Introduction to Node.js

The `node` Command

We can execute Node.js programs in the terminal by typing the `node` command, followed by the name of the file.
The example command above runs **app.js**.

```
node app.js
```

Node.js REPL

Node.js comes with REPL, an abbreviation for read–eval–print loop. REPL contains three different states: *a **read** state where it reads the input from a user, *the **eval** state where it evaluates the user's input *the **print** state where it prints out the evaluation to the console. After these states are finished REPL loops through these states repeatedly. REPL is useful as it gives back immediate feedback which can be used to perform calculations and develop code.

```
//node is typed in the console to access REPL
$ node

//the > indicates that REPL is running
// anything written after > will be evaluated
> console.log("HI")

// REPL has evaluated the line and has printed out HI
HI
```

Node.js Global Object

The Node.js environment has a global object that contains every Node-specific global property. The global object can be accessed by either typing in `console.log(global)` or `global` in the terminal after RPL is running. In order to see just the keys `Object.keys(global)` can be used. Since global is an object, new properties can be assigned to it via `global.name_of_property = 'value_of_property'`.

```
//Two ways to access global
> console.log(global)
//or
> global

//Adding new property to global
> global.car = 'delorean'
```

Node.js Process Object

A process is the instance of a computer program that is being executed. Node has a global process object with useful properties. One of these properties is **NODE_ENV** which can be used in an if/else statement to perform different tasks depending on if the application is in the production or development phase.

```
if (process.env.NODE_ENV ===
'development'){
  console.log('Do not deploy!! Do not
deploy!!');
}
```

Node.js process.argv

`process.argv` is a property that 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 the Node, followed by the path to the file that's running and finally any command-line arguments provided when the process was initiated.

```
// Command line values: node web.js
testing several features
console.log(process.argv[2]); //
'testing' will be printed
```

Node.js process.memoryUsage()

`process.memoryUsage()` is a method that can be used to return information on the CPU demands of the current process. Heap can refer to a specific data structure or to the computer memory.

```
//using process.memoryUsage() will return
an object in a format like this:

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

Node.js Modules

In Node.js files are called modules. Modularity is a technique where one program has distinct parts each providing a single piece of the overall functionality - like pieces of a puzzle coming together to complete a picture. `require()` is a function used to bring one module into another.

```
const baseball = require('./babeRuth.js')
```

Node.js Core Modules

Node has several modules included within the environment to efficiently perform common tasks. These are known as the **core modules**. The core modules are defined within Node.js's source and are located in the `lib/` folder. A core module can be accessed by passing a string with the name of the module into the `require()` function.

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

Listing Node.js Core Modules

All Node.js core modules can be listed in the REPL using the `builtinModules` property of the `module` module. This is useful to verify if a module is maintained by Node.js or a third party.

```
> require('module').builtinModules
```

The `console` Module

The Node.js `console` module exports a `global` console object offering similar functionality to the JavaScript console object used in the browser. This allows us to use `console.log()` and other familiar methods for debugging, just like we do in the browser. And since it is a global object, there's no need to require it into the file.

```
console.log('Hello Node!'); // Logs
'Hello Node!' to the terminal
```

`console.log()`

The `console.log()` method in Node.js outputs messages to the terminal, similar to `console.log()` in the browser. This can be useful for debugging purposes.

```
console.log('User found!'); // Logs 'User
found!' to the terminal
```

The `os` Module

The Node.js `OS` module can be used to get information about the computer and operating system on which a program is running. System architecture, network interfaces, the computer's hostname, and system uptime are a few examples of information that can be retrieved.

```javascript
const os = require('os');

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

The `util` Module

The Node.js `util` module contains utility functions generally used for debugging. Common uses include runtime type checking with `types` and turning callback functions into promises with the `.promisify()` method.

```js
// typical Node.js error-first callback
function
function getUser (id, callback) {
  return setTimeout(() => {
    if (id === 5) {
      callback(null, { nickname: 'Teddy'
});
    } else {
      callback(new Error('User not
found'));
    }
  }, 1000);
}


function callback (error, user) {
  if (error) {
    console.error(error.message);
    process.exit(1);
  }
  console.log(`User found! Their nickname
is: ${user.nickname}`);
}

// change the getUser function into
promise using `util.promisify()`
const getUserPromise =
util.promisify(getUser);

// now you're able to use then/catch or
async/await syntax
getUserPromise(id)
  .then((user) => {
    console.log(`User found! Their
nickname is: ${user.nickname}`);
  })
  .catch((error) => {
    console.log('User not found',
error);
  });
```