

# Today's schedule

## Today

- Intro to Amateur JavaScript
  - What is JavaScript?
  - Tour of language features
  - Basic event handling
- DOM: How to interact with your web page

# Note

## Assignment 1:

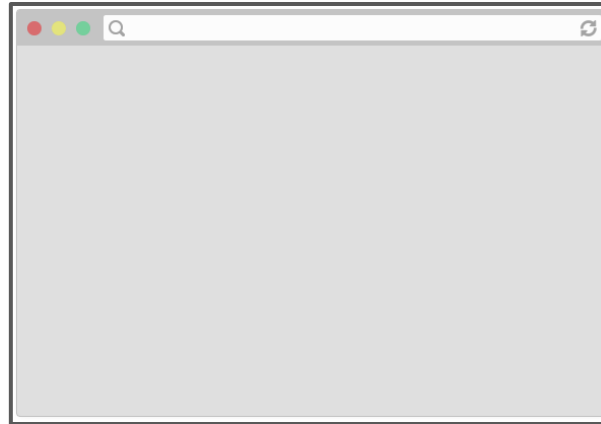
- Released
- Due: **17<sup>th</sup> Oct**
- Marking scheme: available for self-checking
- Check points: from this week (**26<sup>th</sup> Sep**)

How do web pages  
work again?

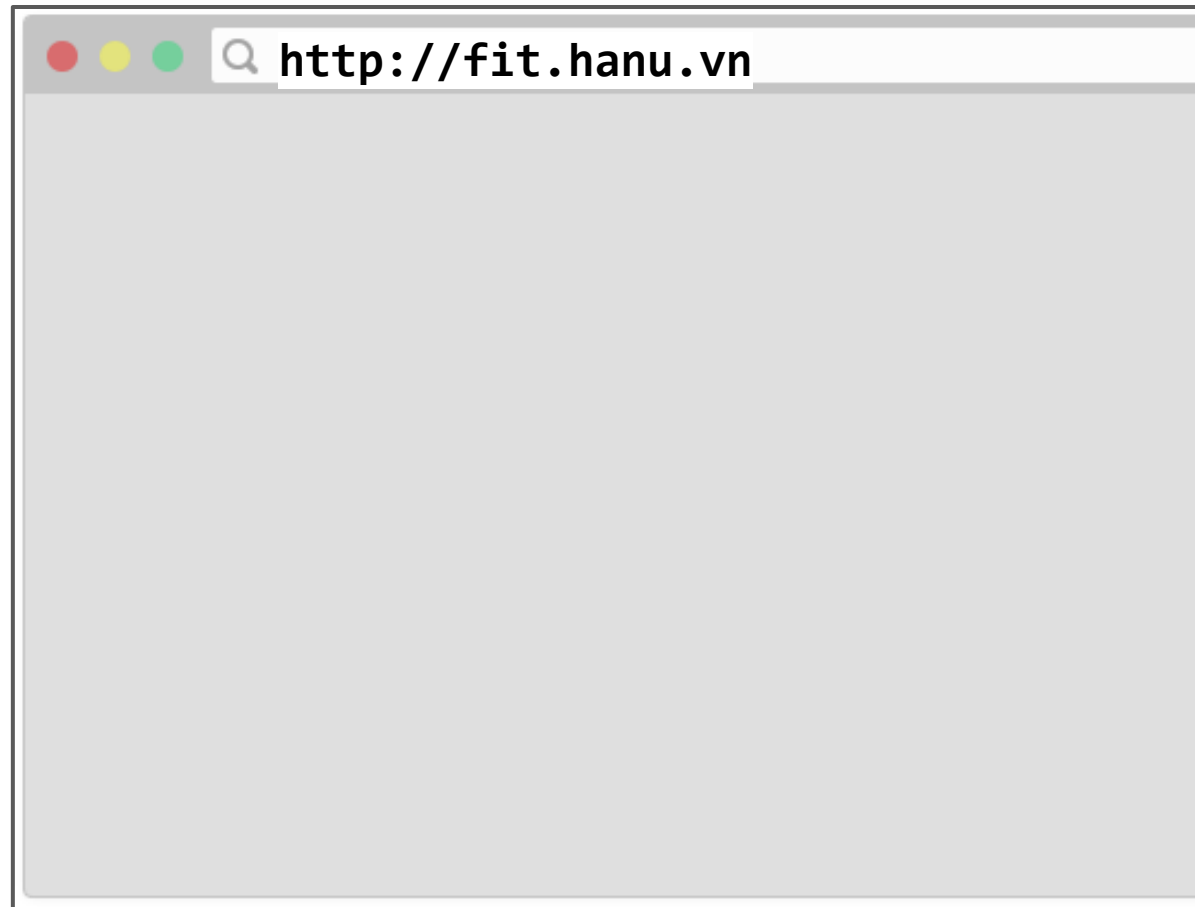
You are on  
your laptop

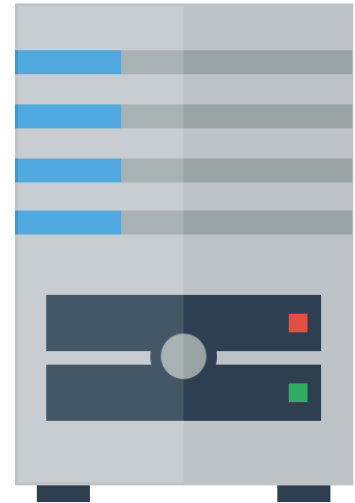
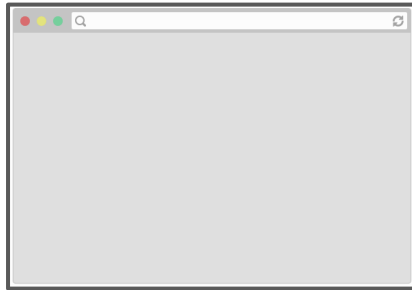


Your laptop is  
running a web  
browser, e.g.  
Chrome



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



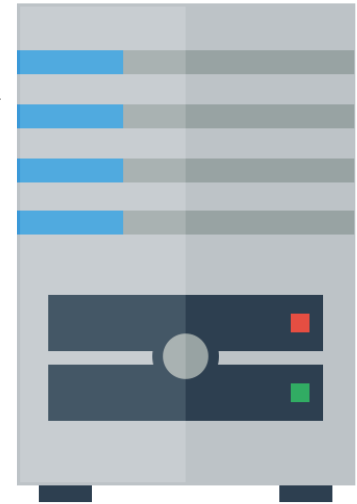
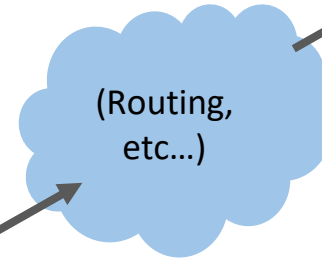
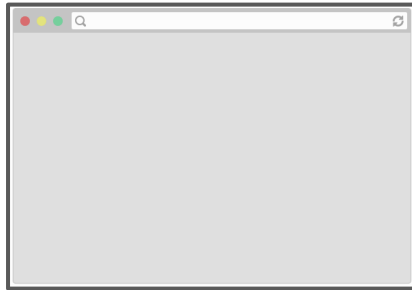


**(Warning:** Somewhat inaccurate,  
massive hand-waving begins now.

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

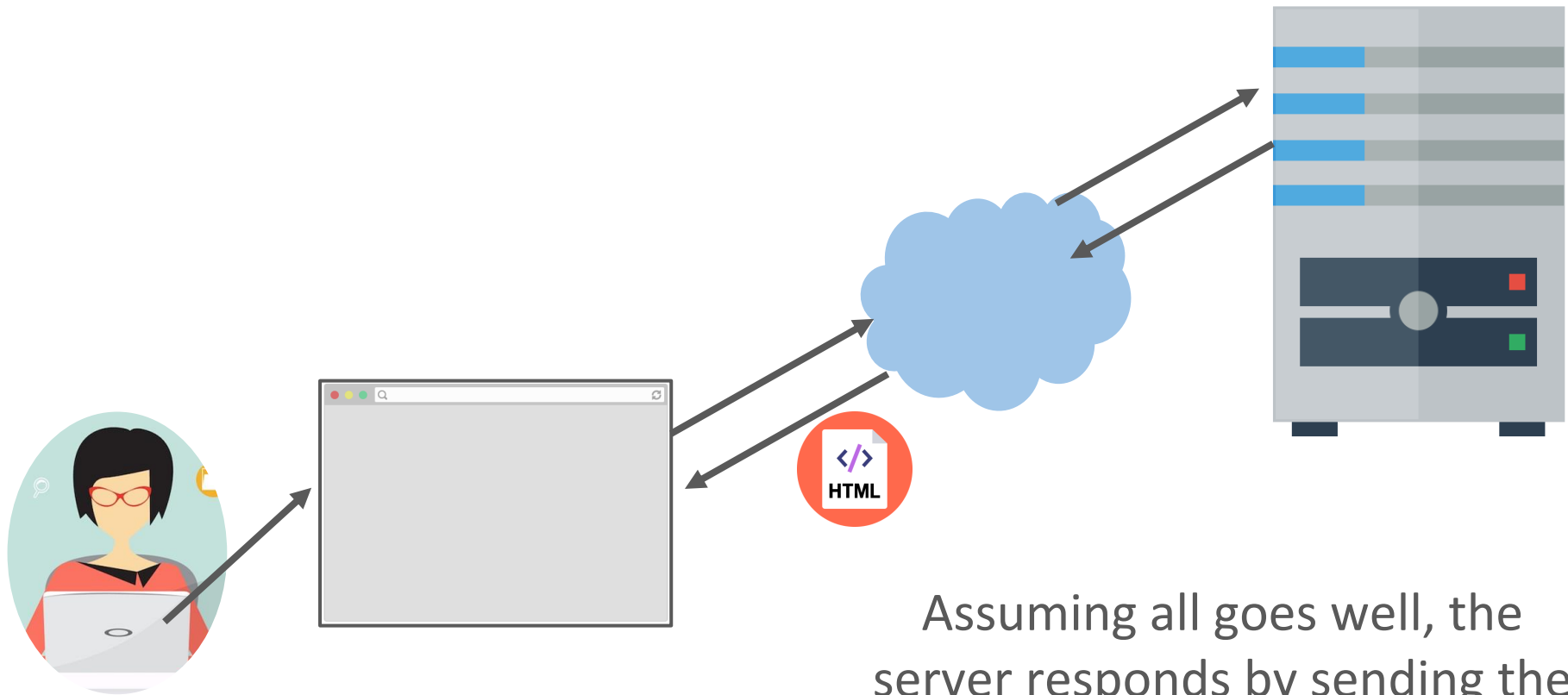
Server at  
<http://fit.hanu.vn>

Browser sends an HTTP request  
saying "Please GET me the  
index.html file at  
<http://fit.hanu.vn>"





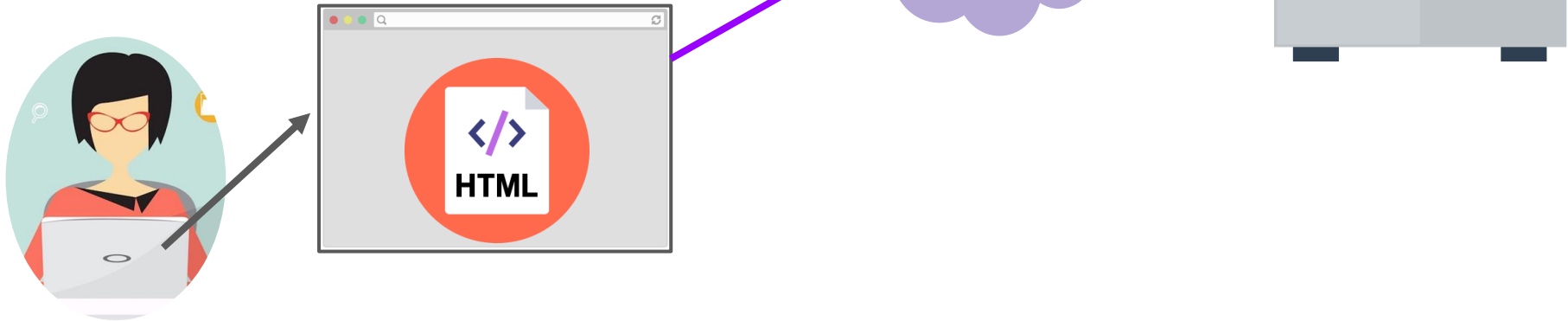
Server at  
<http://fit.hanu.vn>



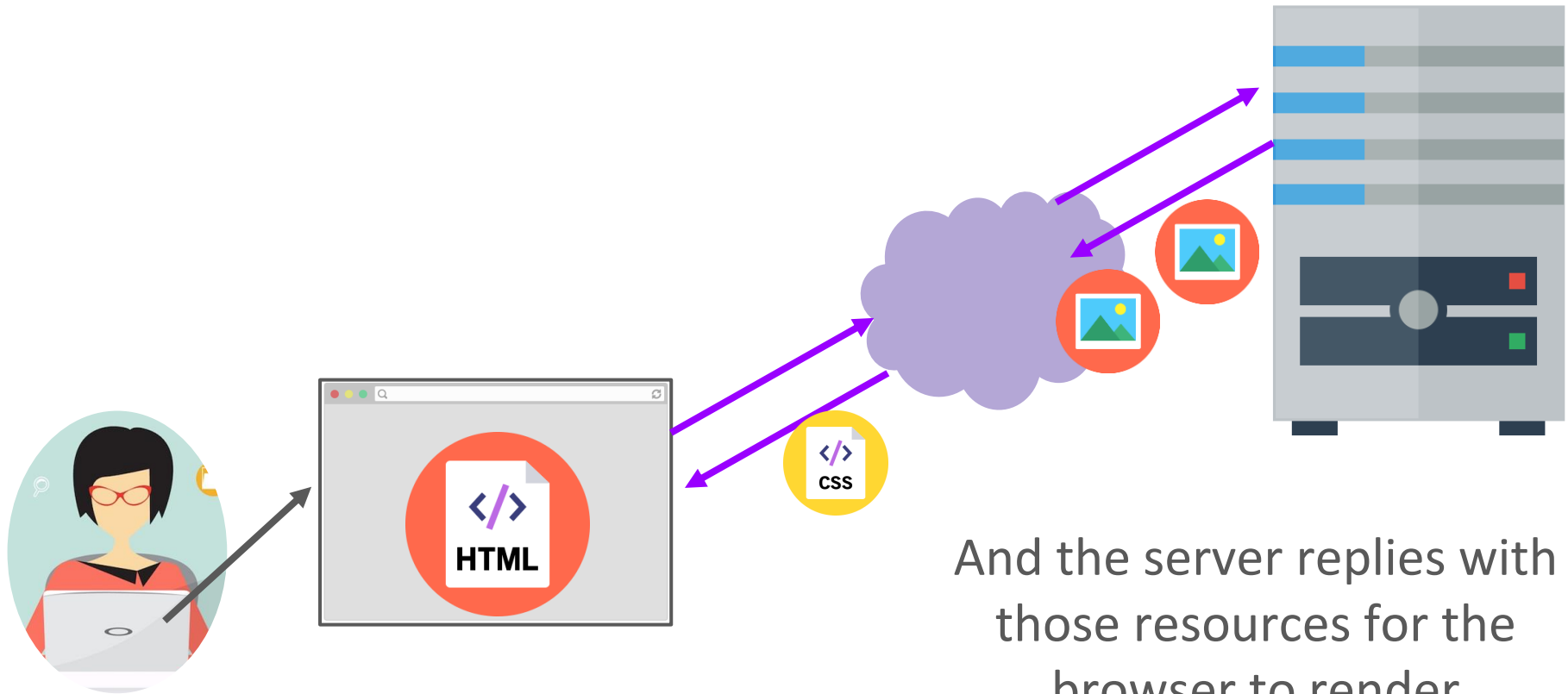
Assuming all goes well, the server responds by sending the HTML file through the internet back to the browser to display.

Server at  
<http://fit.hanu.vn>

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



Server at  
<http://fit.hanu.vn>



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



The screenshot displays a web browser window with the address `http://fit.hanu.vn`. The page is the 'Web Programming' course page for the Faculty of Information Technology (FIT). The navigation bar includes links for 'My courses', 'Faculty of IT', 'Admission', 'Academic programs', 'Student', 'This course', and 'English (en)'. The course title 'Web Programming' is prominently displayed. Below the title is a large illustration of a desk setup with a computer monitor showing '<WEB/>', a tower unit, a cat, and a guitar. The sidebar on the right contains an 'ADMINISTRATION' section with links like 'Course administration', 'Turn editing on', 'Edit settings', 'Users', 'Filters', 'Reports', 'Grades', 'Backup', 'Restore', 'Import', 'Reset', 'Question bank', and 'Switch role to...'. Below this is a 'CALENDAR' section for August 2019, showing a grid of dates. At the bottom of the main content area, there are links for 'Announcements', 'Discussion Forum', 'Resources', and 'Module Description'.



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  
(co-founder of Mozilla; resigned 2014 [due to his homophobia](#))
- 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  
(co-founder of Mozilla; resigned 2014 [due to his homophobia](#))
- 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:

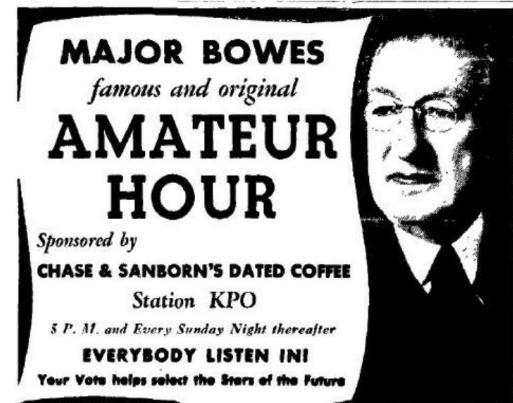
**JavaScript is messy and full of drama...  
and our only option.**

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

# Our JavaScript Strategy

This week: "**Amateur JavaScript**"

- Simple, 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
- Even more "opinionated"

# 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>WPR</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**

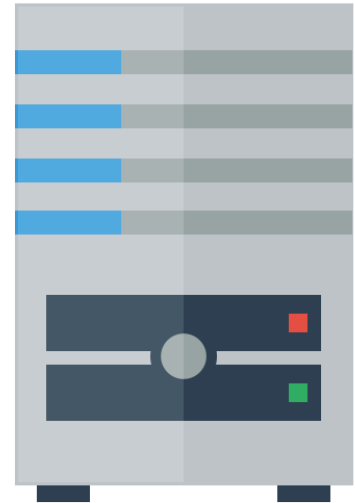
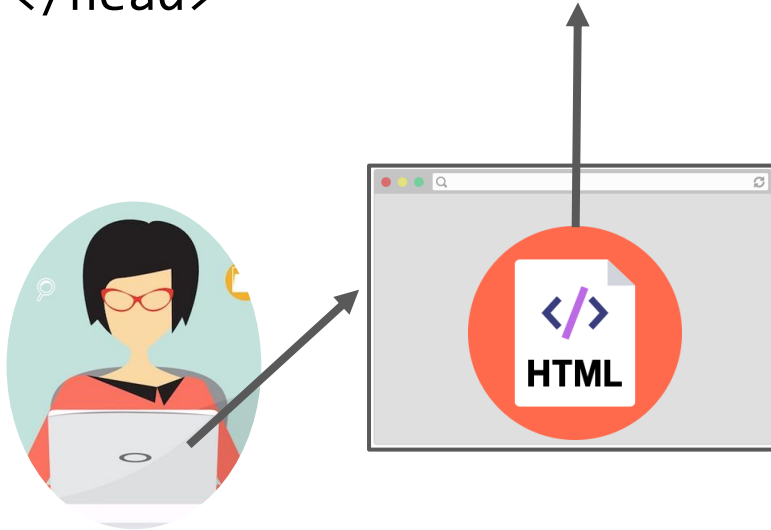
```
console.log('Hello, world!');
```

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

How does JavaScript get loaded?



```
<head>
  <title>WPR</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.

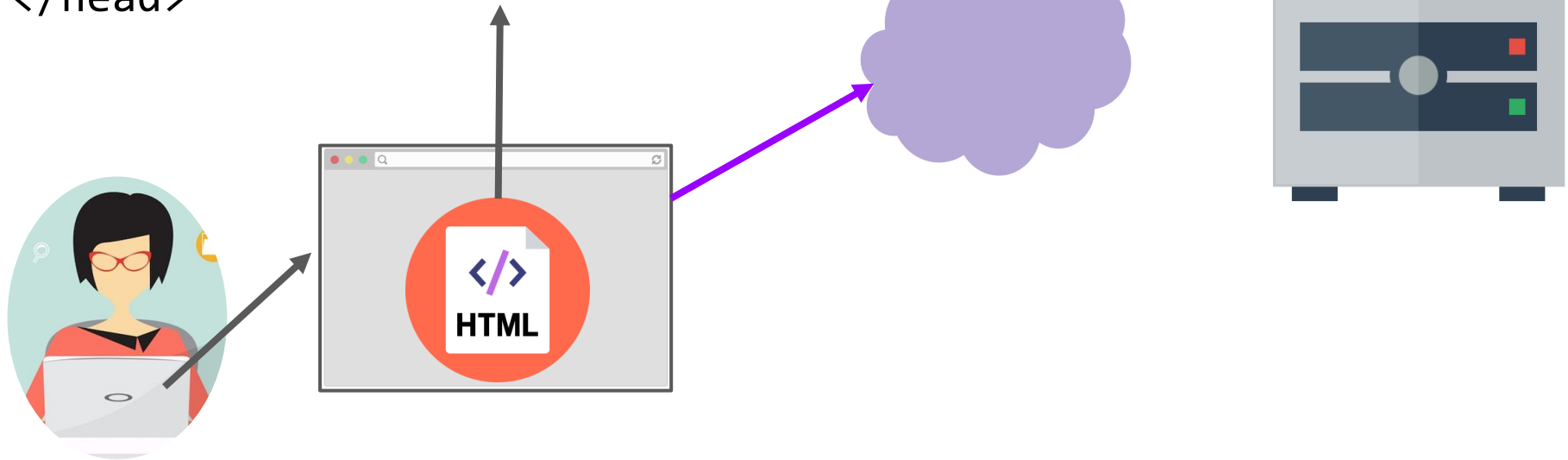
```
<head>
```

```
  <title>WPR</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...

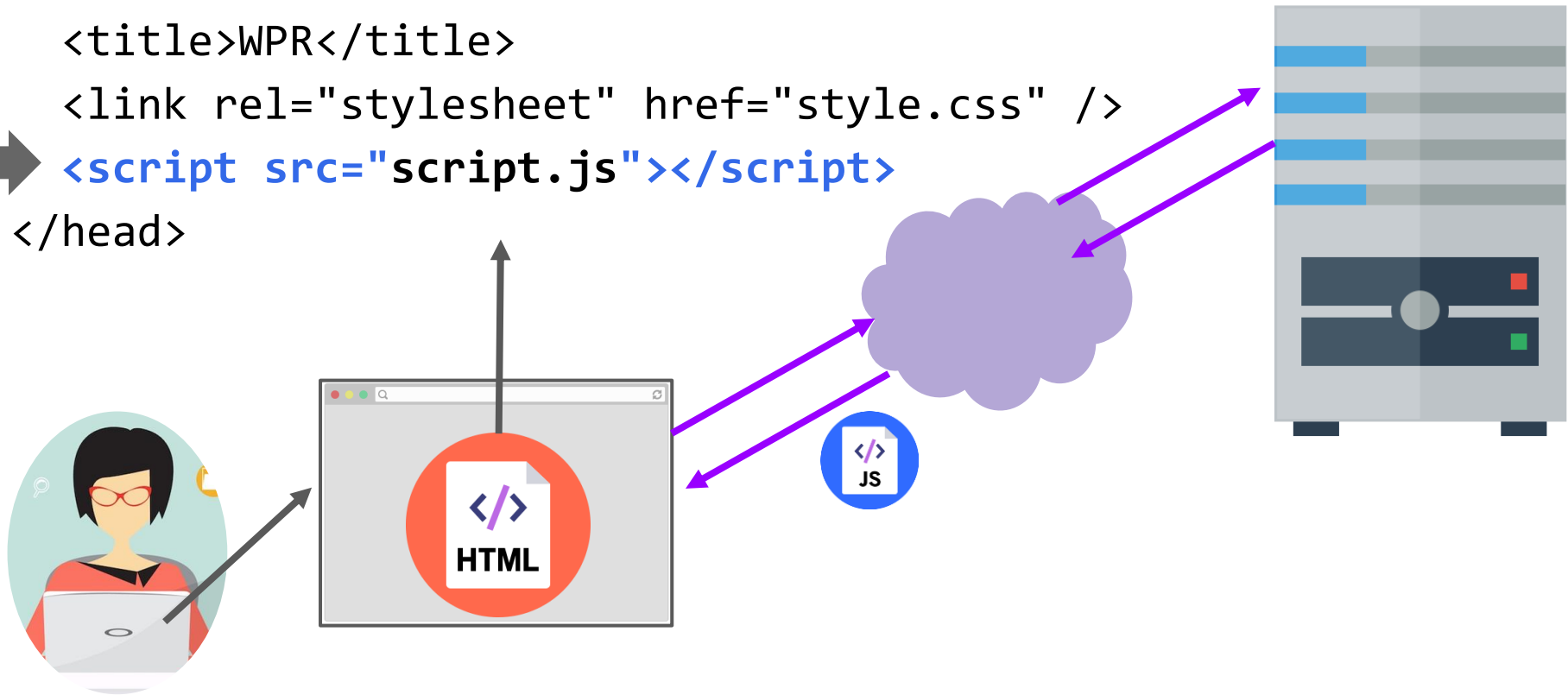
```
<head>
```

```
  <title>WPR</title>
```

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

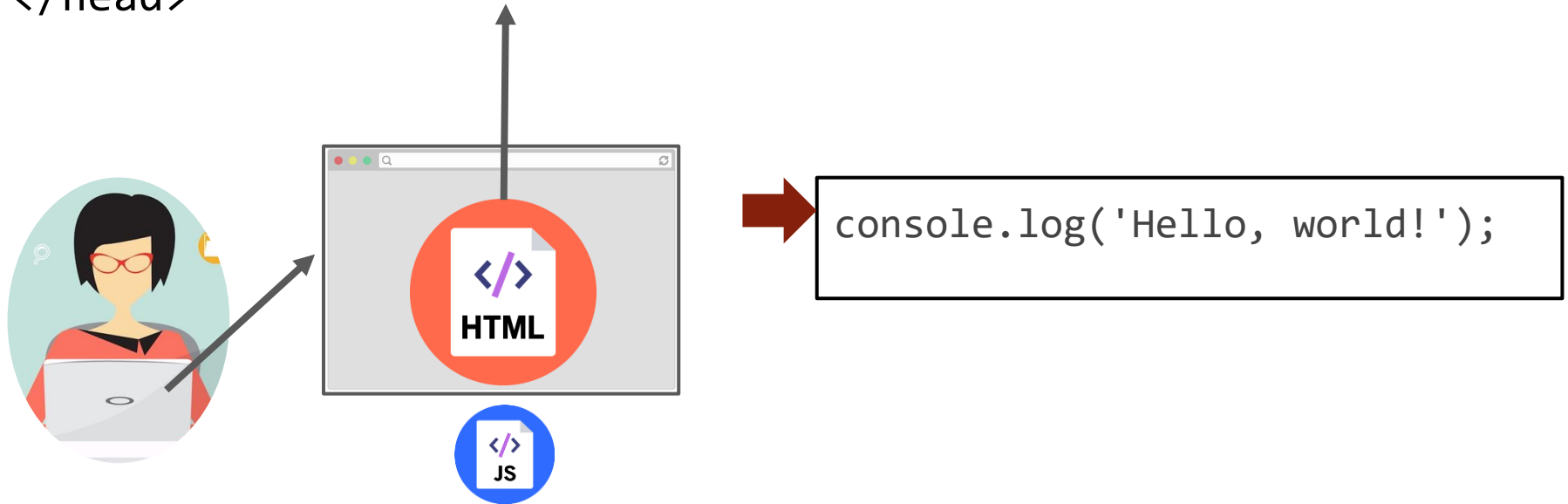
```
➡ <script src="script.js"></script>
```

```
</head>
```



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

```
<head>
  <title>WPR</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

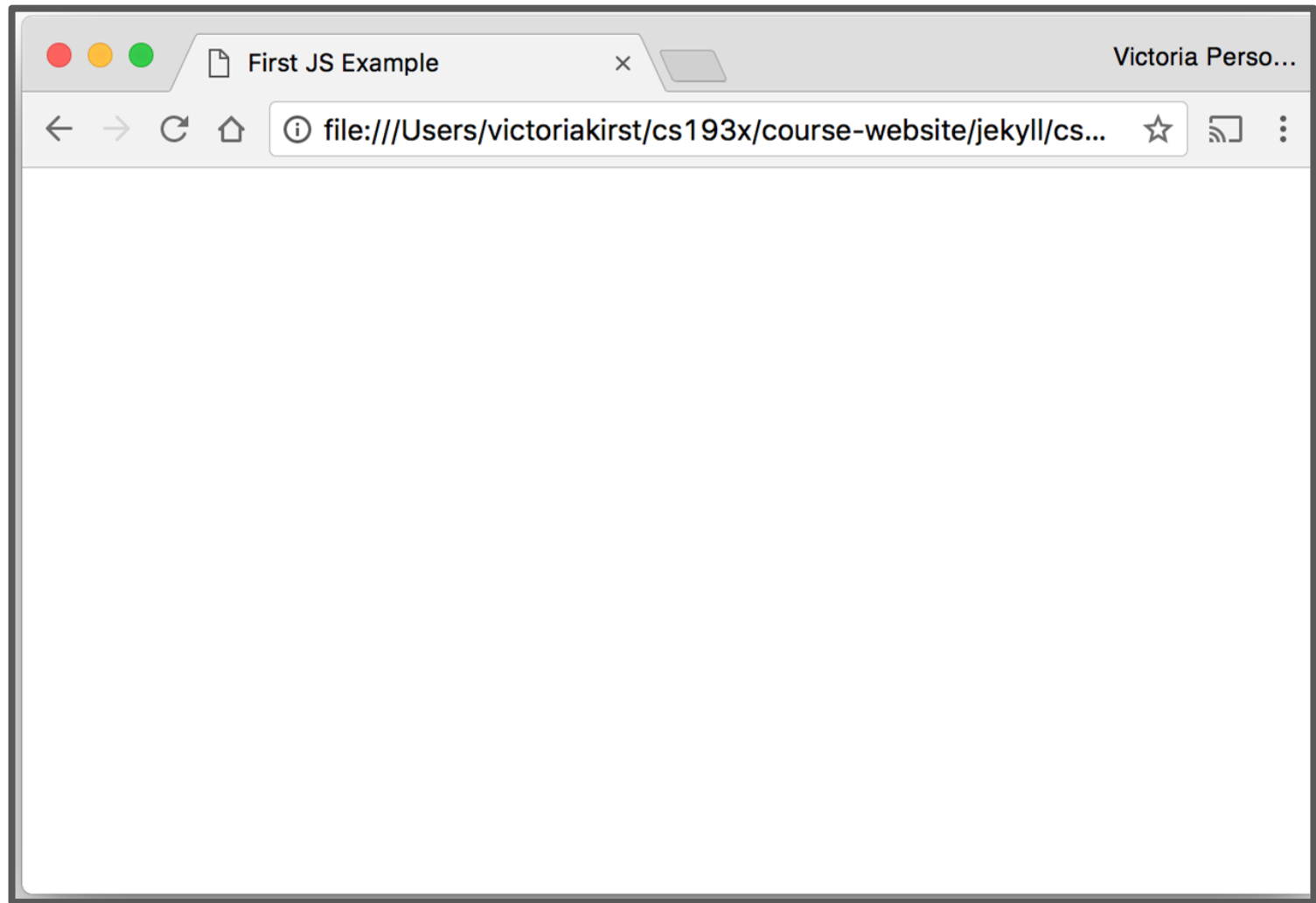
(Note that this is slightly different than being "interpreted": see [just-in-time \(JIT\) compilation](#))

## first-js.html

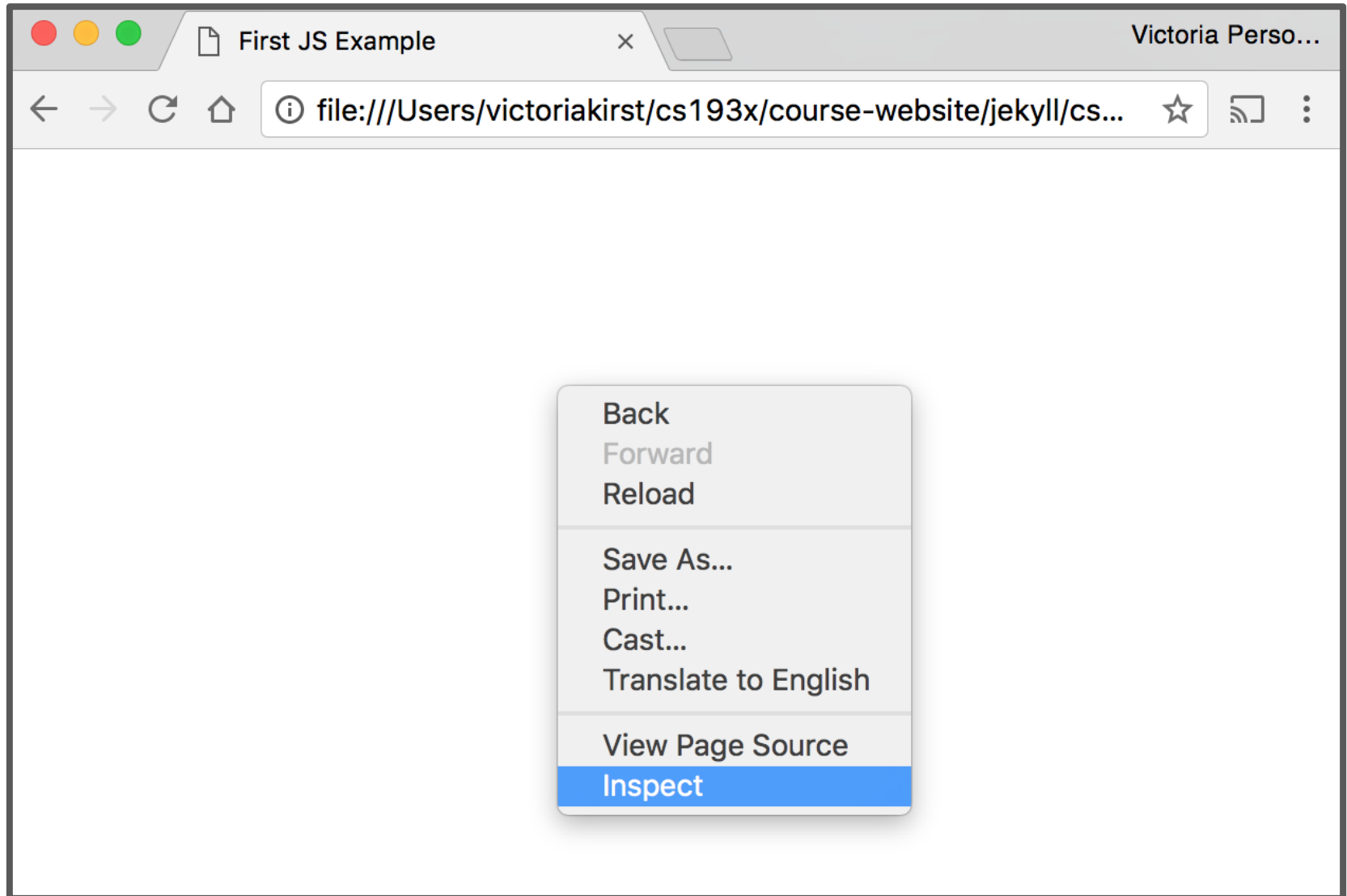
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
  </body>
</html>
```

## script.js

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

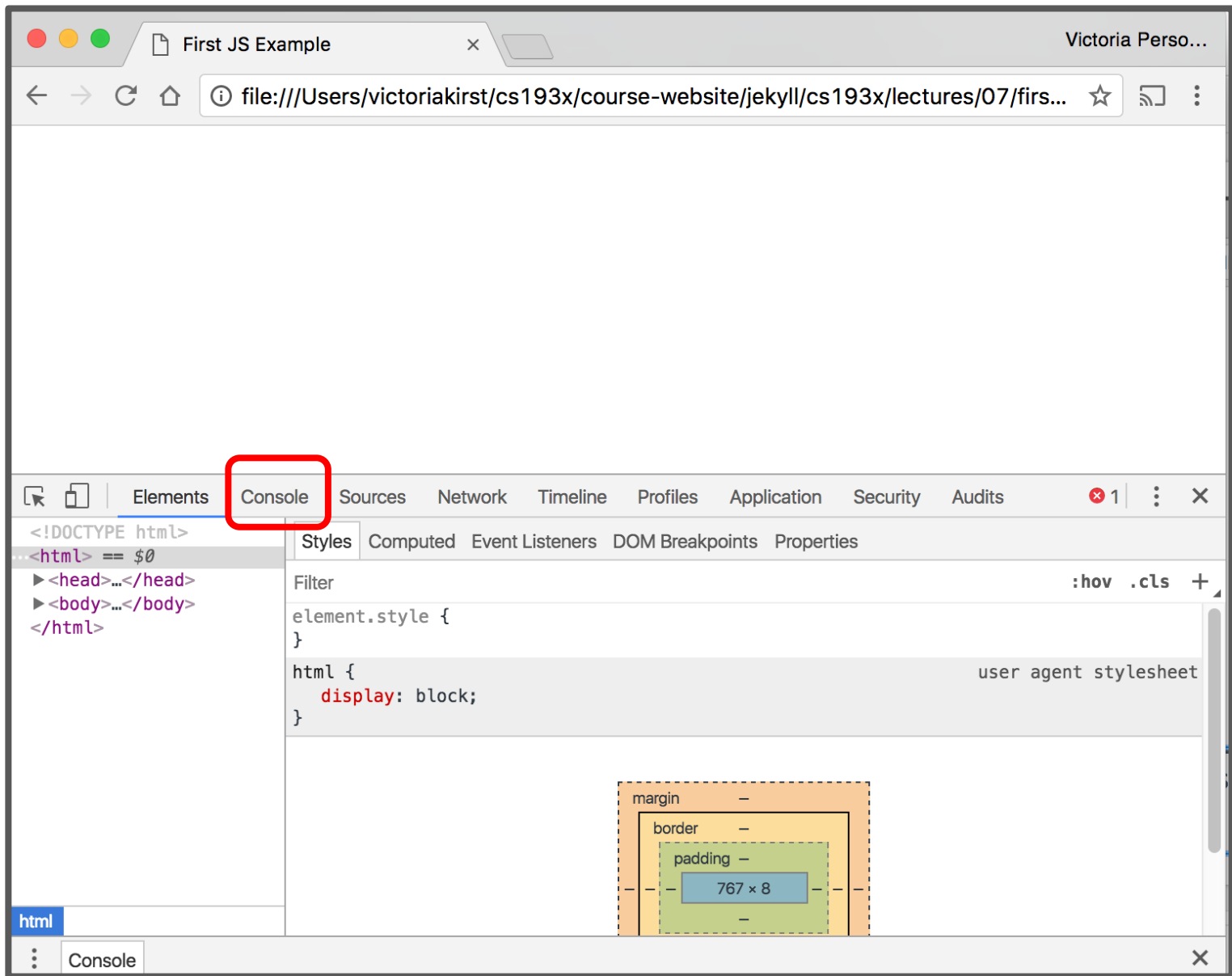


Hey, nothing happened!

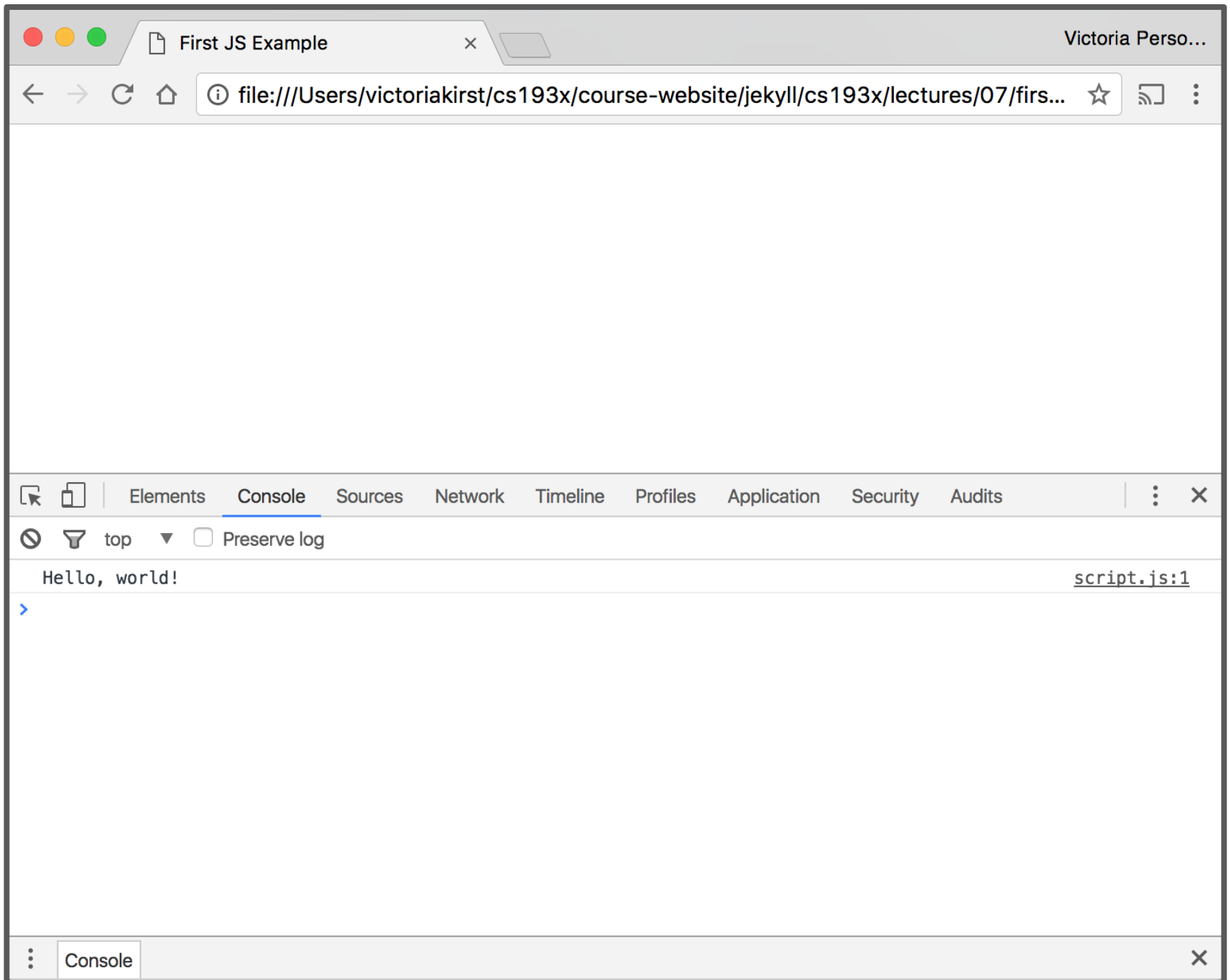


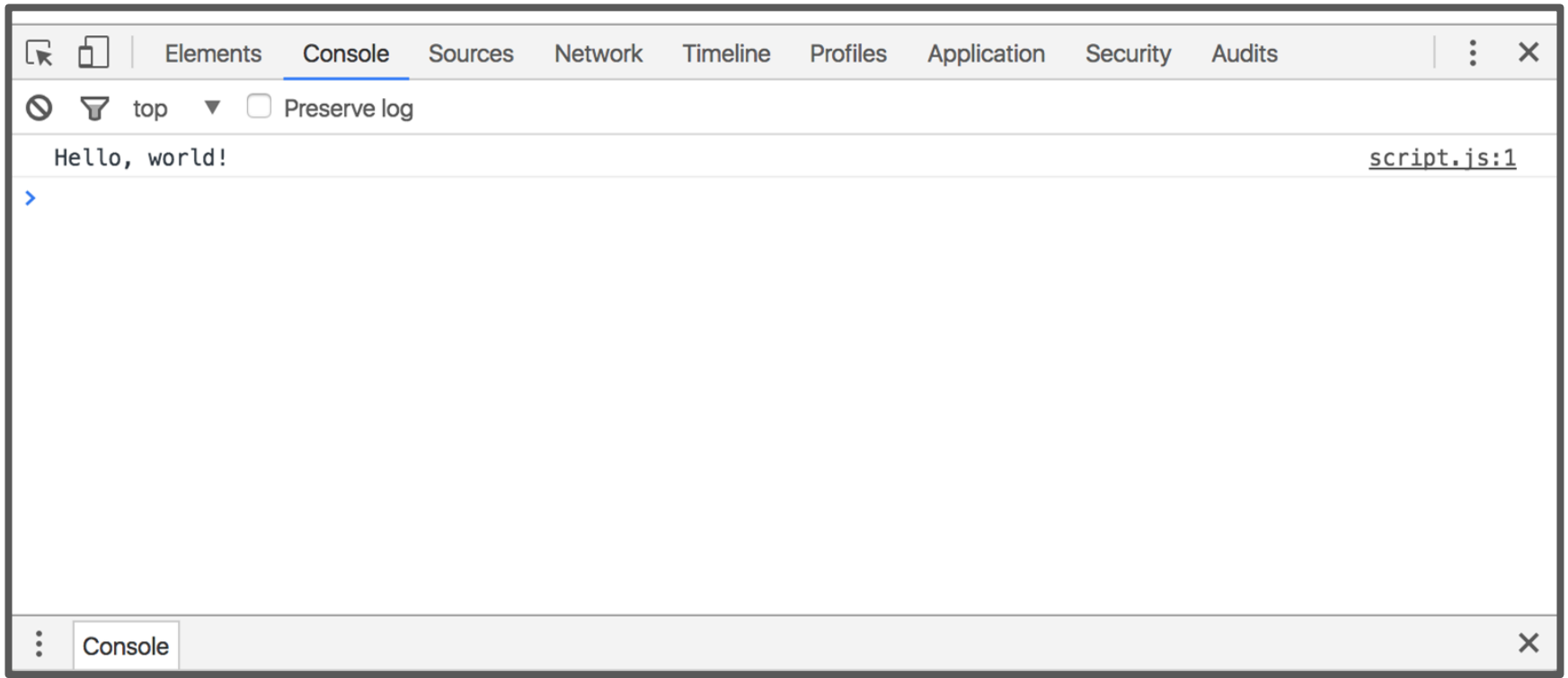
Right-click (or control-click on Mac) and choose "Inspect"





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 tour

# Same as Java

for-loops:

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

while-loops:

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

comments:

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

conditionals (if statements):

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

# 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")

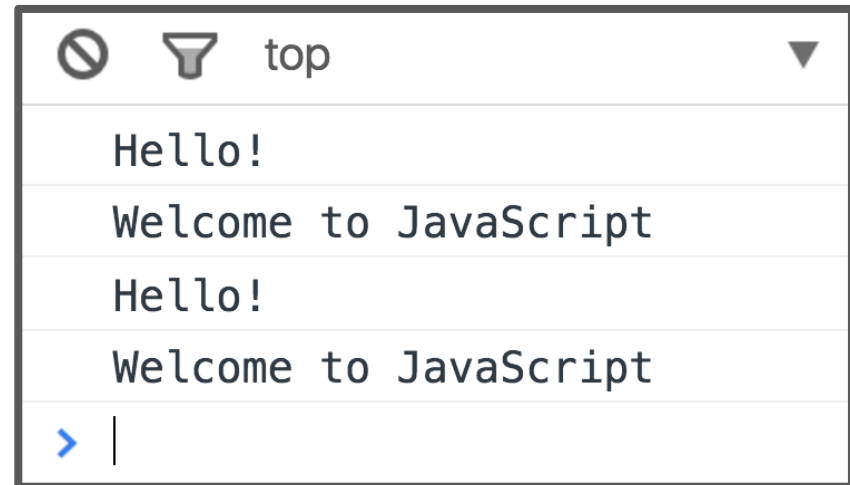
# Functions

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();
```



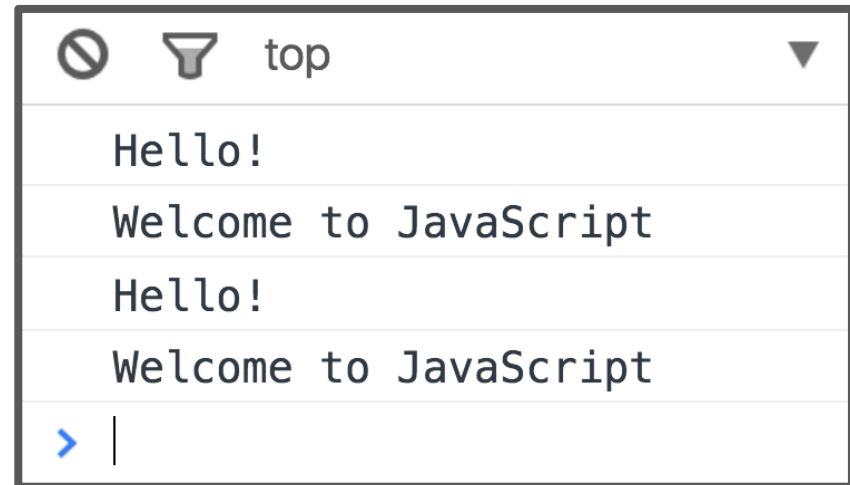
Console output



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.



Console output

**script.js**

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

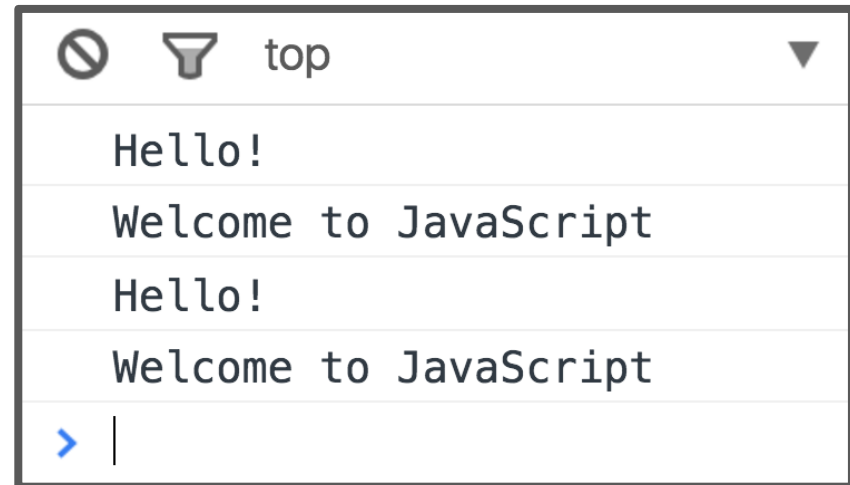
**Q: Does this work?**

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](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting)). 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).



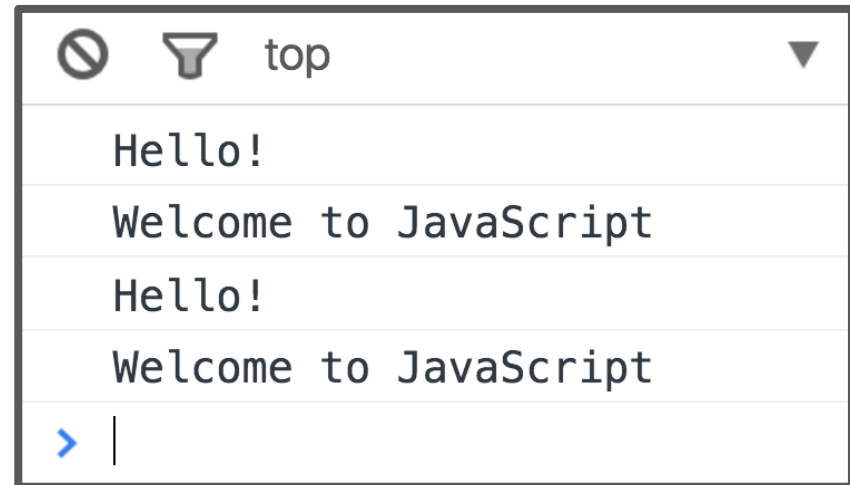
Console output

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

# 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`

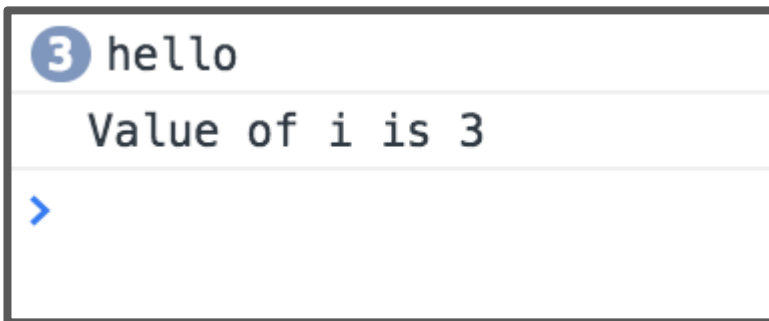
# Understanding var

```
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);
```



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);
```

Value of y is 10

>

- 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);
```

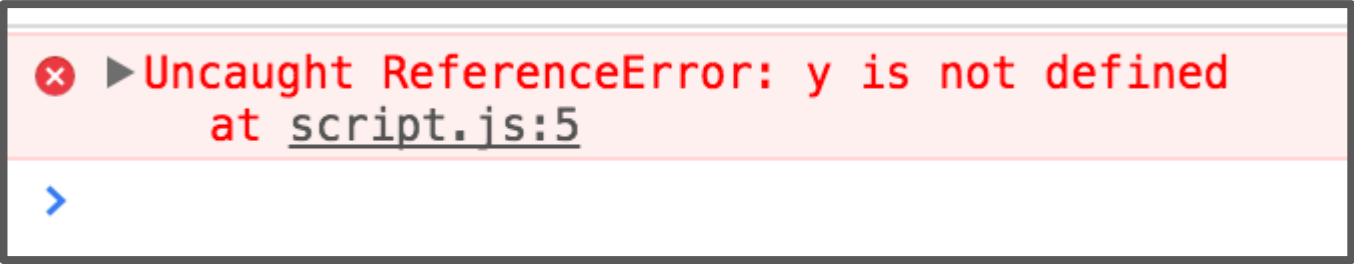
A screenshot of a JavaScript console window. At the top, it shows the output '3 hello'. Below that, a red error message is displayed: 'Uncaught ReferenceError: i is not defined' at 'script.js:5' and 'script.js:8'. A blue prompt character '>' is visible at the bottom of the console.

```
3 hello  
✖ ▶ Uncaught ReferenceError: i is not defined  
   at printMessage (script.js:5)  
   at script.js:8  
>
```

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!
```



✖ ▶ 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)

# 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 six primitive types ([mdn](#)):

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

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of)):

(intuition: **for** each item **of** the groceries list)

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

# List operations

Method	Description
<code>list.push(<i>element</i>)</code>	Add <i>element</i> to back
<code>list.unshift(<i>element</i>)</code>	Add <i>element</i> to front

Method	Description
<code>list.pop()</code>	Remove from back
<code>list.shift()</code>	Remove from front

Method	Description
<code>list.indexOf(<i>element</i>)</code>	Returns numeric index for <i>element</i> or -1 if none found

# splice

Add/remove element at index: [splice](#)

```
list.splice(startIndex, deleteCount, item1, item2, ...)
```

Remove one element at index 3:

```
list.splice(3, 1);
```

Add *element* at index 2:

```
list.splice(2, 0, eLement);
```

# Maps through Objects

- Every JavaScript object is a collection of property-value pairs.
- 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]);  
}
```

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

BREAK TIME!

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

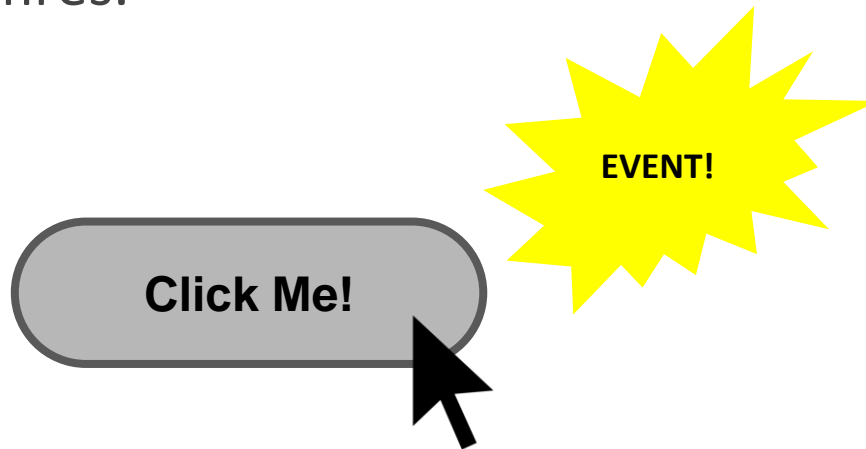
Most JavaScript written in the browser is **event-driven**:  
The code doesn't run right away, but it executes after some event fires.



When the user clicks the button...

# Event-driven programming

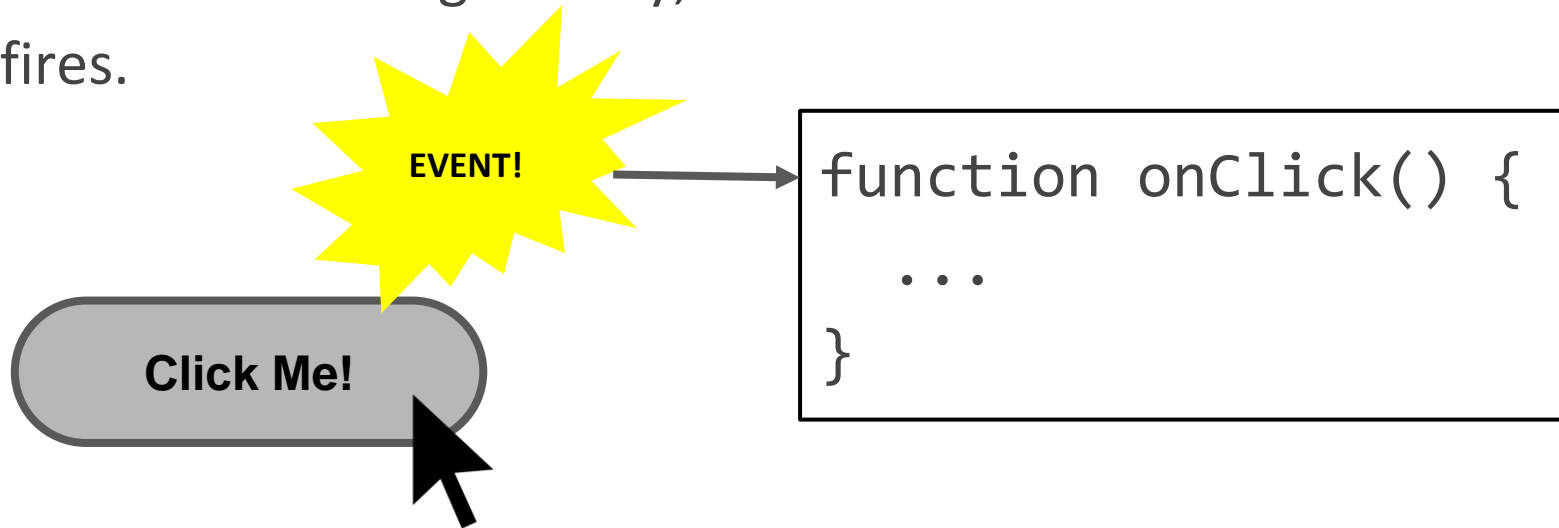
Most JavaScript written in the browser is **event-driven**:  
The code doesn't run right away, but it executes after some event fires.



...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."**

# A few more HTML elements

Buttons:

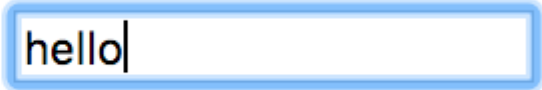
```
<button>Click me</button>
```



Click me

Single-line text input:

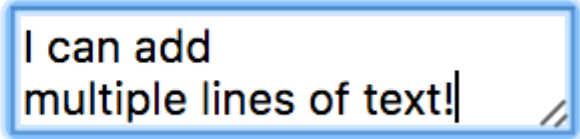
```
<input type="text" />
```



hello

Multi-line text input:

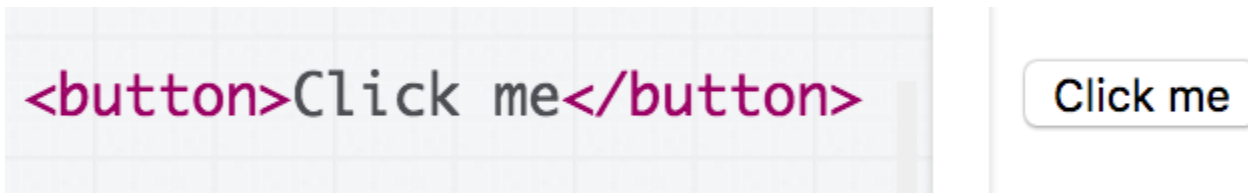
```
<textarea></textarea>
```



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:



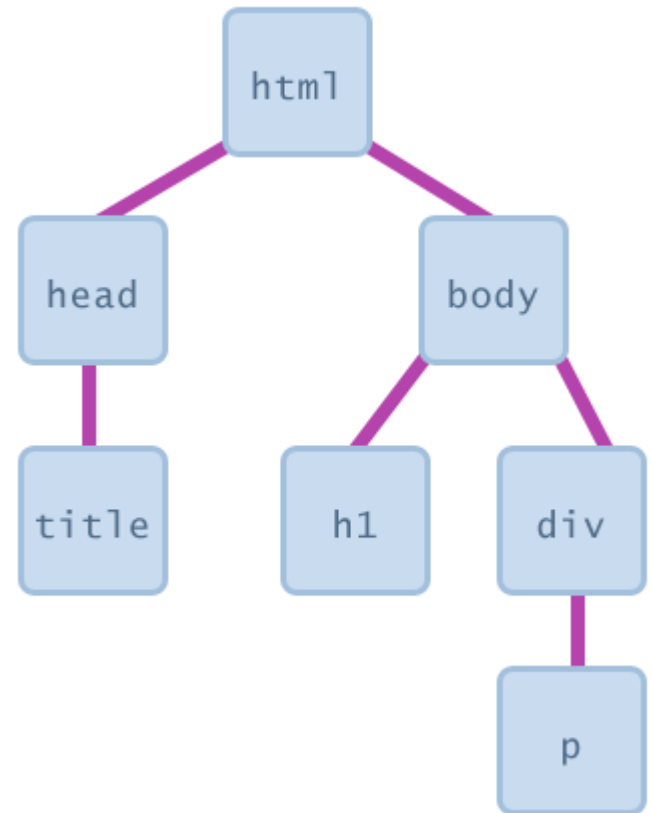
We need to add an event listener to the button...

**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**

- The DOM is the tree of nodes corresponding to HTML elements on a page.
- Can modify, add and remove nodes on the DOM, which will modify, add, or remove the corresponding element on the page.



# 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 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

# 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>
▼<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?

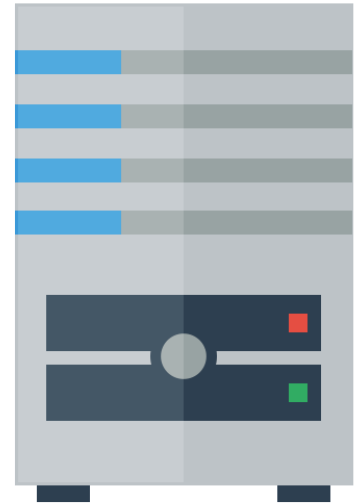
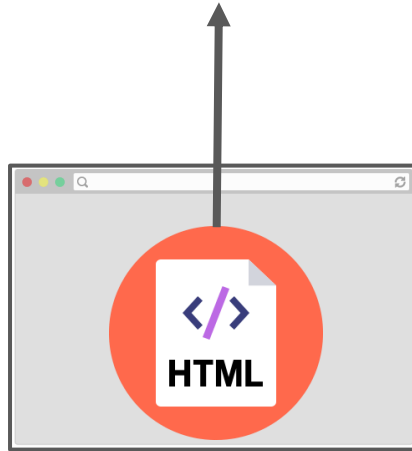
```
<head>
```

```
  <title>WPR</title>
```

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

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

```
</head>
```



```
<head>
```

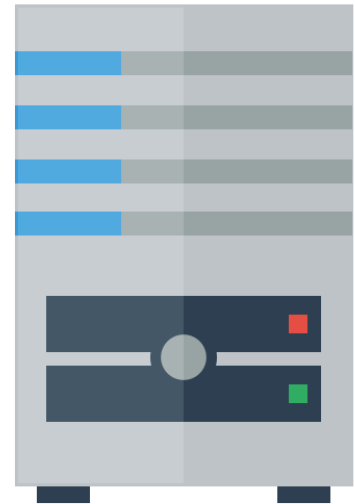
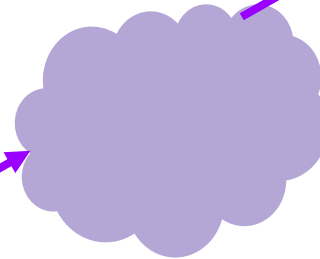
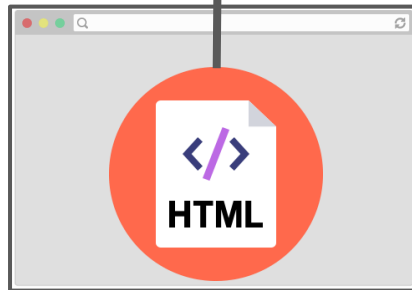
```
  <title>WPR</title>
```

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

➡ 

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

```
</head>
```



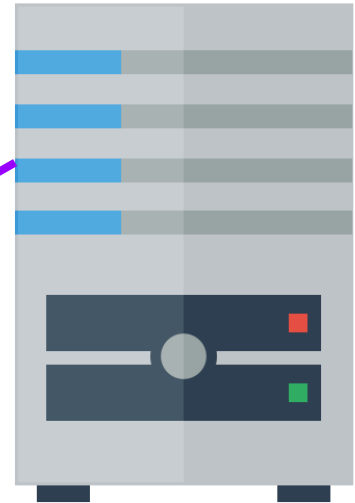
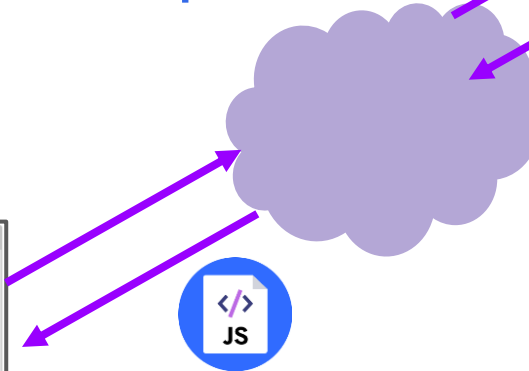
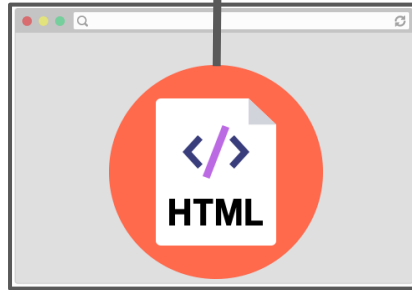
```
<head>
```

```
  <title>WPR</title>
```

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

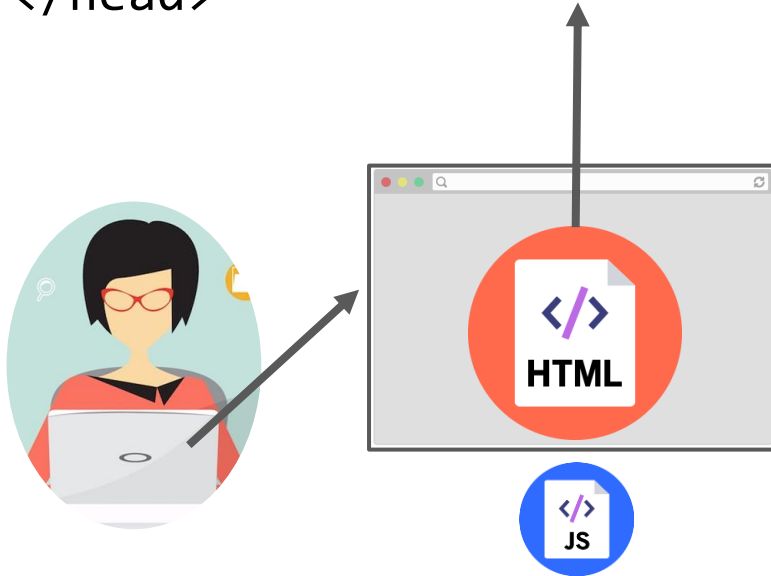
```
➡ <script src="script.js"></script>
```

```
</head>
```





```
<head>
  <title>WPR</title>
  <link rel="stylesheet" href="style.css" />
  ➡ <script src="script.js"></script>
</head>
```

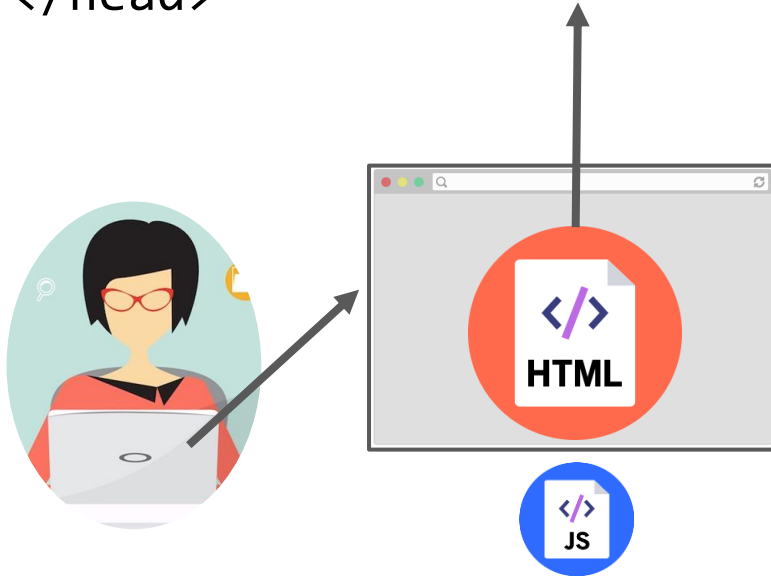


➡

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

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

```
<head>
  <title>WPR</title>
  <link rel="stylesheet" href="style.css" />
  ➡ <script src="script.js"></script>
</head>
```

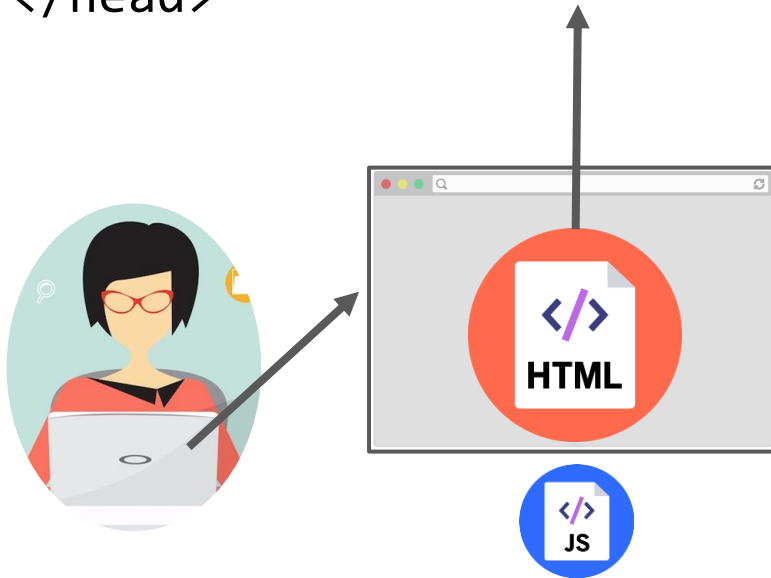


➡

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

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

```
<head>
  <title>WPR</title>
  <link rel="stylesheet" href="style.css" />
  ➡ <script src="script.js"></script>
</head>
```



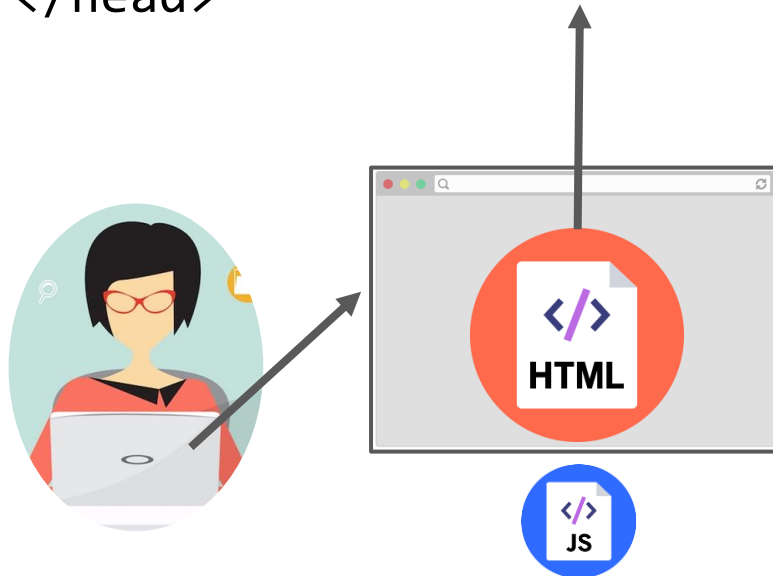
➡

```
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>WPR</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);
```

Click Me!



Elements

Console



top

clicked



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](#).)

# Some properties of Element objects

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

Maybe we can adjust  
the **textContent**!  
[CodePen](#)

More next time!