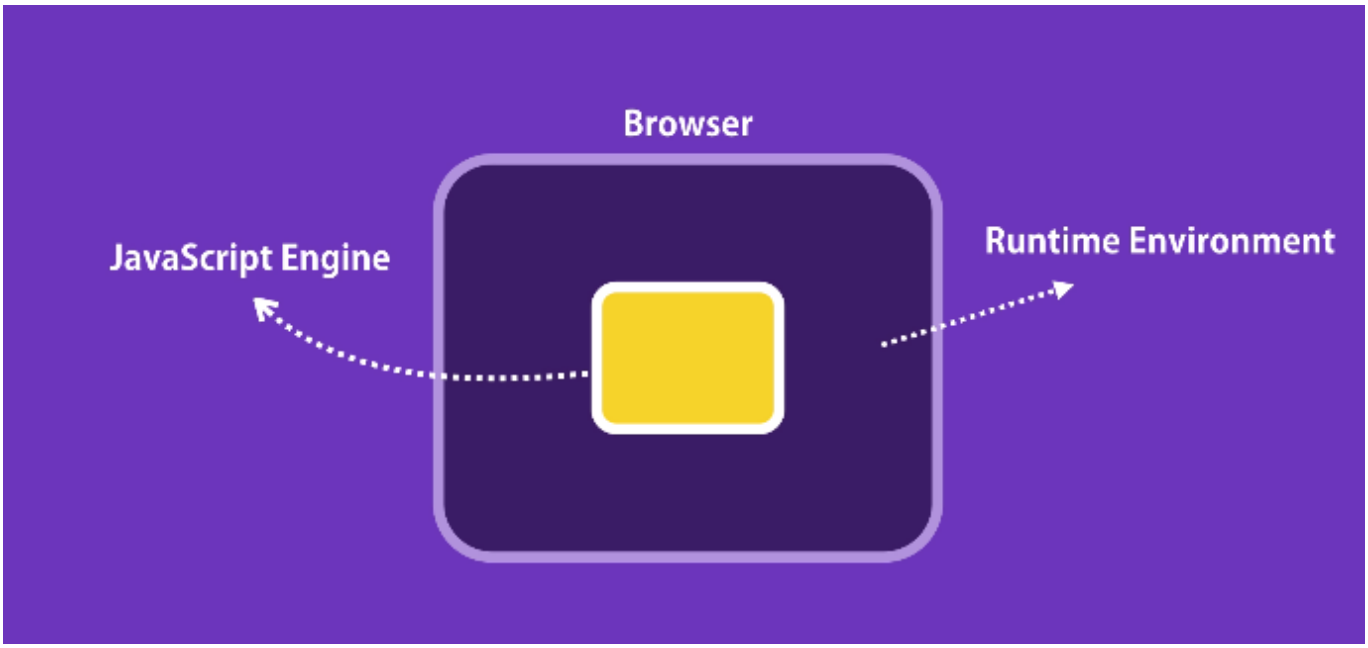
node

É um **Runtime Environment** (Ambiente de Execução) que permite o uso de Javascript **fora do navegador**

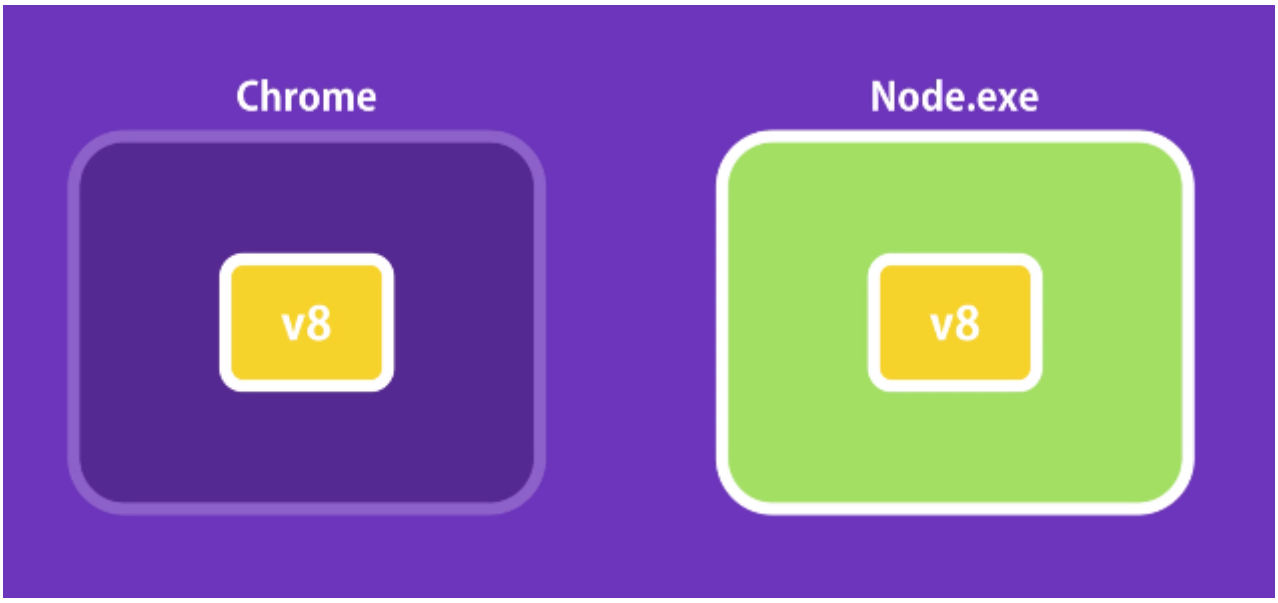
Javascript em Browsers vs Node

Originalmente, javascript só podia ser executado via navegador. Para tal, cada um possui sua própria **JS Engine** (Ou **interpretador javascript**), que interpreta o código em **código de máquina** e o executa, o que faz de JS uma linguagem **interpretada**. O próprio **interpretador é quem se comunica com a CPU e executa o código de máquina**, ao contrário de C (Que é uma linguagem compilada) onde o GCC/**compilador apenas compila o código para código de máquina**, que deve ser executado posteriormente pelo próprio usuário. Exemplos de JS engines: Chackra(Edge) SpiderMonkey(Firefox) e **V8(Google Chrome)**.



O **runtime Environment não é a Engine**, mas todo o **ambiente e funções providas** pelo mesmo que complementam o funcionamento de Engine e permitem um uso mais completo e específico da linguagem. Portanto, **navegadores também possuem Runtime Environment** para Javascript, que são **limitados pelo escopo do Browser** (Podem acessar elementos HTML pelo id ou poppar caixas de alerta, procedimentos que fazem sentido no contexto do navegador, mas não podem, por exemplo, acessar arquivos do HD do usuário, bem como a maioria das funções de sistema no geral).

Node utiliza apenas o **V8 do Chrome**, envolvendo-o em **código c++** para criar um runtime environment **alheio ao navegador** com **acesso a funções do sistema**, desta maneira permitindo uma execução **nativa** e com **funções essenciais** para **servidores**.



Ou seja, o Chrome e o Node possuem **a mesma JS Engine**, mas promovem **diferentes ambientes de execução** para javascript.

Por exemplo:

```
let my_div = document.getElementById("div_principal")
```

É um programa que **funciona apenas em navegadores**, pois `document.getElementById` é definida pelo runtime environment dos mesmos. Já:

```
const fs = require('fs')

fs.readFile('/Users/joe/test.txt', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

Utiliza o `readFile`, que é definido pelo Node, não sendo disponível em navegadores.

window vs global e let vs var: Escopo global

window é o objeto global que representa uma janela em navegadores.

`console.log()` é uma abreviação de `window.console.log()`, por exemplo. Além disso, variáveis declaradas com **var** são, na verdade, adicionadas a window. Por exemplo:

```
var my_var = 3;
console.log(window.my_var)
```

imprime

```
3
```

Este é o motivo de se ouvir que "variáveis declaradas com var são globais". Também é o motivo de **não ser recomendado** o uso de var, pois diferentes arquivos podem possuir variáveis com mesmo nome, que são sobescritas, o que pode gerar comportamentos inesperados.

Ao invés disso, se recomenda o uso de **let**

```
//browser javascript
let my_var = 3;
console.log(window.my_var)
```

não gera o mesmo resultado, imprimindo

```
undefined
```

node **não** possui o objeto window. Ao invés disso, possui o objeto **global**.

```
//node javascript
global.console.log("oi")
```

também imprime

```
oi
```

Entretanto, variáveis decladaras com **var** NÃO são adicionadas ao objeto global em node.

```
//node javascript
var my_var = 3;
global.console.log(my_var)
```

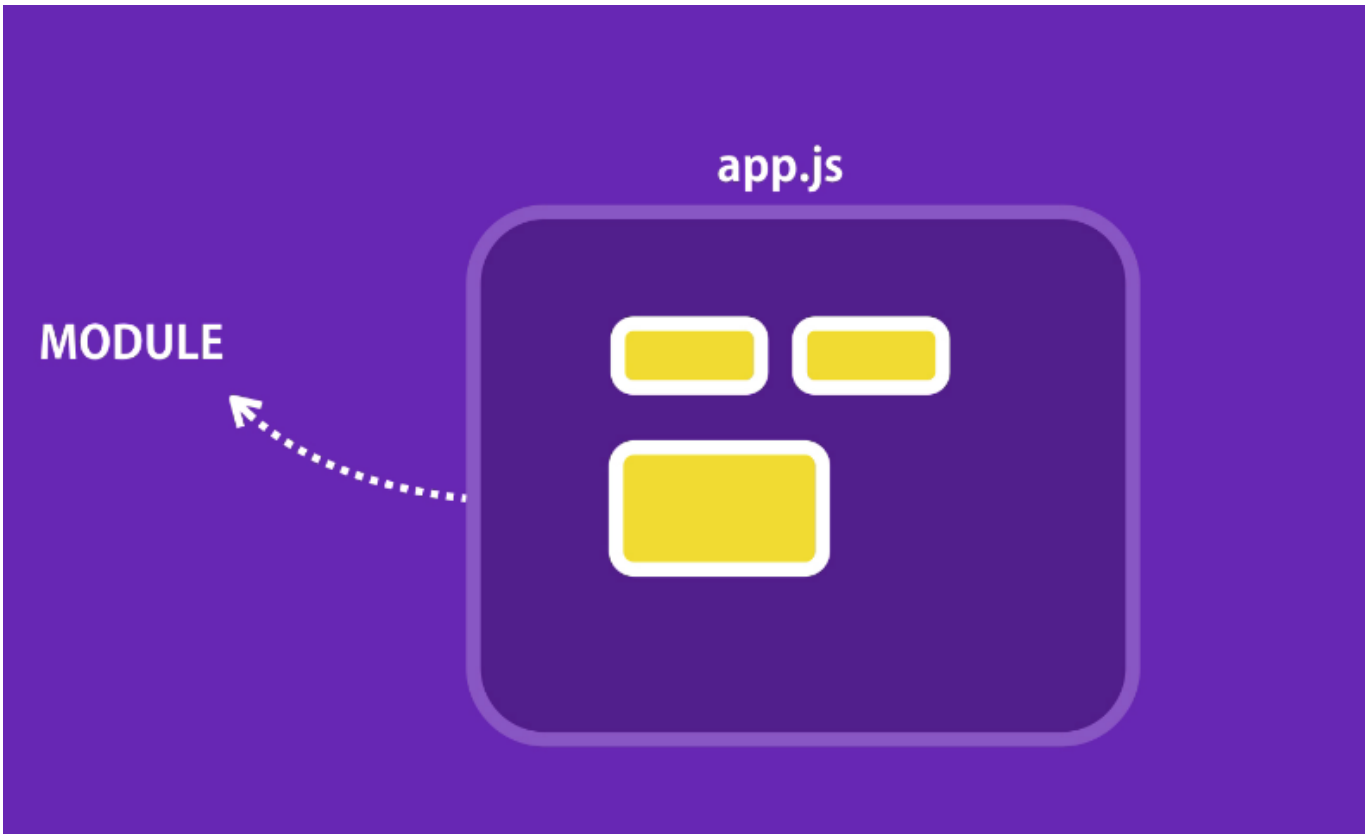
imprime

```
undifined
```

Isto se deve ao conceito de **modularização** de node

Node Modules

Em node, cada arquivo possui um **módulo**, cujas variáveis e métodos são privados (Acessíveis) inicialmente apenas por ele próprio.



`module` é um objeto, que **não pertence** a global.

```
console.log(global.module)
```

retorna

```
undefined
```

Já

```
console.log(module)
```

retorna

```
Module {
  id: '.',
```

```
  exports: {},
  parent: null,
  filename:
   '/home/USUARIOS/9793502/Web-Development-Studies/8 Node/test.js',
  loaded: false,
  children: [],
  paths:
   [ '/home/USUARIOS/9793502/Web-Development-Studies/8 Node/node_modules',
     '/home/USUARIOS/9793502/Web-Development-Studies/node_modules',
     '/home/USUARIOS/9793502/node_modules',
     '/home/USUARIOS/node_modules',
     '/home/node_modules',
     '/node_modules' ] }
```

Para tornar o seu conteúdo público, deve-se **exportar** o módulo.

Exportando um módulo

- `module.exports` cria uma variável/função que será exportada, e a assimila a uma variável/função do módulo.

```
module.exports.nome_como_será_exportado = variavel_do_modulo
```

Suponha um arquivo **library.js**, que exportará variáveis **my_a** e **my_b**, bem como a função **library_print**.

```
//library.js
var my_a = 3;
var my_b = 4;
var my_c = 5;

function library_print(a,b){
    c = a + b;
    console.log(c);
    return c;
}
//module.exports.nome_como_sera_exportado = nome_da_variável_no_módulo
module.exports.any_name_a = my_a; //posso usar qualquer nome
module.exports.my_b = my_b;       //posso usar mesmo nome
module.exports.library_prints = library_print; //posso exportar funcoes
```

Importando um módulo

- `require('path')` é o método de Node que importa um módulo. Possui como **argumento** o **caminho para o arquivo** que será importado.

Suponha um arquivo **app.js**, que importará as variáveis e funções **exportadas de library.js**, este, na mesma pasta de app.js.

```
var library = require('./library.js') //nome qualquer, não precisa ser o
mesmo do arquivo

console.log(library)
```

imprime

```
{ any_name_a: 3,
  my_b: 4,
  library_prints: [Function: library_print] }
```

Um JSON que indica os atributos e métodos disponibilizados por library. Note que **my_c não foi exportada**, e portanto **não é acessível** por arquivos que importam library.

É possível acessar os métodos/atributos de library com o operador `.` :

```
//app.js
var library = require('./library.js') //nome qualquer, não precisa ser o
mesmo do arquivo

console.log('a importado de library:' + library.any_name_a)
console.log('b importado de library: ' + library.my_b)
console.log('Função importada de library que soma os números importados: '
+ library.library_prints(library.any_name_a, library.my_b))
```

imprime:

```
a importado de library:3
b importado de library: 4
7
Função importada de library que soma os números importados: 7
```

É importante notar que **require não é um construtor!**

```
//app.js
let instA = require('./library.js')
let instB = require('./library.js')

instA.my_b = 111;
instB.my_b = 666;

console.log(instA.my_b);
console.log(instB.my_b);
```

tem como saída:

```
666
666
```

Ou seja, **instâncias geradas a partir do require de um mesmo arquivo apontam para o mesmo objeto!!!**.

O resultado esperado é obtido modificando-se library, que terá uma **função que retorna um objeto**:

Em library:

```
//library.js
function library_generates_object(){
    let my_obj = {
        my_a: 3,
        my_b: 4,
    }
    return my_obj;
}

module.exports.library_generates_object =  library_generates_object
```

Em app:

```
//app.js
let library = require('./library.js');

var instA = library.library_generates_object();
var instB = library.library_generates_object();

instA.my_a = 111;
instB.my_a = 666;

console.log(instA.my_a)
console.log(instB.my_a )
```

Que agora imprime corretamente:

```
111
666
```

Ou ainda, utilizar a sintaxe de *classses*:

Em library.js:

```
//library.js

class my_class{
    constructor(){
        this.my_a = 2;
        this.my_b = 3;
    }
}
module.exports.library_generates_object =  my_class
```

Em app.js:

```
//app.js
let library = require('./library.js');

let instA = new library.library_generates_object();
let instB = new library.library_generates_object();

instA.my_a = 111;
instB.my_a = 666;

console.log(instA.my_a)
console.log(instB.my_a)
```

Que também retorna corretamente:

```
111
666
```

Fontes:

.1 https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html

.2 https://www.udemy.com/course/nodejs-master-class