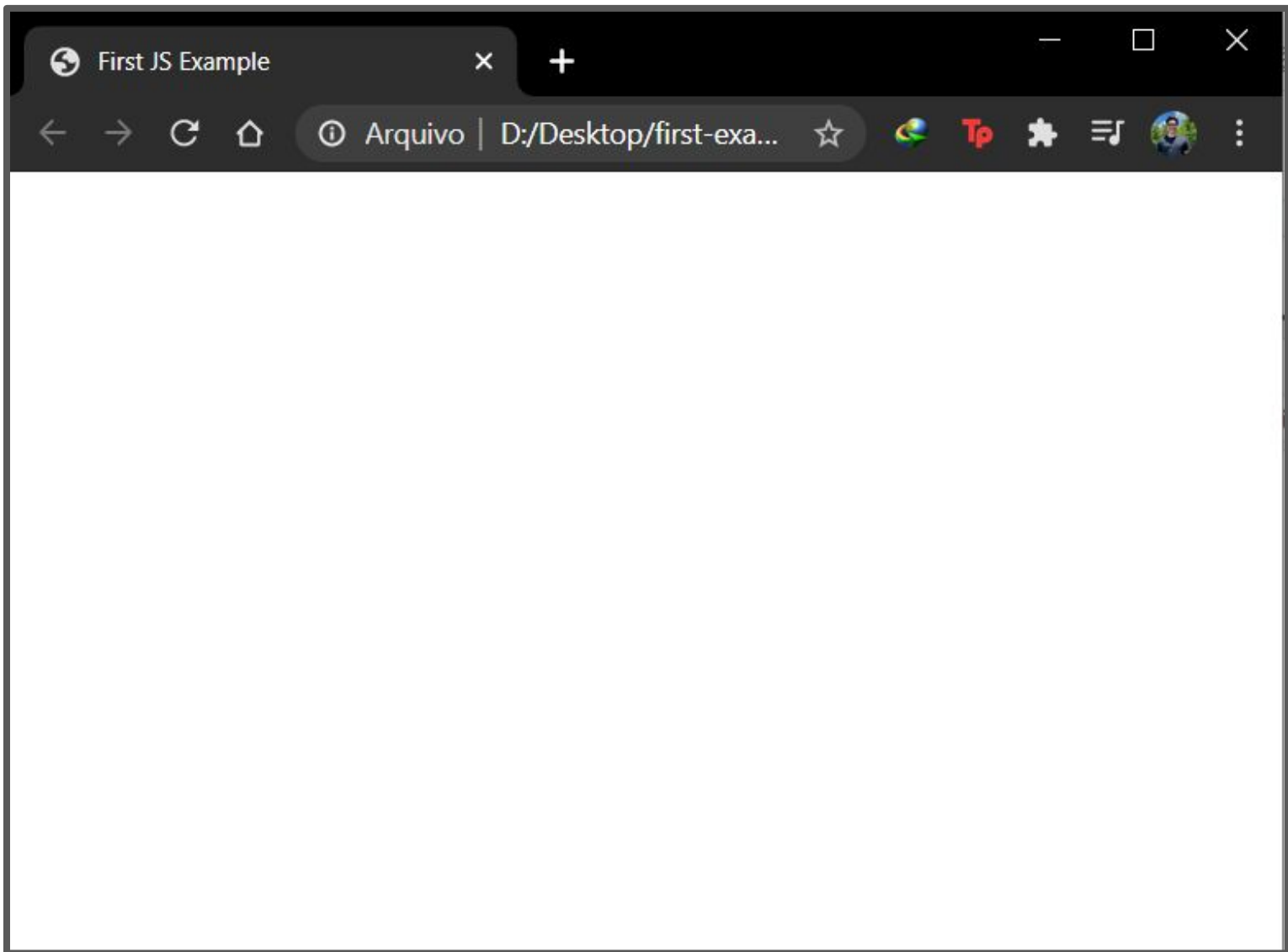
[just-in-time \(JIT\) compilation](#))

first-js.html

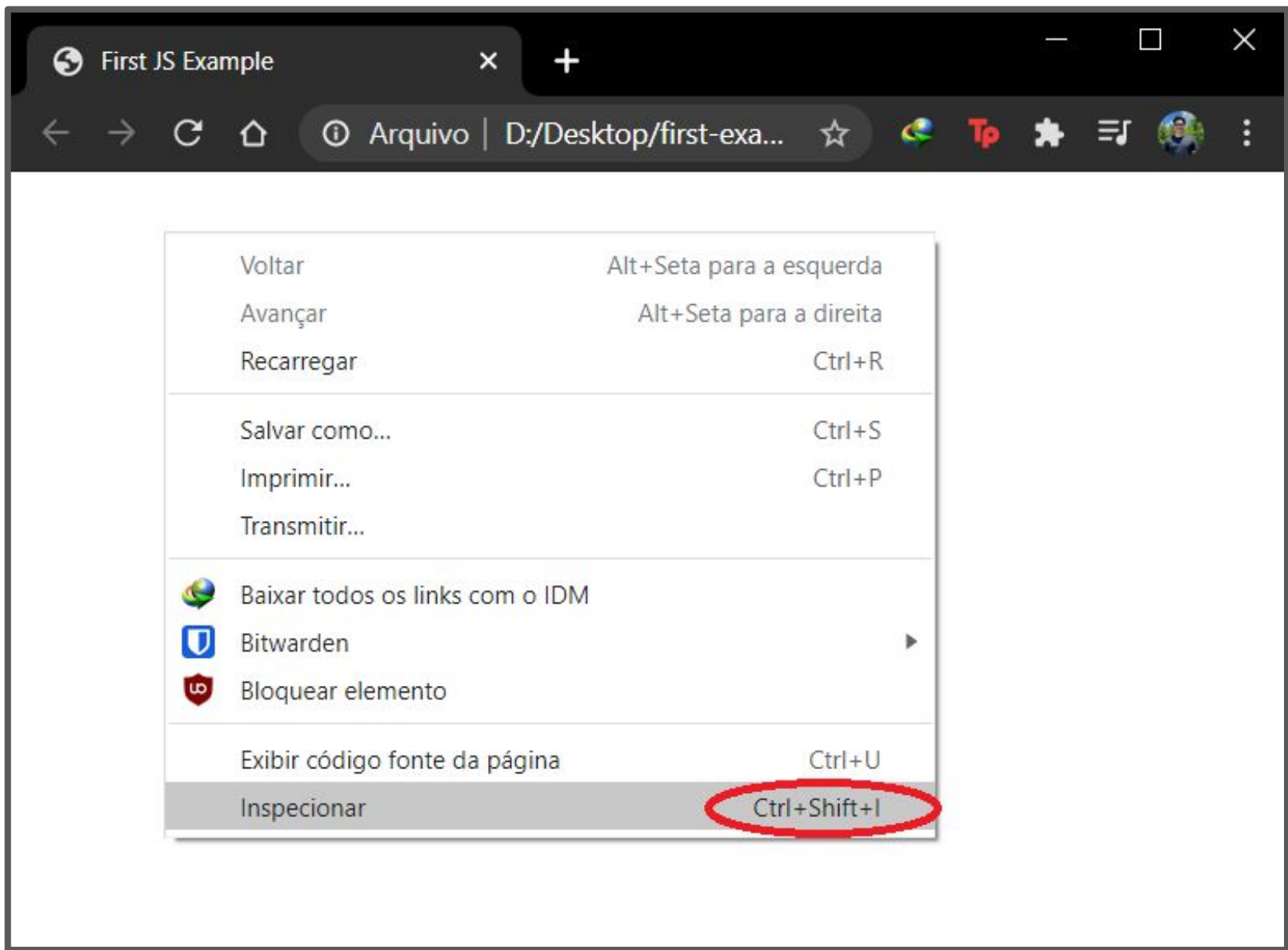
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
  </body>
</html>
```

script.js

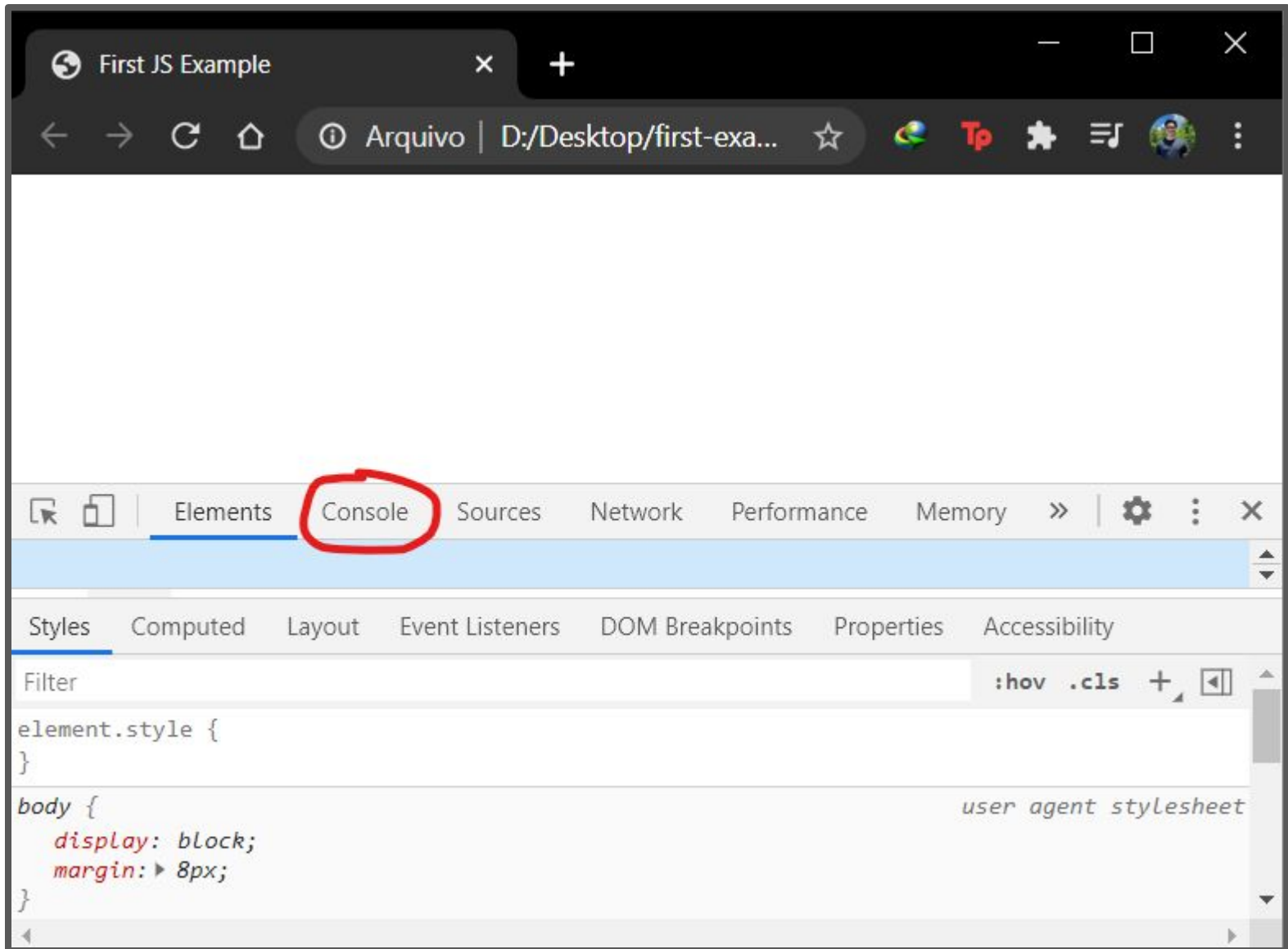
```
console.log('Hello, world!');
```



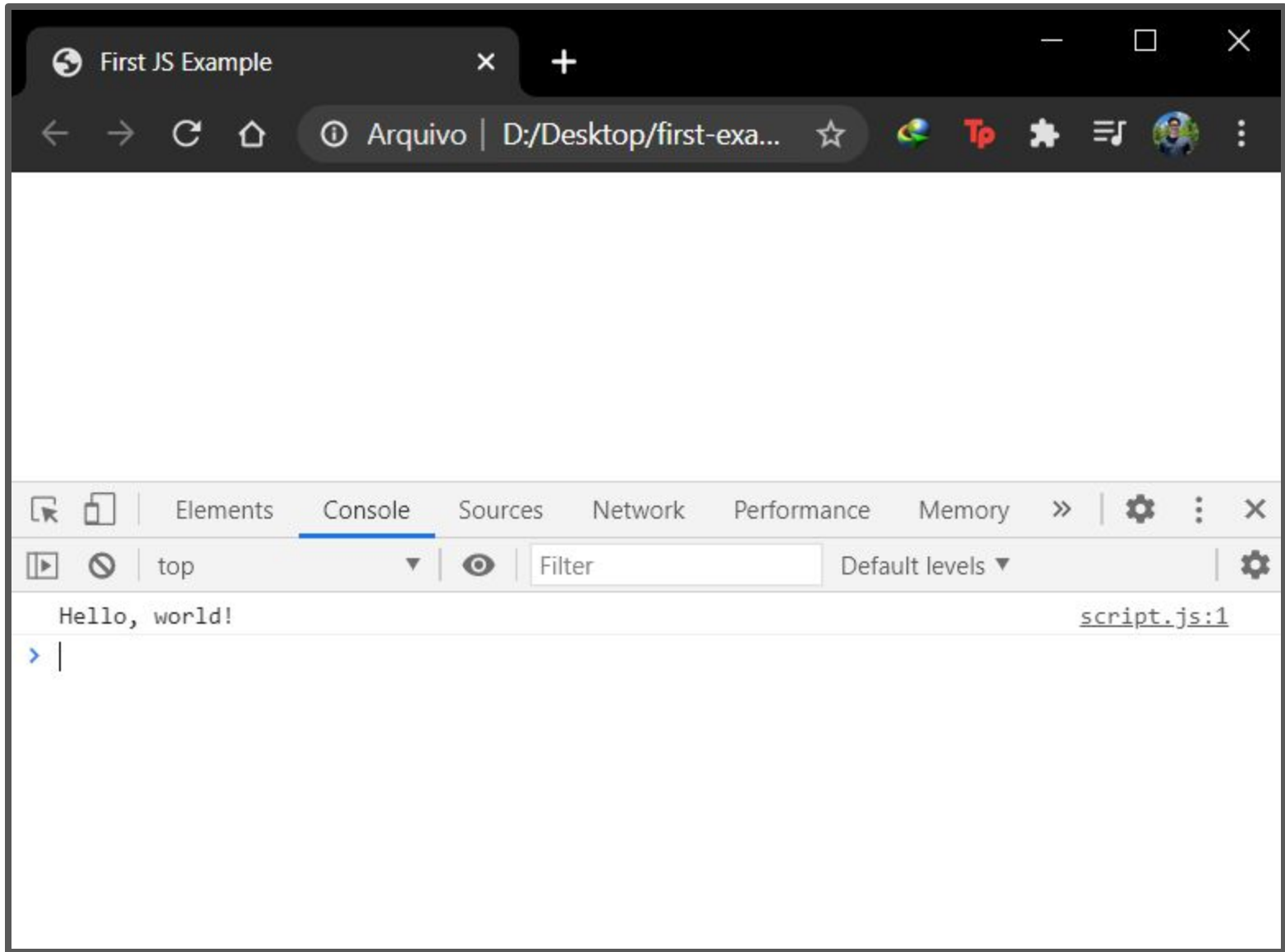
Hey, nothing happened!

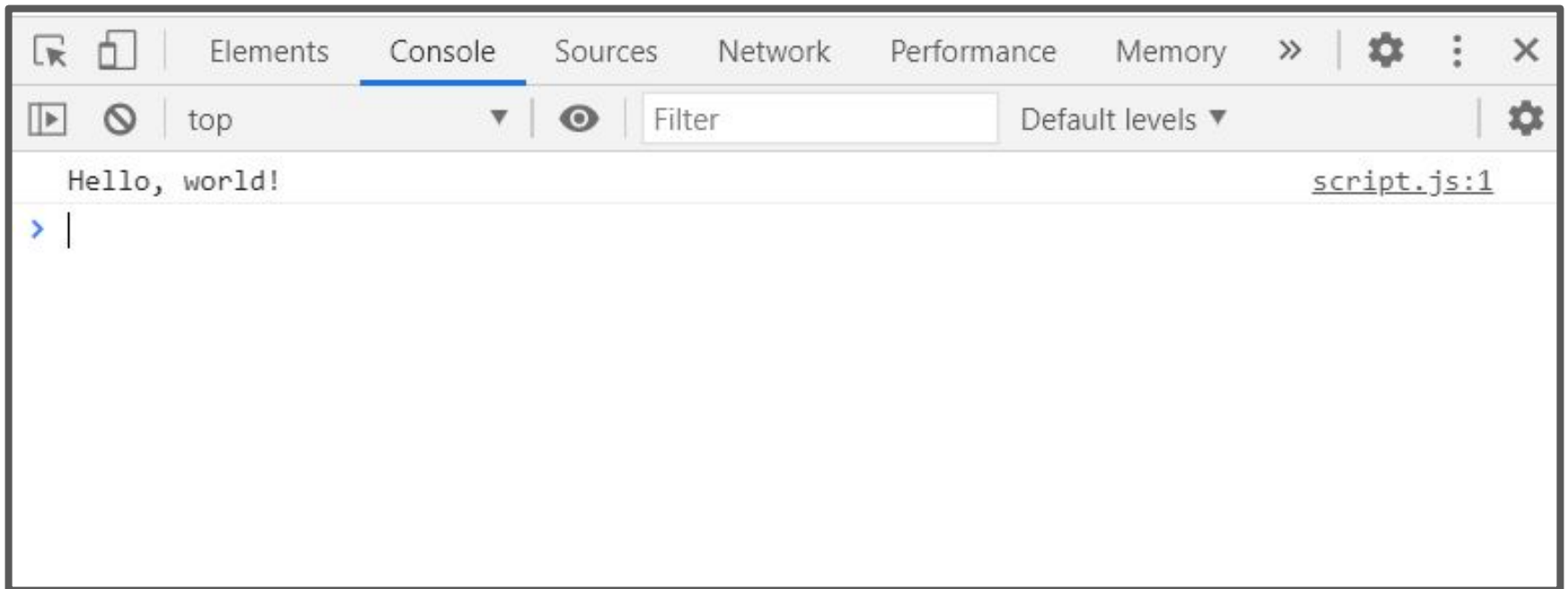


Right-click (or control-click on Mac) and choose "Inspect"
(Ctrl + Shift + I on Chrome)



Click "Console" tab





The "Console" tab is also a [REPL](#), or an interactive language shell, so you can type in JavaScript expressions, etc. to test out the language.

We will be using this throughout the quarter!

JavaScript language features

Same as Java/C++/C-style langs

for-loops:

```
for (let i = 0; i < 5; i++) { ... }
```

while-loops:

```
while (notFinished) { ... }
```

comments:

```
// comment or /* comment */
```

conditionals (if statements):

```
if (...) {  
    ...  
} else {  
    ...  
}
```

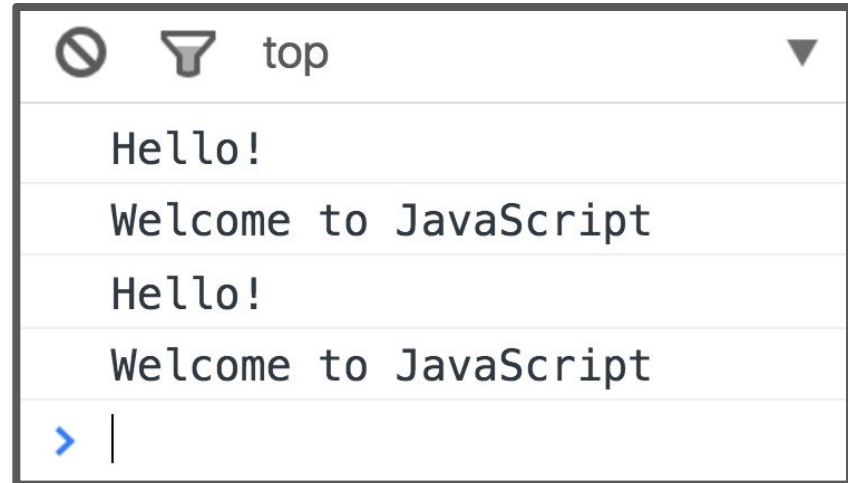
Functions

One way of defining a JavaScript function is with the following syntax:

```
function name() {  
    statement;  
    statement;  
    ...  
}
```

script.js

```
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}  
  
hello();  
hello();
```



```
top  
Hello!  
Welcome to JavaScript  
Hello!  
Welcome to JavaScript  
> |
```

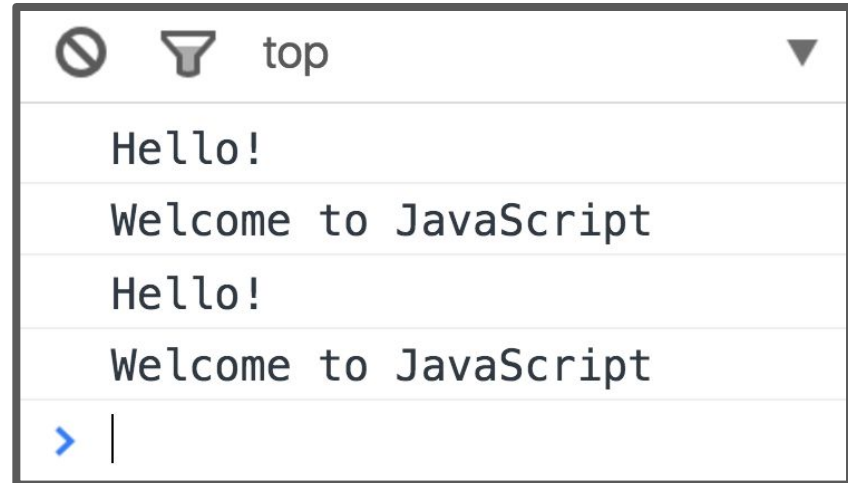
Console output

script.js

```
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

```
hello();  
hello();
```

The browser "executes" the function definition first, but that just creates the `hello` function (and it doesn't run the `hello` function), similar to a variable declaration.



```
top  
Hello!  
Welcome to JavaScript  
Hello!  
Welcome to JavaScript  
> |
```

Console output

script.js

```
hello();  
hello();  
  
function hello() {  
    console.log('Hello!');  
    console.log('Welcome to JavaScript');  
}
```

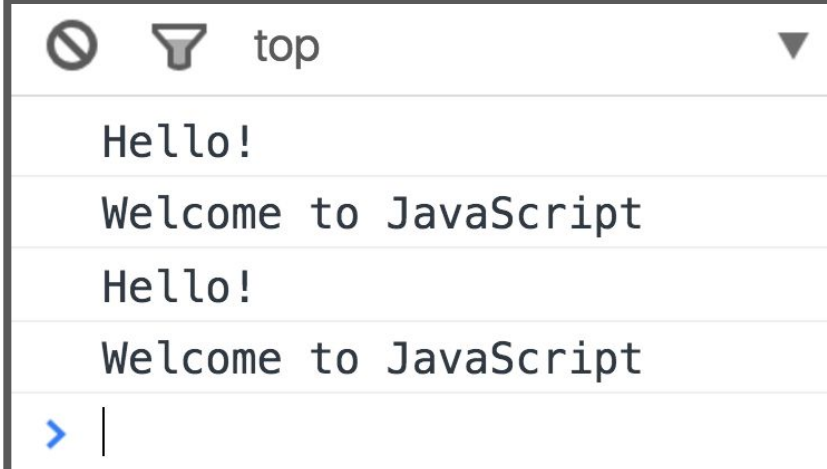
Q: Does this work?

script.js

```
hello();  
hello();  
  
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

A: Yes, for this particular syntax.

This works because function declarations are "**hoisted**" ([mdn](#)). You can think of it as if the definition gets moved to the top of the scope in which it's defined (though that's not what actually happens).



```
top  
Hello!  
Welcome to JavaScript  
Hello!  
Welcome to JavaScript  
> |
```

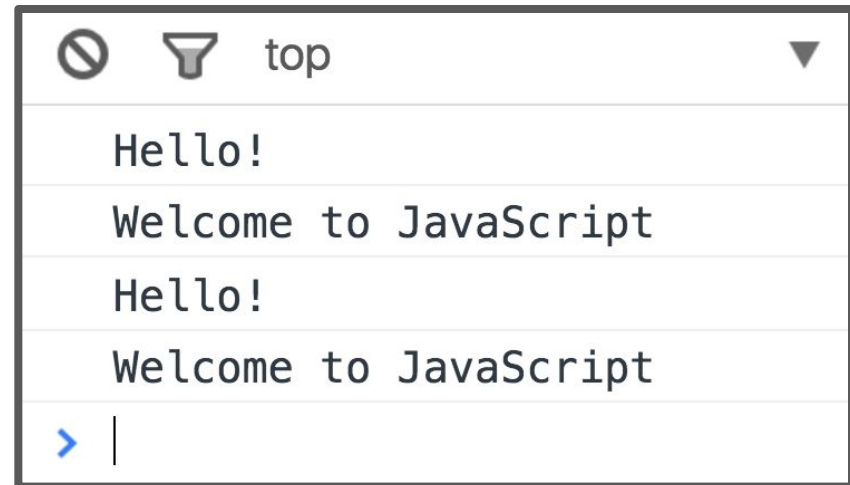
Console output

script.js

```
hello();  
hello();  
  
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

Caveats:

- There are other ways to define functions that do not get hoisted; we'll visit this once we graduate from Amateur JS
- Try not to rely on hoisting when coding. [It gets bad.](#)



Console output

Variables: 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;
```

You do not declare the datatype of the variable before using it ("[dynamically typed](#)")

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.

Function parameters

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
}
```

Function parameters are **not** declared with `var`, `let`, or `const`

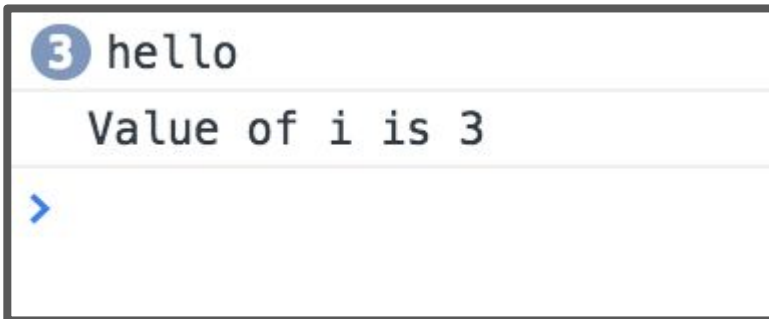
Understanding var

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}  
printMessage('hello', 3);
```

Q: What happens if we try to print "i" at the end of the loop?

Understanding var

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}  
printMessage('hello', 3);
```

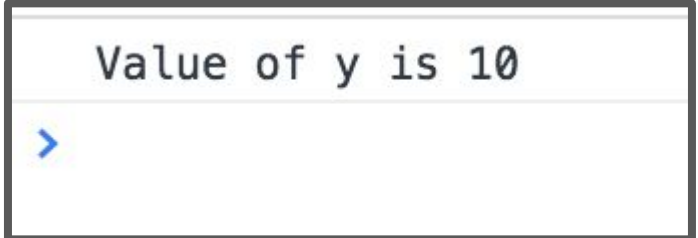


```
3 hello  
Value of i is 3  
>
```

The value of "i" is readable outside of the for-loop because variables declared with var have function scope.

Function scope with var

```
var x = 10;  
if (x > 0) {  
    var y = 10;  
}  
console.log('Value of y is ' + y);
```

A screenshot of a console log window. The text 'Value of y is 10' is displayed in a monospace font. Below the text is a blue prompt character '>'. The entire window is enclosed in a black border.

- Variables declared with "var" have function-level scope and do not go out of scope at the end of blocks; only at the end of functions
- Therefore you can refer to the same variable after the block has ended (e.g. after the loop or if-statement in which they are declared)

Function scope with var

```
function meaningless() {  
  var x = 10;  
  if (x > 0) {  
    var y = 10;  
  }  
  console.log('y is ' + y);  
}  
meaningless();  
console.log('y is ' + y); // error! ❌
```

```
y is 10
```

```
❌ ▶ Uncaught ReferenceError: y is not defined  
   at script.js:9
```

But you can't refer to a variable outside of the function in which it's declared.

Understanding `let`

```
function printMessage(message, times) {  
  for (let i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}  
printMessage('hello', 3);
```

Q: What happens if we try to print "i" at the end of the loop?

Understanding `let`

```
function printMessage(message, times) {  
  for (let i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}  
printMessage('hello', 3);
```



The screenshot shows a browser console with the following content:

- A blue circle with the number 3, followed by the text "hello".
- A red error message: "Uncaught ReferenceError: i is not defined".
- The error stack trace: "at printMessage (script.js:5)" and "at script.js:8".
- A blue prompt character ">" at the bottom.

`let` has block-scope so this results in an error

Understanding `const`

```
let x = 10;
if (x > 0) {
  const y = 10;
}
console.log(y); // error!
```

A screenshot of a browser's developer console showing a red error message. The message reads: "Uncaught ReferenceError: y is not defined" followed by "at script.js:5". There is a red 'x' icon to the left of the message and a blue arrow icon below it.

✖ ▶ Uncaught ReferenceError: y is not defined
at script.js:5

Like `let`, `const` also has block-scope, so accessing the variable outside the block results in an error

Understanding `const`

```
const y = 10;  
y = 0;           // error!  
y++;           // error!  
const list = [1, 2, 3];  
list.push(4);   // OK
```

`const` declared variables cannot be reassigned.

However, it doesn't provide true const correctness, so you can still modify the underlying object

- (In other words, it behaves like Java's `final` keyword and not C++'s `const` keyword)

Contrasting with `let`

```
let y = 10;  
y = 0;           // OK  
y++;            // OK  
let list = [1, 2, 3];  
list.push(4);   // OK
```

Let can be reassigned, which is the difference between `const` and `let`

Variables best practices

- Use `const` whenever possible.
- If you need a variable to be reassignable, use `let`.
- **Don't use `var`.**
 - You will see a ton of example code on the internet with `var` since `const` and `let` are relatively new.
 - However, `const` and `let` are [well-supported](#), so there's no reason not to use them.

(This is also what the [Google](#) and [AirBnB](#) JavaScript Style Guides recommend.)

Variables best practices

- Use `const` whenever possible.
- If you need a variable to be reassignable, use `let`.
- **Don't use `var`.**
 - You will see a ton of example code on the internet with `var` since `const` and `let` are relatively new.
 - However, `const` and `let` are **well-supported** so

Aside: The internet has a **ton** of misinformation about JavaScript!

Including several "accepted" StackOverflow answers, tutorials, etc. Lots of stuff online is years out of date.

Tread carefully.

Types

JS **variables** do not have types, but the **values** do.

There are nine primitive types ([mdn](#)):

- [Boolean](#): true and false
- [Number](#): everything is a double (no integers)
- [BigInt](#): new definition for integers that falls outside the allowed range for representation
- [String](#): in 'single' or "double-quotes"
- [Symbol](#): *(skipping this today)*
- [Undefined](#): the value of a variable with no value assigned
- [Null](#): null: a value meaning "this has no value"

There are also [Object](#) types, including Array, Date, String (the object wrapper for the primitive type), etc.

Numbers

```
const homework = 0.45;  
const midterm = 0.2;  
const final = 0.35;  
const score =  
    homework * 87 + midterm * 90 + final * 95;  
console.log(score);    // 90.4
```

Numbers

```
const homework = 0.45;
const midterm = 0.2;
const final = 0.35;
const score =
    homework * 87 + midterm * 90 + final * 95;
console.log(score);    // 90.4
```

- All numbers are floating point real numbers. No integer type.
- Operators are like Java or C++.
- Precedence like Java or C++.
- A few special values: [NaN](#) (not-a-number), +Infinity, -Infinity
- There's a [Math](#) class: `Math.floor`, `Math.ceil`, etc.

Strings

```
let snack = 'coo';  
snack += 'kies';  
snack = snack.toUpperCase();  
console.log("I want " + snack);
```

Strings

```
let snack = 'coo';  
snack += 'kies';  
snack = snack.toUpperCase();  
console.log("I want " + snack);
```

- Can be defined with single or double quotes
 - Many [style guides](#) prefer single-quote, but there is no functionality difference
- Immutable
- No char type: letters are strings of length one
- Can use plus for concatenation
- Can check size via `length` property (not function)

Boolean

- There are two literal values for boolean: `true` and `false` that behave as you would expect
- Can use the usual boolean operators: `&&` `||` `!`

```
let isHungry = true;
```

```
let isTeenager = age > 12 && age < 20;
```

```
if (isHungry && isTeenager) {  
    pizza++;  
}
```

Boolean

- Non-boolean values can be used in control statements, which get converted to their "truthy" or "falsy" value:
 - `null`, `undefined`, `0`, `NaN`, `' '`, `''` evaluate to `false`
 - Everything else evaluates to `true`

```
if (username) {  
    // username is defined  
}
```

Equality

JavaScript's == and != are basically broken: they do an implicit type conversion before the comparison.

```
' ' == '0' // false
' ' == 0   // true
0 == '0'  // true
NaN == NaN // false
[''] == '' // true
false == undefined // false
false == null      // false
null == undefined  // true
```

Equality

Instead of fixing `==` and `!=`, the ECMAScript standard kept the existing behavior but added `===` and `!==`

```
' ' === '0' // false
' ' === 0 // false
0 === '0' // false
NaN == NaN // still weirdly false
[''] === '' // false
false === undefined // false
false === null // false
null === undefined // false
```

Always use `===` and `!==` and don't use `==` or `!=`

Null and Undefined

What's the difference?

- `null` is a value representing the absence of a value, similar to `null` in Java and `nullptr` in C++.
- `undefined` is the value given to a variable that has not been a value.

```
let x = null;  
let y;  
console.log(x);  
console.log(y);
```

```
null  
undefined  
>
```

Null and Undefined

What's the difference?

- `null` is a value representing the absence of a value, similar to `null` in Java and `nullptr` in C++.
- `undefined` is the value given to a variable that has not been a value.
 - ... however, you can also set a variable's value to `undefined` 😞

```
let x = null;  
let y = undefined;  
console.log(x);  
console.log(y);
```

```
null  
undefined
```



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.)

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

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

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]);  
}
```

- Use **for...in** on object types
- Use **for...of** 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.



Example:

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

Event-driven programming

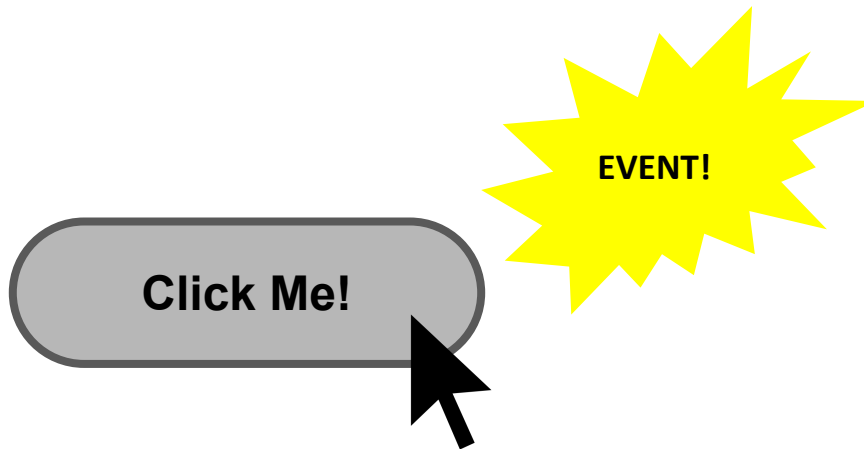
Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.



When the user clicks the button...

Event-driven programming

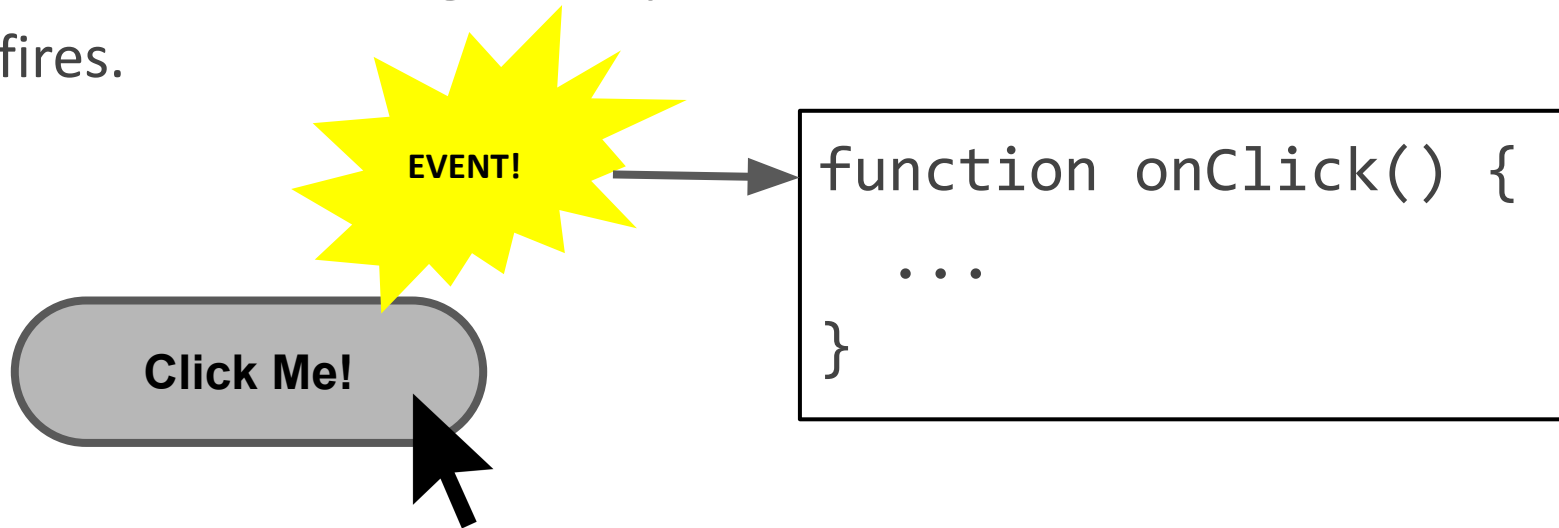
Most JavaScript written in the browser is **event-driven**:
The code doesn't run right away, but it executes after some event fires.



...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:

```
<button>Click me</button>
```



Click me

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:



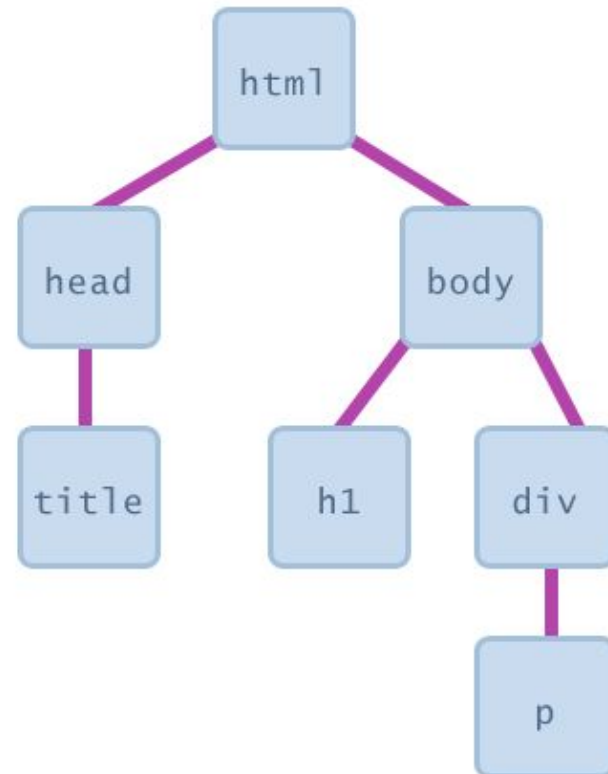
We need to add an event listener to the button...

How do we talk to 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.

Adding event listeners

Each DOM object has the following function:

```
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

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

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

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

```
script.js x
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?

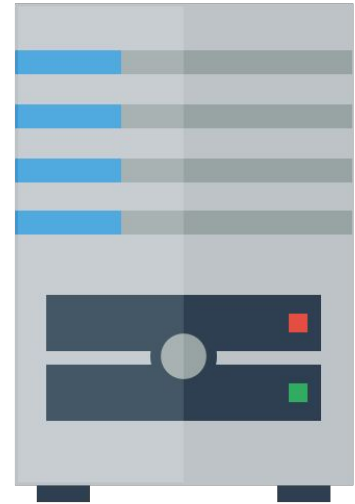
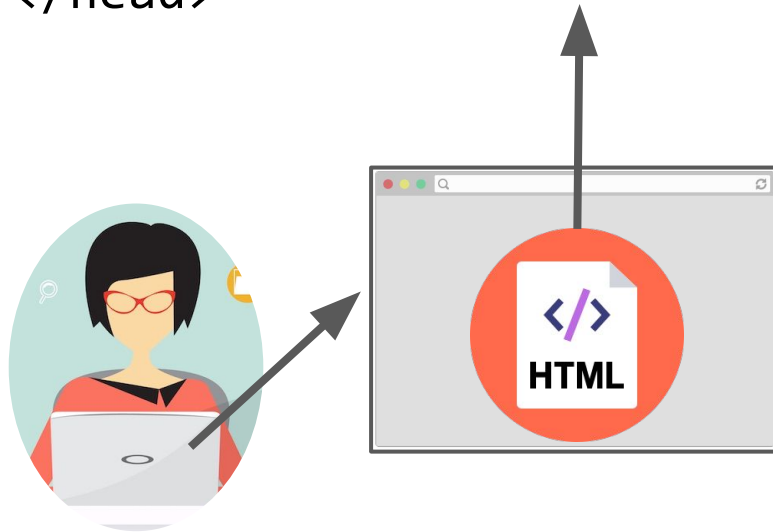
```
<head>
```

```
  <title>Interactive Web Programming</title>
```

```
  <link rel="stylesheet" href="style.css" />
```

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

```
</head>
```



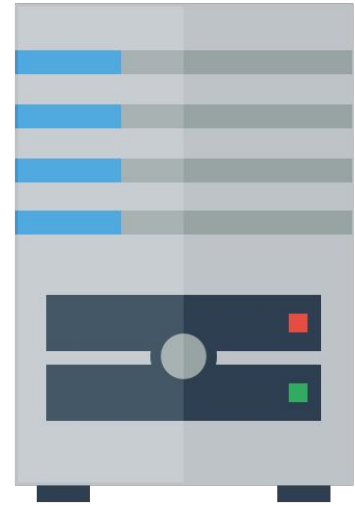
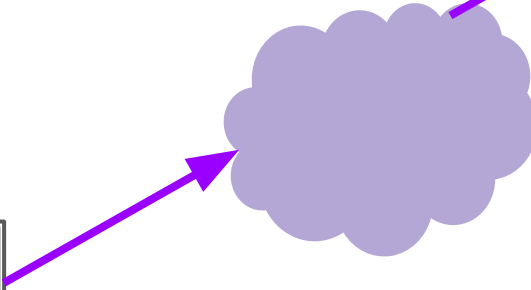
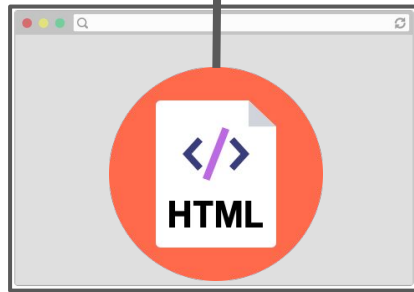
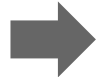
```
<head>
```

```
  <title>Interactive Web Programming</title>
```

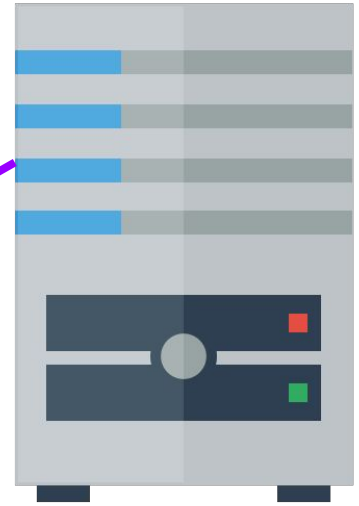
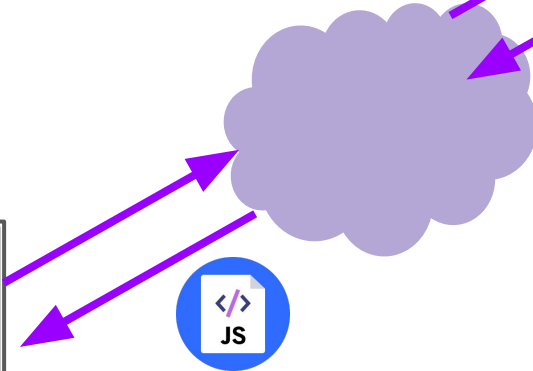
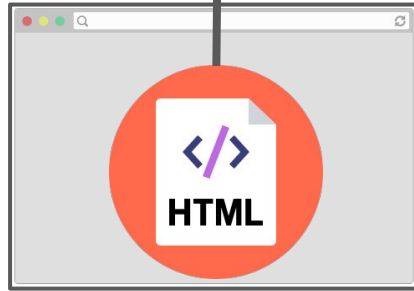
```
  <link rel="stylesheet" href="style.css" />
```

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

```
</head>
```




```
<head>
  <title>Interactive Web Programming</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```



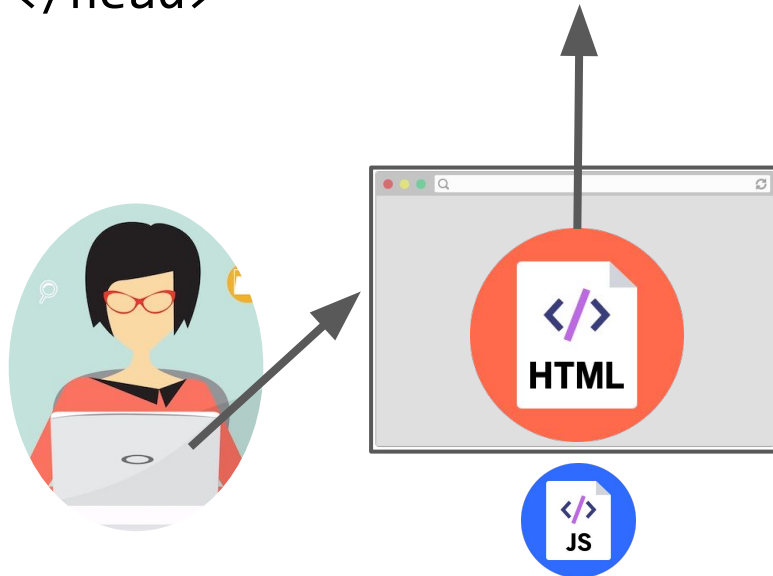
```
<head>
```

```
  <title>Interactive Web Programming</title>
```

```
  <link rel="stylesheet" href="style.css" />
```

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

```
</head>
```



```
function onClick() {  
  console.log('clicked');  
}  
  
const button = document.querySelector('button');  
button.addEventListener('click', onClick);
```

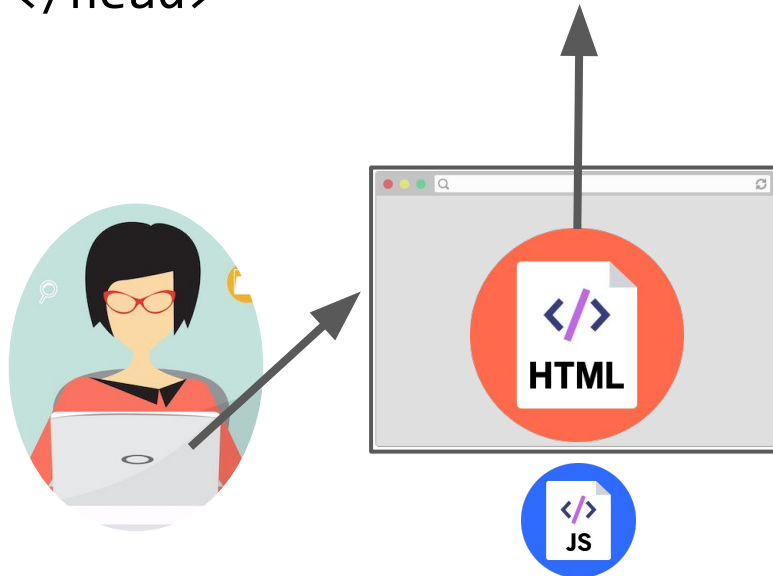
```
<head>
```

```
  <title>Interactive Web Programming</title>
```

```
  <link rel="stylesheet" href="style.css" />
```

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

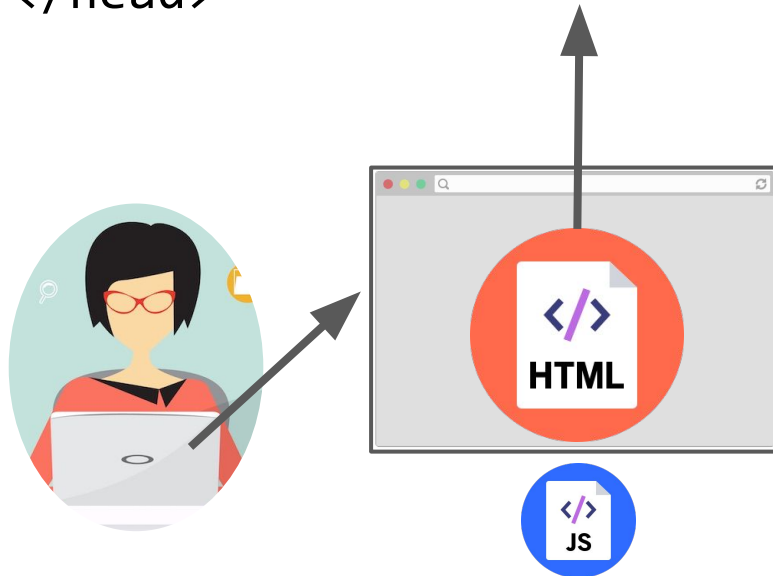
```
</head>
```



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

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

```
<head>
  <title>Interactive Web Programming</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```

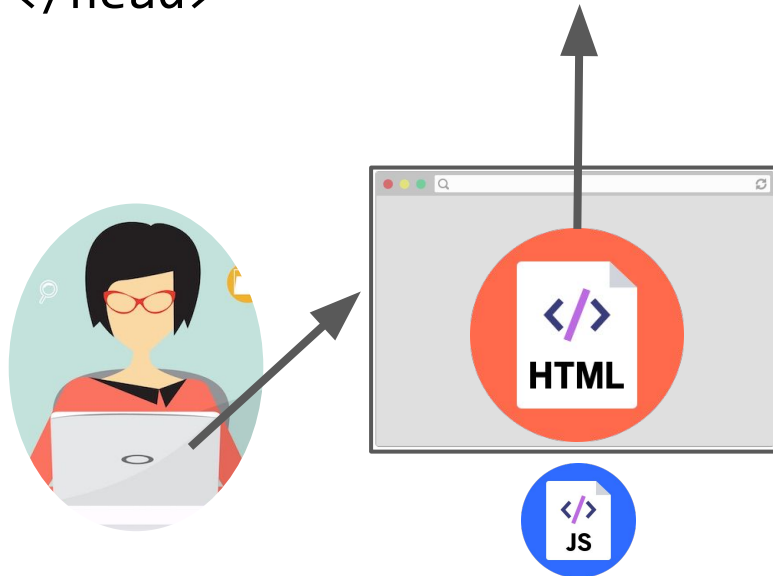


```
function onClick() {
  console.log('clicked');
}

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

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

```
<head>
  <title>Interactive Web Programming</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js"></script>
</head>
```



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

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

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

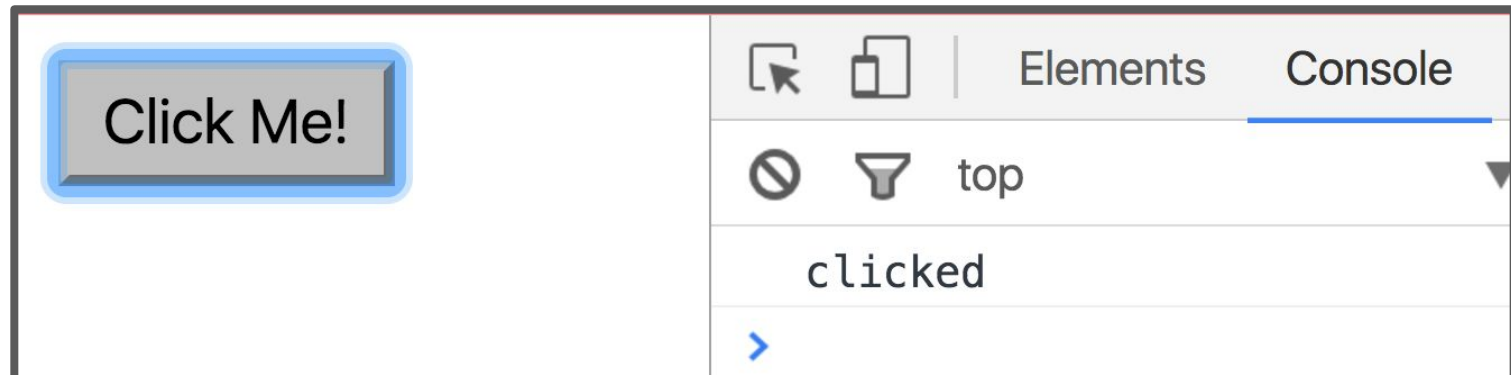
You will see tons of examples on the internet that do this.

They are out of date. `defer` is [widely supported](#) and better.

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

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

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



The screenshot shows a web browser interface. On the left, there is a button with the text "Click Me!". On the right, the developer console is open, showing a log message "clicked". The console has tabs for "Elements" and "Console", with "Console" selected. There are also icons for a mouse cursor, a document, a filter, and a dropdown menu.

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

```

```

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](#))

Next time...

Example: Present

Click for a present:



See the [CodePen](#) -
much more exciting!

```
function openPresent() {  
  const image = document.querySelector('img');  
  image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';  
}
```

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