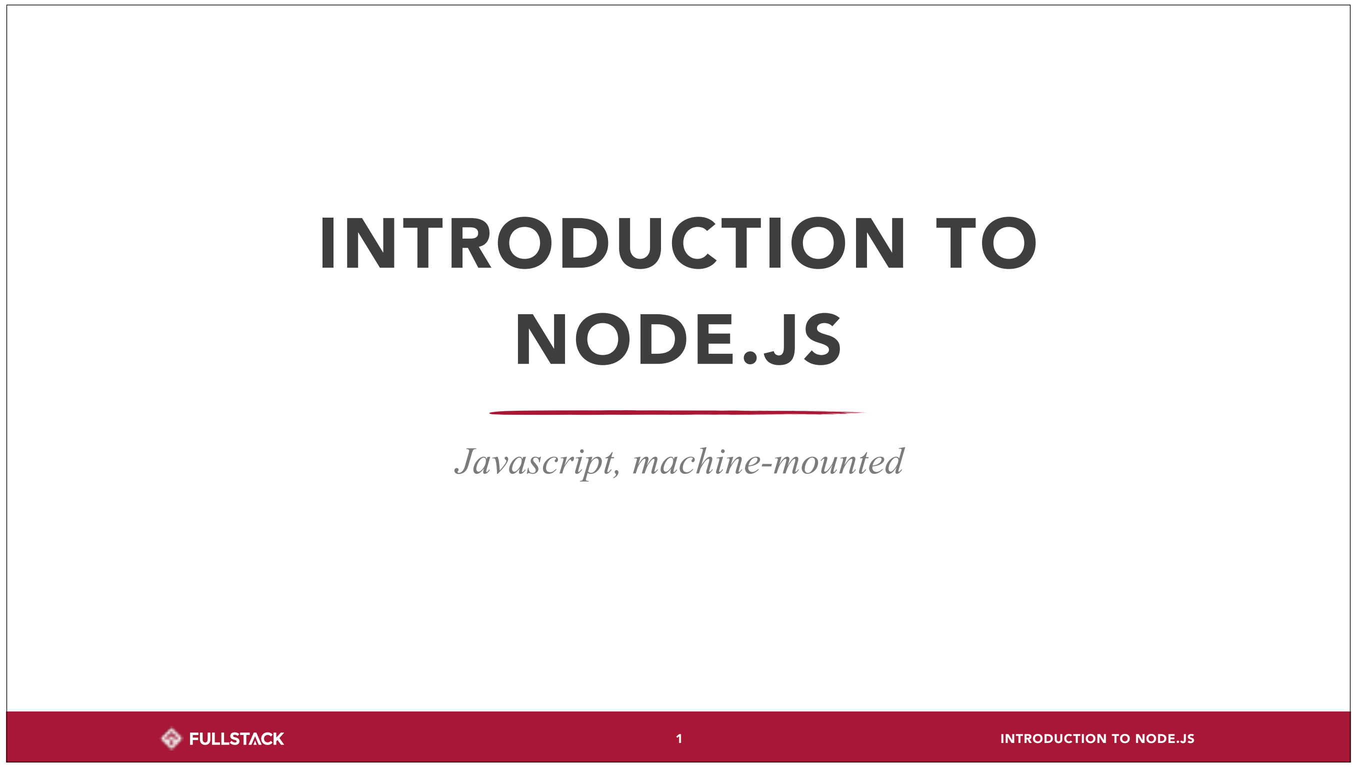


INTRODUCTION TO NODE.JS

Javascript, machine-mounted

So normally around this point in the curriculum, we begin instruction on Node.js. The instructor staff has, however, had heated discussion about reform and we have decided Javascript is outdated and just plain awful. Instead, the remainder of the curriculum will be about . . .

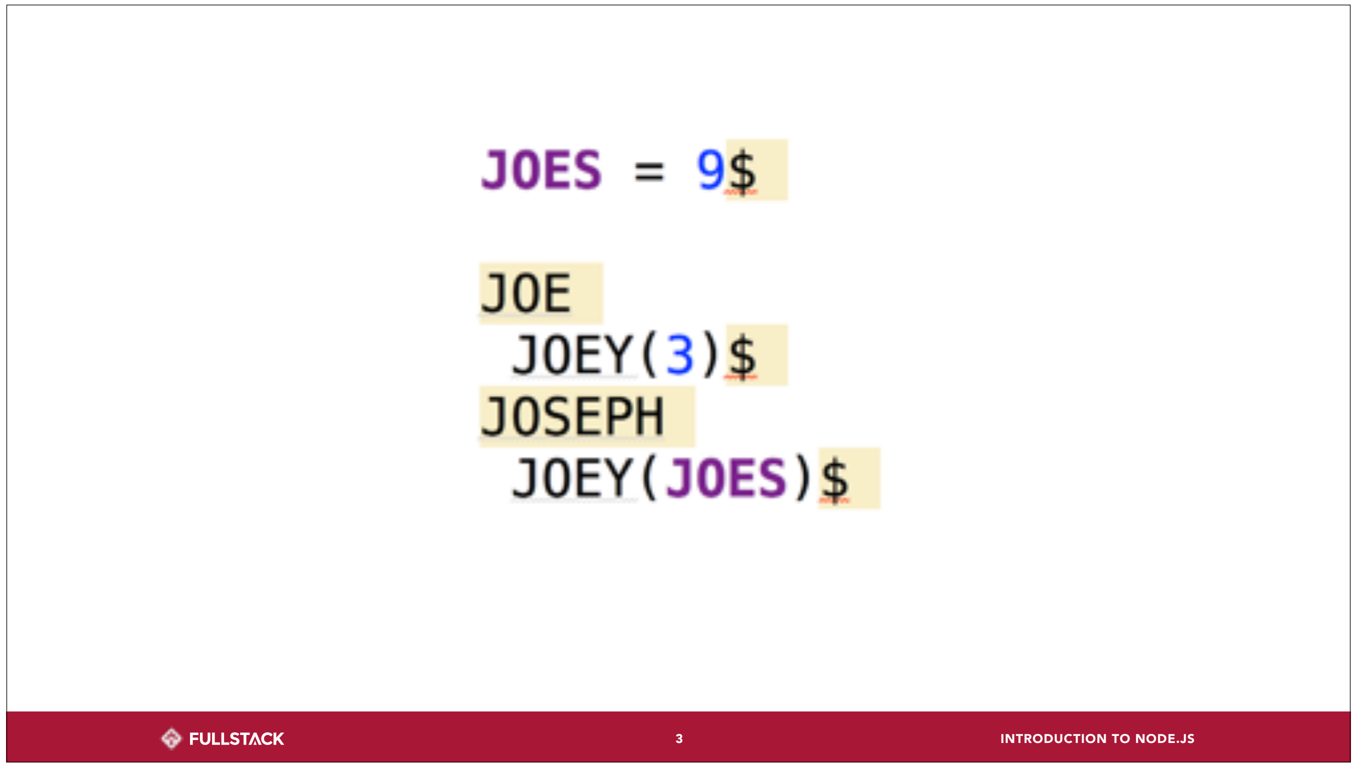
INTRODUCTION TO JOESCRIPT

The last programming language you'll ever need.

JoeScript.

JoeScript is a really fantastic programming language, invented by yours truly (Joe). I have also given this language its namesake.

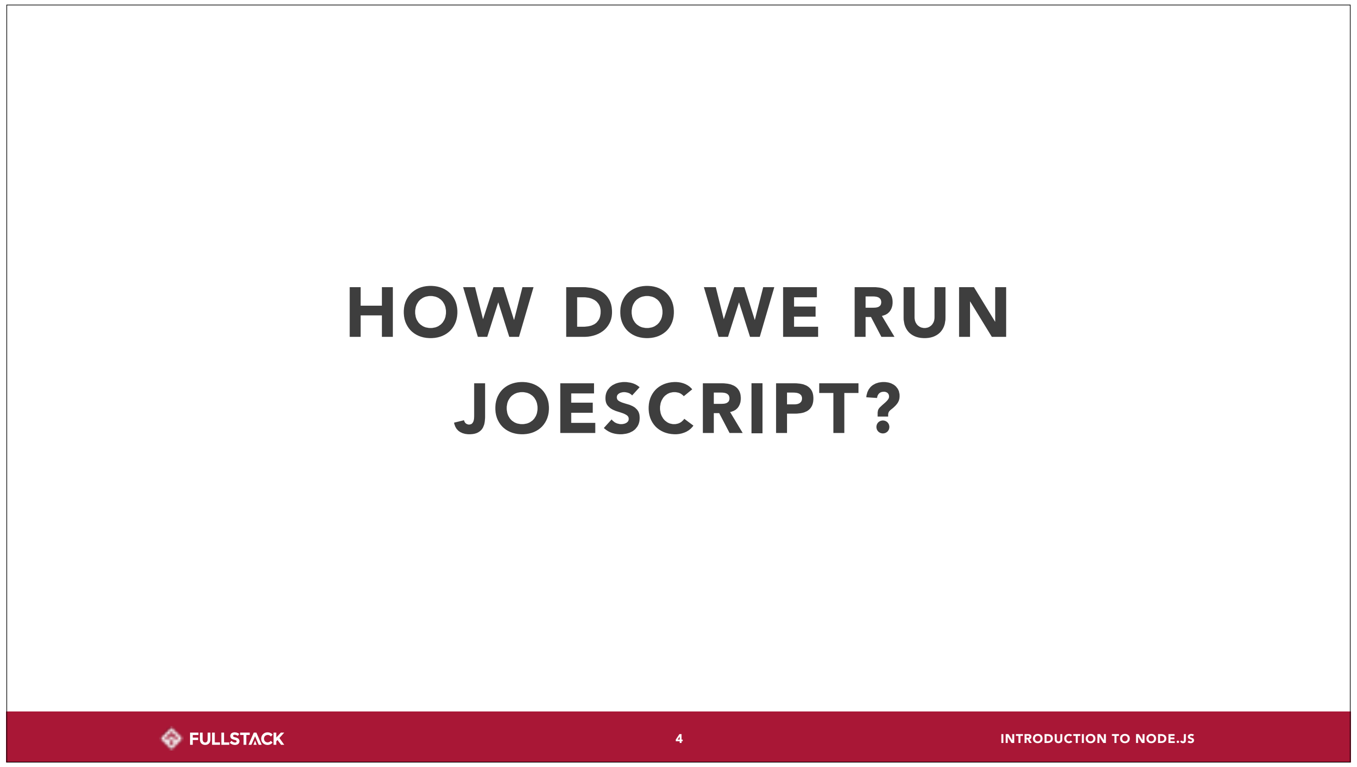
Now, you might be pretty upset about this but, trust me, JoeScript is pretty sweet.



```
JOES = 9$

JOE
 JOEY(3)$
JOSEPH
 JOEY(JOES)$
```

Here’s an example of JoeScript. We have a variable JOES and we make that equal to 9. We use dollar signs instead of semicolons in JoeScript, because JoeScript allows you to make programs that make mad monies.

Then we have a JOE statement and if JOE we called JOEY with 3, but if JOSEPH we call JOEY with JOES. JOEY takes the given value and emails it to me, joe@fullstackacademy.com.

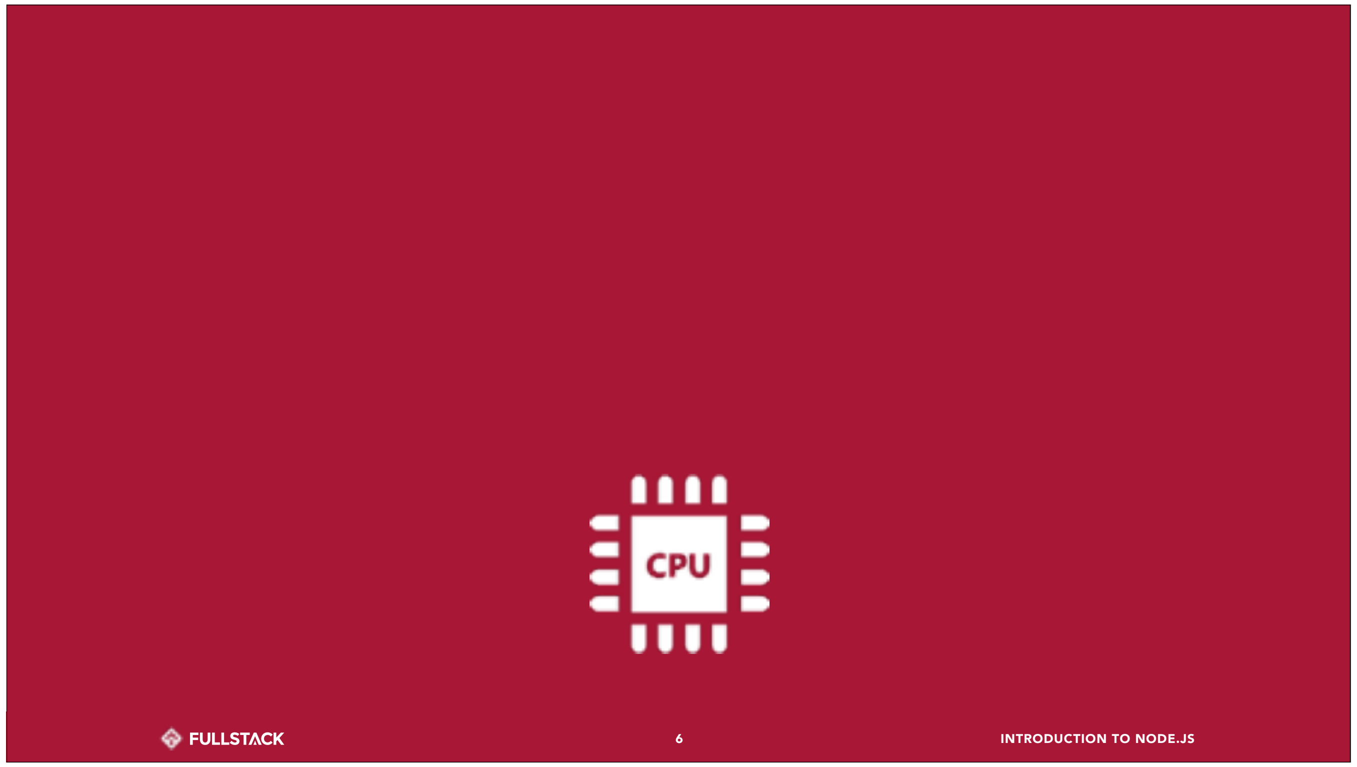


HOW DO WE RUN JOESCRIPT?

So, now that you have learned every feature JoeScript, you're ready to write some programs. But, how do we run our JoeScript code? How do we get this code to actually run on a computer?

HOW DO WE RUN ANY PROGRAMMING LANGUAGE?

I think the better question to ask is, how do we run any programming language? What does executing a computer program mean?

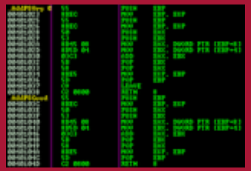


We all know what the CPU is. If you don't, the CPU is a "Central Processing Unit". It is a tiny chip physically inside your computer that is responsible for the computer operating.

Everything your computer does is a result of work done by this CPU. All software, in any language, at some deep level of abstraction, boils down to providing direct instructions to the CPU.

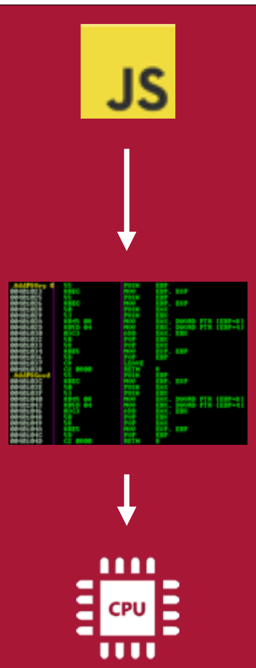
These direct instructions are called “machine code”. These are extremely low-level instructions sent to the CPU to perform operations. If your program, in any language, evaluates “2+2”, it is translating that “2+2” into something the CPU understands.

JoeScript

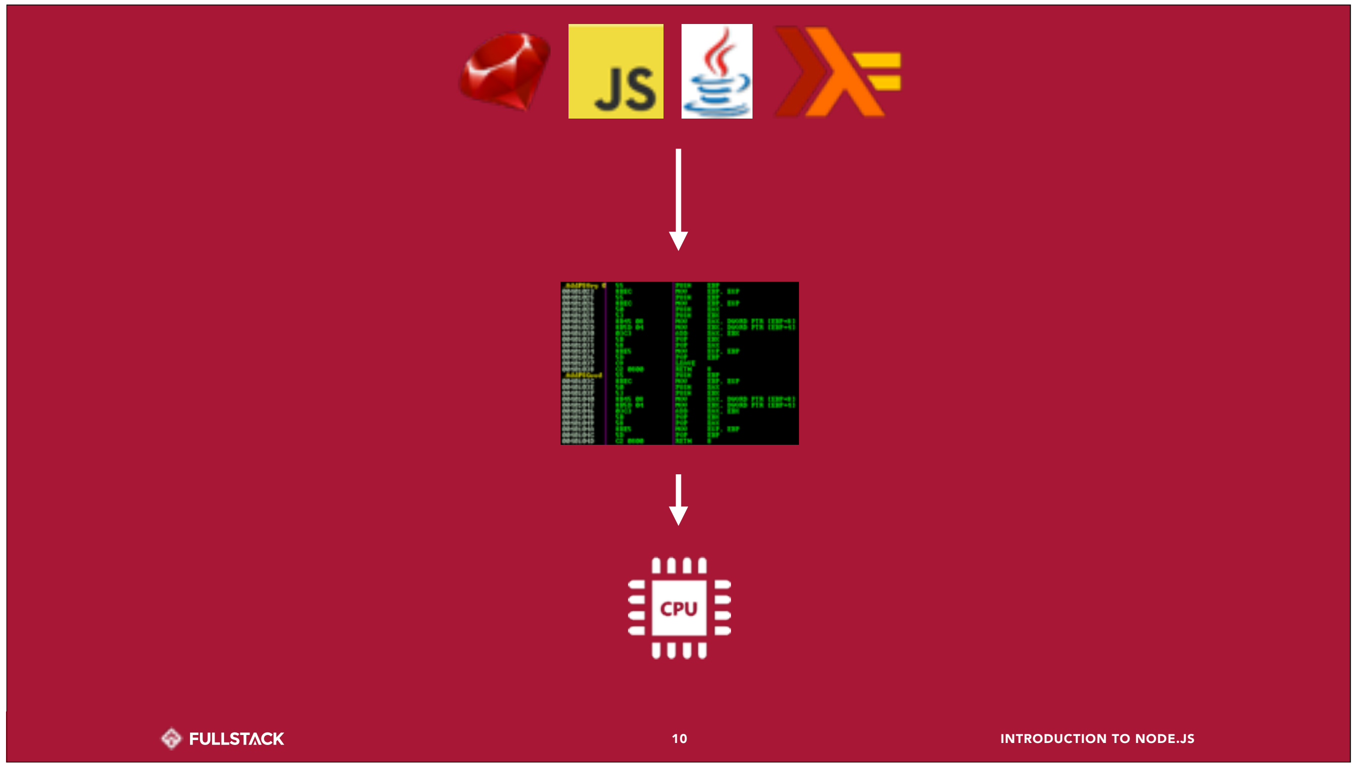


CPU

So in order for you to run JoeScript, we need to translate Joescript code into machine instructions for the CPU.



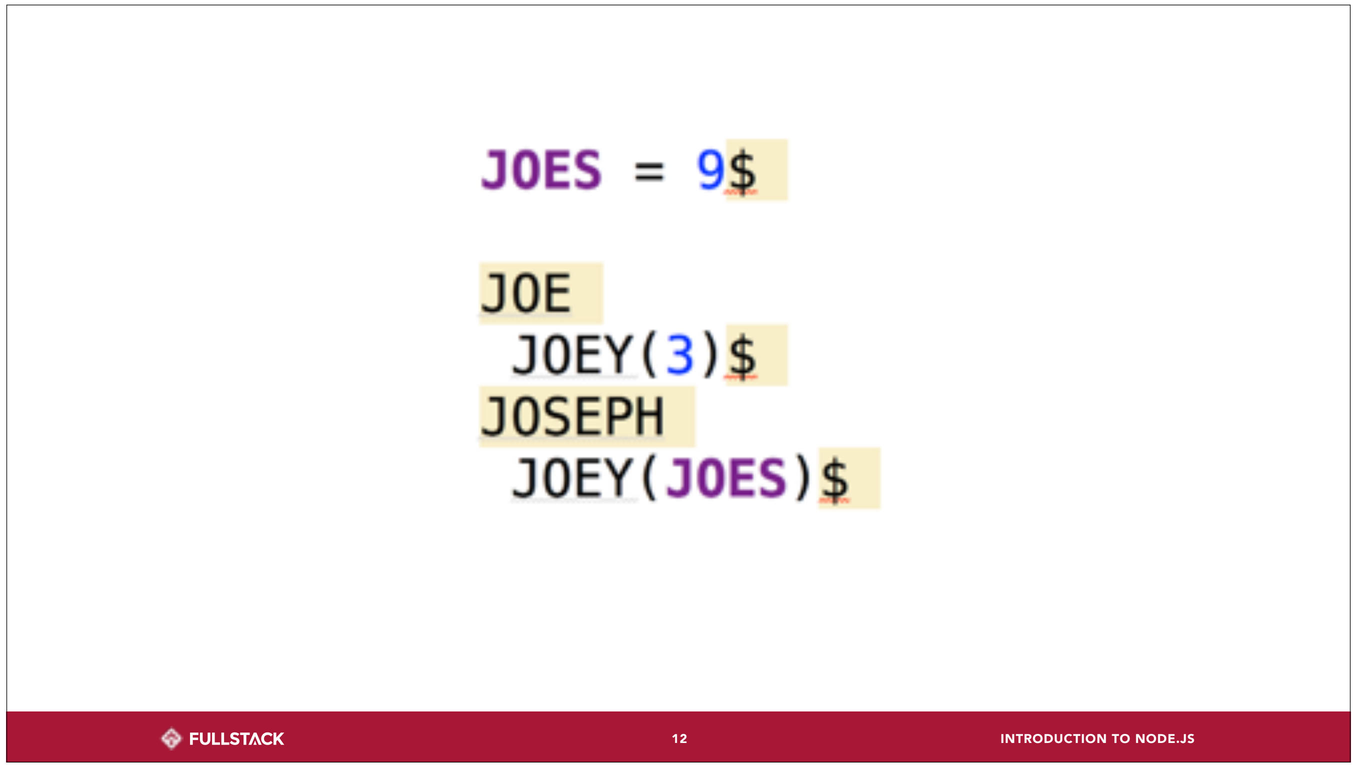
The same need exists with Javascript. Javascript itself doesn't just magically run. It needs to be translated into machine instructions for the CPU.



In fact, nearly every programming language needs to be translated into machine instructions for the CPU.

IT'S ALL JUST TEXT

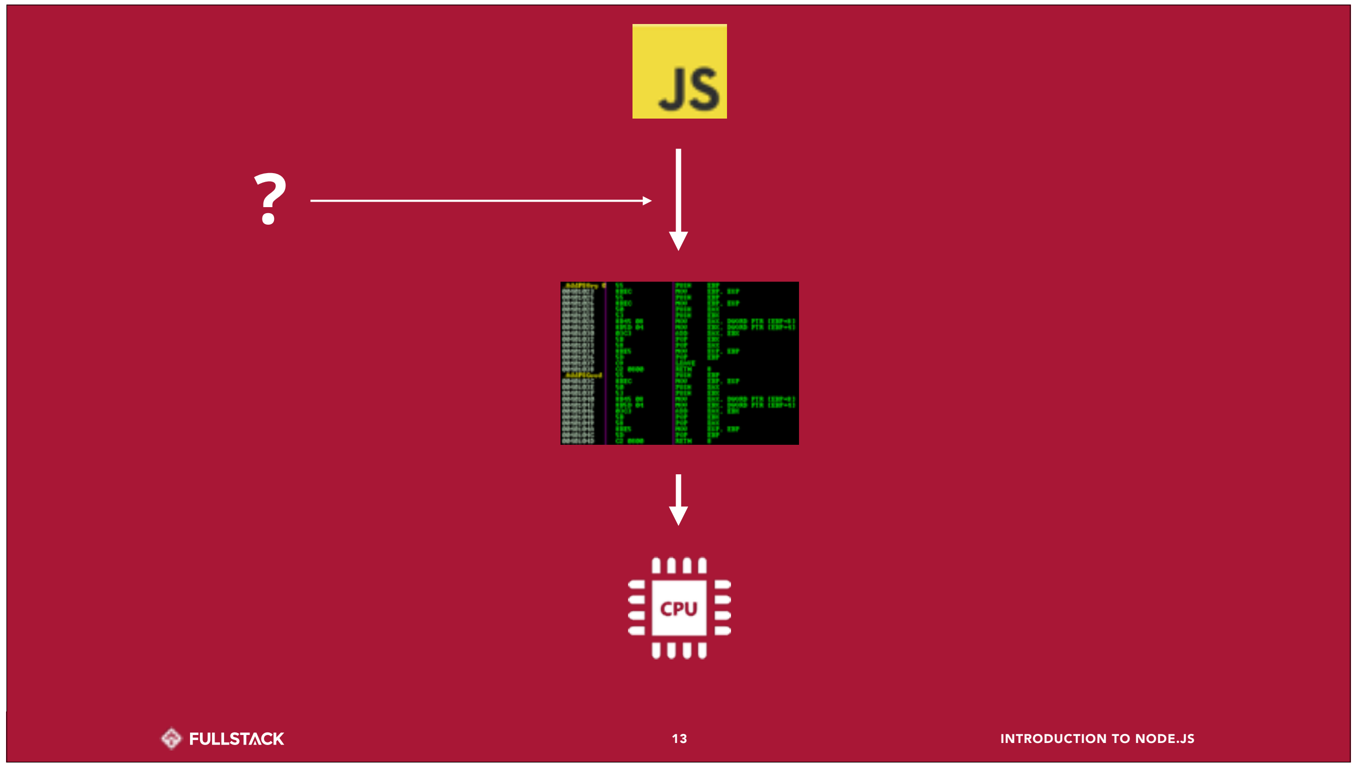
Because programming languages don't just magically work. Code is just text, nothing more.



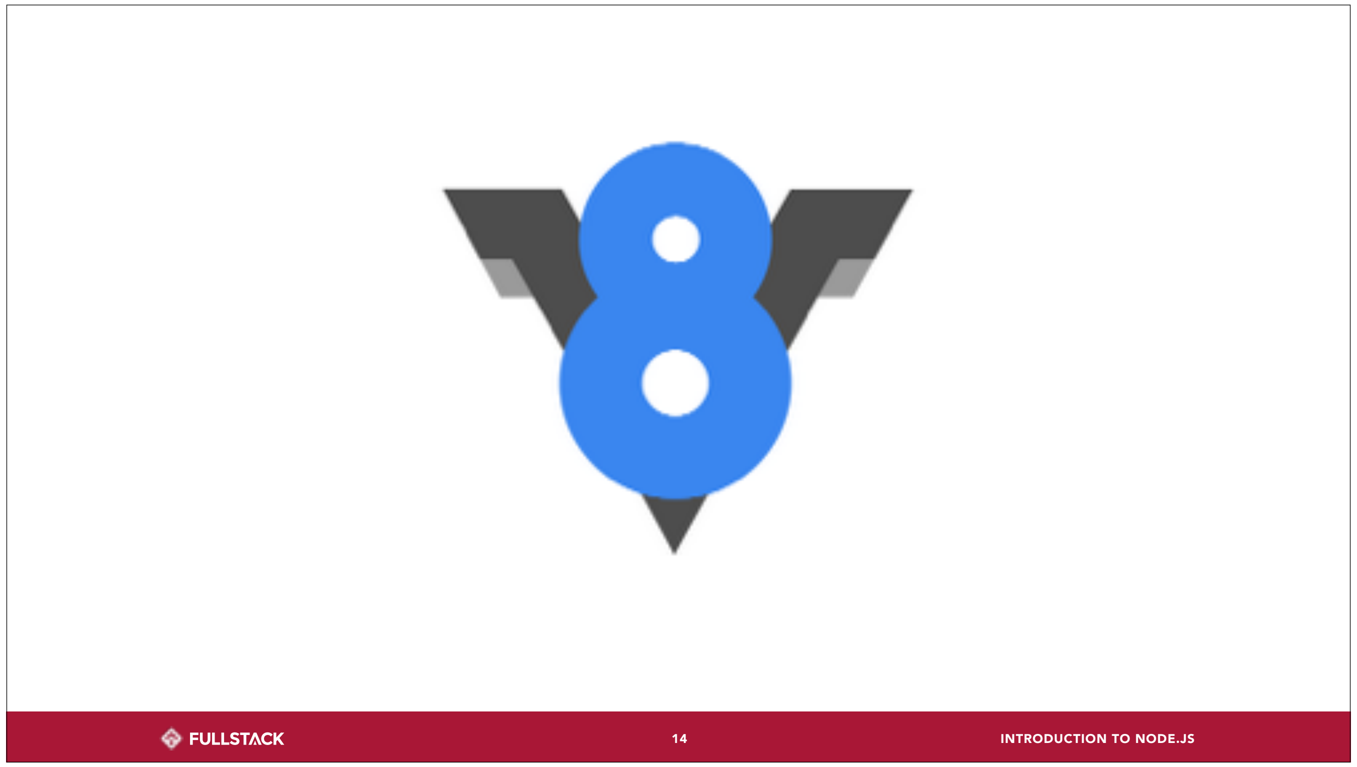
```
JOES = 9$

JOE
 JOEY(3)$
JOSEPH
 JOEY(JOES)$
```

When a new programming language is invented, like I just invented JoeScript, syntax is just only half the work. There needs to be a way to take this text, this “code”, and turn it into machine instructions.

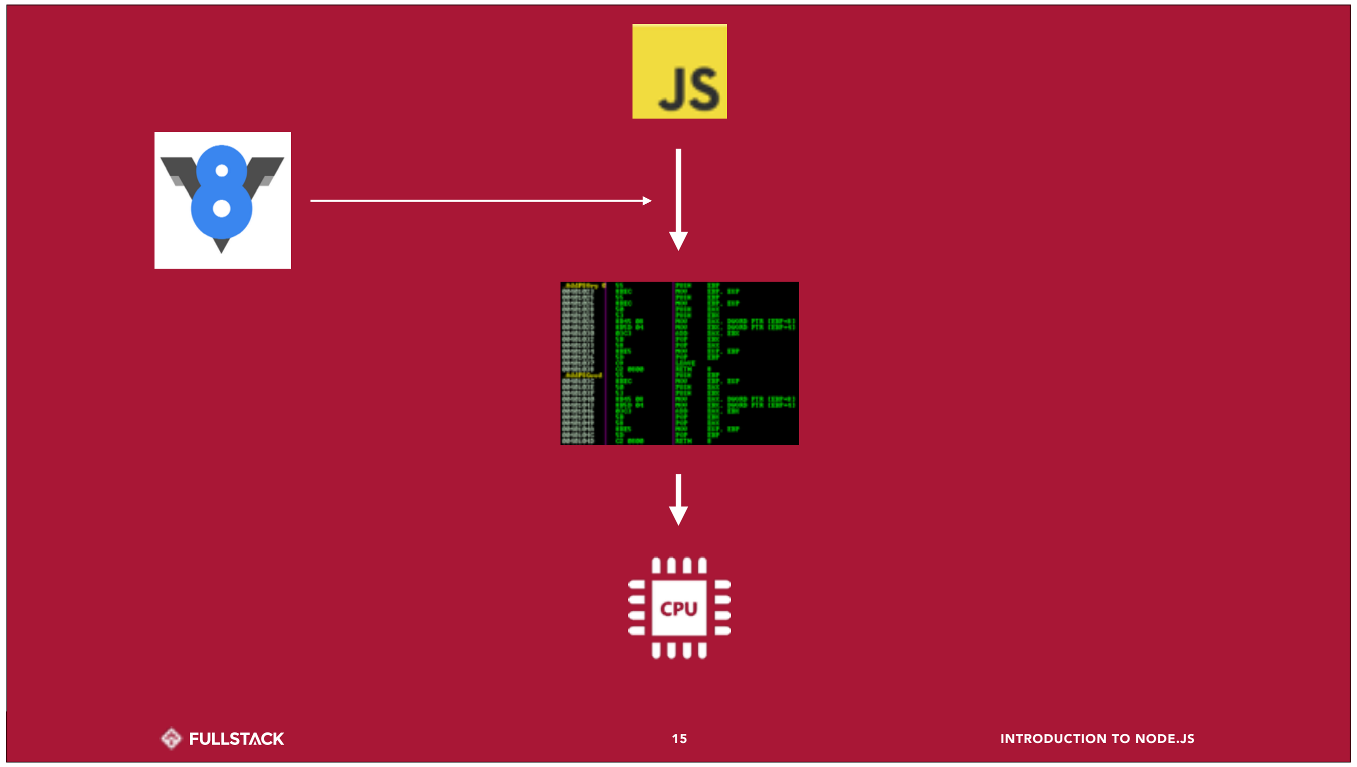
So, how do you run JoeScript? Well, you can’t. I haven’t written anything that will translate JoeScript code, this text, into machine instructions for the CPU to actually have it *do* something. Sorry.



But we have written Javascript code, and that code has executed, which means that the CPU was given machine instructions by some means. But how? What is translating our Javascript code into machine instructions?



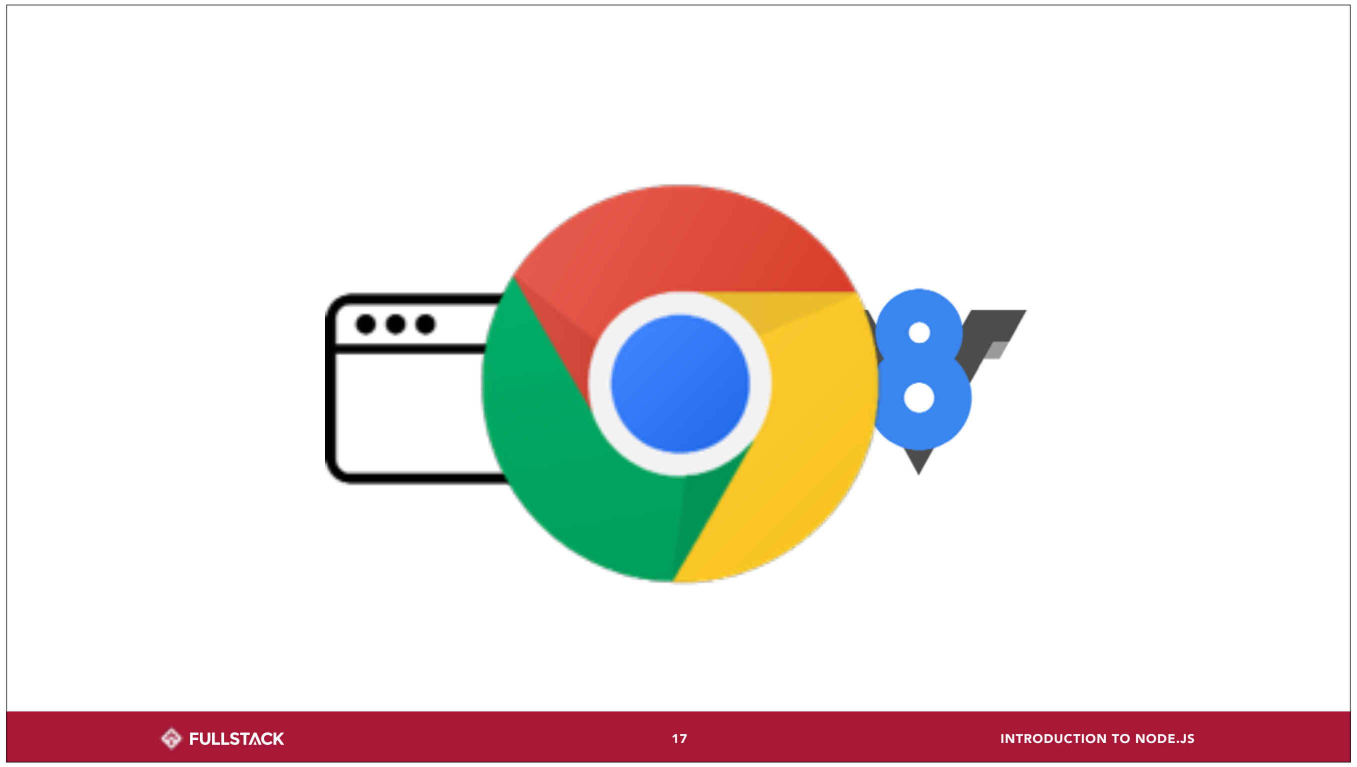
V8—yes, like the vegetable drink—is a piece of software considered a Javascript “engine” (thus V8). It is one of many engines, but talking about V8 is important. This “engine” parses Javascript code, translates that Javascript into machine instructions and then runs those instructions on your CPU. The engine itself is written in C++.



This is how our Javascript is being translated into machine code, and that makes our programs execute. V8 is a piece of software that accomplishes this.

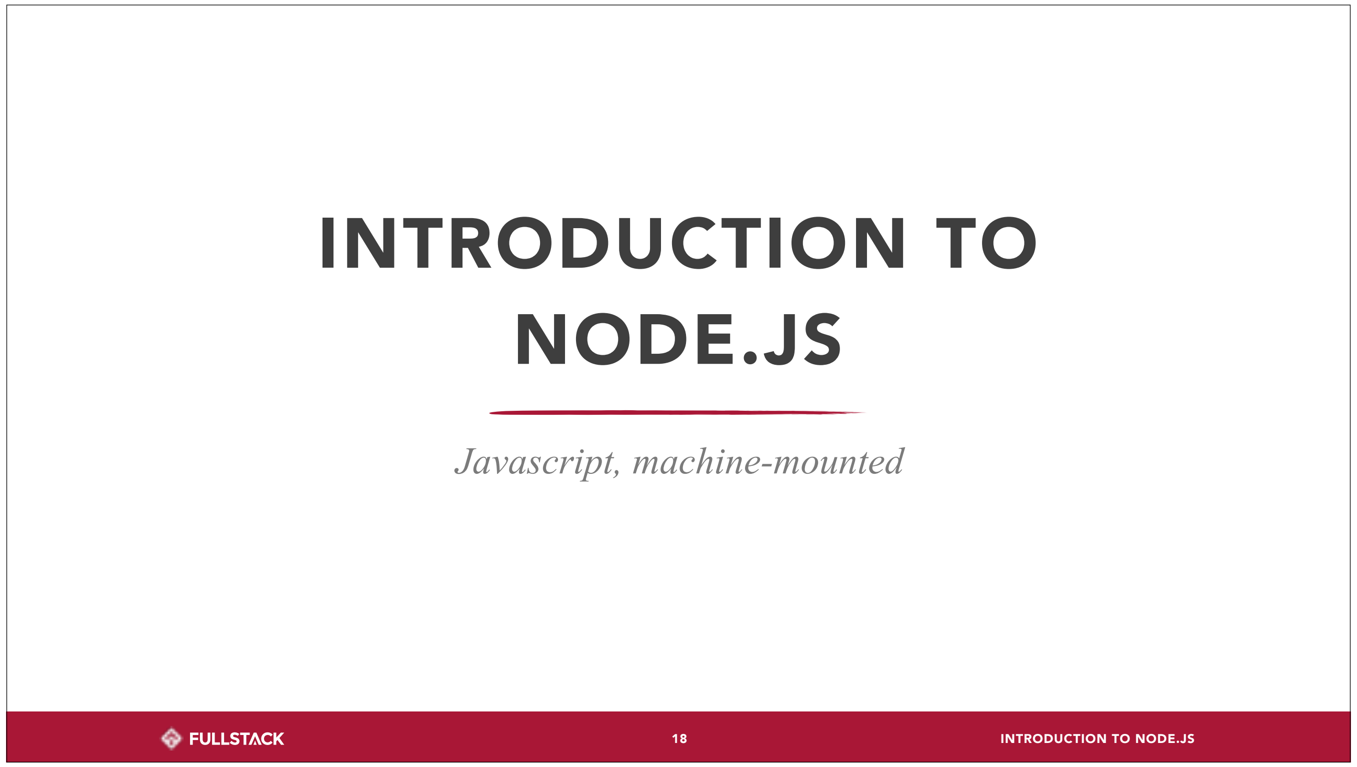
But where does V8 exist? . . . Maybe a better way to phrase our question, where can we run Javascript?

One place is in our browser, right? Chrome specifically is a piece of software that has a bunch of different components . . .



Including a way to view web pages, but also a Javascript interpreter that can take script files—Javascript code—and run operations based on that code.

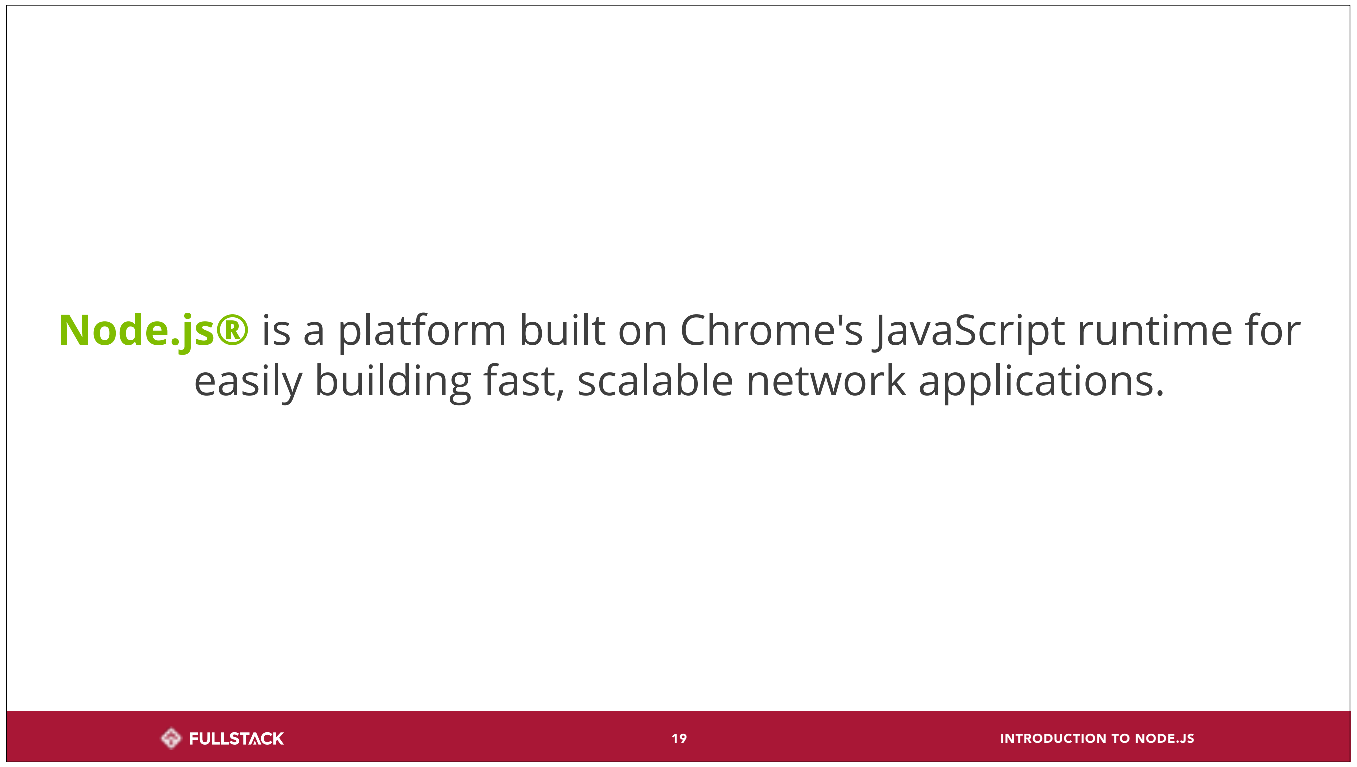
So, where else can we use V8; in other words, run Javascript code?



INTRODUCTION TO NODE.JS

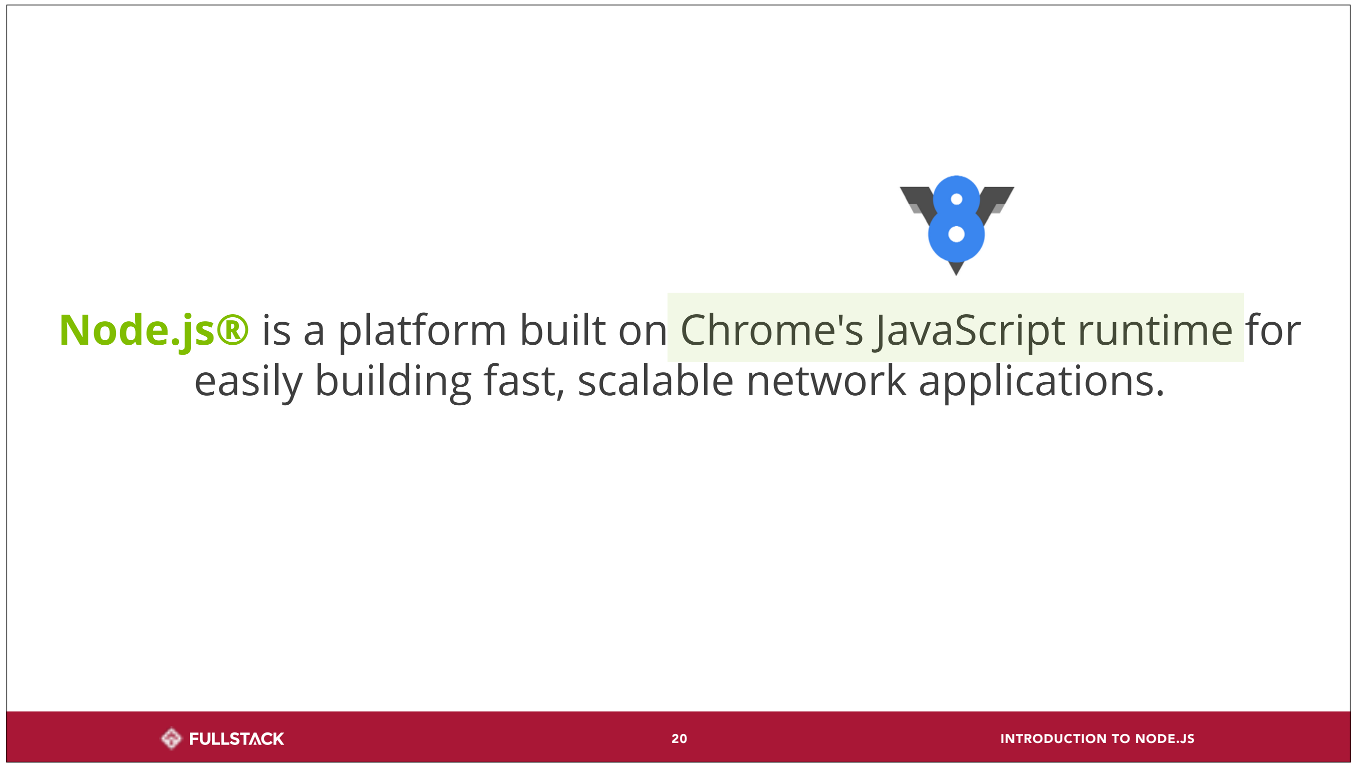
Javascript, machine-mounted

Well, back to slide 1. We are indeed learning about Node.js; I first had to make sure you were ready.



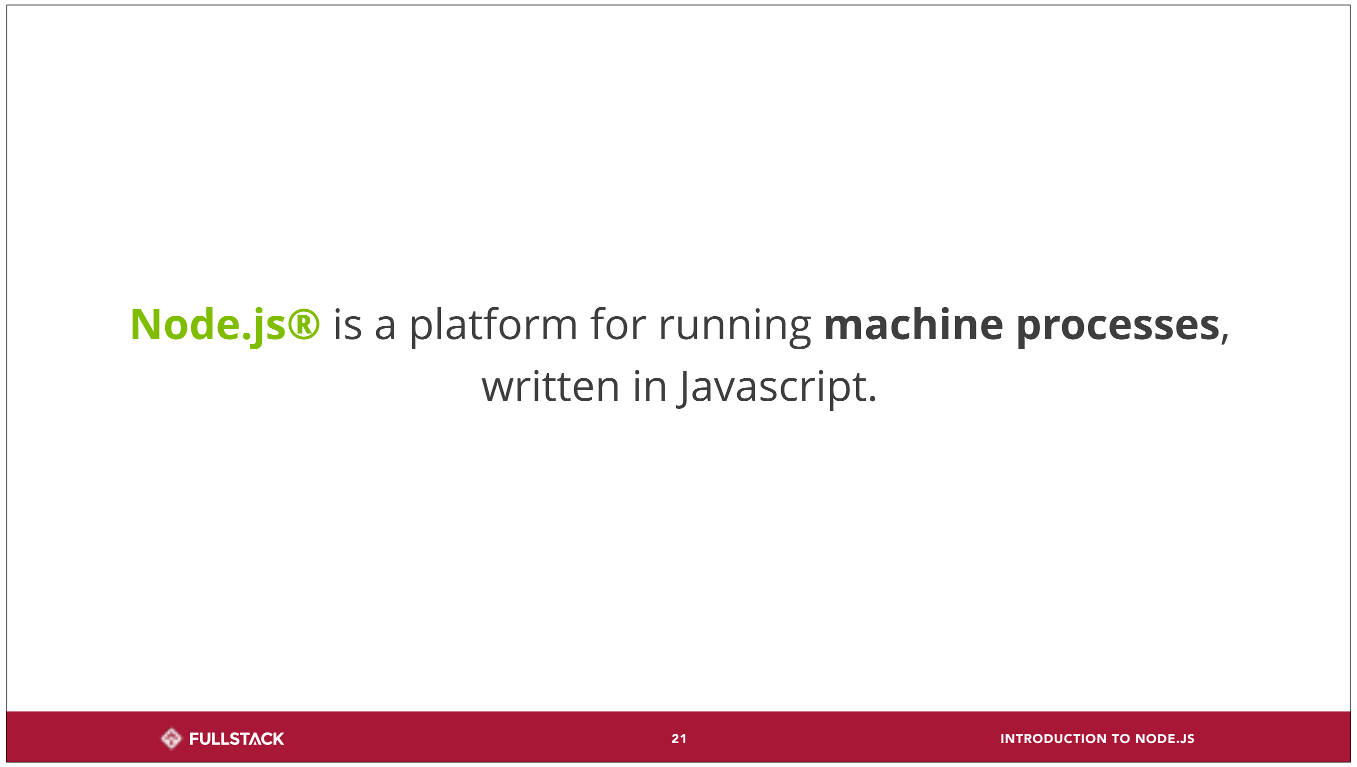
Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

This is straight from the node.js official site. What is meant by “Chrome’s Javascript runtime”? . . .

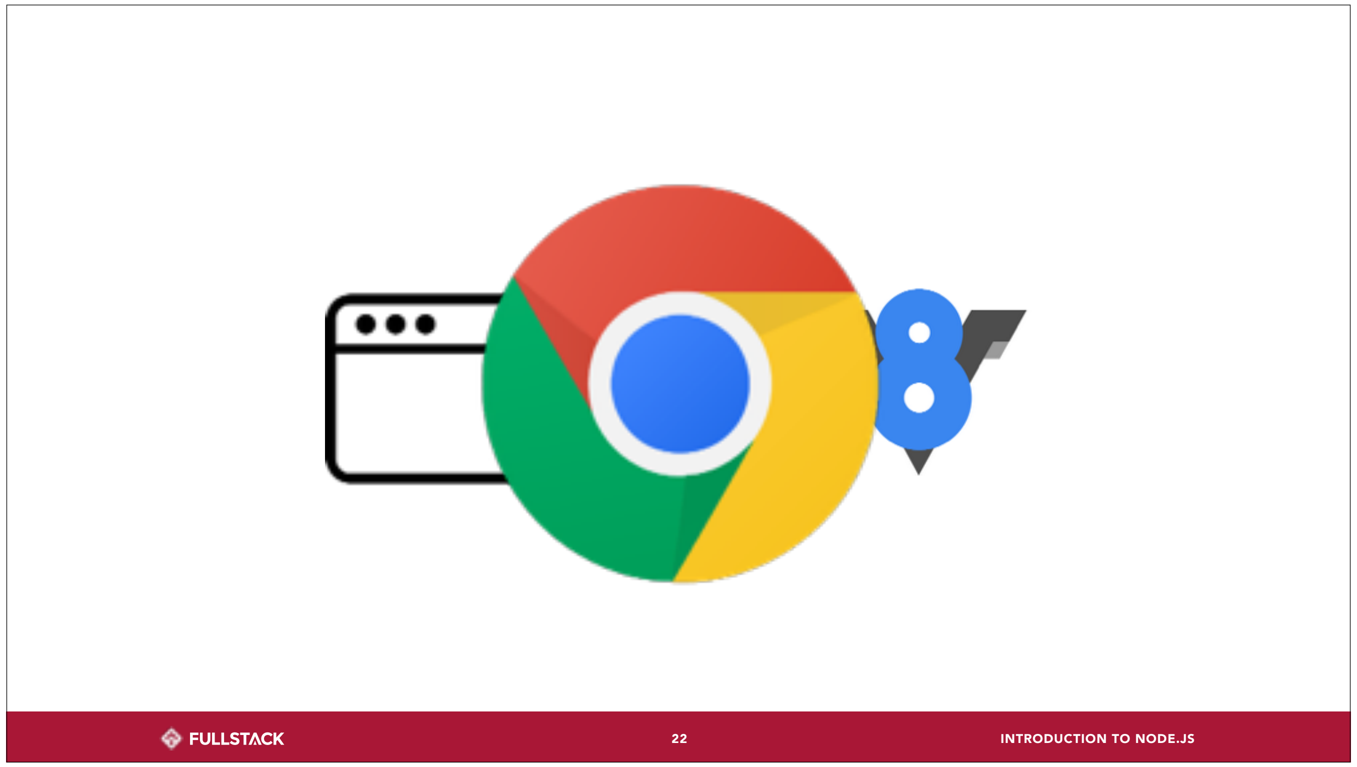


Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

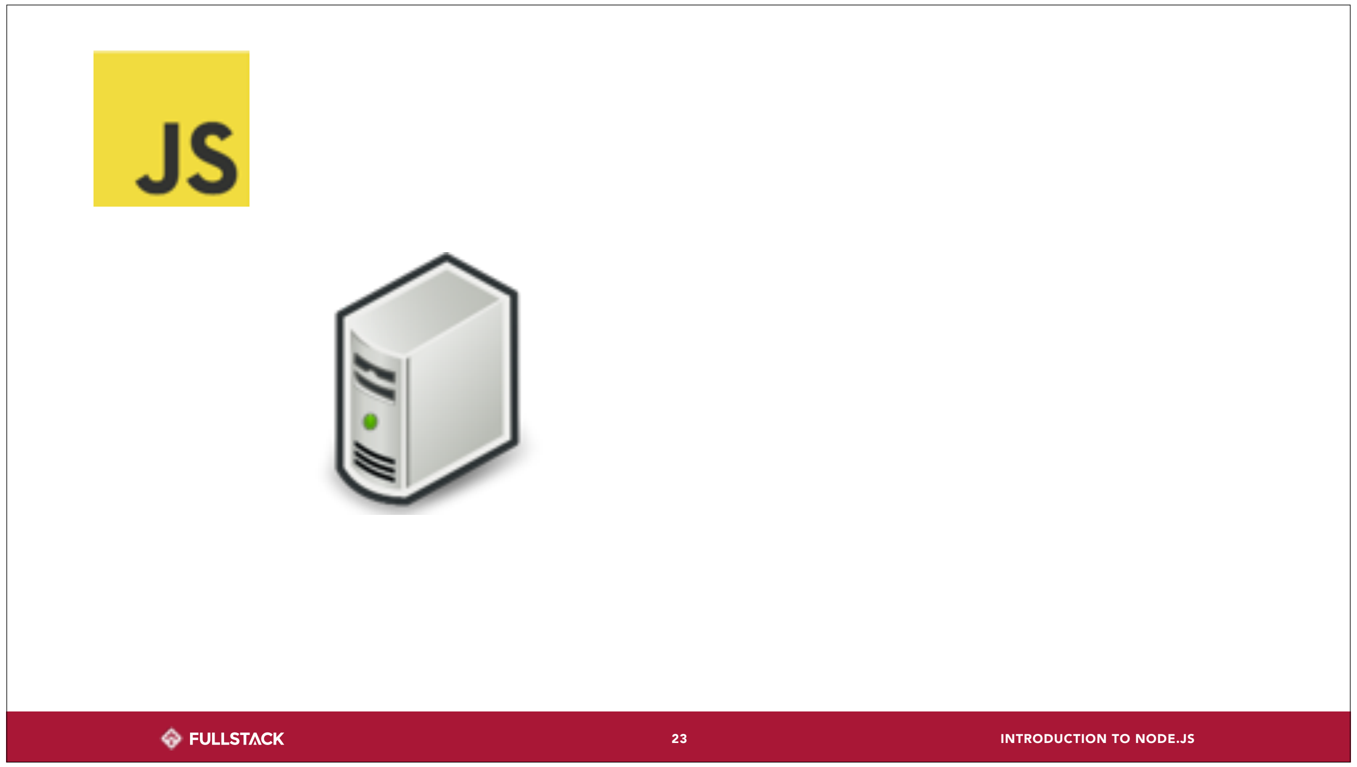
Right, it’s V8. I’m not a big fan of the official site’s description. How I like to describe node is:


Node.js® is a platform for running **machine processes**, written in Javascript.

we are running Javascript on a machine itself.

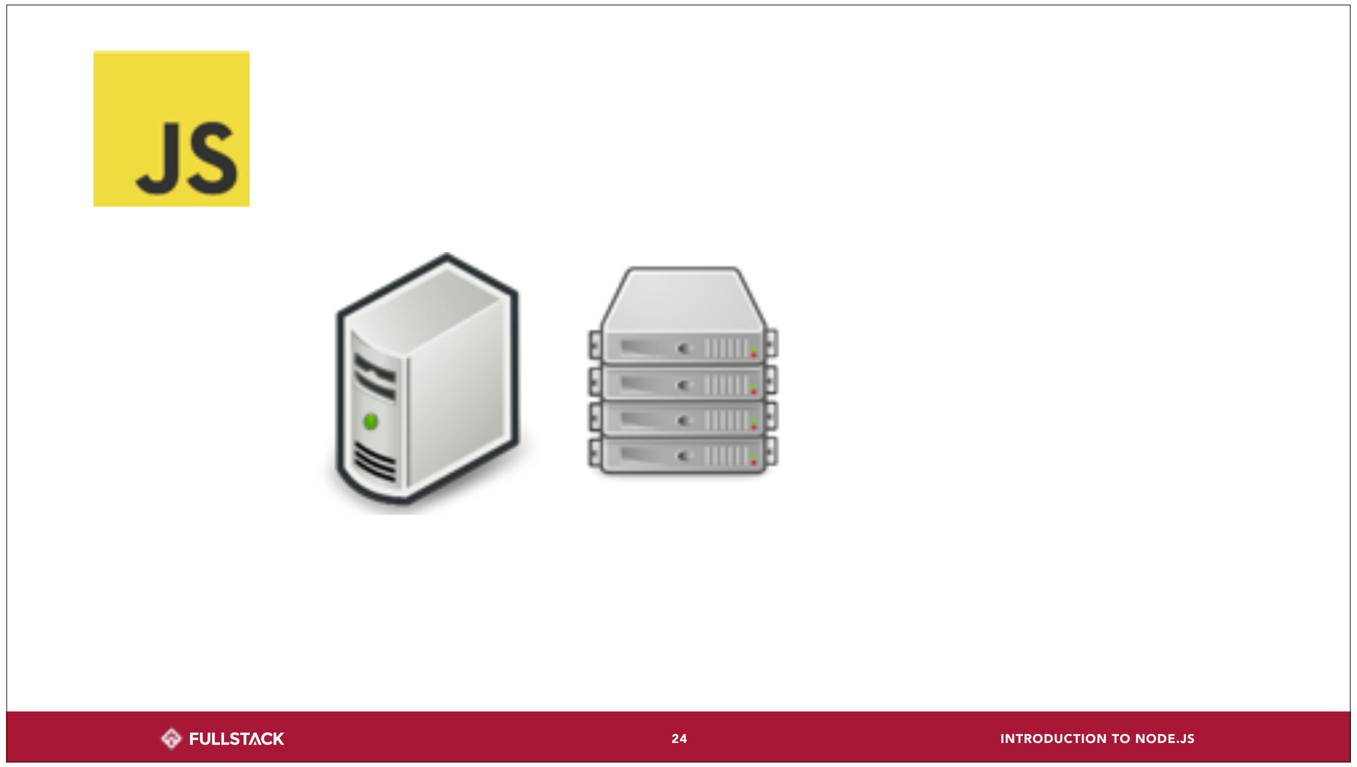


This means we’re not running Javascript to interact with browser functionality, like updating HTML or handling click events . . .

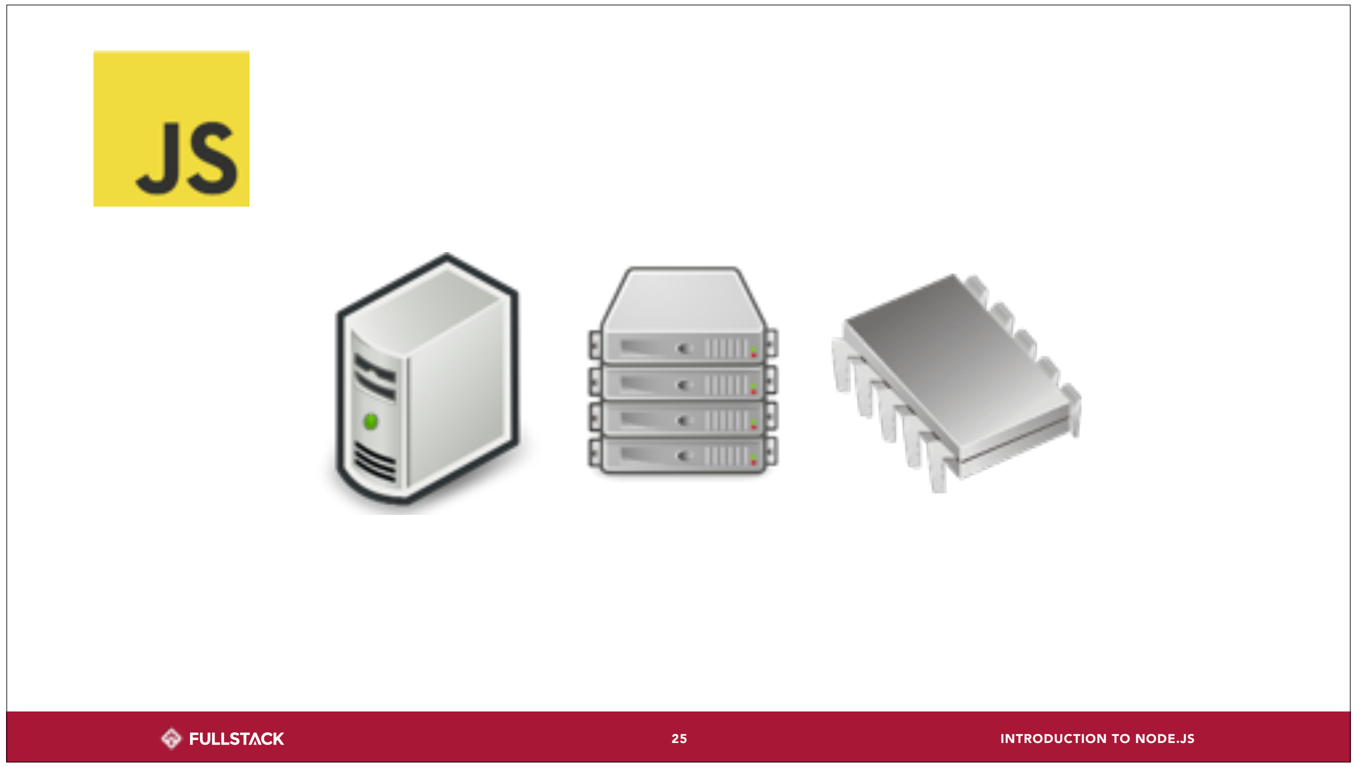


JS

We are writing programs in Javascript that run on the machine itself, not in a browser context. This means that we have non-restricted access, in our programs, to the files of that machine, processes on that machine. And when I say machine, I don’t just mean a computer. It could also be …



a remote server computer that will host your website …



A microcontroller that interacts with other hardware, like Arduino or Tessel machine

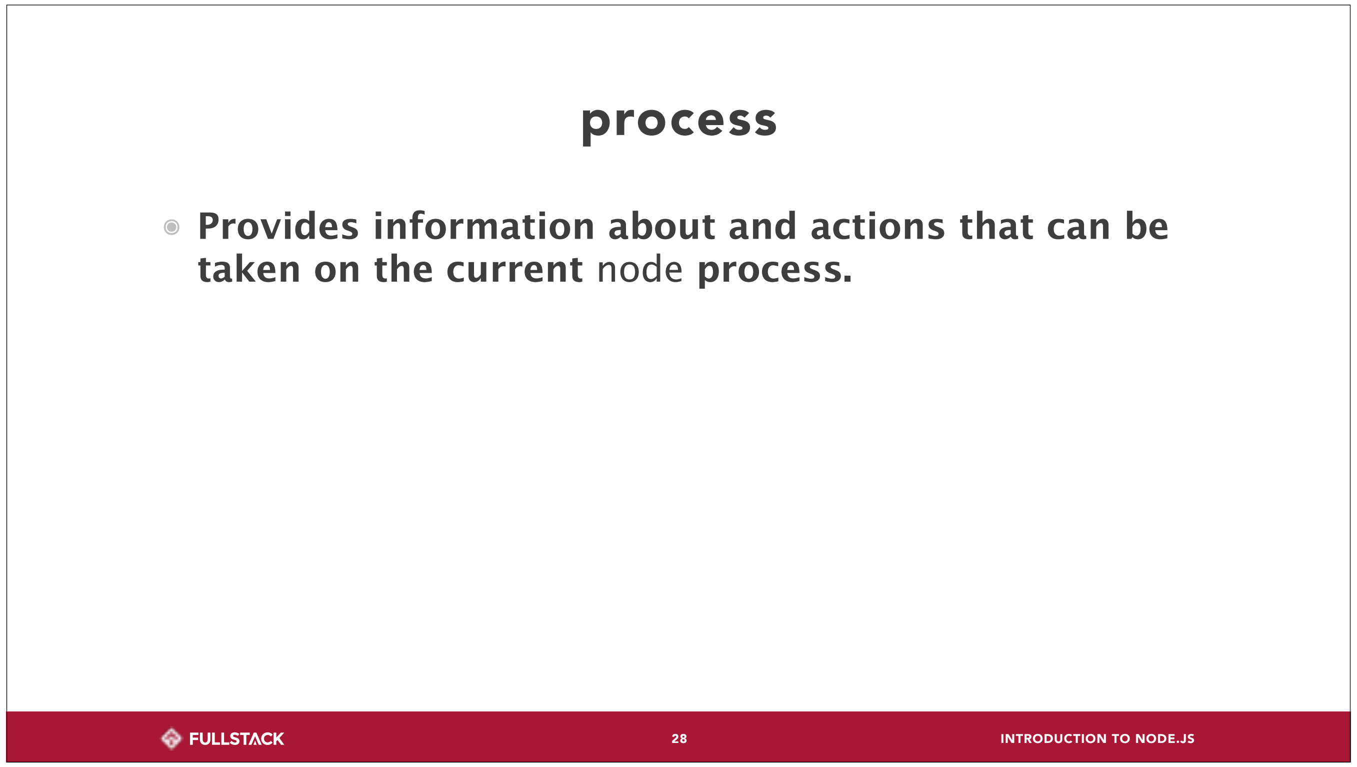
So that’s what node.js is. Let’s explore a bit.

(BASIC NODE EXAMPLES)

NODE.JS GLOBALS

FULLSTACK

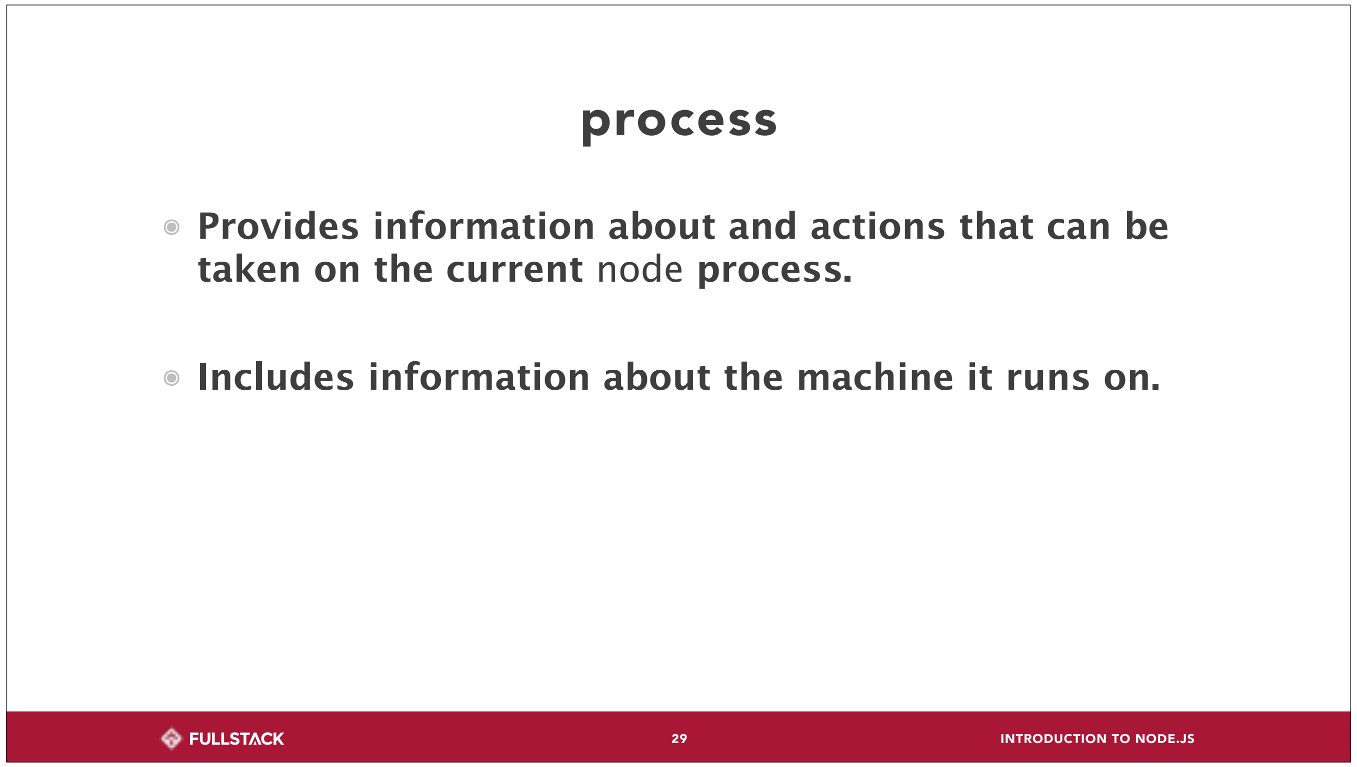
When you write Javascript programs in node, there are some built-in global variables that you can access anywhere in your program. We'll talk about a few of these. The first one is very interesting:


process

- **Provides information about and actions that can be taken on the current** node **process.**

FULLSTACK

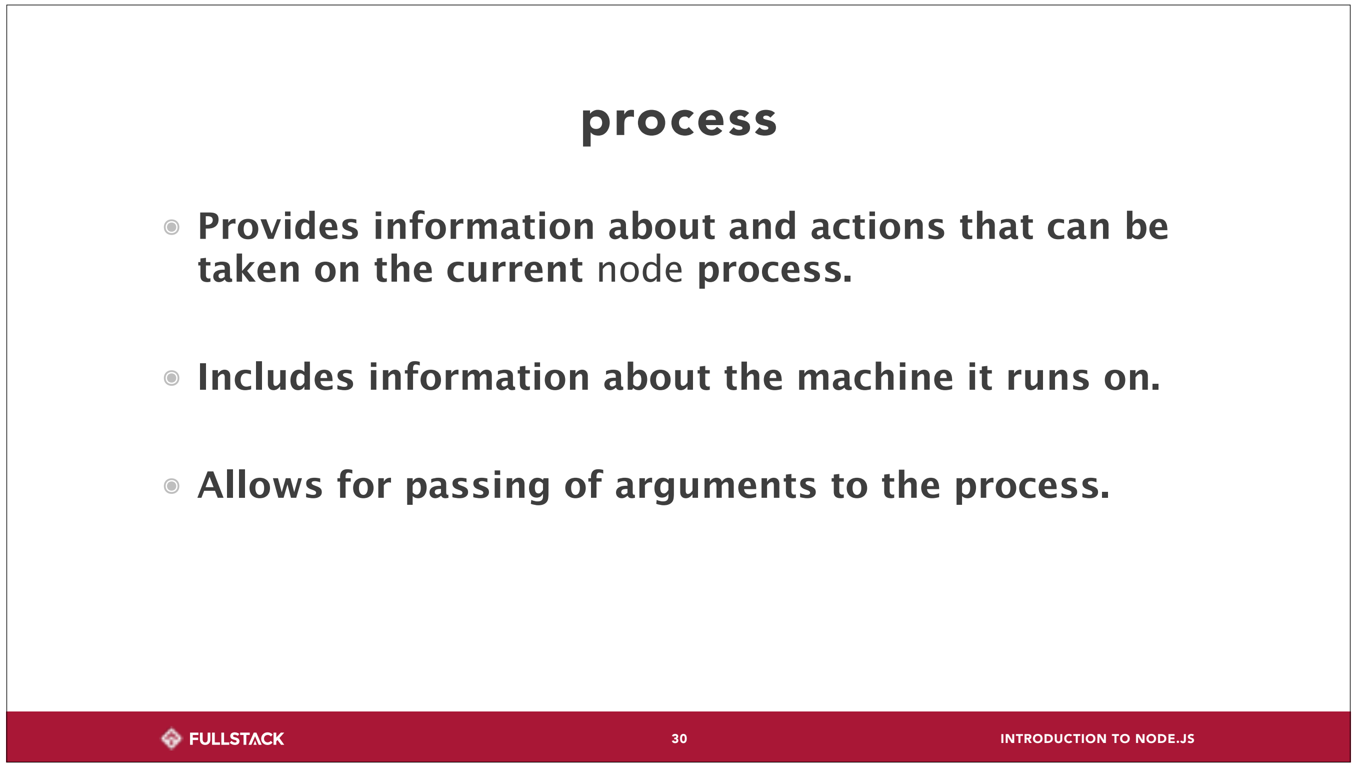
The process object is available in any of your node programs and provides interesting properties about the process itself and some methods for you to control the process.



process

- Provides information about and actions that can be taken on the current node process.
- Includes information about the machine it runs on.

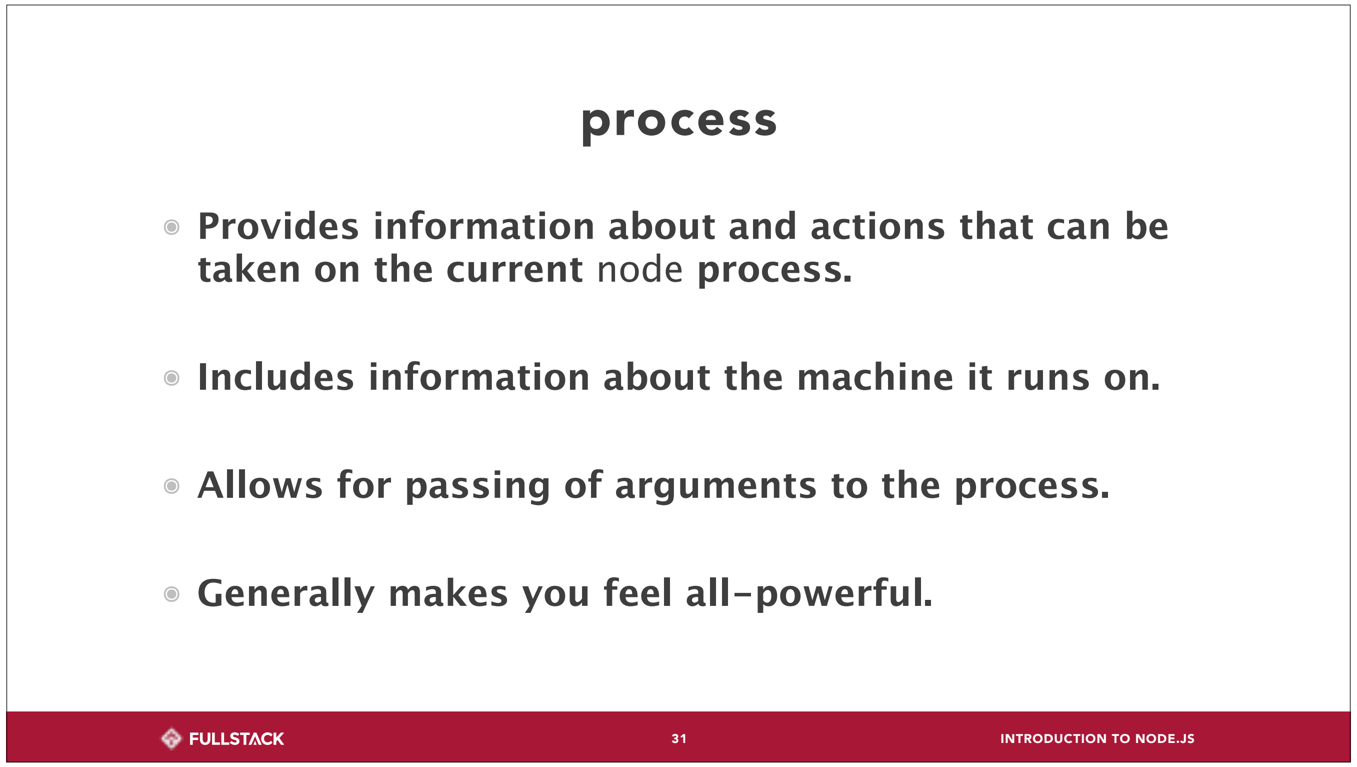
Some of these properties describe the machine you’re running on as well. Things like environment variables, OS architecture, the OS platform.



process

- Provides information about and actions that can be taken on the current node process.
- Includes information about the machine it runs on.
- Allows for passing of arguments to the process.

We begin our node programs, or processes, from the terminal, so we can use the process object to receive arguments when our programs are started. We'll see this in a minute.


process

- Provides information about and actions that can be taken on the current node process.
- Includes information about the machine it runs on.
- Allows for passing of arguments to the process.
- Generally makes you feel all-powerful.

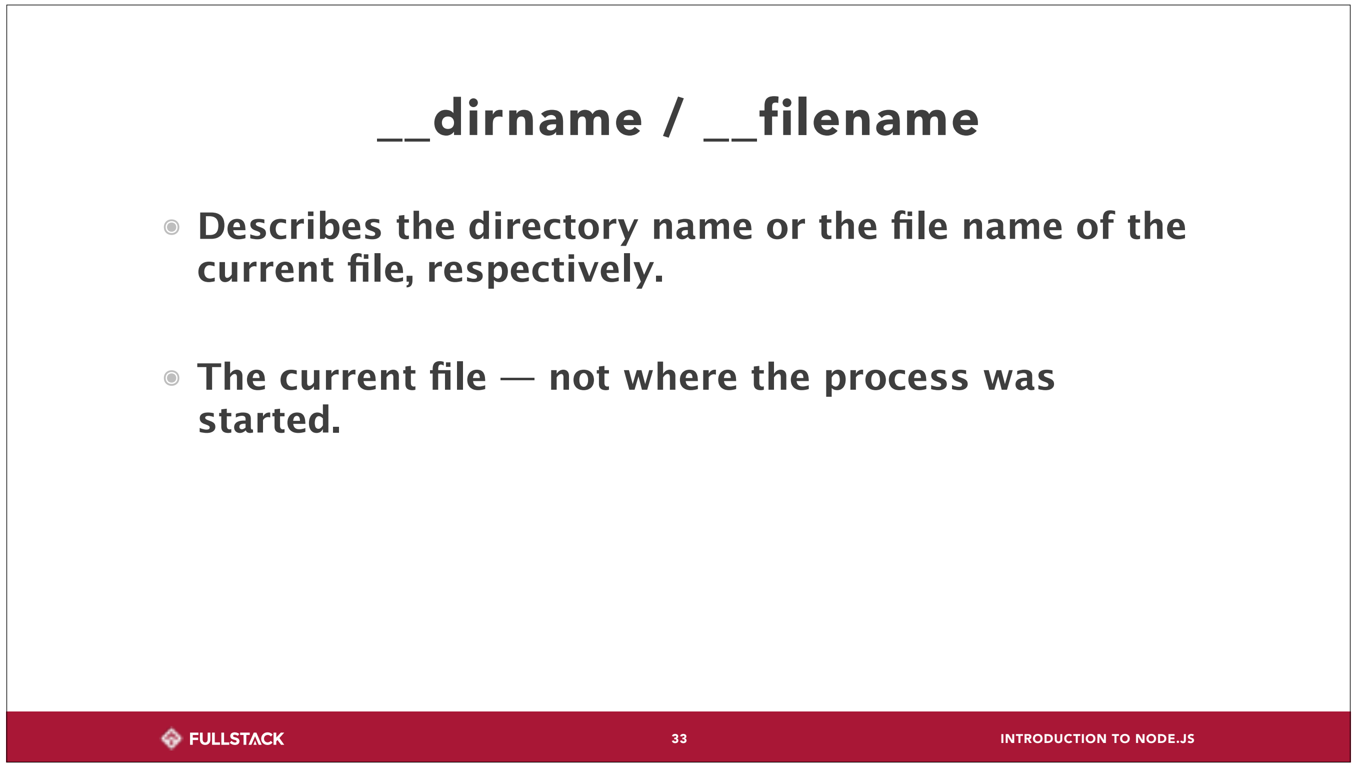
Overall, the process object makes you feel open to lots of possibilities in your programs because you have fine-grain access to the machine and information about the machine you're running on. Let's go through a few examples:

[PROCESS EXAMPLES]

__dirname / __filename

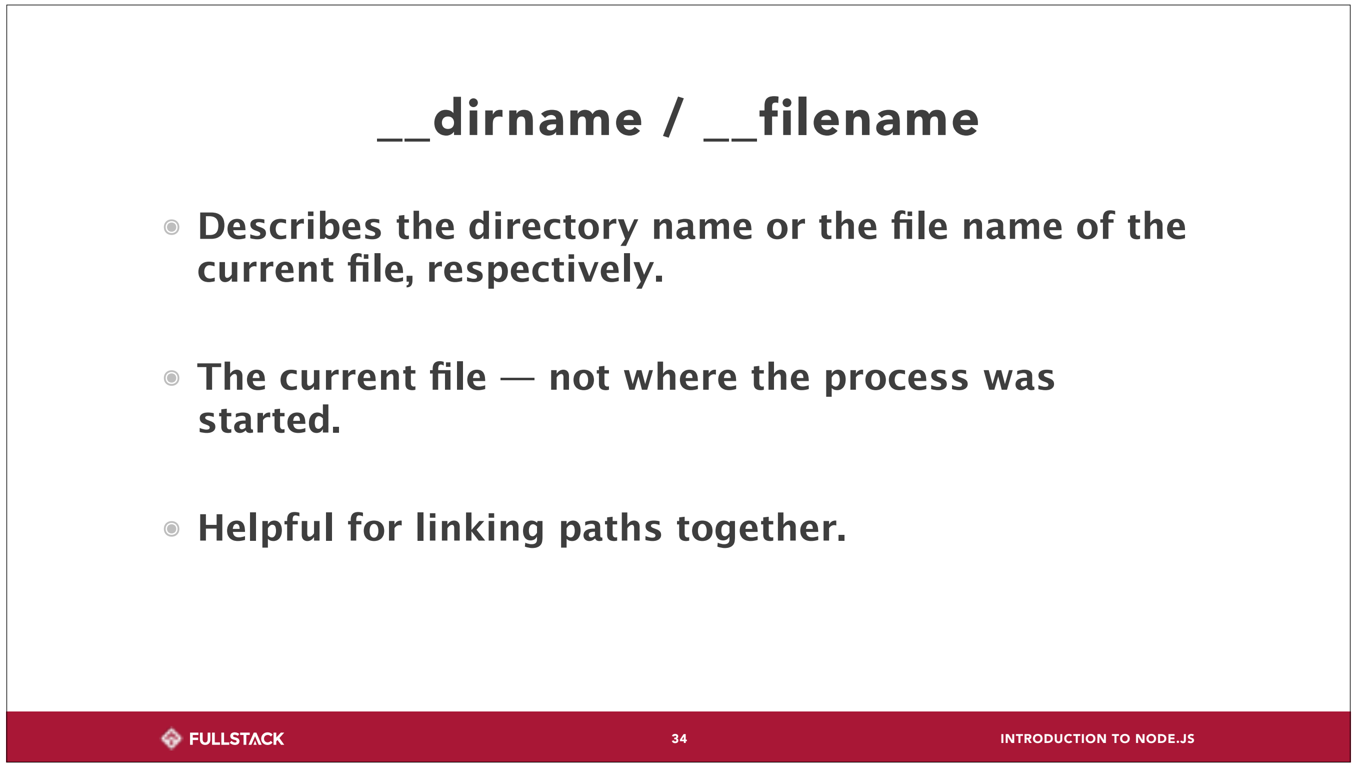
- Describes the directory name or the file name of the current file, respectively.

Another pair of globals are double-underscore dirname and double-underscore filename. These globals are strings that represent, when they are used, the name or directory name of the current file.


__dirname / __filename

- Describes the directory name or the file name of the current file, respectively.
- The current file — not where the process was started.

Note, I said current file, not where the process started. Wherever the code that uses these variables is written — that’s the file we’re talking about.

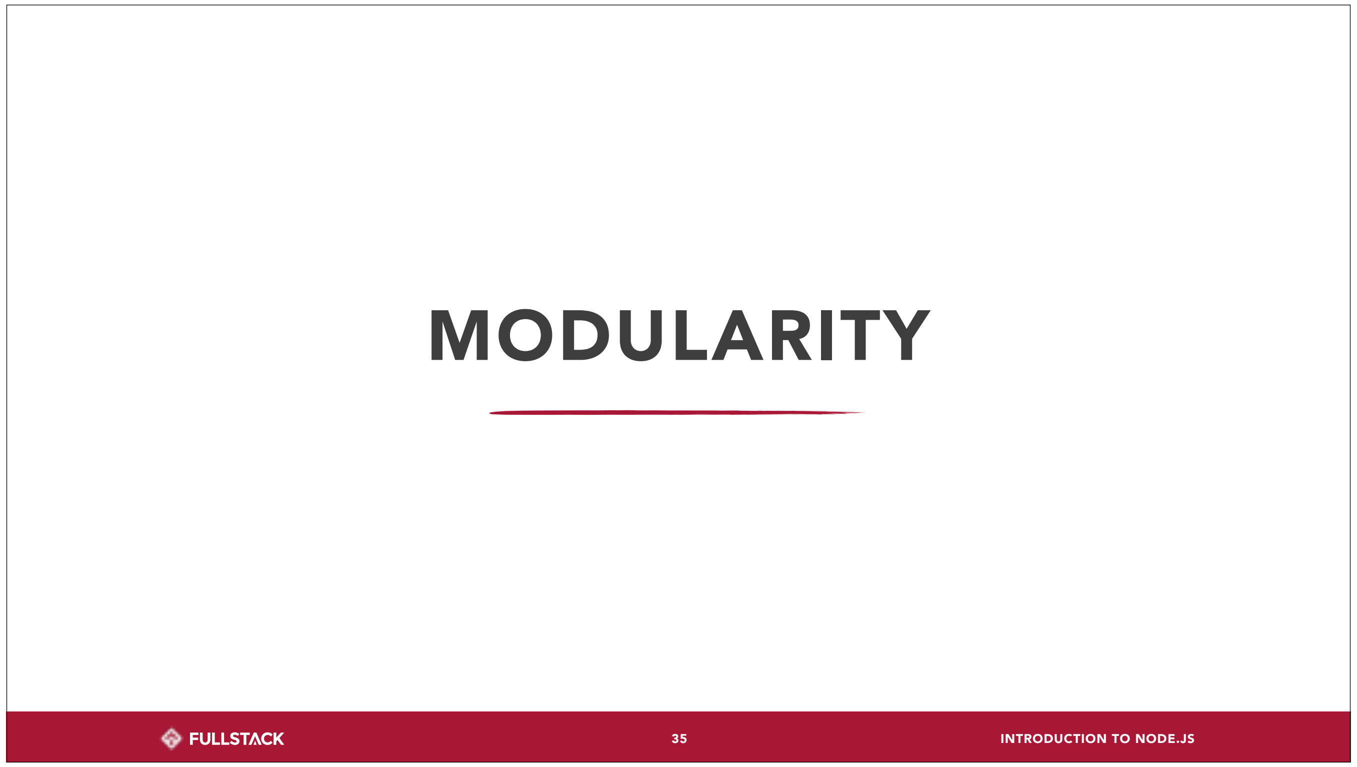


__dirname / __filename

- Describes the directory name or the file name of the current file, respectively.
- The current file — not where the process was started.
- Helpful for linking paths together.

These variables are helpful for linking paths together. Let’s look at a few examples:

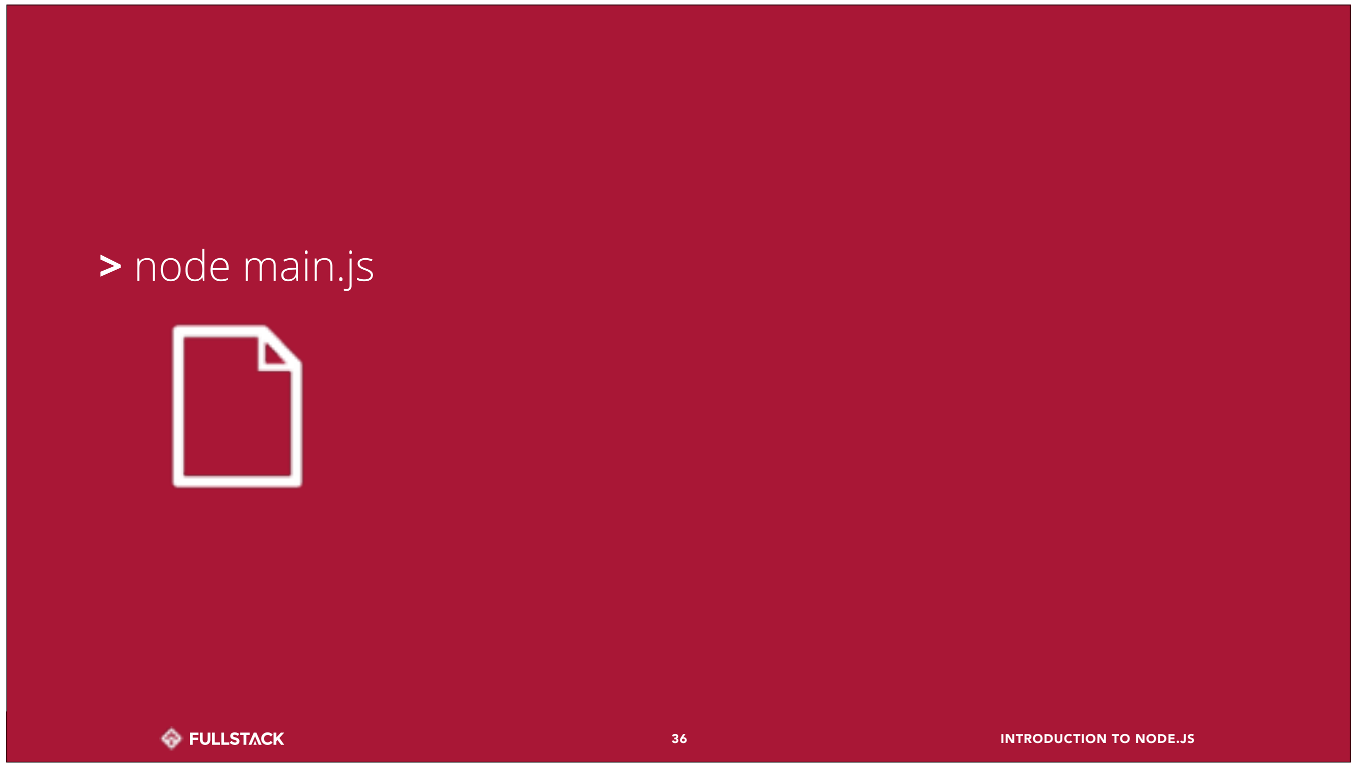
[DIRNAME/FILENAME EXAMPLES]



MODULARITY

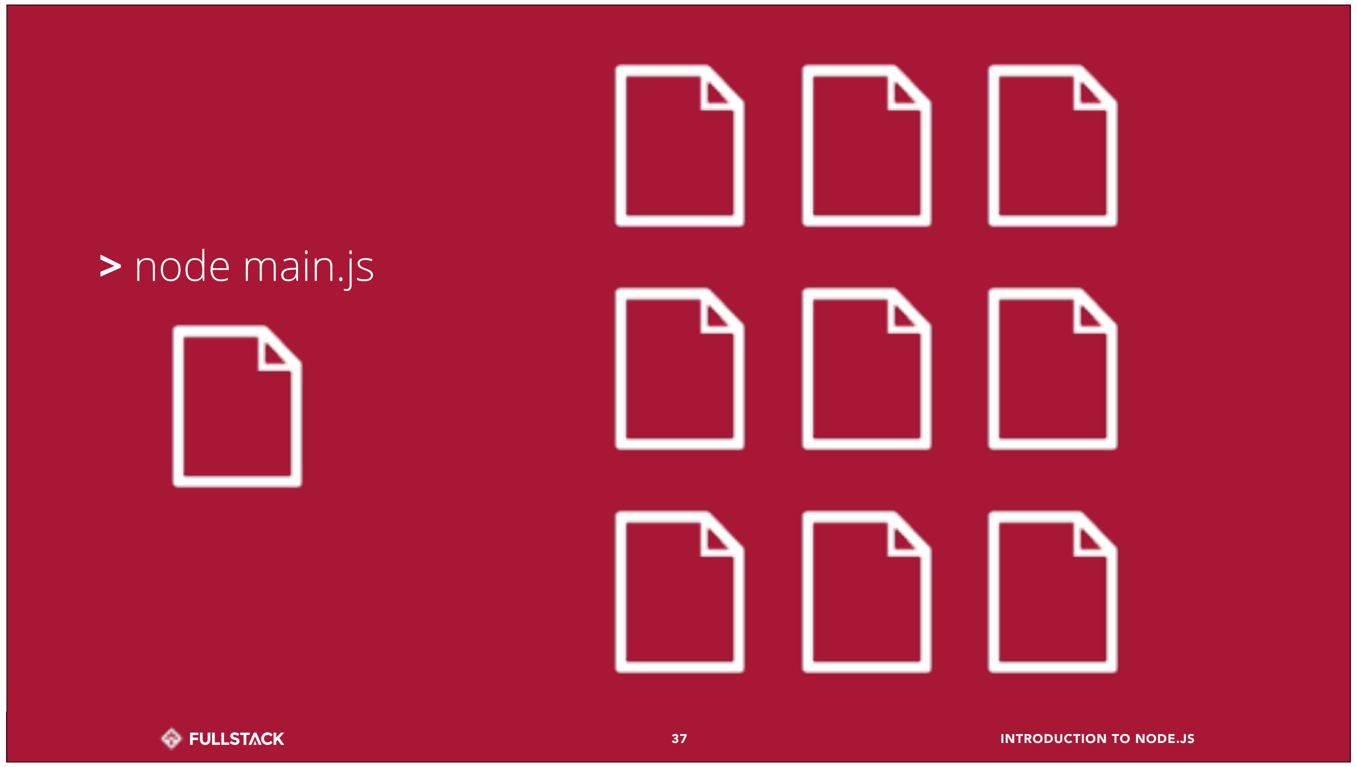
Next, we’ll talk about modularity with node.js. What is modularity and why do we care to make code “modular”? . . .

Modularity is an extremely important characteristic of application code, especially at a large size. We want to create modules in order to reuse code, separate the concerns of our code and organize our codebase.

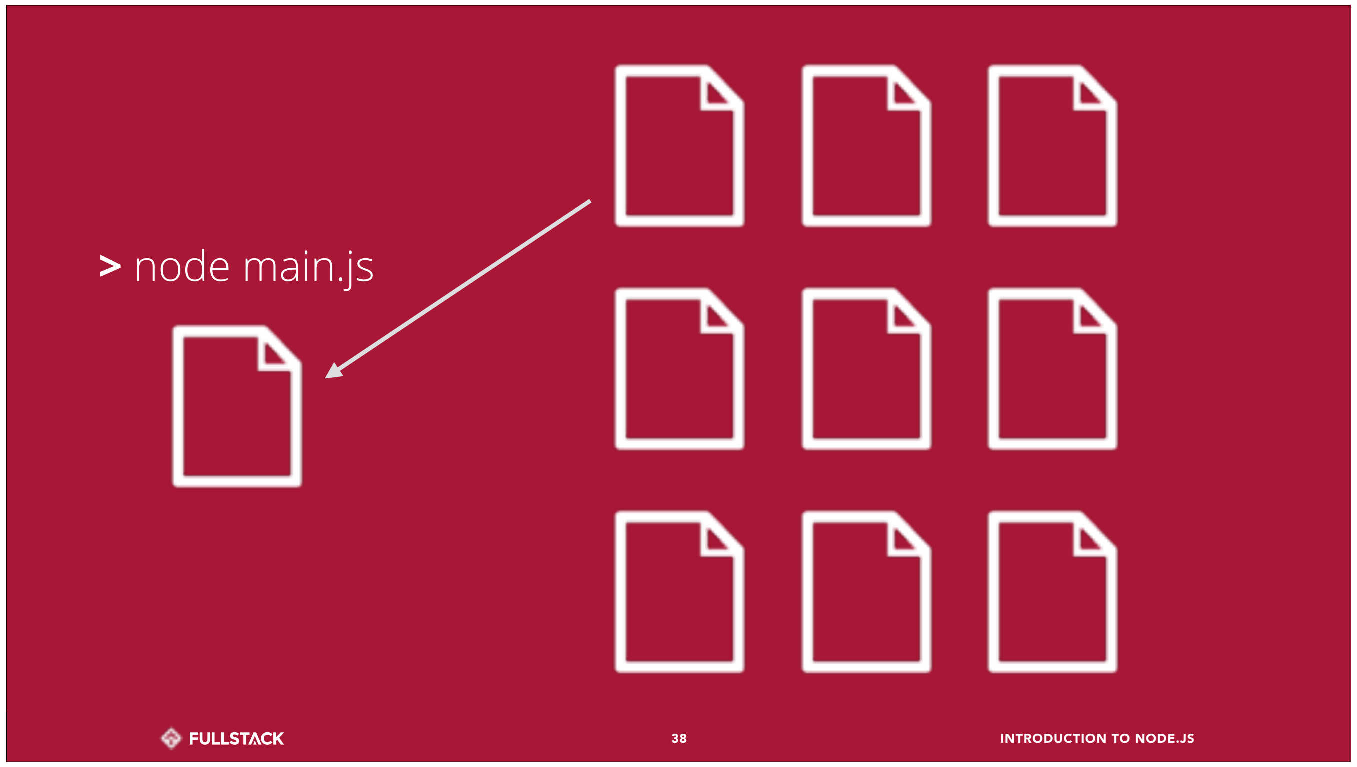


```
> node main.js
```

Every node process starts with exactly one file. We execute the node program by calling the command on this file.

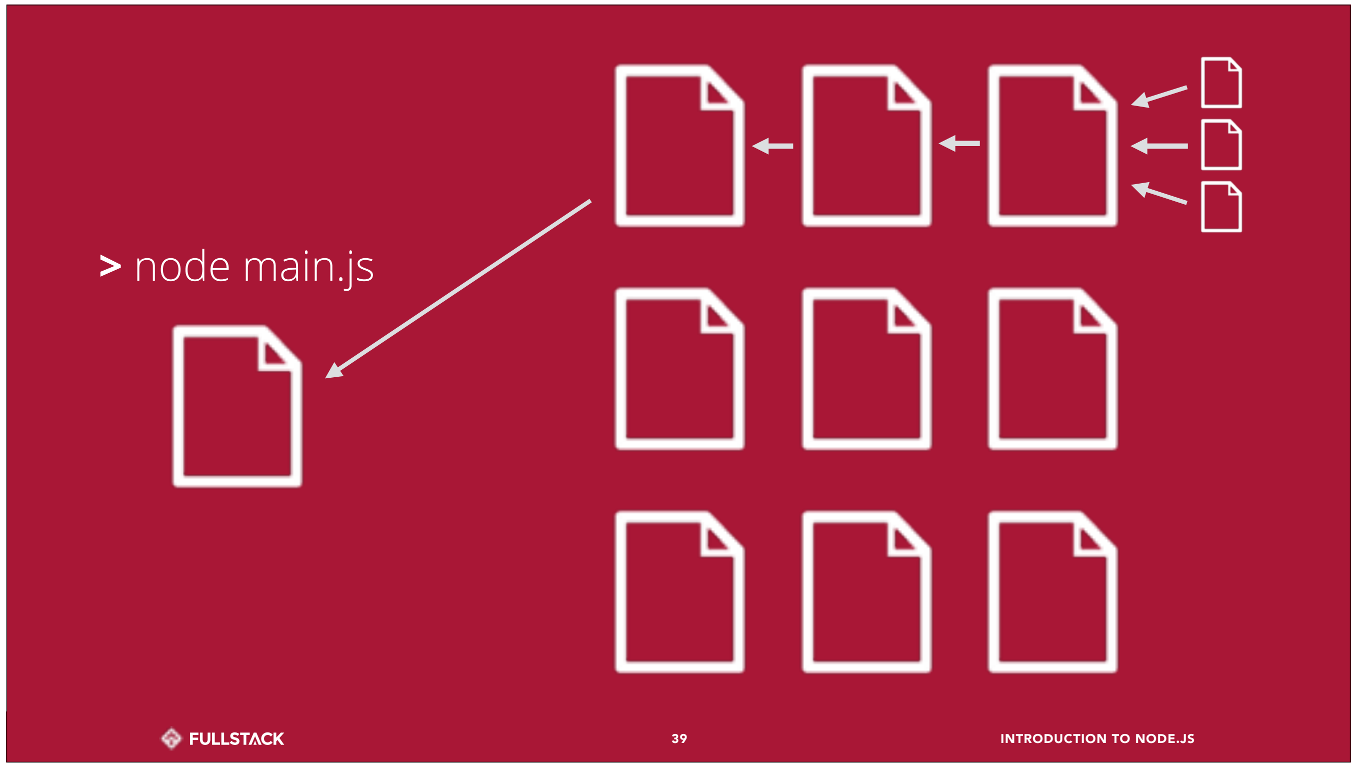

> node main.js

But, as our programs get bigger, we begin to start putting our code and logic into smaller compartments — other files. We still start our process at main-js . . .



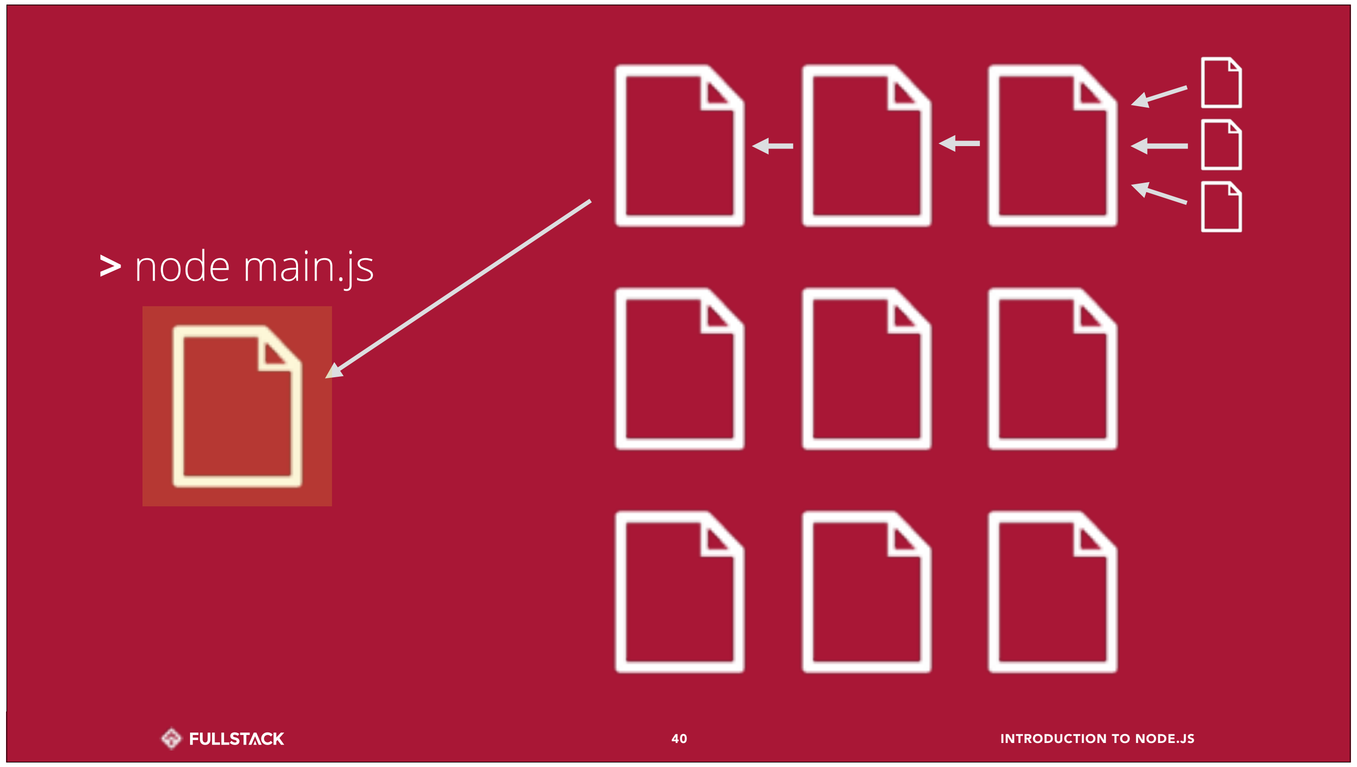
```
> node main.js
```

But then we use some kind of mechanism to bring the code, or the evaluation of the code, from another file into our main file.



```
> node main.js
```

And the evaluation of that code may have been bringing it in from another file, from another file, from another bunch of files.



> node main.js

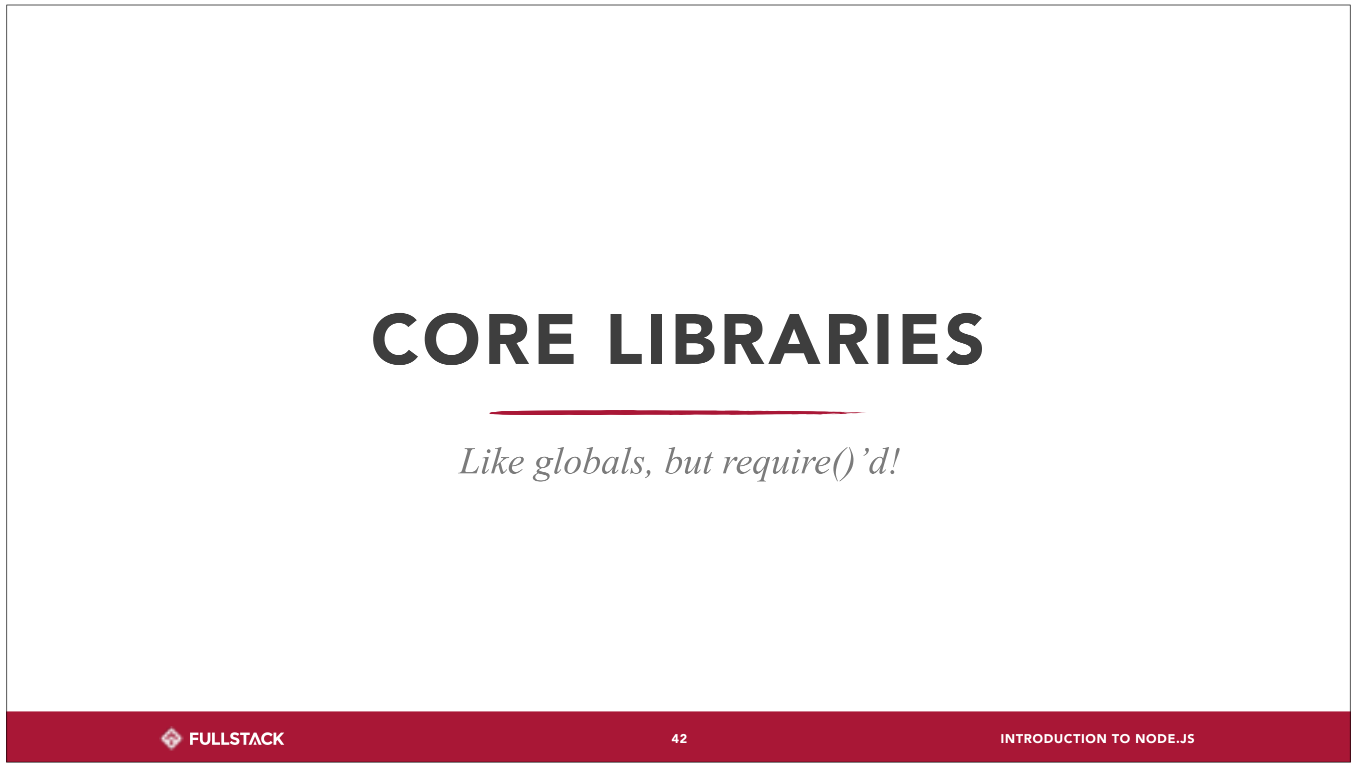
But our node processes always start from this one file and then bubbles out through this modularizing mechanism.

module and require

Modularizing our codebases
... tomorrow, the world.

This mechanism is broken into two halves: the module global and the require global.

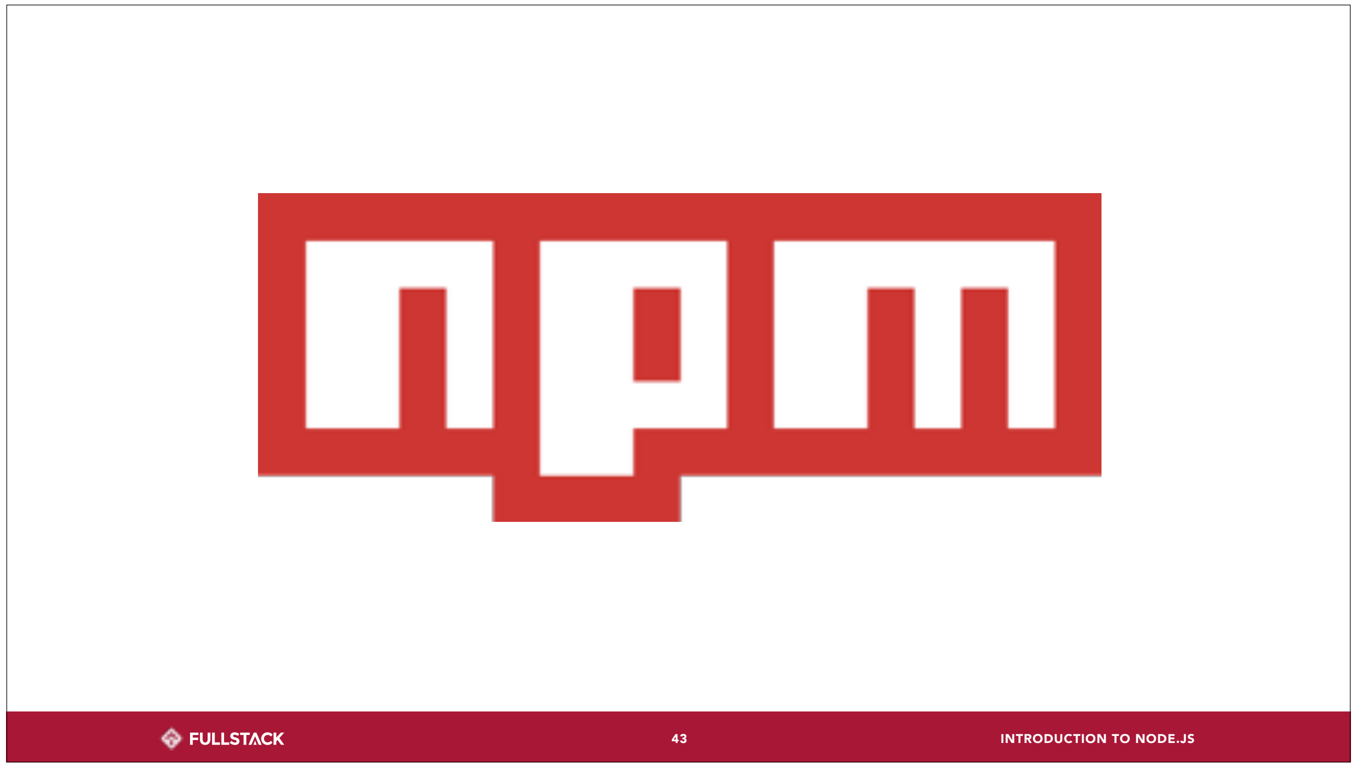
[COMMONJS EXAMPLES]



CORE LIBRARIES

Like globals, but require()'d!

[CORE LIBRARIES EXAMPLES]



[USING NPM]