**Introduction**

For a long time, the browser was the only place JavaScript code could be executed. Web developers had to use a different programming language on the front-end than the back-end. It also meant that, even as JavaScript evolved into a more robust and powerful language, it remained a front-end only language.

Though multiple attempts to create off-browser JavaScript environments have been attempted, [Node.js](https://nodejs.org/en/), invented by Ryan Dahl in 2009, found unprecedented popularity and is currently being used by numerous top-tier companies including Netflix, Uber, Paypal, and eBay. Node.js is a JavaScript *runtime*, or an environment that allows us to execute JavaScript code outside of the browser. A “runtime” converts code written in a *high-level*, human-readable, programming language and compiles it down to code the computer can execute.

Though Node was created with the goal of building web servers and web applications in JavaScript, it can also be used for creating command-line applications or desktop applications. In this lesson, we’ll explore some features of Node so you start to feel comfortable with running JavaScript in the Node environment and gain some familiarity with features unique to Node. For more advanced development, Node can be combined with any number of robust frameworks like the [Express.js framework](https://expressjs.com/) for creating effective web application back-ends.

There’s more to learn about Node than we could ever fit in one lesson. We’ll try to point to great resources like [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript) and the [Node.js documentation](https://nodejs.org/api/). Take your time exploring and use the documentation.

**Instructions**

**1.**

In this lesson, we’ll be providing you a terminal with Node.js already installed. If you’d like to download Node on your local machine and follow along, check out [this article](https://www.codecademy.com/content-items/c4fe3060dbc61fc82d810c4ea06c29a8).

Let’s see what version of Node we have installed. Type node -v in the terminal and then press and hit enter (or return).

**The Node REPL**

[REPL](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop) is an abbreviation for **r**ead–**e**val–**p**rint **l**oop. It’s a program that **l**oops, or repeatedly cycles, through three different states: a **r**ead state where the program **r**eads input from a user, the **e**val state where the program **e**valuates the user’s input, and the **p**rint state where the program **p**rints out its evaluation to a console. Then it **l**oops through these states again.

When you install Node, it comes with a built-in JavaScript REPL. You can access the REPL by typing the command node (with nothing after it) into the terminal and hitting enter. A > character will show up in the terminal, indicating the REPL is running and prompting your input. The Node REPL will evaluate your input line by line.

By default, you indicate the input is ready for eval when you hit enter. If you’d like to type multiple lines and then have them evaluated at once, you can type .editor while in the REPL. Once in “editor” mode, you can type control + d when you’re ready for the input to be evaluated. Each session of the REPL has a single shared memory; you can access any variables or functions you define until you exit the REPL.

A REPL can be extremely useful for performing calculations, learning a language, and developing code. It’s a place where you can explore language features and try things out while receiving immediate feedback. Figuring out how to do this outside of the browser or a website can be really empowering.

The Node environment contains a number of Node-specific global elements in addition to those [built into the JavaScript language](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects). Every Node-specific global property sits inside the [the Node global object](https://nodejs.org/api/globals.html" \t "_blank). This object contains a number of useful properties and methods that are available anywhere in the Node environment.

Let’s try out the Node REPL. This will be a good way for you to explore the Node global object!

**Instructions**

**1.**

Let’s enter the Node REPL. Type node in the terminal and press enter.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Experiment on your own within the REPL to get a better sense of it. Here are some suggestions for things to try:

* Access the global object. You can console.log(global) or, since the REPL displays the return of each evaluated line, simply type global and then enter.
* Woah… it looks huge. A lot of that is because of the global.process object. Check out an easier to read list of the properties on the global object with Object.keys(global).
* The global object has a lot of useful properties and methods, and it’s not common to add any to it. However, it is just an object, so we can! Add a property to the global object, eg. global.cat = 'meow!'.
* Now print or return the property you just added:

> console.log(global.cat)  
'meow!'

* If you’re familiar with running JavaScript on the browser, you’ve likely encountered the Window object. Here’s one major way that Node differs: try to access the Window object (this will throw an error). The Window object is the JavaScript object in the browser that holds the DOM, since we don’t have a DOM here, there’s no Window object.

You’ll learn more about the global object as you explore Node, but remember that, at its core, it’s just a JavaScript object!

**Running a Program with Node**

Node was designed with server-side web development in mind and has a lot of thoughtful functionality towards that end. At its most simple, however, it provides the ability to run JavaScript programs on our own computers instead of just in the browser’s console or embedded in HTML.

In this lesson, we’ll explore some of the functionality and properties specific to the Node environment, but first, let’s see how we run a program.

We’ll need to create a file with a .js extension. We’ll call ours **myProgram.js**. Next, we’ll open that file with a text editor and add our code:

// Inside myProgram.js  
console.log('Hello World');

Our code is complete! Now, we want to execute it. We’ll open our terminal and navigate to the directory that contains **myProgram.js**. Finally, we’ll type the command node myProgram.js into our terminal.

$ node myProgram.js

The results of our program will print to the terminal.

Hello World

Let’s write a program and run it in Node.

**Core Modules**

*Modularity* is a software design technique where one program has distinct parts, each providing a single piece of the overall functionality. These separate *modules* come together to build a cohesive whole. Modularity is essential for creating scalable programs which incorporate libraries and frameworks and separate the program’s concerns into manageable chunks. Essentially, a module is a collection of code located in a file. Instead of having an entire program located in a single file, code is organized into separate files based on the concerns they address. These files can then be included in other files by using the require() function.

To save developers from reinventing the wheel each time, Node.js has several built-in modules to perform common tasks efficiently. These are known as the *core modules*. The core modules are defined within Node.js’s source code and are located in the **lib/** folder. Core modules can be required by passing a string with the name of the module into the require() function:

// Require in the 'events' core module:  
const events = require('events');

The example above shows how the events module is required into a file and stored in an events variable. Understanding the specifics of this module isn’t necessary at this point, but the events module is a Node.js core module that provides key functions for working with, well… events. You’ll learn more about it in a later lesson.

Some core modules are actually used inside other core modules. For instance, the util module can be used in the console module to format messages. We’ll cover these two modules in this lesson, as well as two other commonly used core modules: process and os.

Node.js has several core modules, far too many to cover in this lesson. We’ll learn how to get the full list and then dive deeper into the aforementioned modules throughout the next few exercises.

**The Console Module**

One of the most commonly used Node.js core modules is the console module. In Node.js, the terminal is used to send and receive text feedback to and from a program, often for debugging purposes. This may sound familiar to how we use the console in the web browser. That’s because in Node.js, the built-in console module exports a global console object that gives the terminal similar functionality. The console object provides many of the same familiar methods such as:

* .log() — to print messages to the terminal.
* .assert() — to print a message to the terminal if the value is falsy.
* .table() — to print out a table in the terminal from an object or array.

Since console is a global module, its methods can be accessed from anywhere, and the require() function is not necessary.

**Instructions**

**1.**

Inside **app.js**, we have an array of pets, cleverly named petsArray.

Use console.log() to print petsArray to the terminal.

For these checkpoints, we’ll check the program with node app.js behind the scenes when you press the Run button.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Next, using the console.table() method, print petsArray to the terminal as a table.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Lastly, inside the console.assert() method, check if the length of petsArray is greater than 5.

const petsArray = ['dog', 'cat', 'bird', 'monkey'];

// Add console methods below!

console.log(petsArray)

console.table(petsArray)

console.assert(petsArray.length>5)

**The Process Module**

In computer science, a *process* is the instance of a computer program that is being executed. You can open Task Manager if you’re on a Windows machine or Activity Monitor from a Mac to see information about the various processes running on your computer right now. Node has a global process object with useful methods and information about the current process.

The process.env property is an object which stores and controls information about the environment in which the process is currently running. For example, the process.env object contains a PWD property which holds a string with the directory in which the current process is located. It can be useful to have some if/else logic in a program depending on the current environment— a web application in a development phase might perform different tasks than when it’s live to users. We could store this information on the process.env. One convention is to add a property to process.env with the key NODE\_ENV and a value of either production or development.

if (process.env.NODE\_ENV === 'development'){  
  console.log('Testing! Testing! Does everything work?');  
}

The process.memoryUsage() returns information on the CPU demands of the current process. It returns a property that looks similar to this:

{ rss: 26247168,  
  heapTotal: 5767168,  
  heapUsed: 3573032,  
  external: 8772 }

*Heap* can mean different things in different contexts: a heap can refer to [a specific data structure](https://en.wikipedia.org/wiki/Heap_(data_structure)), but it can also refer to a block of [computer memory](https://en.wikipedia.org/wiki/Memory_management). The process.memoryUsage().heapUsed method will return a number representing how many bytes of memory the current process is using.

The process.argv property holds an array of command line values provided when the current process was initiated. The first element in the array is the absolute path to Node, which ran the process. The second element in the array is the path to the file that’s running. The following elements will be any command line arguments provided when the process was initiated. Command line arguments are separated from one another with spaces.

node myProgram.js testing several features

console.log(process.argv[3]); // Prints 'several'

We’ve only covered a few of the properties of the process object, so make sure to check out the [documentation on the process object](https://nodejs.org/api/process.html) to learn more about it and explore some of its other methods and properties.

Let’s get some practice using the process object!

**Instructions**

**1.**

We want the program in **app.js** to store the starting amount of memory used (heapUsed), perform an operation, and then compare the final amount of memory used to the original amount. Right now, the initialMemory variable is assigned to null. Change this line, so that initialMemory is instead assigned the value of the heapUsed property on the object returned from invoking the process.memoryUsage() method.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

We want the user of the program to be able to fill in their own word when they run the program. Right now word is assigned to null. Change the program so that when a user initiates the program with an additional command line argument, word will be assigned that value. For example, running the program with the command: node app.js Codecademy would result in word being assigned the value 'Codecademy'

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Awesome! Now let’s run the program. Type node app.js followed by any word you like and then hit enter.

**The OS Module**

When developing or debugging an app, it can be helpful to have information about the computer, operating system, and network on which the program is running. Before Node, this information could not be retrieved using JavaScript due to the language being confined to the browser. However, Node.js is a JavaScript runtime, which means it can execute code outside of the browser, and we’re able to get access to much of this information through the os core module.

Unlike process and console, the os module is not global and needs to be included into the file in order to gain access to it’s methods. You can include the os module into your file by typing:

const os = require('os');

With the os module saved to the os variable, you can call methods like:

* os.type() — to return the computer’s operating system.
* os.arch() — to return the operating system CPU architecture.
* os.networkInterfaces — to return information about the network interfaces of the computer, such as IP and MAC address.
* os.homedir() — to return the current user’s home directory.
* os.hostname() — to return the hostname of the operating system.
* os.uptime() — to return the system uptime, in seconds.

Let’s take a look at an example:

const os = require('os');  
   
const local = {    
  'Home Directory': os.homedir(),      
  'Operating System': os.type(),  
  'Last Reboot': os.uptime()  
}

In the above example code, we first require the os module and store it in a variable, os. Below that, we have an object, local, that will hold some information about the user’s computer: the name of the home directory, the type of operating system, and the time since the computer was last rebooted.

  {  
    'Home Directory': '/Users/luca',  
    'Operating System': 'Darwin',  
    'Time since reboot': 86997  
  }

When we run the program, the local object stores all the requested information:

* the user’s home directory — '/Users/luca',
* the operating system — 'Darwin' ([Darwin](https://en.wikipedia.org/wiki/Darwin_(operating_system)) is the underlying operating system of macOS.),
* and the time since the computer was last rebooted — 86997 seconds.

Feel free to try running some of the os module methods on your own computer to get more information about the hardware, OS, and network you’re on!

Note that in the exercises below, the os module will return information from the learning environment hosted by Codecademy, and not your local computer.

**Instructions**

**1.**

In **app.js**, using the const keyword, declare a variable, os, that stores the os module.

Stuck? Get a hint

**2.**

Next, below the line requiring in the os module, use const to create an empty object named server.

Stuck? Get a hint

**3.**

Inside server, add the keys: type, architecture, and uptime. Store their respective values using [methods](https://nodejs.org/docs/latest-v14.x/api/os.html) from the os module.

Stuck? Get a hint

**4.**

Finally, console.log() the server object, and run the program by typing node app.js in the terminal.