

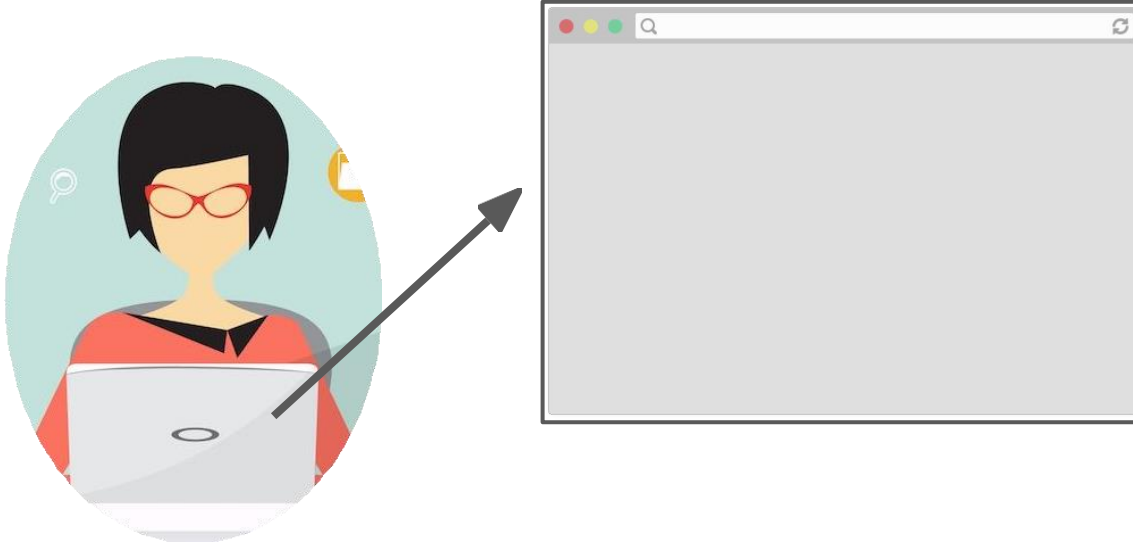
# CS353: Javascript

How do web pages  
work?

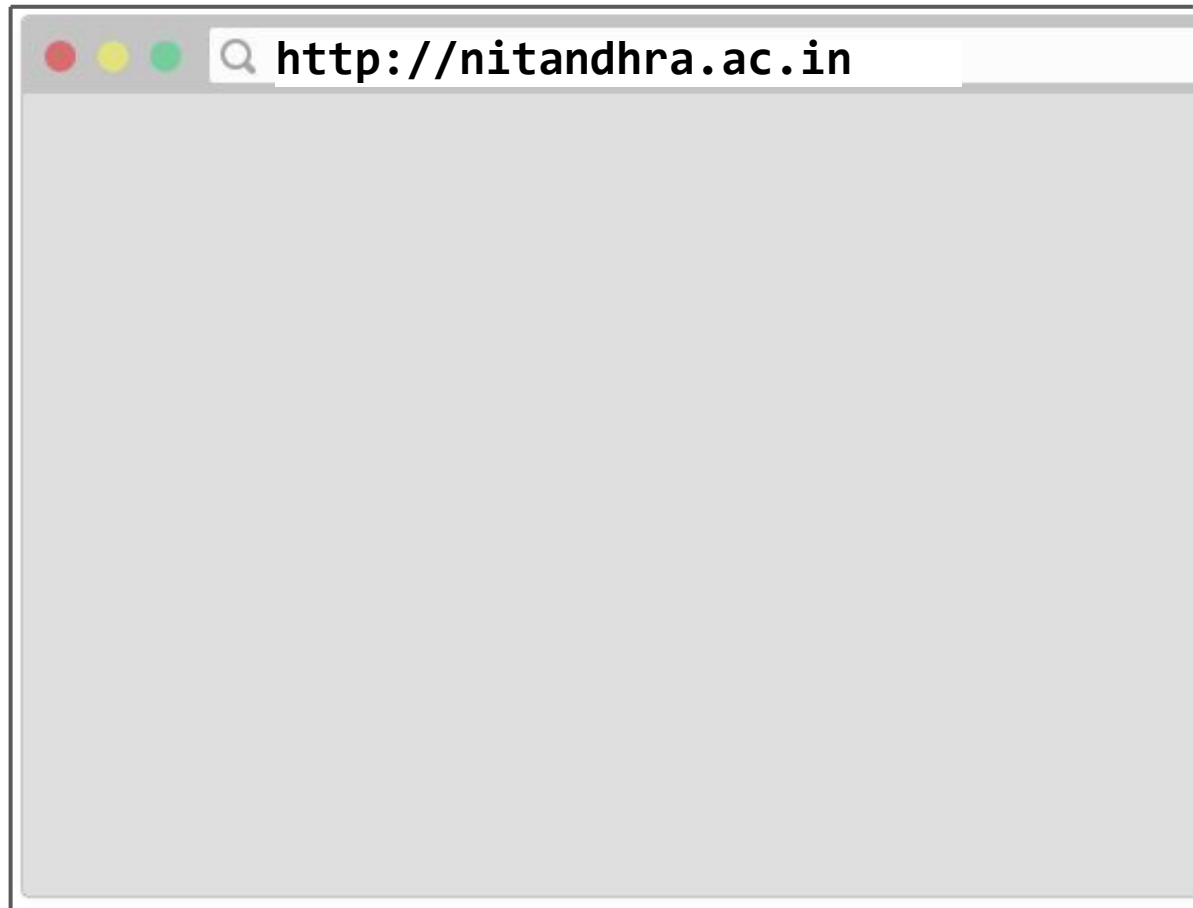
You are on  
your laptop



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

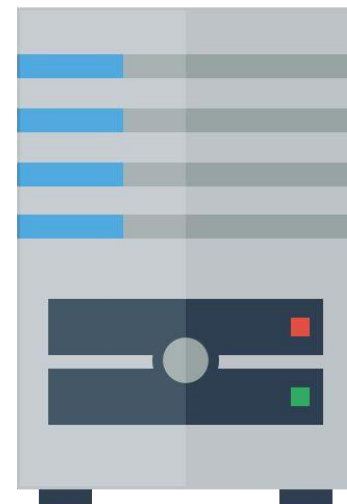
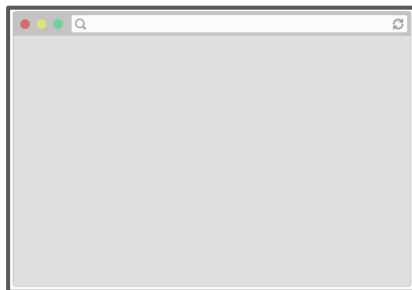


You type a URL in  
the address bar and  
hit "enter"



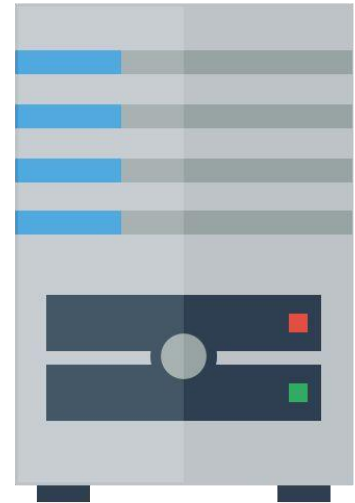
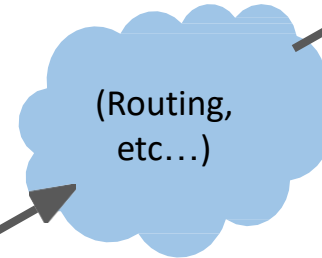
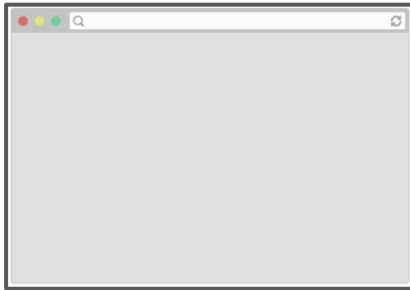
Server at

<http://nitandhra.ac.in>



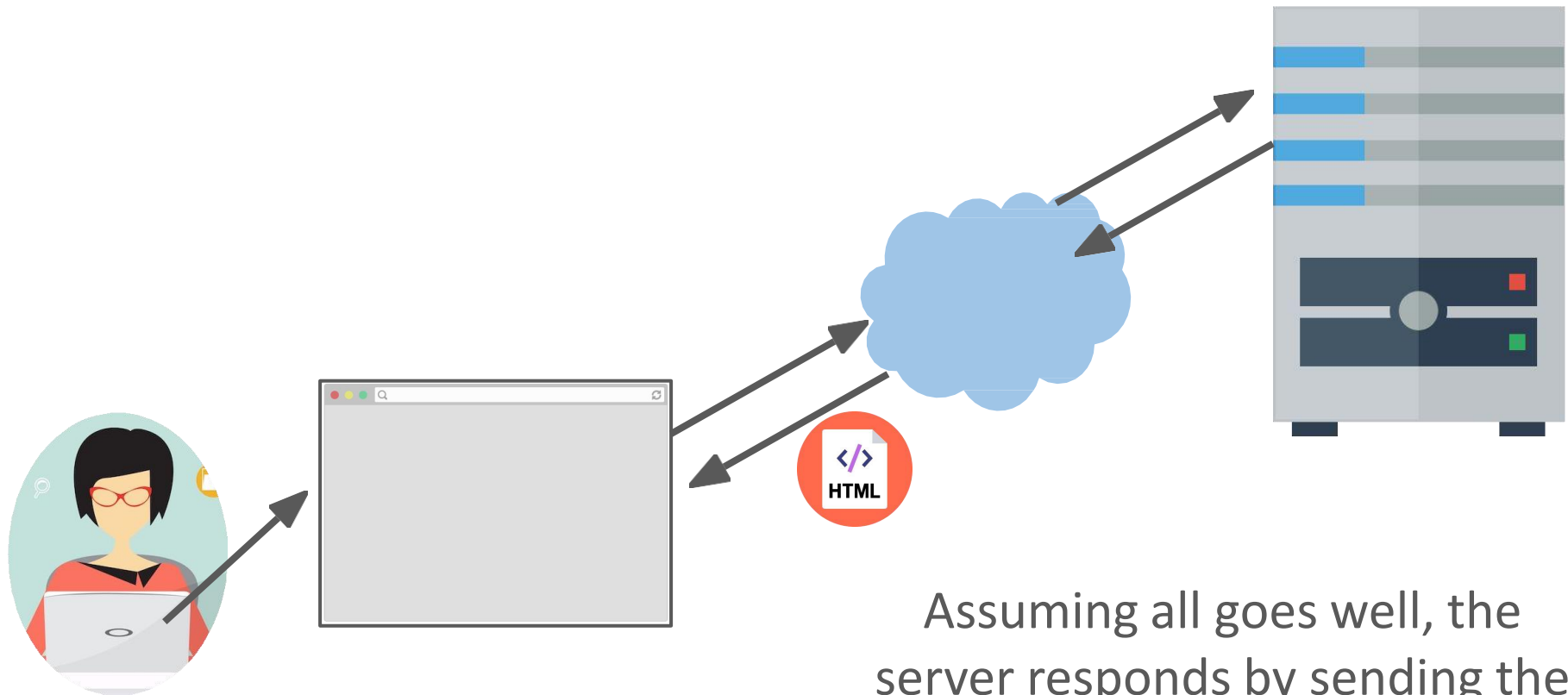
Server at  
<http://nitandhra.ac.in>

Browser sends an HTTP request  
saying "Please GET me the  
index.html file at  
<http://nitandhra.ac.in>"



Server at

<http://nitandhra.ac.in>



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



Server at

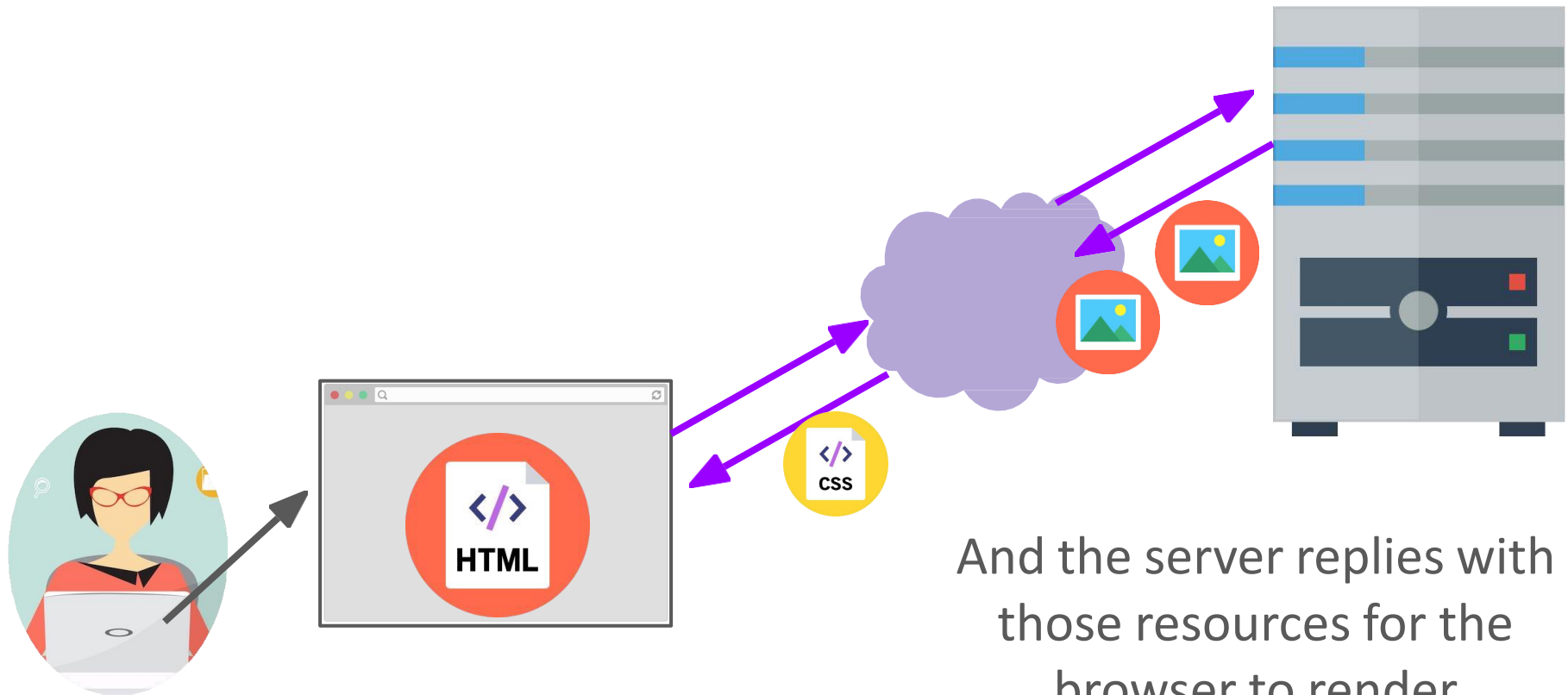
<http://nitandhra.ac.in>

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



Server at

<http://nitandhra.ac.in>



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



http://nitandhra.ac.in

राष्ट्रीय प्रौद्योगिकी संस्थान आंध्र प्रदेश  
NATIONAL INSTITUTE OF TECHNOLOGY ANDHRA PRADESH  
(An autonomous Institute under the aegis of Ministry of Education, Govt. of India)

HOME | DIRECTOR | ADMINISTRATION | ADMISSIONS | ACADEMICS | RESEARCH | PEOPLE | T & P CELL | FACILITIES | STUDENT WELFARE | NIRF

76th Independence Day

**Announcements**

- List of eligible candidates for interview- PhD/MS Admission- January 2023 session
- Applications are invited from interested candidates for a temporary Junior Research Fellow (JRF) position in a project sponsored by SERB, GOI, in the

**About the Institute**

**Vision**

**Mission**

National Institute of Technology, Andhra Pradesh is the 31st institution among the chain of NITs started by the Government of India. NIT Andhra Pradesh is established in the state of Andhra Pradesh recently in the academic year 2015 – 2016. Currently the Institute is offering B.Tech., M.Tech., Ph.D. and MS programmes.

A new campus has been established with 172.6 acres of land adjacent to

**Publications/Achievements**

- Savitha Chirasmayee ( FT Scholar) and Dr. T. Reshma's research work titled "Mapping Cropland Extent Using Sentinel-2 Datasets and Machine Learning Algorithms for an Agriculture Watershed" has been Published in *Smart Agricultural Technology, Elsevier*



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!

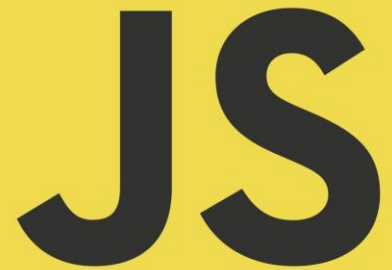
# JavaScript

# 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)

# 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>CS 353</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?

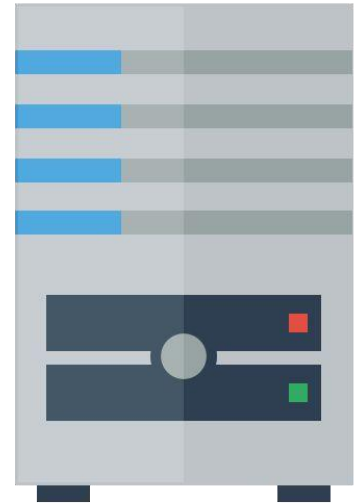
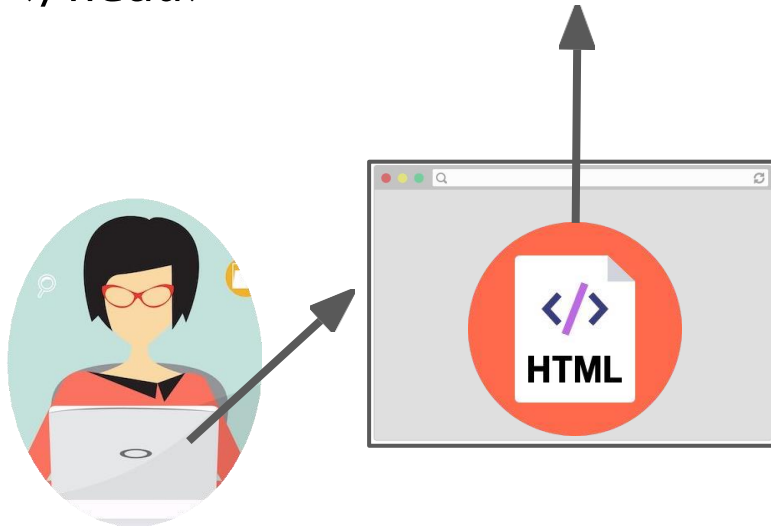
```
<head>
```

```
  <title>CS 353</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.

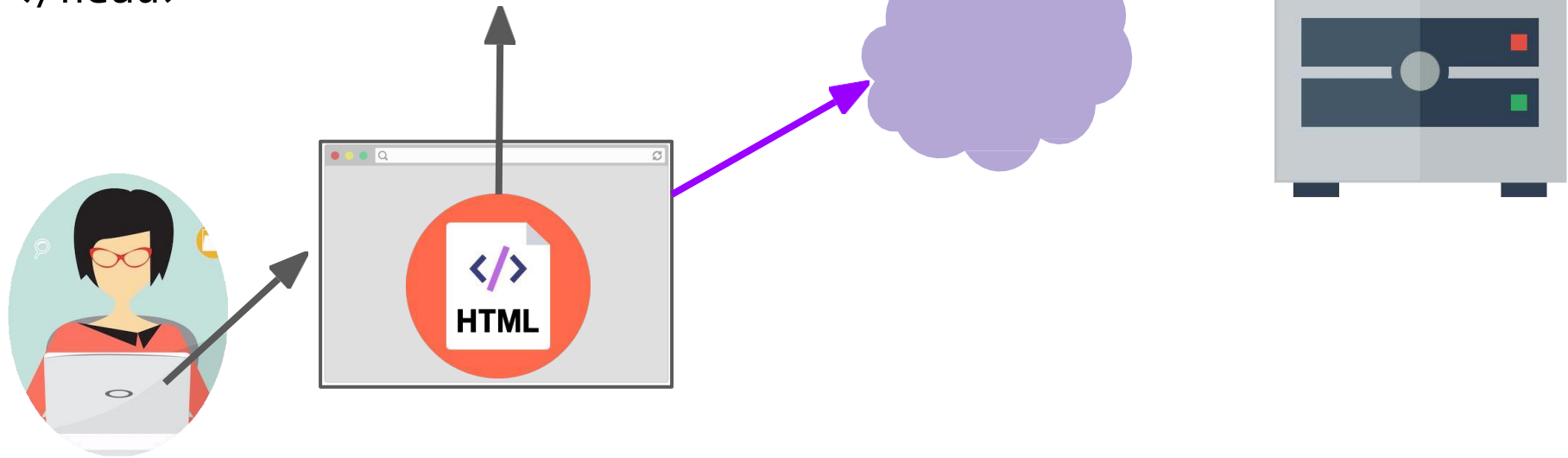
```
<head>
```

```
  <title>CS 353</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>
```

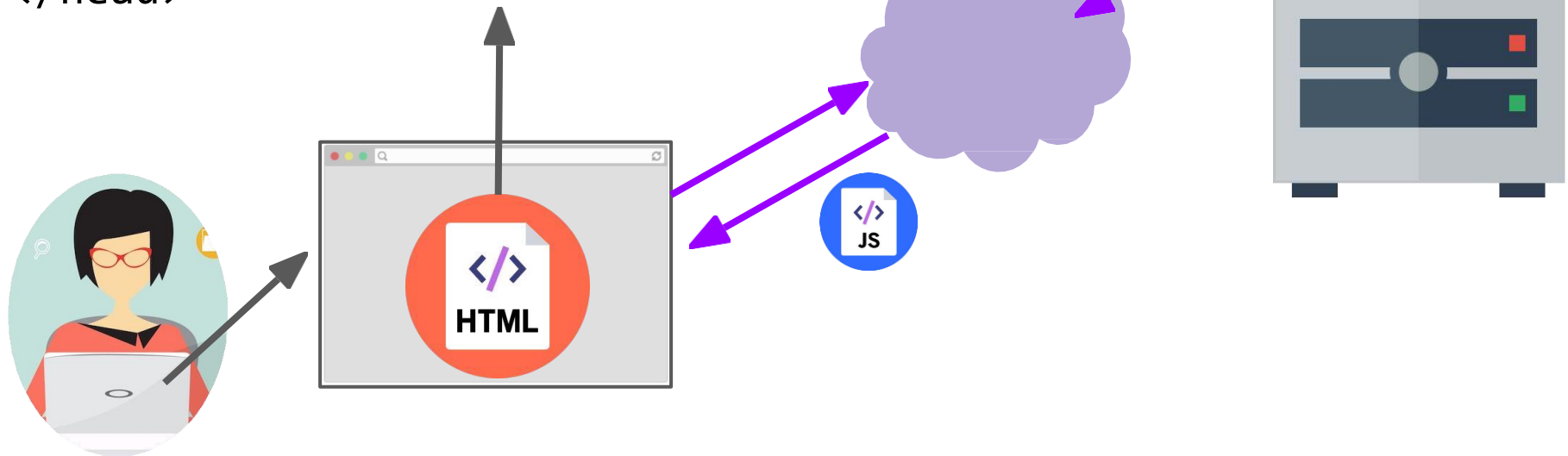
```
  <title>CS 353</title>
```

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

➔ 

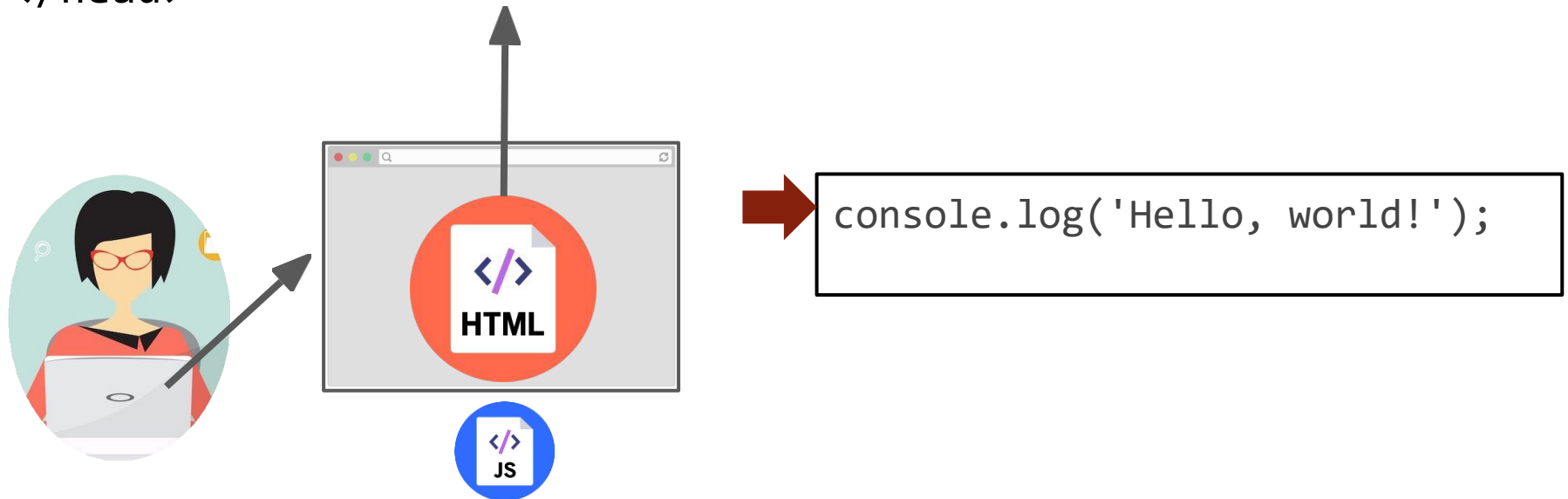
```
  <script src="script.js"></script>
```

```
</head>
```



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

```
<head>
  <title>CS 353</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

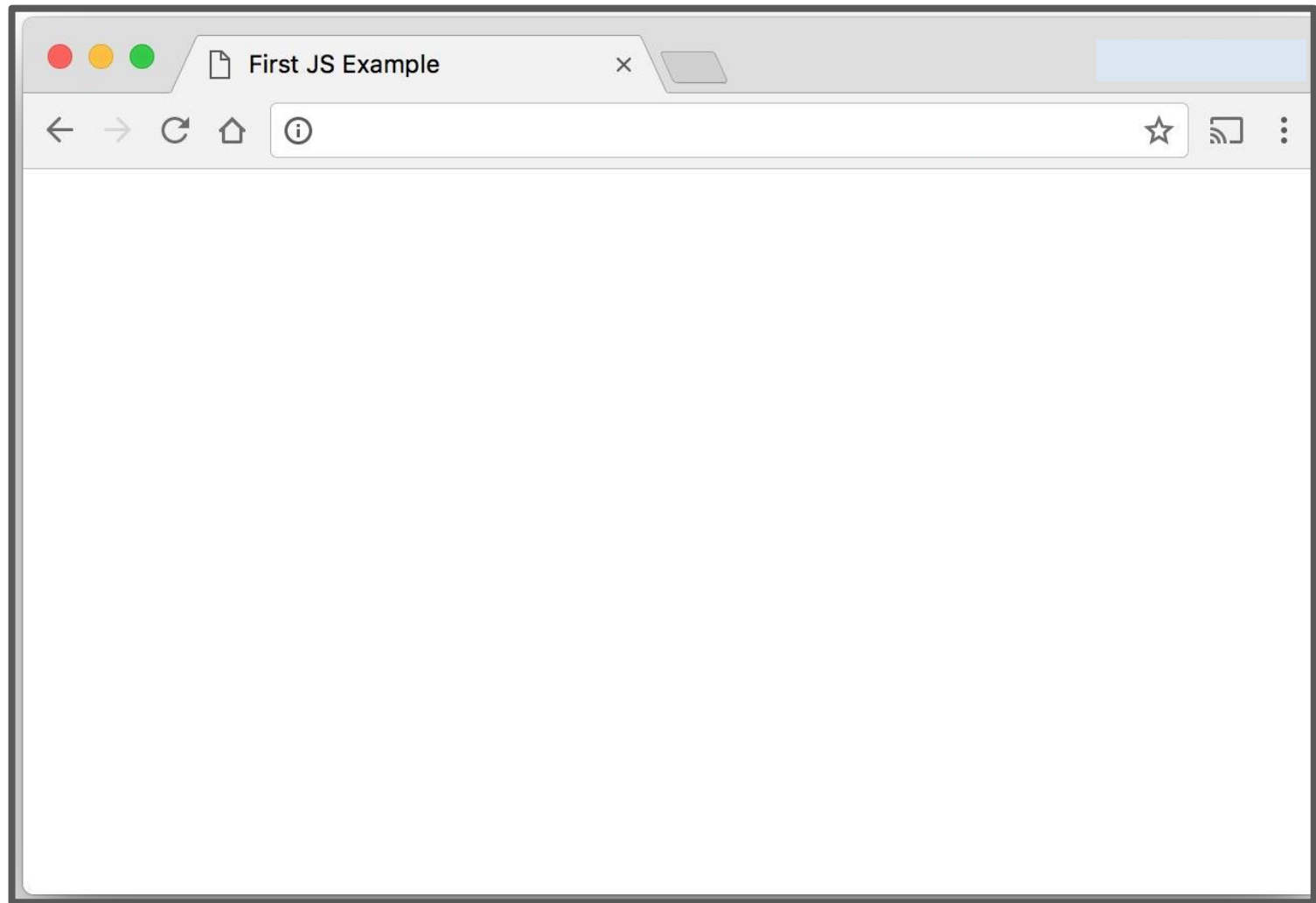
(Note that this is slightly different than being "interpreted": see [just-in-time \(JIT\) compilation](#))

## first-js.html

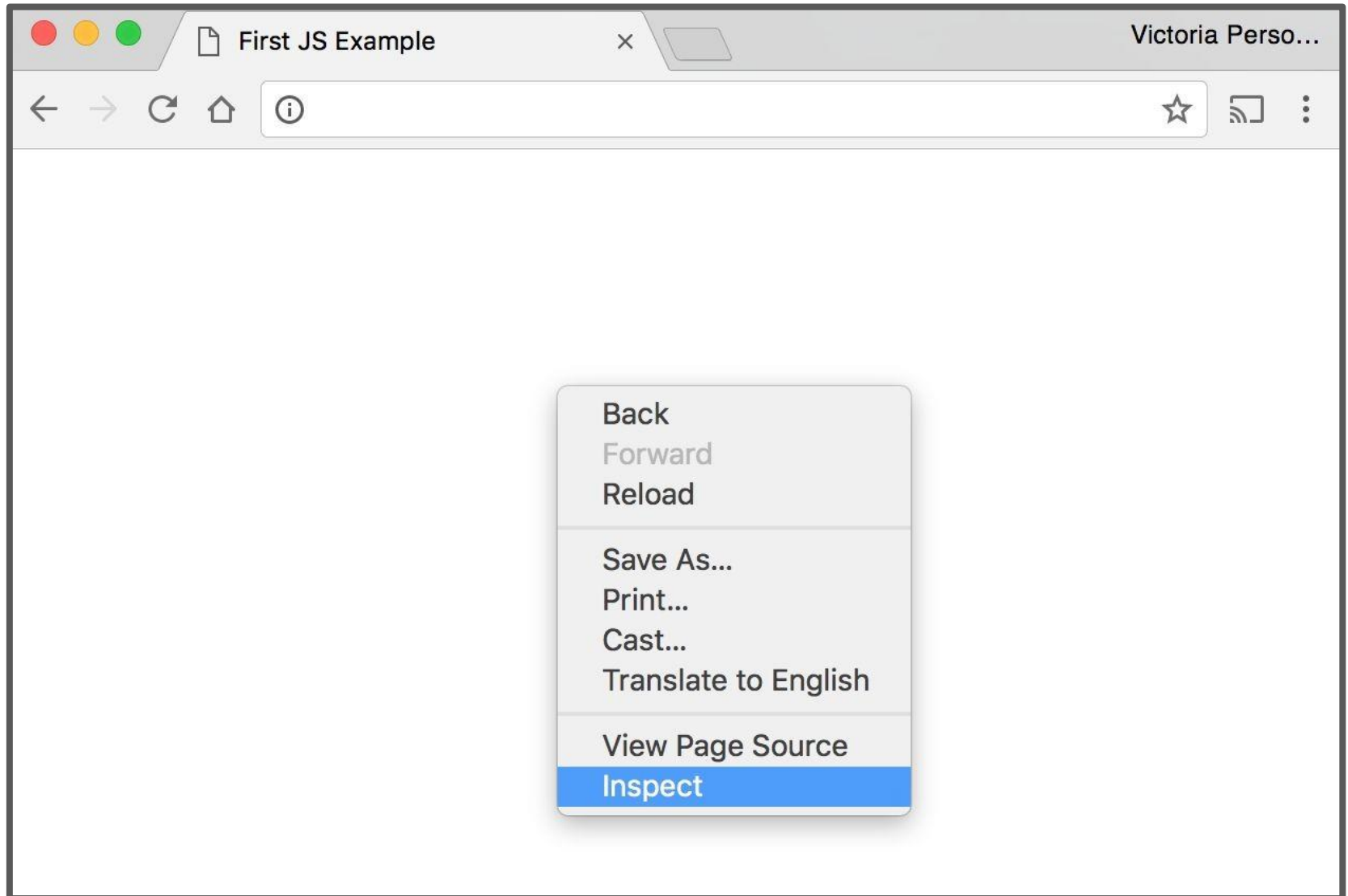
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
  </body>
</html>
```

## script.js

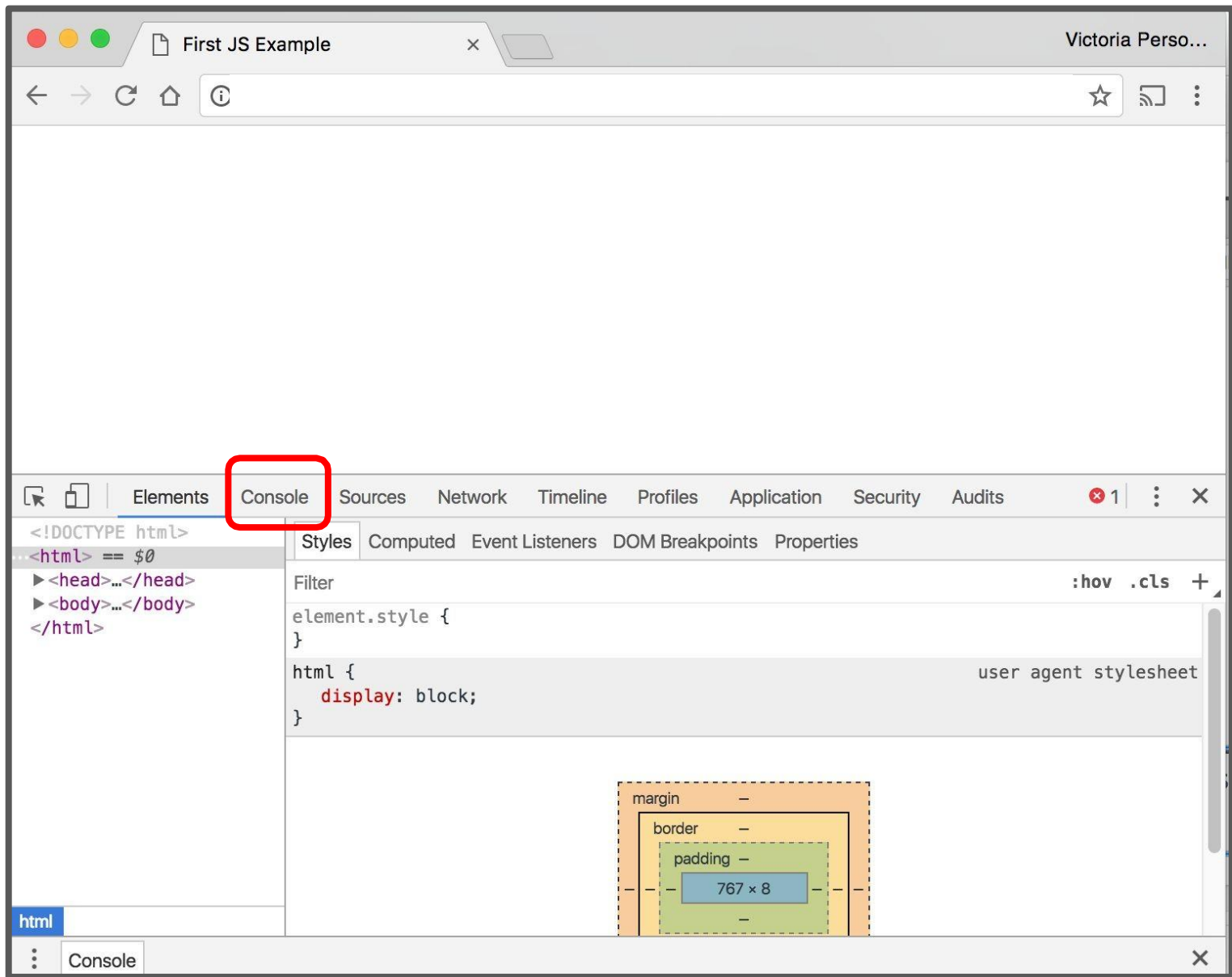
```
console.log('Hello, world!');
```



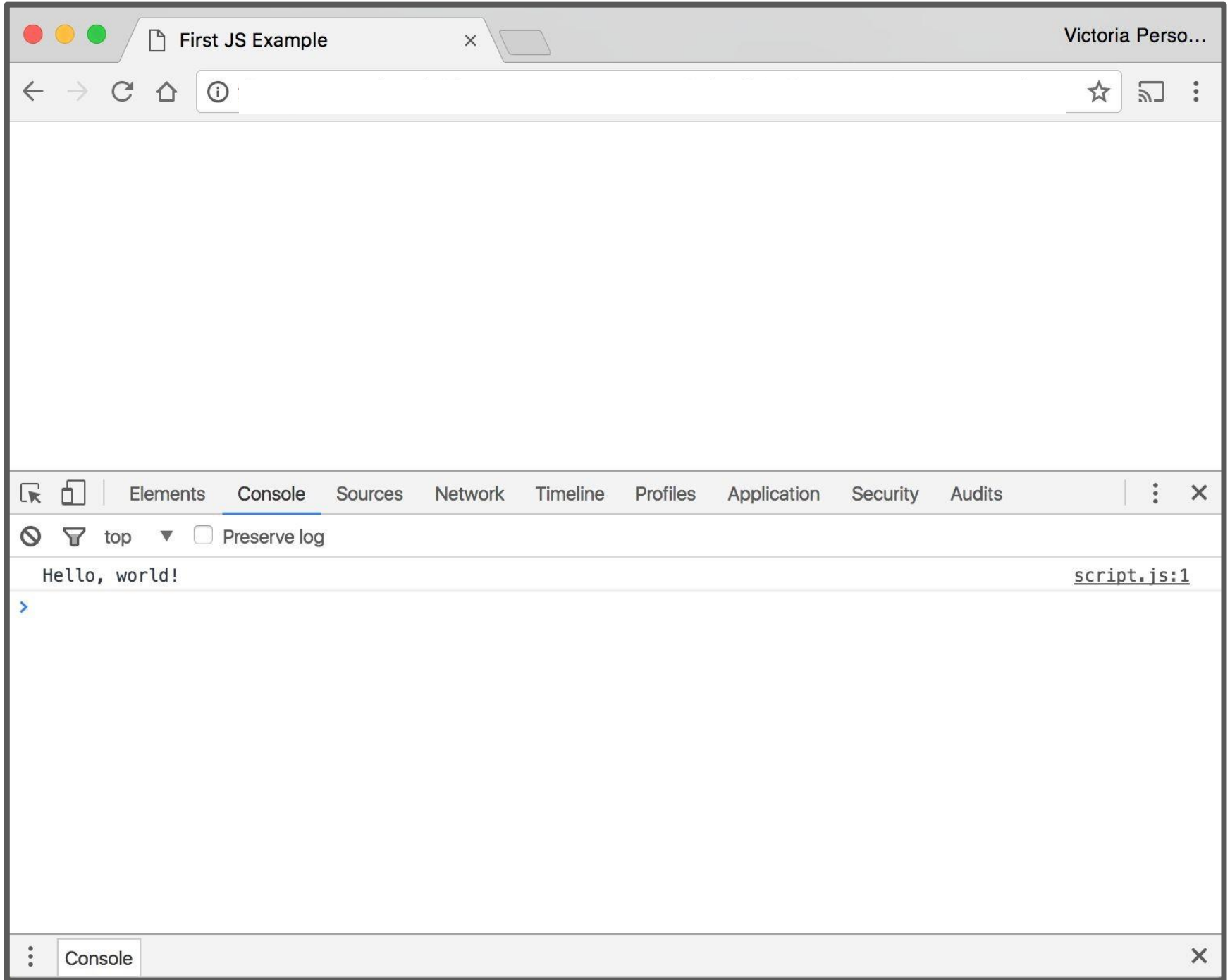
Hey, nothing happened!



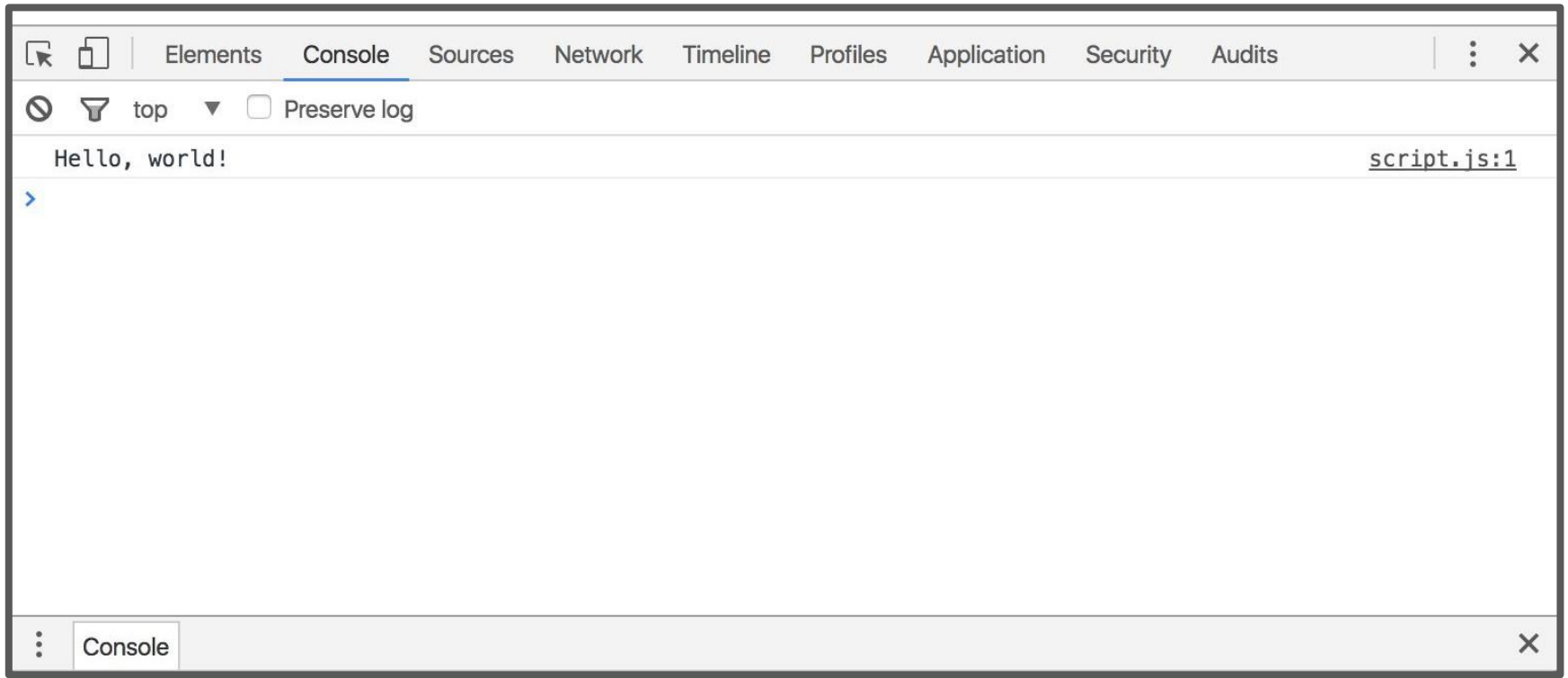
Right-click (or control-click on Mac) and choose "Inspect"



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 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();
```



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](#)). 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

# 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")

# 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`

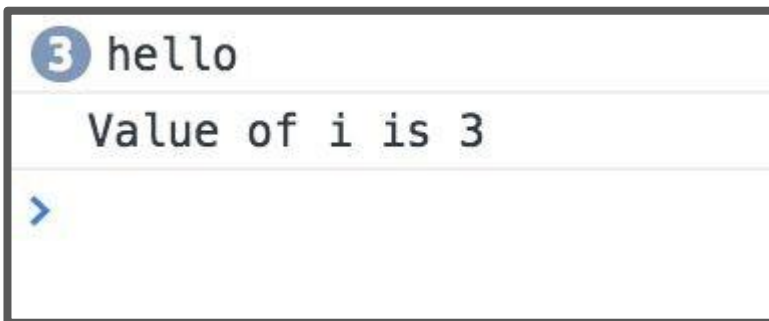
# Understanding var

```
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

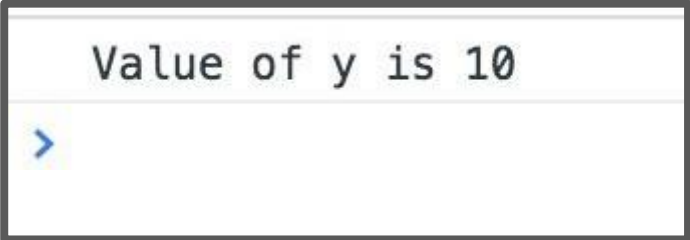
```
function printMessage(message, times)
{ for (var i = 0; i < times; i++) {
    console.log(message);
}
  console.log('Value of i is ' + i);
}
printMessage('hello', 3);
```



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);
```

A screenshot of a console output window. The text "Value of y is 10" is displayed in a monospace font. Below the text is a prompt character ">" on a new line.

- 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, there is a blue circle with the number '3' followed by the text 'hello'. Below this, there is a red error message: 'Uncaught ReferenceError: i is not defined'. The message is followed by two lines of stack trace: 'at printMessage (script.js:5)' and 'at script.js:8'. At the bottom of the console, there is a blue prompt character '>'.

```
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 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.)

# 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`





# object type

- Object is an unordered collection of name-value pairs called **properties**  
`let foo = {};`  
`let bar = {name: "Alice", age: 23, state: "California"};`
- Name can be any string: `let x = { "": "empty", "---": "dashes" }`
- Referenced either like a structure or like a hash table with string keys:  
`bar.name` or `bar["name"]`  
`x["---"]` // have to use hash format for illegal names  
`foo.nonExistent == undefined`
- Global scope is an object in browser (i.e. `window[prop]`)

# Properties can be added, removed, enumerated

- To add, just assign to the property:

```
let foo = {};  
foo.name = "Fred";      // foo.name returns "Fred"
```

- To remove use delete:

```
let foo = {name: "Fred"};  
delete foo.name; // foo is now an empty object
```

- To enumerate use Object.keys():

```
Object.keys({name: "Alice", age: 23}) = ["name", "age"]
```

# 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)

# Arrays

```
let anArr = [1,2,3];
```

Are special objects: `typeof anArr == 'object'`

Indexed by non-negative integers: `(anArr[0] == 1)`

Can be **sparse** and **polymorphic**: `anArr[5]='FooBar'; //[1,2,3,,, 'FooBar']`

Like strings, have many methods: `anArr.length == 3`

`push, pop, shift, unshift, sort, reverse, splice, ...`

# Dates

```
let date = new Date();
```

Are special objects: `typeof date == 'object'`

The number of milliseconds since midnight January 1, 1970 UTC

Timezone needed to convert. Not good for fixed dates (e.g. birthdays)

Many methods for returning and setting the data object. For example:

```
date.valueOf() = 1452359316314
```

```
date.toISOString() = '2016-01-09T17:08:36.314Z'
```

```
date.toLocaleString() = '1/9/2016, 9:08:36 AM'
```

# Regular Expressions

```
let re = /ab+c/;    or    let re2 = new RegExp("ab+c");
```

Defines a pattern that can be searched for in a string

String: `search()`, `match()`, `replace()`, and `split()`

RegExp: `exec()` and `test()`

Uses:

Searching: Does this string have a pattern I'm interested in?

Parsing: Interpret this string as a program and return its components

## Regular Expressions by example - search/test

```
/HALT/.test(str);    // Returns true if string str has the substr HALT
/halt/i.test(str);   // Same but ignore case
/[Hh]alt [A-Z]/.test(str); // Returns true if str either "Halt L" or "halt L"

'XXX abbbbbbbc'.search(/ab+c/);    // Returns 4 (position of 'a')
'XXX ac'.search(/ab+c/);           // Returns -1, no match
'XXX ac'.search(/ab*c/);           // Returns 4

'12e34'.search(/^[^d]/);           // Returns 2
'foo: bar;'.search(/...\s*:\s*...\s*/); // Returns 0
```

# Default parameters - Parameters not specified

## Old Way

```
function myFunc(a,b) {  
    a = a || 1;  
    b = b || "Hello";  
}
```

Unspecified parameters are set to undefined.  
You need to explicitly set them if you want a different default.

## New Way

```
function myFunc (a = 1, b = "Hello") {  
}
```

Can explicitly define default values if parameter is not defined.



# Destructuring assignment

## Old Way

```
var a = arr[0];  
var b = arr[1];  
var c = arr[2];
```

```
var name = obj.name;  
var age = obj.age;  
var salary = obj.salary;
```

```
function render(props) {  
  var name = props.name;  
  var age = props.age;
```

## New Way

```
let [a,b,c] = arr;
```

```
let {name, age, salary} = obj;
```

```
function render({name, age}) {
```

# Template string literals

## Old Way

```
function formatGreetings(name, age) {  
  
  var str =  
    "Hi " + name +  
      " your age is " + age;  
  ...  
}
```

Use string concatenation to build up string from variables.

## New Way

```
function formatGreetings(name, age) {  
  
  let str =  
    `Hi ${name} your age is ${age}`;  
}
```

Also allows multi-line strings:

```
`This string has  
two lines`
```

Very useful in frontend code. Strings can be delimited by " ", ' ', or ` `

# For of

## Old Way

```
var a = [5,6,7];  
var sum = 0;  
for (var i = 0; i < a.length; i++) {  
    sum += a[i];  
}
```

Iterator over an array

## New Way

```
let sum = 0;  
for (ent of a) {  
    sum += ent;  
}
```

Iterate over arrays, strings, Map, Set, without using indexes.