Introduction to Node.js concepts

Node.js is a JavaScript runtime environment. This is what *executes* your JS code. This runtime environment runs on your server. Previously, you've written JavaScript to manipulate the DOM in the browser. In that case, your client-side JavaScript was being run by your *browser's* runtime environment.

For Node.js, since it is the runtime environment on the server, this means that you can execute **server-side JavaScript** in addition to client-side JavaScript (what we did previously). By using JavaScript for both server-side and client-side code, this means that you can do your development all in JavaScript without the need for many different languages.

Node.js is event-driven, and uses a non-blocking I/O model. The idea behind it is that, on the web, things are triggered by events, but we also don't know when events occur. This means that code should be written so that, in general, you don't have code blocking other code from running if they're not dependent.

Take the example when you are shopping in a store: you have to select an item to buy, then go the cashier, and pay for your item before exiting the store. Every customer executes these tasks when buying an item. Now, if this process was written synchronously (i.e. things must always follow a sequence), the shopping experience will be severely impacted because code run in a single thread needs to go through one customer before moving to the next. This creates a bottle neck.

On the other hand—even while using a single thread—if you allow customers to execute tasks as they complete them (e.g. customer 2 can go to pay before customer 1 if customer 1 has not finished browsing), that is, execute tasks **asynchronously**, this is much more efficient. This is the foundational idea behind Node.js.

Before we actually get into Node.js though, there are a couple key concepts you need to understand. These concepts are the foundation upon which Node.js is built. They are:

- modularity
- asynchronous programming
- event loop

Modularity

The idea of modular code is a core concept in programming (object-oriented, especially). By splitting up functionality into reusable bits of code, this makes maintaining code easier. There are different techniques:

- functions
- data encapsulation (classes, interfaces, superclasses, etc)

Functions

Functions are simply reusable chunks of code you can use to execute (usually) a common task.

Data encapsulation

The idea behind data encapsulation is that you encapsulate data into objects. Your object (class) definitions allow you to define specific properties (characteristics) and methods (actions the object can do) to your object. Data encapsulation and functions help to ease code maintenance because other coders (or other code) using these functions and objects only need to worry about what the function/class/property/method names are and what they need to input and expect back, but other than that, they have no idea how everything has been *implemented*. This blackbox-like experience is a key concept because it allows you to completely decouple the various functionalities of your code so your code won't affect the classes' and other functions' code you've used. It makes it somewhat more of a plug-and-play experience.

In JavaScript, you can create modules to hold specific properties and methods. This *encapsulates* the properties and methods within the module to prevent naming/coding conflicts. This conflict prevention or protection occurs because these properties and methods belong to the *module* (think of it like a namespace). We can do this with objects.

Example. There is an object used for logging messages named `logger`.

```
//OBJECT LITERAL
//You need commas between properties and methods.
//DO NOT add a comma after the last one.
var logger = {
  message: "Hello, how are you?",
  logMessage: function() {
    console.log(logger.message);
  },
  logError: function() {
    console.error(logger.message);
  }
};
```

In the above code, the object literal prevents naming conflicts but the properties and methods are still publicly available (i.e. you can access them directly).

In order to better "protect" your properties and methods (e.g. you want private property variables and methods), you'll need to use the module design pattern (https://www.patterns.dev/vanilla/module-pattern). Notice that there are exported functions below.

```
//MODULE PROPERTY (message) and METHODS (getMessage and setMessage)
//Only exported variables and functions are accessible outside the module. By
encapsulating in a “module” the non-exported items are “protected” and inaccessible
from outside the module.
var message = "My hidden message";
export function getMessage() {
  console.log(message);
};
export function setMessage(newMessage) {
    message = newMessage;
}
```

We can then use the module by first adding the script in the HTML like so:

```
<script type="module" src="[path-to-module]"></script>
```

Now, for the script using the module:

```
<script type="module" src="[path-to-script-using-module]"></script>
```

Notice that both script tags have **`type="module"`**. Any script which is using import or export (i.e. the script using the module and the module itself) needs to use **`type="module"`**.

To import from the module, you can use `import {getMessage, setMessage} from "<path_to_module>";`

If you don't have a default import, you need to import by name.

Note about export

There can only be one default export per module but you can have multiple non-default exports (e.g. exporting objects or functions). When not using default, you need to import the exact names of things you want to import in curly brackets. For example:

```
import { myFunction } from "<path_to_module>";
```

whereas when using default, it is already known what will be imported without any specific name (because it's the default) so you can write an arbitrary name without curly brackets. For example:

```
import myArbitraryDefault from "<path_to_module>";
```

Modules can have both non-default exports and a default export.

See the following for more information about export statements: https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export

Asynchronous programming

Asynchronous programming is programming in such a way that tasks can be called and run at any moment (i.e. you're not waiting for things to happen only in sequence). Previously, you've already done some asynchronous programming by creating event handlers.

Event handlers are asynchronous because they can be called to run at any point. You **don't know when a user will click on something or trigger some event**. In the meanwhile, your other Javascript code can be executed while the event handler is waiting to be called. This functionality that you want to execute upon triggering an event is called a **callback function** because it's the function that is **called** once the event is done.

Other types of asynchronous programming you've done are through the use of timers. The functions that are run upon a timer reaching it's set time are also callback functions. Any function which is set to run upon something occurring is a callback function.

Node.js uses a **lot** of callback functions. This is because to take advantage of having server-side JS, you want to code in such a way that it is **non-blocking**. This means that you don't want your code to have a bottleneck where other code **cannot** be run because it's waiting for your main code to complete running. If you instead write your code to be non-blocking by using asynchronous programming, this results in a much more efficient and faster experience. This is the advantage of using JS-based code.

Promises, async and await

Callback functions are nice but sometimes you end up nesting code a lot because of having many callback functions meant to trigger upon subsequent events. An alternative to callback functions is by using promises along with async/await.

Take a look at the following resources:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Event loop

Your Node.js code runs on a single thread (think of a one-lane road). This means that only one task (car) can be run at a time (as opposed to multi-threaded). This is why it's so important to write asynchronous code. If there's only a single thread, any task which *blocks* will prevent the other tasks (i.e. the cars behind the running task) from executing. In Node.js, this thread is called the **event loop**.

When you code your Node.js app, program in such a way that you **do not block the event loop!**

- https://nodejs.org/en/guides/event-loop-timers-and-nexttick/
  This is a more in-depth explanation of the various phases for the event loop (for your knowledge).

Install Node.js and test it

Go to the Node.js website and install the **latest stable release of Node.js**.

Once it's installed, you can check if it's installed by opening your command line interface (CLI) and type the following:

```
node -v
```

Let's try the basic server example from the Node.js docs (https://nodejs.org/en/about).

Create a new file and name it ***server.js***. This will hold our server code.

Type in the following and save:

```
const http = require('http');

const port = process.env.PORT || 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.end('<h1>Hello, World!</h1>');
});

server.listen(port, () => {
  console.log(`Server running at port ${port}`);
});
```

This server code imports the built-in http library and sets the port for the server to 3000. Upon navigating to the server at port 3000 (http://localhost:3000), it prints the HTML "Hello, World!".

Open your command line interface (CLI) and navigate to the folder where you have saved ***server.js***.

Run the Node.js server app by entering the following in the CLI:

```
node server.js
```

Exit the app by hitting **`Ctrl + C`**.