



# JAVASCRIPT IMPRESSIONADOR

Curso Completo de JavaScript da Hashtag Programação





## Módulo 2

DADOS, VARIÁVEIS E OPERAÇÕES

DADOS, VARIÁVEIS E OPERAÇÕES

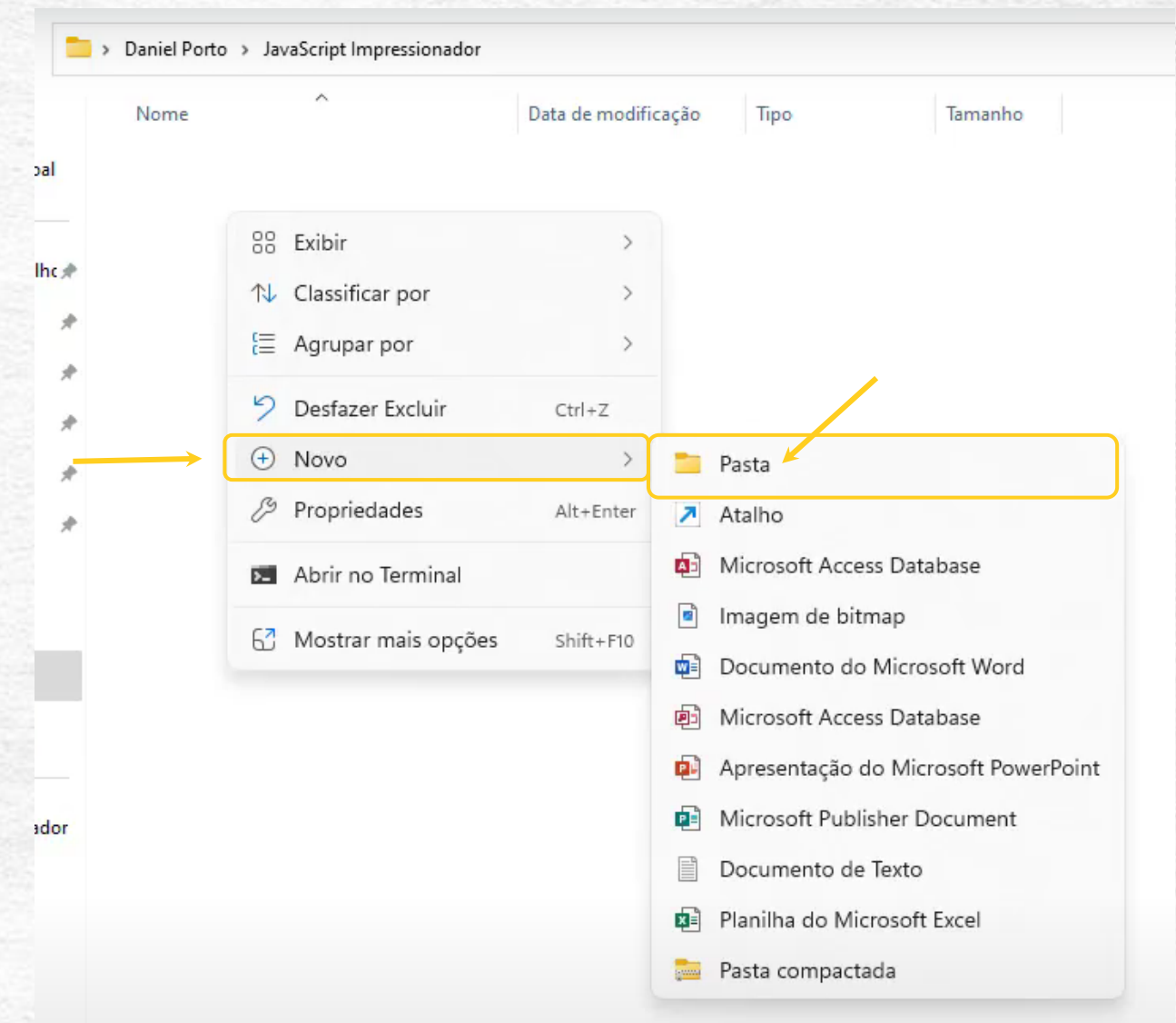
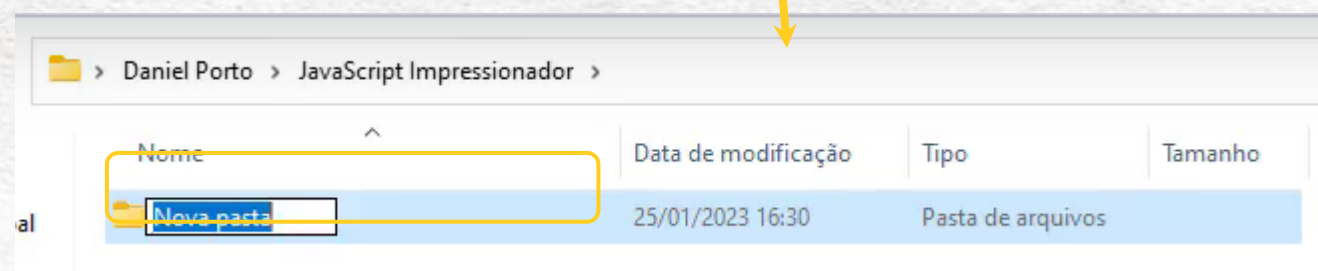
DADOS, VARIÁVEIS E OPERAÇÕES



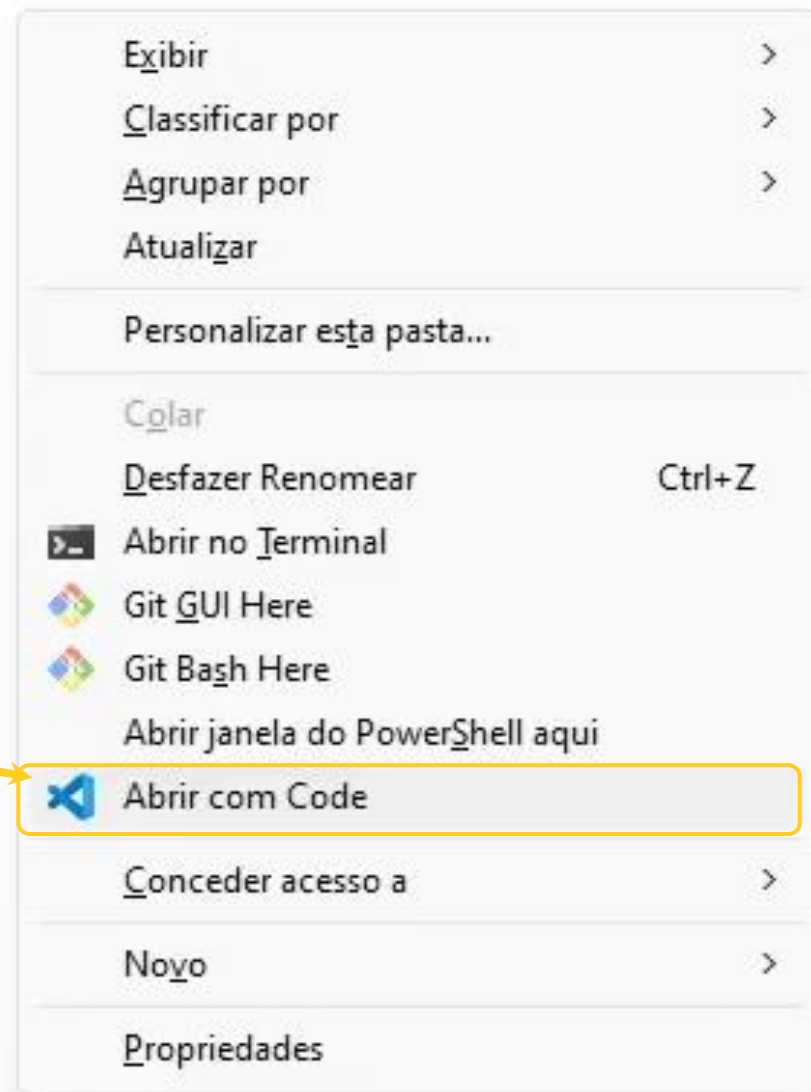
Neste módulo, iremos escrever nossos primeiros códigos em JAVASCRIPT.

Para iniciarmos, vamos realizar os próximos passos para que os códigos e as aulas fiquem organizados em uma estrutura de pastas e de fácil compreensão.

- **1º Passo:** Criaremos uma pasta chamada JavaScript Impressionador no seu computador (no local que faça sentido para você- exemplo: Pasta Pessoal, Área de Trabalho, Documentos ).
- **2º Passo:** Dentro da pasta JavaScript Impressionador, clicando com o botão direito iremos clicar em 'Novo', depois 'Pasta', iremos nomear ela de acordo de onde você estará no curso, que nesse momento será 'modulo\_2'. Onde iremos armazenar todos os códigos de Javascript feitos.

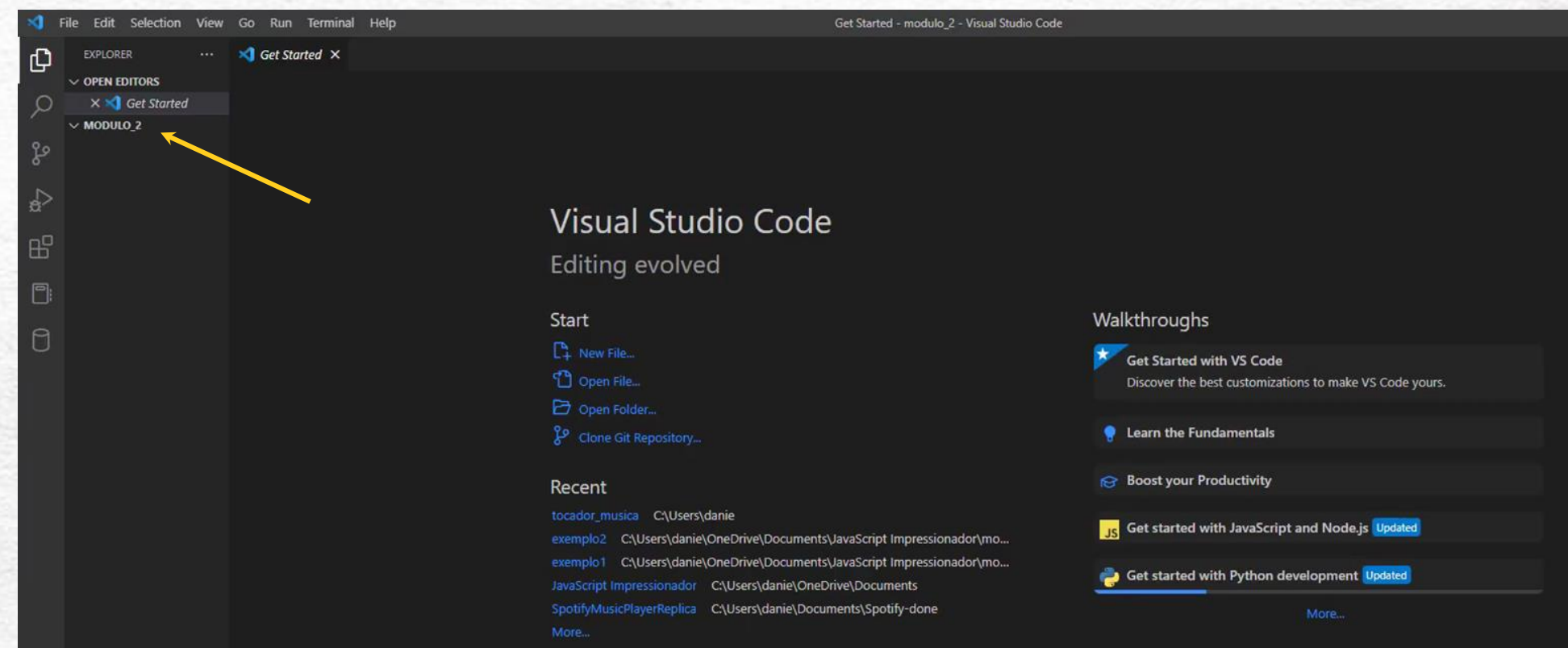






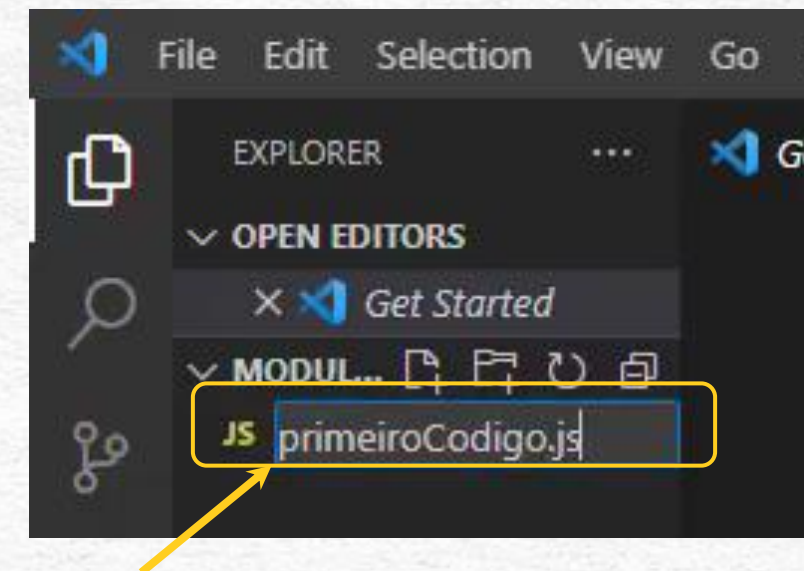
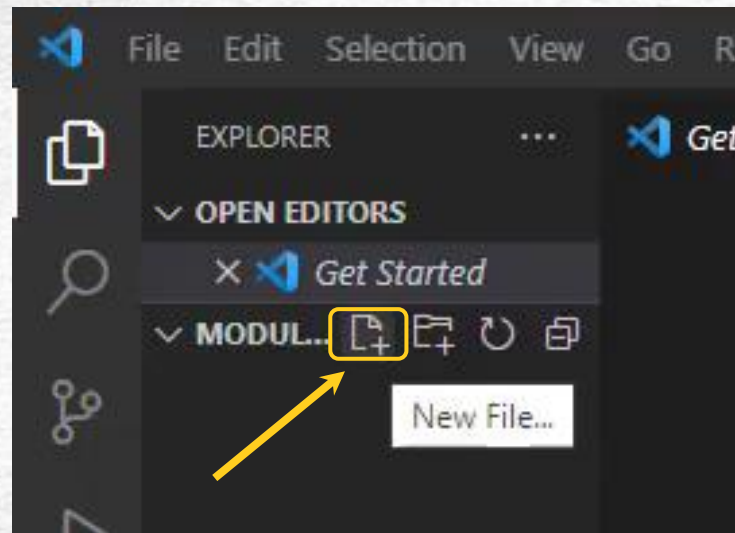
- **3º Passo:** Dentro da pasta modulo\_2, iremos clicar com '*SHIFT + botão direito do mouse*' e clicar na opção '**Abrir com Code**'.

Ao finalizar o 3º passo, o programa *Visual Studio Code* (VS Code) irá abrir “olhando” para esta pasta, ou seja, todo o código que trabalharmos no programa estará dentro da pasta que criamos. E assim a nossa organização e leitura ficará mais simples.





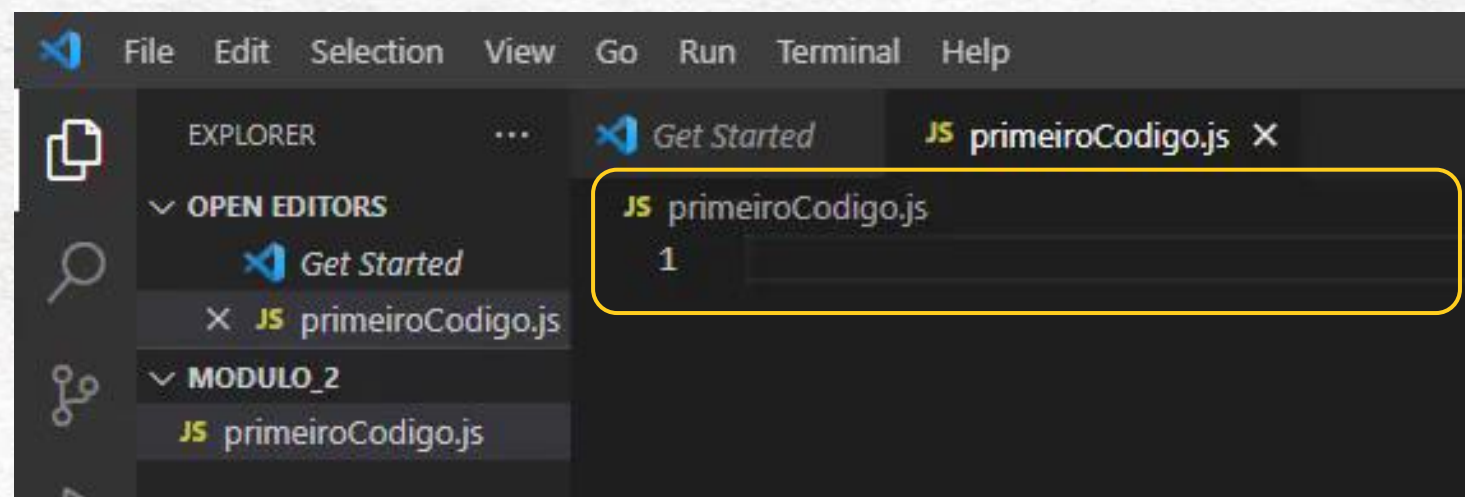
- 4º Passo: Iremos iniciar um novo arquivo, clicando no ícone de 'New File', nomeando-o **“primeiroCodigo.js”**



Repare que ao colocarmos **“.js”** no final do nosso arquivo e salvar um ícone **JS** foi gerado. Mas o que é e por que isso ocorre??

O VS Code, ao receber um arquivo **“.js”** automaticamente já faz a leitura de que se trata de um arquivo de código de Javascript, e isso ocorre porque essa extensão **“.js”** é utilizada para além de sinalizar, criar um arquivo do tipo de Javascript.

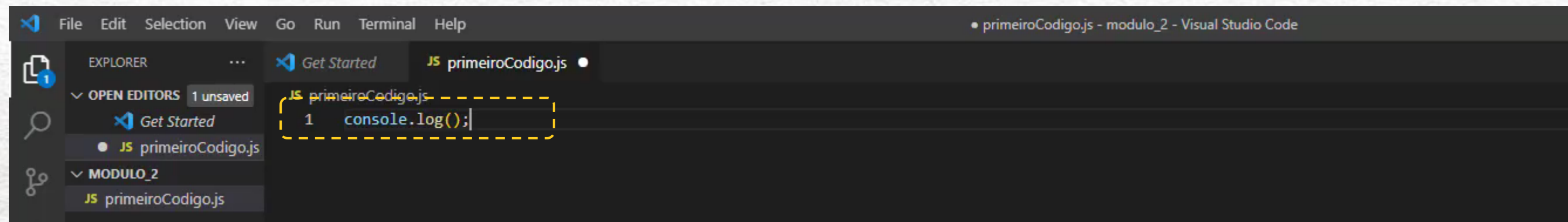
Isso facilita o VS Code a interpretar e tratar ele, gerando dicas e atalhos que facilitam o processo de escrever códigos.





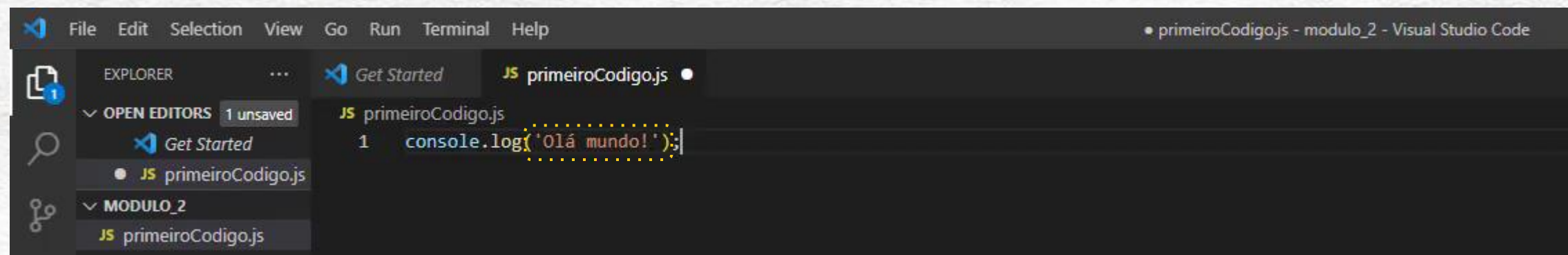
Finalmente vamos dar início. O JavaScript é uma sequência de comandos a serem executados, ou seja, um comando JavaScript tem o propósito de dizer ao programa o que fazer, como se fosse uma instrução.

O comando **console.log()** imprime o texto no console como uma mensagem de log dentro da tela, basicamente as informações que colocamos dentro dos parentes serão impressas na saída do VS Code.



Normalmente usamos essa técnica de programação para conseguir enxergar o caminho que o nosso código está fazendo, mas entenderemos isso mais à frente.

No JavaScript ao escrever **TEXTOS** devemos colocá-los entre aspas para que o programa entenda que tipo de informação estamos passando.





1 Javascript é uma linguagem interpretada e leve. O computador recebe o código JavaScript em sua forma de texto original e executa o seu 'script' em tempo de execução, e não antes, ou seja, é interpretado/traduzido diretamente no código da linguagem por um interpretador chamado JavaScript Engine. E durante o curso utilizaremos o interpretador **NODE.JS**.

2

O código JavaScript é uma sequência de comandos JavaScript. Cada comando é executado pelo computador na sequência que eles são escritos, de cima para baixo. Isso significa que você precisa tomar cuidado com a ordem que você coloca as coisas dentro do seu código.



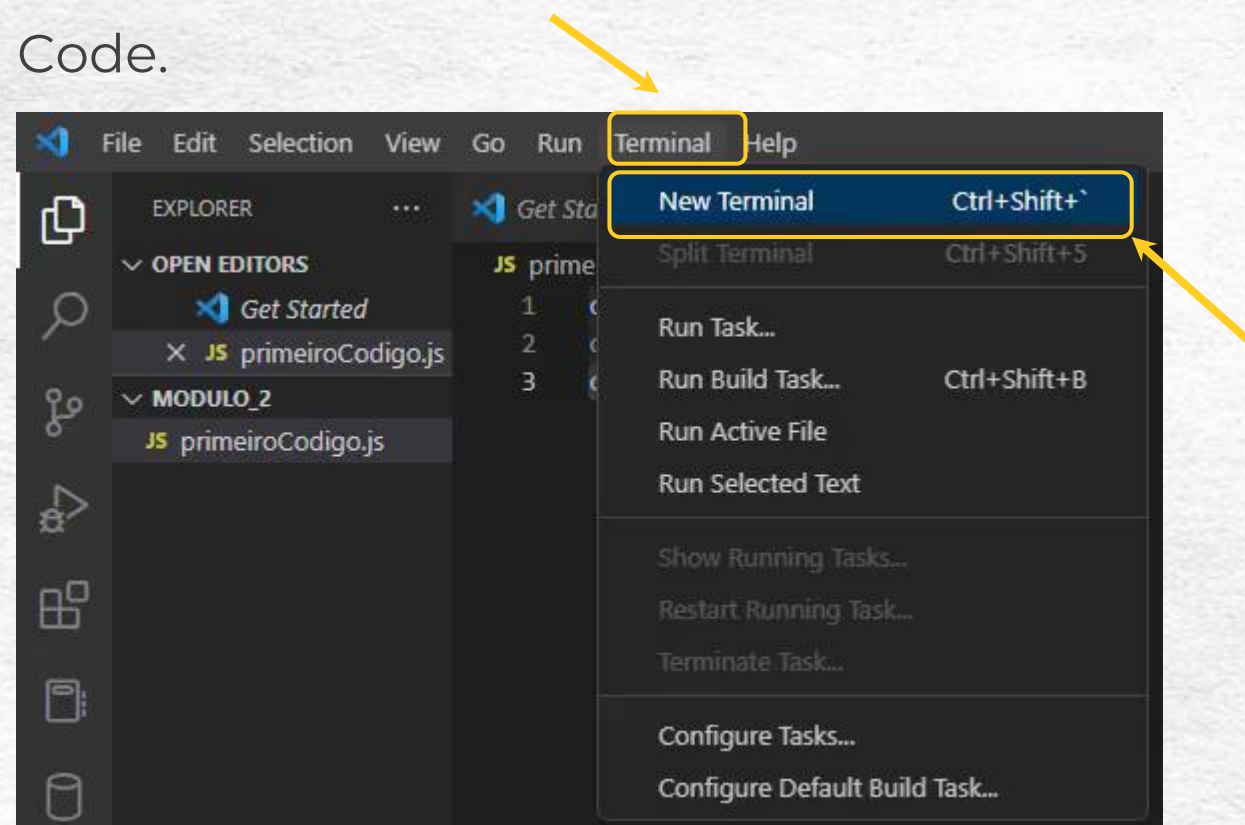
```
JS primeiroCodigo.js X
JS primeiroCodigo.js
1 console.log('Olá mundo!'); 1°
2 console.log('Olá mundo!'); 2°
3 console.log('Olá mundo!'); 3°
```



Para executarmos o Node.js e interpretar nosso código JavaScript iremos realizar as seguintes instruções!

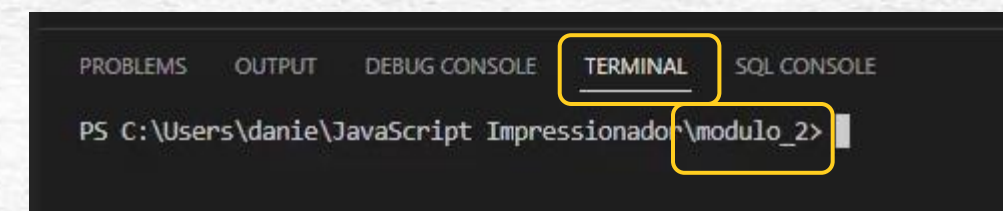
## Instrução 1º

Iremos clicar com o botão esquerdo em 'Terminal' e logo em seguida clicar em 'Novo Terminal', para iniciarmos o Terminal de comando dentro do VS Code.



## Instrução 2º

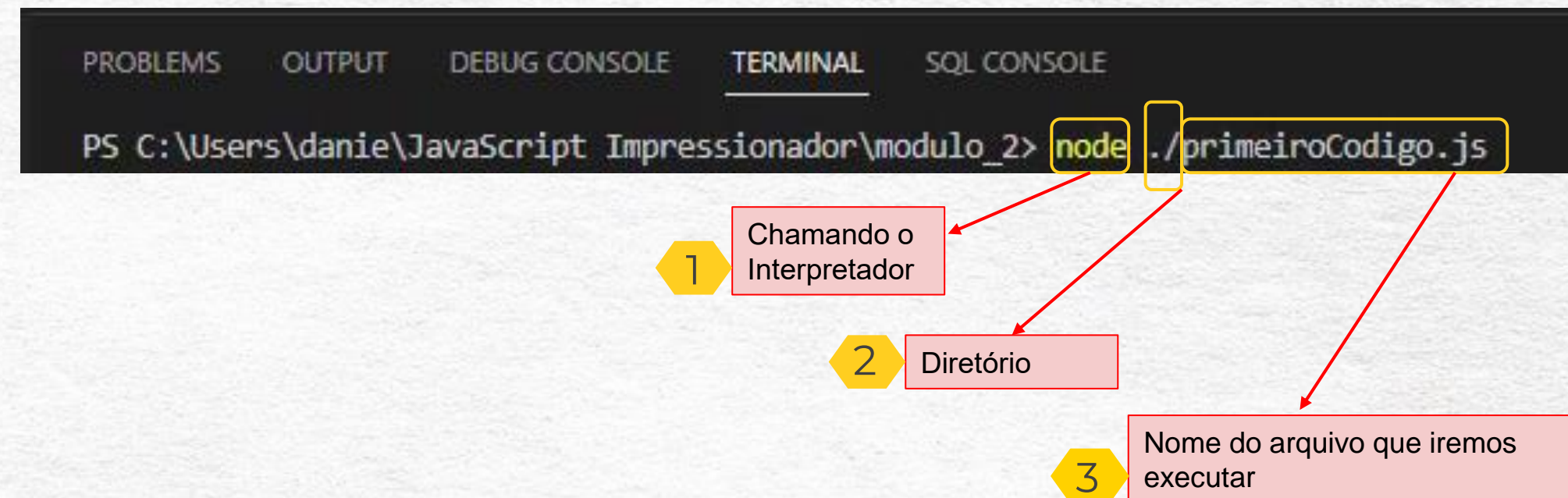
Na parte inferior do VS Code, o Terminal irá abrir e apontar para o local / pasta que estamos trabalhando. Nesse caso, no diretório 'modulo\_2'. E dentro do terminal poderemos escrever comandos para interpretar e executar o nosso primeiro programa.





## Instrução 3º

No terminal iremos digitar o comando **node**, pois será o interpretador que utilizaremos dentro do nosso programa para ler e executar o nosso código. Seguindo de **./**, que corresponde “a partir do diretório que estou trabalhando”, que é o local / pasta que está salvo o nosso arquivo de Javascript. E para finalizar escrevemos o nome do arquivo que queremos executar, que no nosso caso é **primeiroCodigo.js**.







The image shows a VS Code editor window with a file named `primeiroCodigo.js`. The code contains three lines of JavaScript:

```
1 console.log('Olá mundo!');  
2 console.log('Olá mundo!');  
3 console.log('Olá mundo!');
```

Below the editor, the terminal window shows the command `node ./primeiroCodigo.js` being executed, resulting in three lines of output: `Olá mundo!`, `Olá mundo!`, and `Olá mundo!`. A yellow box highlights the output, and a blue button labeled "Enter" is shown next to it.

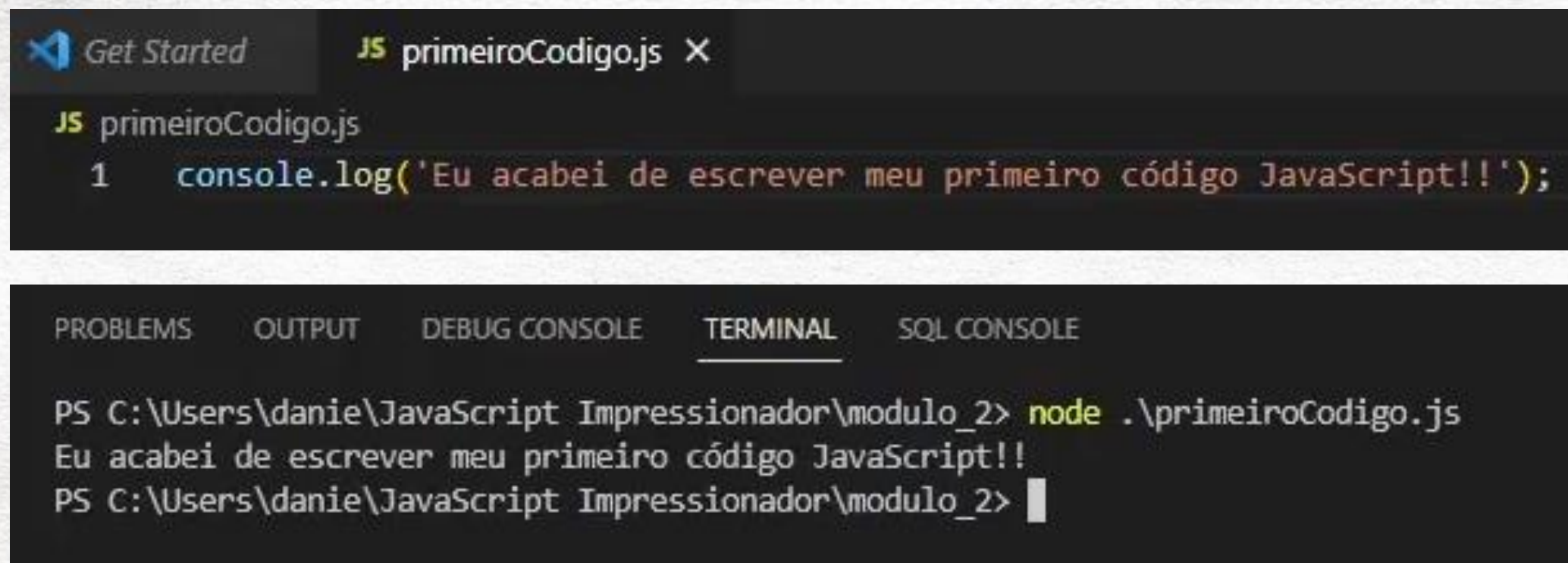
## Instrução 4º

E agora é hora de executarmos o nosso primeiro programa em JavaScript apertando a tecla **'ENTER'** e verificar se a saída é a instrução que queríamos que nosso código fizesse. No nosso programa esperamos que o comando **console.log** IMPRIMA três vezes o texto "Olá mundo!" que escrevemos dentro do seus parênteses ().

### “TUDO COMEÇA COM UM ‘Olá Mundo!’ (ou ‘Hello World!’)”

Uma curiosidade para você é que o 'Olá Mundo!', é uma frase que foi imortalizada pelos criadores da linguagem de programação C, porém foi aderida por toda a comunidade de programação como uma brincadeira, para iniciarmos com o pé direito em uma linguagem, criamos o nosso primeiro programa dizendo 'Olá Mundo!'.





The screenshot shows the Visual Studio Code interface. At the top, there's a tab for 'primeiroCodigo.js'. The editor displays a single line of JavaScript code: `console.log('Eu acabei de escrever meu primeiro código JavaScript!!');`. Below the editor, the 'TERMINAL' pane is active, showing the command `node .\primeiroCodigo.js` being executed. The output of the command is `Eu acabei de escrever meu primeiro código JavaScript!!`.

## Meu Primeiro Programa JavaScript

E agora para finalizarmos iremos trocar o texto do nosso comando `console.log()`, por “Eu acabei de escrever meu primeiro código JavaScript!!” e executamos as mesmas instruções do “Olá mundo!”.

Pronto!! Você acabou de escrever e executar o seu primeiro programa, incrível não!?





### Limpar o terminal

Dentro do seu terminal você irá escrever o comando **cls** ou **clear** e apertar a tecla 'ENTER' para executar. A tela do seu terminal será totalmente limpa e estará pronta para receber novas instruções.



### Caminhar pelo terminal

Ao clicarmos na tecla '**para cima**' e '**para baixo**' no seu teclado, você caminhará através dos comandos que você já executou no seu terminal. São atalhos para facilitar seu dia a dia como programador.

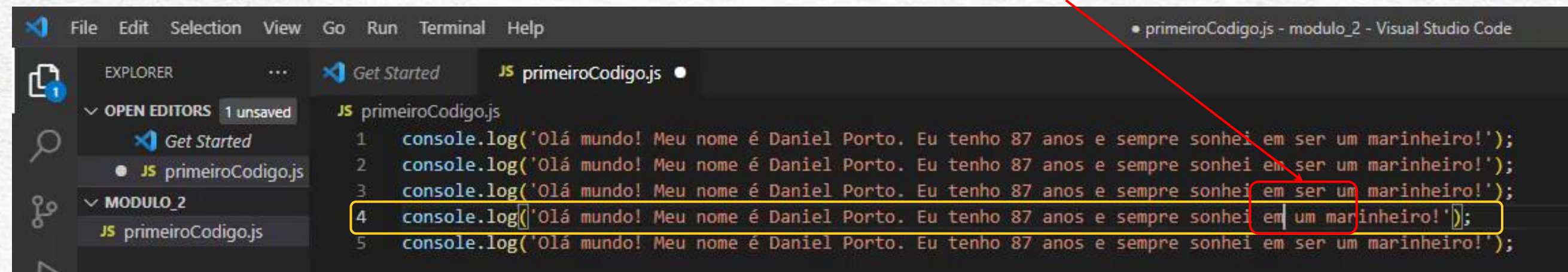


Recapitulando o que vimos anteriormente, aprendemos que o JavaScript possui um comando chamado **console.log()** que tem a função de imprimir na nossa tela as informações passadas entre seus **()**. E os comandos sempre são executados uma linha de código depois da outra, de cima para baixo.

E para executarmos o nosso código JavaScript precisamos salvá-lo e depois executar dentro do terminal o comando:

**node ./nomeDoArquivo.js**

Perfeito! Mas agora imagine uma situação na qual queremos imprimir diversas vezes a mesma informação, e eu precisasse copiar e colar ou escrever á cada `console.log()`, concordam que alguns erros poderiam ocorrer, como colar ou digitar algo errado?

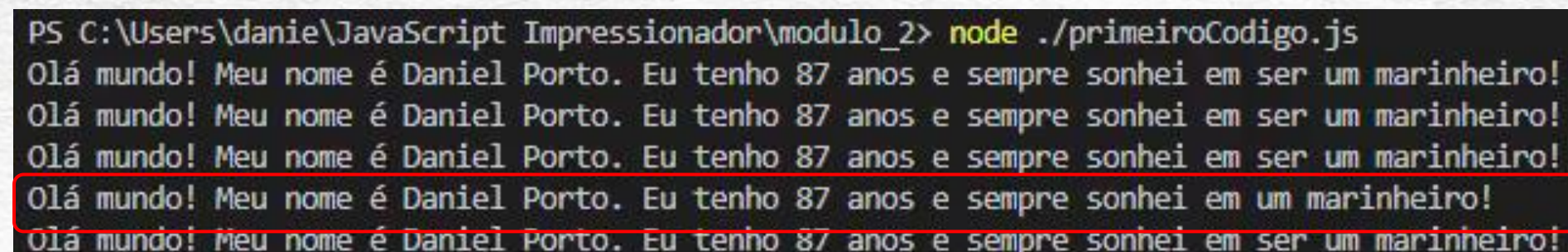


```
File Edit Selection View Go Run Terminal Help
primeiroCodigo.js - modulo_2 - Visual Studio Code

EXPLORER
  OPEN EDITORS 1 unsaved
    Get Started
    JS primeiroCodigo.js
  MODULO_2
    JS primeiroCodigo.js

JS primeiroCodigo.js
1 console.log('Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!');
2 console.log('Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!');
3 console.log('Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!');
4 console.log('Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em um marinheiro!');
5 console.log('Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!');
```

Esse é um exemplo simples que de certa forma seria fácil corrigir, mas concorda que ao executarmos esse código teríamos um erro em sua resposta / saída? Agora imagine em uma aplicação muito maior com mais linhas de código, a tarefa teria um processo de identificação de erro muito mais longo.



```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
```



Uma das formas para evitarmos esses possíveis erros é guardando essas informações em caixinhas. Todas as vezes que quisermos utilizar essa informação iremos dizer ao nosso código para utilizar a caixinha que possui a informação.

Essa caixinha é a nossa **VARIÁVEL**, que é onde iremos armazenar valores que serão manipulados por nossos programas JavaScript. E para utilizarmos elas precisamos colocar uma etiqueta nelas, ou seja, dar um **nome simbólico** a nossa variável (geralmente esse nome terá relação com o que estaremos armazenando dentro dela).

E também precisamos identificar o **tipo de declaração** que aquela variável se caracteriza (**let ou const**).

Então a **estrutura** de uma **variável** será:

**declaração do tipo + nome da variável + sinal de igual ( = ) + informação armazenada.**

```
JS primeiroCodigo.js > ..  
1  const mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!';  
2
```

E para utilizarmos essa variável dentro do nosso console.log() se tornou muito mais simples:

```
3  console.log(mensagemTela);  
4  console.log(mensagemTela);  
5  console.log(mensagemTela);  
6  console.log(mensagemTela);  
7  console.log(mensagemTela);
```



Agora em um cenário que precisasse alterar a informação do meu `console.log()` seria muito mais simples, ao invés de a cada `console.log()` eu efetuar a alteração, é só alterar dentro da variável o conteúdo que quero imprimir, já que o comando `console.log()` irá capturar o valor que está dentro da minha variável e utilizar.

```
JS primeiroCodigo.js > ...
1  const mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade';
2
3  console.log(mensagemTela);
4  console.log(mensagemTela);
5  console.log(mensagemTela);
6  console.log(mensagemTela);
7  console.log(mensagemTela);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  SQL CONSOLE

PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
```

Lembre-se sempre de salvar e depois executar o seu programa, para que essas alterações sejam alteradas e salvas dentro do seu programa.





As diferentes formas de declaração de variáveis são utilizadas para o mesmo propósito de armazenar uma informação ou valor, mas é importante entender que existem diferenças entre elas para saber em qual momento devemos utilizar cada tipo de declaração.

### LET

A variável do tipo **let** permite que você declare variáveis que podem receber outros valores. Em outras palavras, o valor original dessa variável pode ser substituído por outra informação, como se a variável fosse atualizada. Sua característica é ser **MUTÁVEL**.

### CONST

A variável do tipo **const** permite que você declare variáveis **constantes**. Em outras palavras, as informações atribuídas a esse tipo de variável não podem ser alteradas. Geralmente utilizamos para guardar informações fixas, que não serão alteradas nunca em nosso código. Sua característica é ser **IMUTÁVEL**.

### VAR

Você pode se deparar com códigos que utilizem o tipo **var** para variáveis ela atua de forma parecida com o tipo **let**, porém com uma facilidade de alteração muito maior, o que pode ocasionar muitos bugs no seu código, então a partir das atualizações do JavaScript (ES6), ⚠ **o uso do tipo var não é uma boa prática.**



Como podemos observar o exemplo abaixo, nesse momento declaramos nossa variável com o tipo **LET**, pois iremos demonstrar como podemos substituir o valor e verificar como que o JavaScript faz a leitura desses comandos e suas atualizações. Sabemos que para criar uma variável precisamos construir toda sua estrutura iniciando com o tipo de declaração. Mas e para atualizarmos uma variável existente?

```
JS primeiroCodigo.js > ...
1  let mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade';
2
3  console.log(mensagemTela);
4  console.log(mensagemTela);
5  console.log(mensagemTela);
6  console.log(mensagemTela);
7  console.log(mensagemTela);
8
9
10 mensagemTela = 'Agora o que eu quero ser é um idoso filósofo!';
11
12 console.log(mensagemTela);
```

Para atualizarmos os valores de uma variável precisamos apenas chamá-la ou invocá-la pelo seu nome, nesse caso **'mensagemTela'** + **=** + escrevemos o **novo valor** que aquela variável irá armazenar. Lembrando que isso é possível, pois estamos utilizando o tipo **LET**.

O retorno do seu programa seguirá os comandos de cima para baixo, então efetuará a impressão com o valor original 5 vezes , atualizará a variável e irá imprimir o seu valor novo, ou seja, até o momento em que nossa variável não foi atualizada todo o código acima disso será com o seu valor antigo / original.

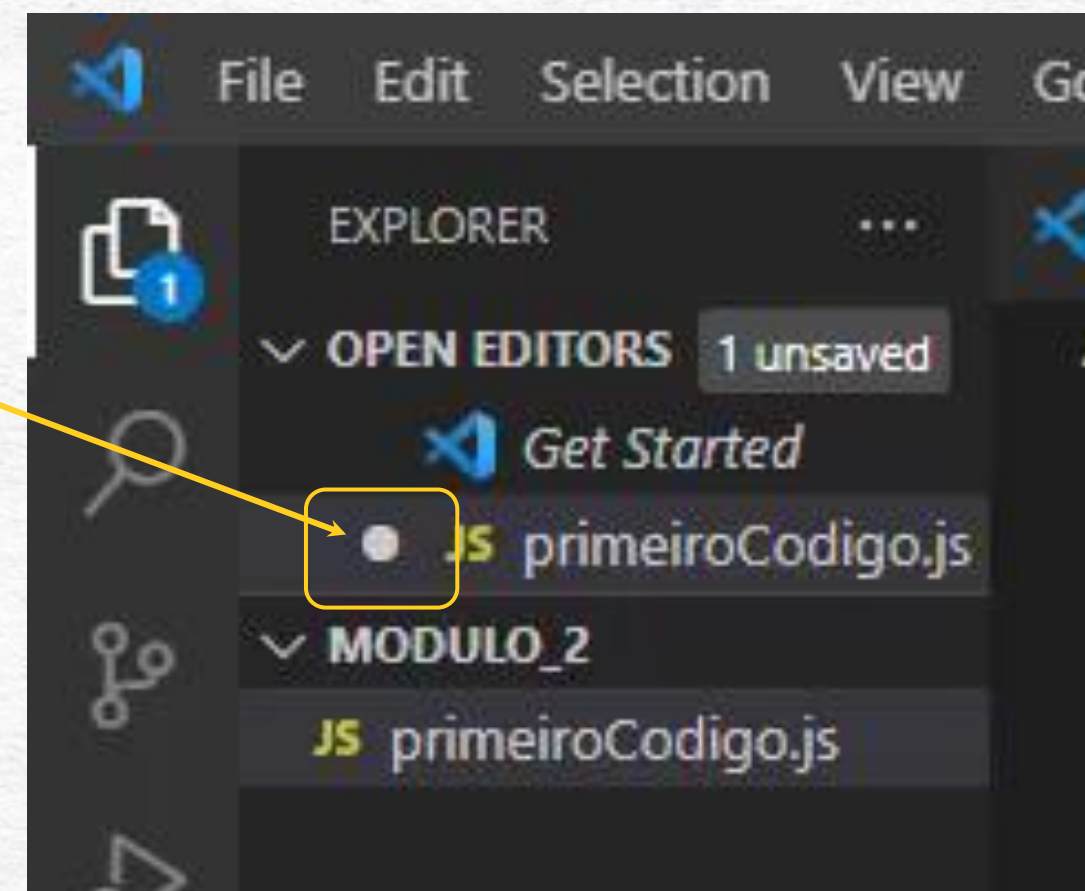
```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Agora o que eu quero ser é um idoso filósofo!
```





### Atualizar e Salvar

No VS Code toda atualização que não está salva será indicada por uma bolinha branca, você precisa salvar ( **CTRL+S** ) toda alteração que fizer no seu código para depois executar o seu programa com essas novas alterações.







### Copiando o código

Um atalho do VS Code para você copiar a linha de código para cima ou para baixo é utilizar o comando SHIFT + ALT + tecla para baixo ou tecla para cima.



### O ponto e vírgula ( ; )

O ponto e vírgula em JavaScript é opcional, mas é considerado uma boa prática entre os programadores. É utilizado como separador de instruções, apenas nos casos de mais de uma instrução na mesma linha o ; é obrigatório.



Vamos falar um pouco mais sobre **VARIÁVEIS**, especificamente algumas regras de como devemos escolher as etiquetas delas, ou seja, a sua nomenclatura (nome).

Você irá perceber ao longo do curso que o JavaScript possui boas práticas, ou seja, situações que são opcionais mas que fazem o nosso código ser mais limpo, legível, simples e organizado, porém o JavaScript também possui regras que devemos seguir. Com relação às variáveis, utilizamos elas através de nomes simbólicos para armazenar os valores / informações dentro dos nossos programas.

O nome das variáveis, são chamados de **identificadores** e precisam obedecer determinadas regras na sua criação, para não gerar erros no seu programa.

E uma boa prática é relacionar o nome de sua variável com o que ela irá armazenar. Um exemplo simples é se queremos armazenar nome completo de uma pessoa, podemos criar uma variável como:

```
let nomeCompleto = 'Daniel Porto';
```

Mas fique tranquilo, iremos mostrar o que pode ou não pode ao declararmos o identificador de uma variável!



## 1 Começar com uma letra

Um identificador Javascript pode começar com qualquer letra de “a” a “z”. E não pode haver nomes duplicados, ou seja, duas variáveis chamadas com o mesmo nome.

## 2 Podem iniciar com cifrão(\$) ou underline(\_)

Não podemos utilizar caracteres especiais ou termos espaços no identificador. Apenas é permitido o \$ e o \_.

## 3 Não podem começar com números

Não podemos iniciar os identificadores com números. Porém, podemos utilizar números ao longo do seu nome.

## 4 Não podem ser palavras reservadas

Não podemos utilizar palavras reservadas do JavaScript, são aquelas que já fazem parte da sintaxe da linguagem Javascript.

## 5 Padrão camelCase

Devido ao JavaScript ser case-sensitive, respeitamos o padrão camelCase, onde intercalamos palavras minúsculas com maiúsculas, nessa ordem.

```
JS declaracaoVariaveis.js • regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js > [?] $variavelTambemValida
1 let $variavelTambemValida CERTO
```

O VS Code identifica se houver um erro no nome da sua variável, ficando com um sublinhado vermelho.

```
JS declaracaoVariaveis.js 3 • regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js
1 let 1variavelTambemValida ERRADO
```



1

Nomes iniciando com letra

```
1 let soma
2 const registroNacional
3 let variavelSomatorio
```

Nomes com \$ e \_

Exemplos de identificadores permitidos: \$caixinha , variavelSomatorio  
Não permitido: @transgressora, variavel-errada

2

3

Nome com números

Exemplos corretos: valorProduto1, temTracao4Rodas  
Exemplo incorreto: 1NomeErrado

Exemplos de Palavra Reservada

Exemplos não permitidos: let, const, function, if

4

5

Exemplos de Padrão camelCase

Exemplos: javascriptImpressionador, variavelCriadaParaEsseExemplo,  
desenvolvimentoComJavascript, programacaoEhLegal





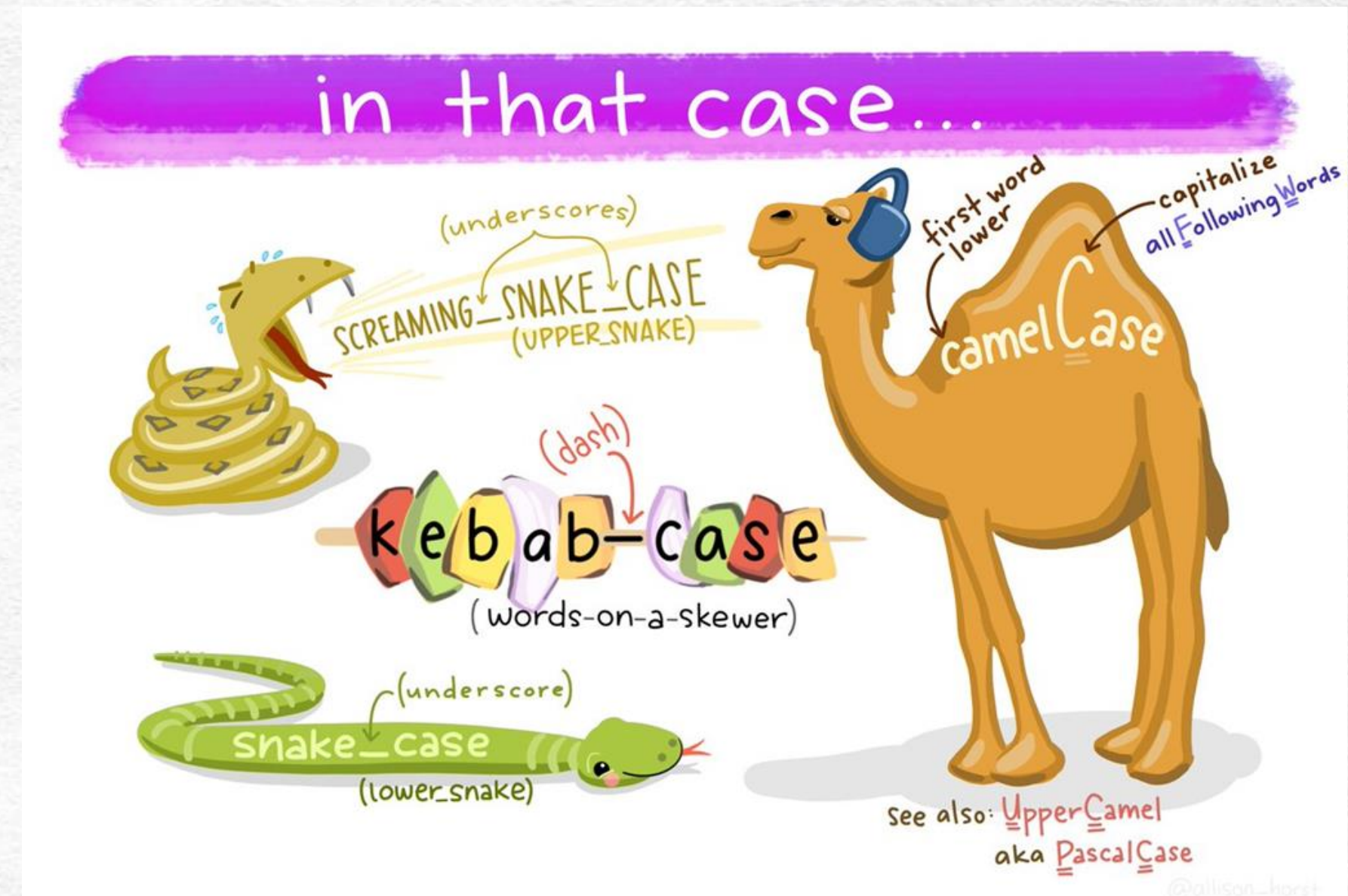
O padrão **camelCase** é uma convenção de nomenclatura utilizada em JavaScript e algumas outras linguagens, para facilitar a padronização e compreensão na leitura de códigos. O seu padrão é escrever palavras compostas ou frases, onde há primeira palavra é iniciada com letra minúscula e é intercalada com palavras de letra maiúscula e unidas sem espaços.

O nome camelCase é dado devido a aparência das costas de um camelo (camel em inglês).

Outro padrão que podemos utilizar é o **snakeCase** que é um estilo de escrita em que cada espaço é substituído por um sublinhado(\_) e a primeira letra de cada palavra é escrita de forma minúscula (lower\_snake), é utilizada convencionalmente para declarar nomes de **campos de bancos de dados, variáveis e funções**.

Existe a padronização com letras maiúsculas que é denominada **CAPA\_SNAKE** (upper\_snake) que é frequentemente utilizada como convenção para declarar **constantes**.

Além de algumas convenções como **kebab-case** (dash-case) que os espaços são substituídos com hífen(-), **PascalCase** que combina palavras colocando todas em letra maiúsculas, inclusive a primeira.



O padrão que iremos utilizar em JavaScript será o **camelCase**







## Tab (auto-completar)

Ao começar digitar um comando no terminal, como nome de arquivo, diretório ou até mesmo opções de comando. Ao apertar a tecla tab ela irá automaticamente completar o que você estava digitando ou mostrará todos os resultados possíveis.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\dec
```

Aperte tecla  
TAB

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
```



## Módulo 2 - Dados, Variáveis e Operações - Declarando e Definindo Variáveis (6 / 7)

```
JS declaracaoVariaveis.js 1 x regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js
1 let variavelNova = 'valorDessaVariavel';
2 variavelNova = 'novoValor';
3
4 const ultimaVariavelDaAula;
5
6 ultimaVariavelDaAula = 'finalmente coloquei um valor nessa variável';
7
8 console.log(variavelNova);
9 console.log(ultimaVariavelDaAula);
```

const declarations must be initialized. ts(1155)

View Problem (Alt+F8) No quick fixes available

```
JS declaracaoVariaveis.js x regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js > ...
1 const variavelNova = 'valorDessaVariavel';
2 variavelNova = 'novoValor';
3
4 console.log(variavelNova);
```

## Constantes

Recapitulando sobre as variáveis do tipo constantes, as regras de nomenclatura irão ser aplicadas a este tipo, mas lembre-se que uma constante precisa ser **inicializada**, ou seja, na hora de criarmos uma variável do tipo const, devemos atribuir o valor que queremos que ela guarde, e esta informação não poderá ser alterada futuramente, e ao executar seu código irá gerar erros de execução.

## Erros de execução

Ao executarmos no terminal um código com erro de declaração de constante, ele irá apontar o erro gerado, tipo e o local onde está o erro.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
C:\Users\danie\JavaScript Impressionador\modulo_2\declaracaoVariaveis.js:2
variavelNova = 'novoValor';
    ^
TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador\modulo_2\declaracaoVariaveis.js:2:14)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47

Node.js v18.13.0
```



Já a estrutura da variável do tipo `let`, nos permite tanto reatribuir valores para as variáveis já criadas, com também permite inicializar a variável sem atribuir um valor inicialmente:

```
let nomeVariavel;
```

O formato acima é a forma de declararmos uma variável sem atribuir um valor a ela.

E quando precisarmos dessa variável, iremos atribuir um valor a ela, em linhas futuras do nosso código, dessa forma:

```
nomeVariavel = ' Estou recebendo meu primeiro valor, pois sou uma LET';
```

Exemplos de declarações de variáveis tipo `let` e atribuições de valores:

```
JS declaracaoVariaveis.js X regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js > ...
1 let variavelNova = 'valorDessaVariavel';
2 variavelNova = 'novoValor';
3
4 let ultimaVariavelDaAula;
5
6 ultimaVariavelDaAula = 'finalmente coloquei um valor nessa variável';
7
8 console.log(variavelNova);
9 console.log(ultimaVariavelDaAula);
```

Saída do código acima ao ser executado:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
novoValor
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
novoValor
finalmente coloquei um valor nessa variável
PS C:\Users\danie\JavaScript Impressionador\modulo_2> |
```





### O que são operadores JavaScript?

Os **OPERADORES** são símbolos especiais que envolvem um ou mais operandos com a finalidade de produzir um determinado resultado, ou seja, símbolos que sinalizam ao nosso interpretador JavaScript a operação que desejamos realizar com os nossos dados. Pense em uma equação de matemática simples como a Adição, quando escrevemos  $2 + 3$ , esperamos que o **operador +** realize a soma de 2 com 3, gerando um resultado 5, correto?

Adição:  $2 + 3 = 5$

Os nossos operadores funcionarão da mesma maneira, recebendo uma tarefa para fornecer um resultado. Existem alguns tipos de operadores em JavaScript, no caso da **Adição (+)**, ela corresponde a um tipo de operador chamado **ARITMÉTICO**. E vamos falar mais um pouco sobre eles a seguir.



Operadores Aritméticos

Esse operadores podem ser velhos conhecidos das aulas de matemática. São usados para fazer operações com valores numéricos (sejam literais ou variáveis) como seus operandos e retornam um único valor numérico.

Na tabela ao lado, identificamos alguns operadores padrões como adição, subtração, multiplicação e divisão. Mas existem outros que complementam o nosso quadro de operadores, como a exponenciação, resto da divisão, incremento e decremento.

Descrição	Operador	Exemplo
ADIÇÃO	+	2 + 8
SUBTRAÇÃO	-	2 - 8
MULTIPLICAÇÃO	*	2 * 8
DIVISÃO	/	2 / 8
EXPONENCIAÇÃO (OU POTENCIALIZAÇÃO)	**	2 ** 8
RESTO DA DIVISÃO (OU MODULUS)	%	2 % 8
INCREMENTO	++	1++
DECREMENTO	--	1--



2

O operador de **Exponenciação (ou Potencialização)** calcula a base elevada à potência do expoente, ou seja, na matemática falávamos o 'número elevado a 3' (ao cubo).

$$2^3 \rightarrow 2^{**}3$$

O operador **Modulus (Módulo) ou Resto da divisão** retorna o inteiro restante da divisão dos dois operandos.

**12 % 5 retorna resto 2**

**10 % 5 retorna resto 0**

3

Os operadores de **Incremento ( ++ ) e Decremento ( -- )** diferente dos outros operadores precisam apenas de um número ou uma variável para realizar sua operação.

O Incremento adiciona um ( 1 ) ao seu operando e o Decremento subtrai um ( 1 ) ao seu operando.

```
let x = 10;  
x++;  
Retorna x = 11
```

```
let x = 10;  
x--;  
Retorna x = 9
```



Porém você pode estar se perguntando, será que os Operadores Aritméticos são utilizados apenas com valores numéricos?

A resposta é **não**, existem algumas exceções, como o operador de Adição ( + ) que conseguimos utilizar como **operador de textos**.

Esse operador é chamado de **CONCATENAÇÃO** é utilizado para unir / juntar textos em JavaScript.

O resultado que ele produz pode ser visto como uma soma da matemática, mas como funciona esse processo?

O **operador de Adição ( + ) avalia os valores** dos operandos e retorna um novo texto que é a junção desses valores. Vale lembrar que neste caso não precisam necessariamente ser todos do tipo texto, pois o JavaScript tentará transformar os valores para fazer a concatenação.

```
JS operadores.js X operadores.javascript.md
aula4 > JS operadores.js > ...
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 + texto2;
5  console.log(textoFinal);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Eu tenho 29 anos.
```

```
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Minha idade é ';
3  const numeroIdade = 29;
4
5  const textoFinal = texto1 + texto2 + numeroIdade;
6  console.log(textoFinal);
```

```
PS C:\Users\daniel\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Minha idade é 29
```



```
1  const texto1 = 'Oi, meu nome é Daniell! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 + texto2;
5
6
7  const variavelNumerica1 = 27;
8  const variavelNumerica2 = 35;
9
10 const resultadoMultiplicacao = variavelNumerica1 * variavelNumerica2;
11 console.log(textoFinal);
12 console.log(resultadoMultiplicacao);
```

Com os nossos operadores aritméticos as operações podem ser feitas diretamente dentro de uma variável ou entre variáveis, como nos exemplos acima, em que cada variável armazena um valor, seja numérico ou texto, e são realizadas as operações entre elas. E irão retornar o resultado das operações.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniell! Eu tenho 29 anos.
945
```



Mas precisamos tomar cuidado, pois nem todos os operadores podem ser utilizados em casos que não sejam valores numéricos. Como a Subtração ( - ) que se colocado no exemplo anterior, irá gerar um resultado do tipo **NaN**.

**NaN** é um acrônimo em inglês para **Not A Number**, ou seja, Não é um número. É um valor que representa um valor numérico indefinido ou irrepresentável.



```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
NaN
```

```
aula4 > JS operadores.js > [e] textoFinal
1  const texto1 = 'Oi, meu nome é Daniell! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 - texto2;
5  console.log(textoFinal);
```

No exemplo ao lado, estamos tentando subtrair um texto de outro, mas o JavaScript ao ler os valores, verifica que tal operação não é possível, porém ele não irá lançar um erro nesse caso, mas sim dará um retorno indicando que não foi possível após avaliar, transformar esses valores em texto.



E se eu contar que existe um tipo de operador que você vem utilizando desde do início das nossas aulas??

Isso mesmo que você leu!

O nosso **Operador de Atribuição**, o símbolo de **igual ( = )**. Utilizamos ele para atribuir um valor a uma variável.

```
1  const primeiraVariavel = 'essa é a minha variavel';
```

Independentemente do tipo de declaração de variável que escolhermos utilizar (let ou const), após nomearmos a nossa variável, utilizamos nosso **operador de atribuição =** para armazenar um novo valor ou substituir o valor original da variável.



```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
```

No caso acima, o arquivo que queremos executar **operadores.js**, está dentro da pasta **aula4**, que está dentro da pasta **modulo\_2** (indicado pelo **./**).



### Executando arquivos

Caso você esteja em outra pasta, ao invés da pasta que está o seu arquivo código, é só passar o caminho com os nomes do diretórios até chegar aonde seu programa está após o comando node. E o seu programa será executado de forma e no local correto.



Nesse momento, iremos revisar alguns conceitos para fortalecer a compreensão deles. Vamos então retornar um pouco sobre o tema **variáveis**.

Variáveis são "caixas" nas quais iremos armazenar informações ou valores, e ao declararmos elas, utilizamos os tipos **Const** ou **Let**. A escolha para cada tipo será de acordo com a sua mutabilidade, ou seja, se queremos que seja uma variável do tipo constante, aquela que não terá seu valor alterado, utilizamos o tipo **const**, ou se queremos alterar o seu valor ao longo do nosso programa, utilizamos o tipo **let**.

Após a declaração do tipo, precisamos nomear essa variável, como se colocássemos uma "etiqueta" na nossa caixa, para conseguir utilizar, invocar, chamar ou alterar ela no nosso programa.

Utilizamos o operador de atribuição **igual ( = )** , para atribuir um dado ou valor para essa variável, como exemplo um texto ( sempre entre **aspas simples ' '** ou **duplas " "**).

Exemplo de declaração de uma variável

```
JS aula5.js > ...  
1  let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
```



Independentemente do tipo de variável que criarmos ou utilizarmos, elas não podem ter a mesma nomenclatura, ou seja, não pode existir variáveis com o mesmo nome, o próprio VS Code irá mostrar o erro.

```
JS aula5.js > [❌] variavelCaixinha1
1  let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2  const variavelCaixinha1
```

Outro erro comum, e que também o VS Code irá mostrar, é quando não inicializamos uma variável do tipo const, ou seja, quando não atribuímos um valor no momento em que criamos a variável.

```
const variavelCaixinha2
```

Então **SEMPRE** que criarmos uma variável **constante**, precisamos atribuir o seu valor, porque se ela não inicializar com um valor na mesma linha que sua criação, ela nunca teria um valor atribuído devido a sua característica de ser imutável.

Agora criarmos uma variável do tipo let e não atribuirmos nenhum valor inicial, não teria problema, é como se estivéssemos preparando a nossa "caixinha" / variável para receber algum valor futuramente, pois o **tipo let é mutável**, e nos permite alterar o seu valor original.



Aprendemos que o JavaScript recebe instruções, em forma de comandos, para que execute uma tarefa. Um comando que utilizamos bastante até agora é o **console.log()**. Ele é o comando que permite imprimir os valores, que são passados entre seus parênteses ( ), na sua tela / terminal de texto.

```
JS aula5.js > ...
1 let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2 const variavelCaixinha2 = 'aqui se encontra o meu segundo texto'
3 |
4 console.log(variavelCaixinha1);
5 console.log(variavelCaixinha2)
```

**Instrução**

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2\aula5> node ./aula5.js
Esse é o meu primeiro texto do arquivo!
aqui se encontra o meu segundo texto
```

**Saída Terminal**

Agora vamos analisar um pouco mais de perto o código acima. Você percebeu que utilizamos **aspas simples ' '** em uma variável, depois **duplas " "**. Uma hora colocamos o **ponto e vírgula ( ; )**, outros momentos não. E vimos que o nosso programa funcionou sem nenhum erro, resultando na saída esperada do nosso console.log().

Sabemos que para utilizar textos, podemos utilizar os dois tipos de aspas, e que o nosso ponto e vírgula é opcional.

**Mas então qual é o Certo e Errado??**



O JavaScript é muito flexível, em termos de algumas regrinhas, como o ponto e vírgula e as aspas. Ao rodarmos um programa em JavaScript o seu interpretador, consegue identificar quando começa e termina uma linha de código, que tanto as aspas simples como duplas são utilizados para informações do tipo texto. Então para ele, tudo estará certo, o importante é seguir um **PADRÃO**.

Então para entender como escrever um código com boa qualidade, vamos falar sobre **PADRONIZAÇÃO DE CÓDIGO**.

A Padronização de código em JavaScript é manter a **consistência do estilo de código** que está escrevendo, pensando que futuramente você precise revisitar o programa que criou, ou até mesmo criar um projeto em Pair Programming (você com mais pessoas), o que é muito comum na vida diária de um programador, para isso vocês precisam ter uma comunicação única em termos de código.

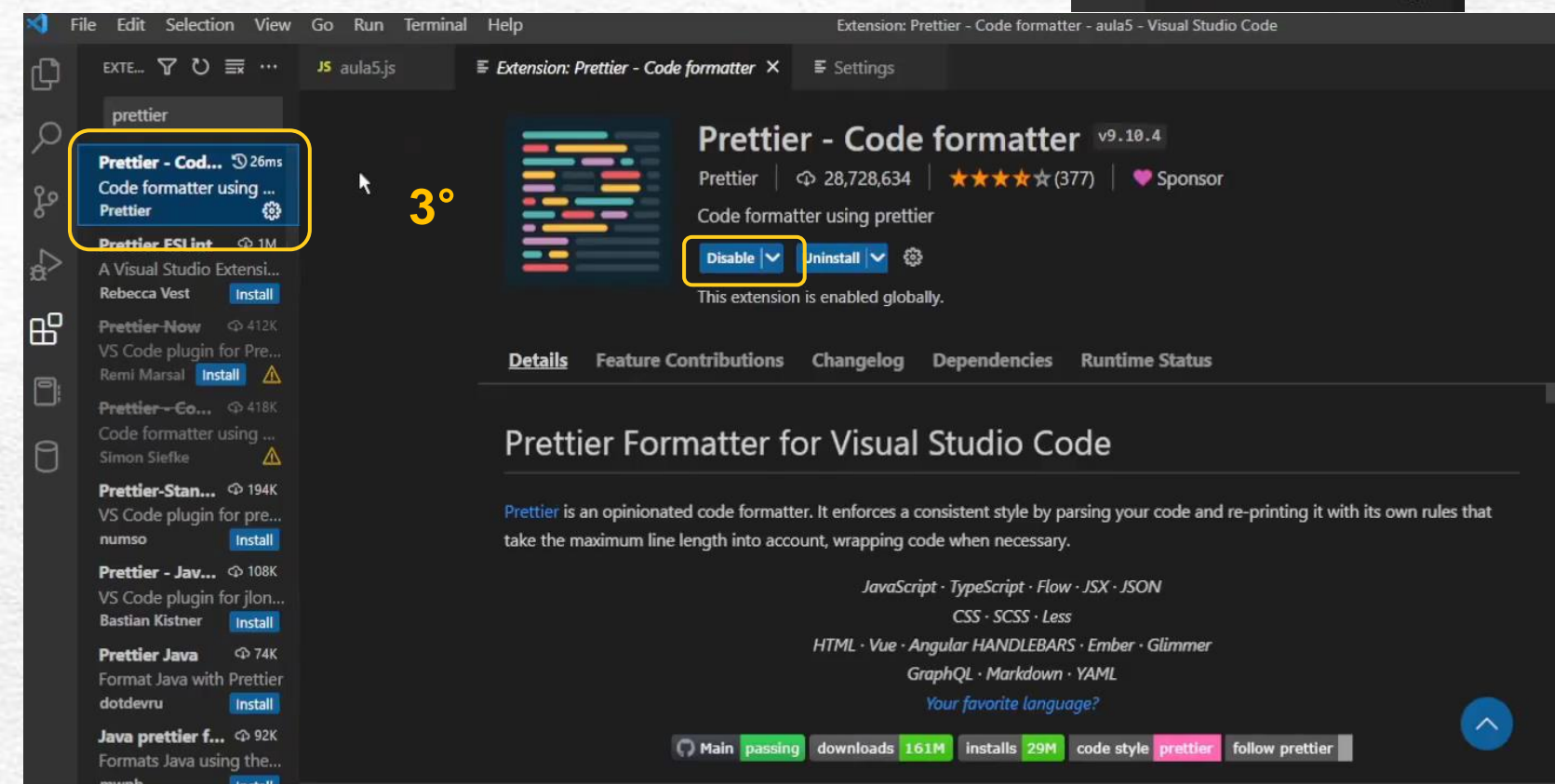
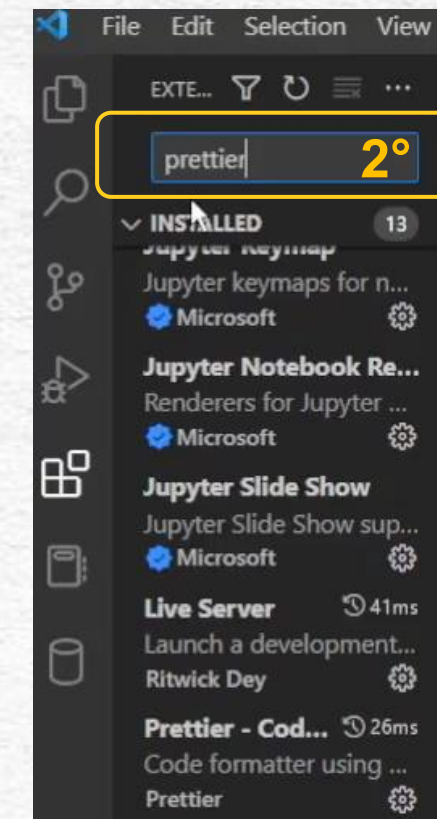
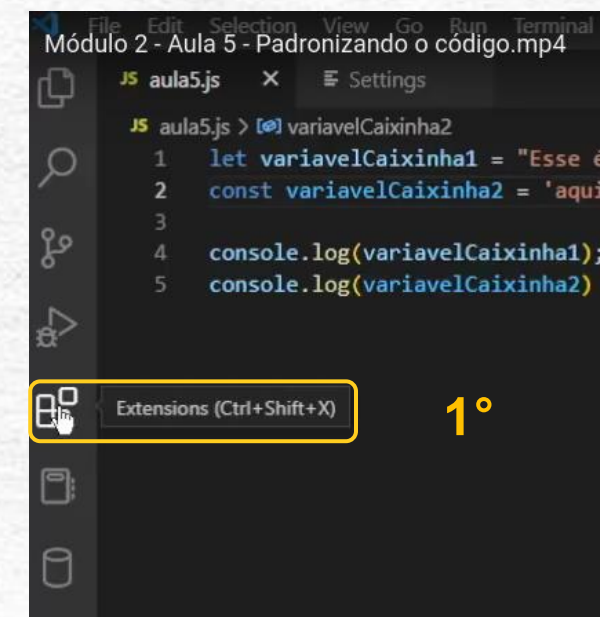
As vantagens da padronização é a **Legibilidade** do código, a fácil **Manutenção**, **Uniformidade** e melhores práticas ao desenvolver seu programa, o que aumenta a qualidade do código.



Para facilitar a padronização o VS Code, possui uma ferramenta chamada Extensões, onde conseguimos baixar facilitadores de código, legibilidade, formatação, auto preenchimento, entre muitos outros.

Hoje vamos ver a **Extensão Prettier - Code formatter**, que é uma extensão de formatação de código, ele permite você deixar o seu código em um formato pré-estabelecido, ou seja, em um padrão.

1. Clique no ícone 'Extensions', do lado direito do VS Code.
2. Digite no campo de busca: Prettier ( o primeiro já é a extensão que queremos).
3. Clique no primeiro resultado, e clique em instalar (caso ainda não tenha).







### Formatação do código

Para utilizar a **Extensão Prettier - Code formatter**, O VS Code possui um atalho – Alt + Shift + F, que é utilizado para formatação do código e a extensão torna o atalho mais versátil e mais funcional para a linguagem JavaScript.

```
JS aula5.js > ...
1  let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2  const variavelCaixinha2 = 'aqui se encontra o meu segundo texto'
3  |
4  console.log(variavelCaixinha1);
5  console.log(variavelCaixinha2)
```

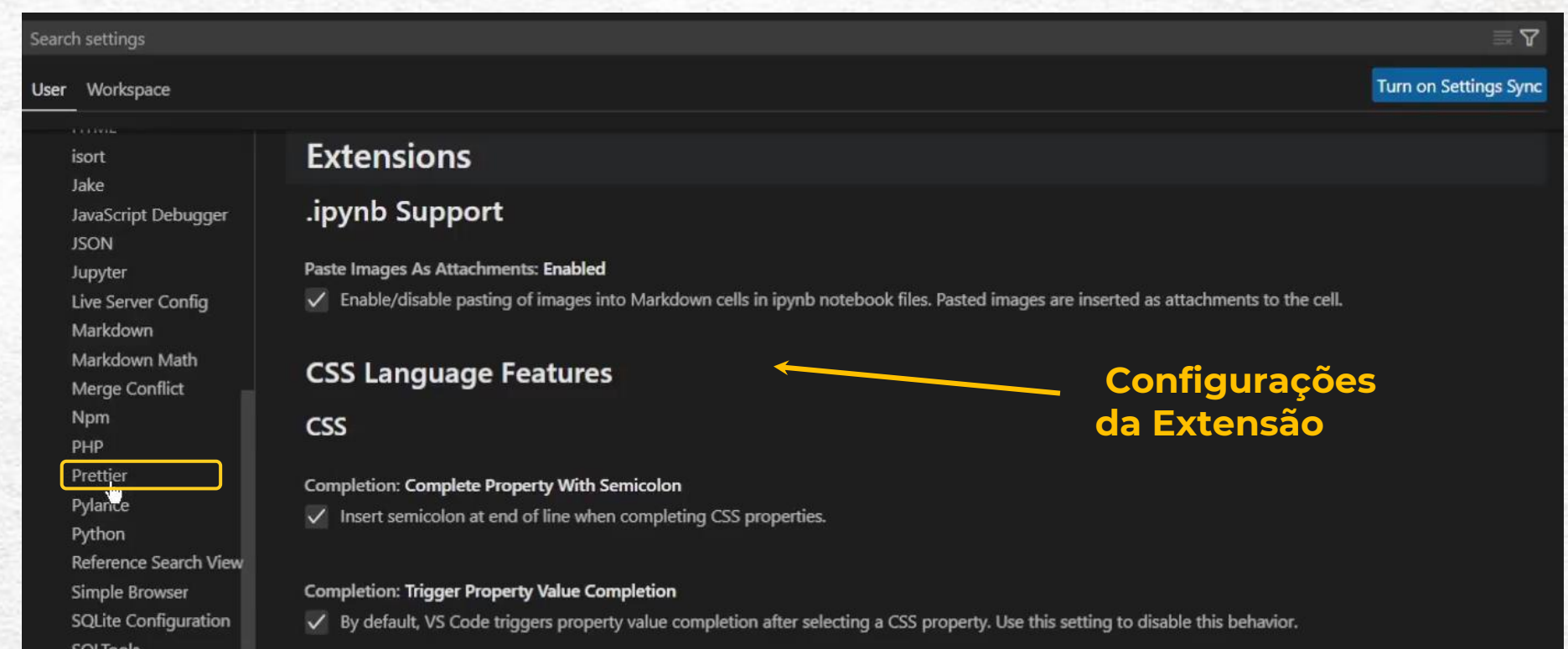
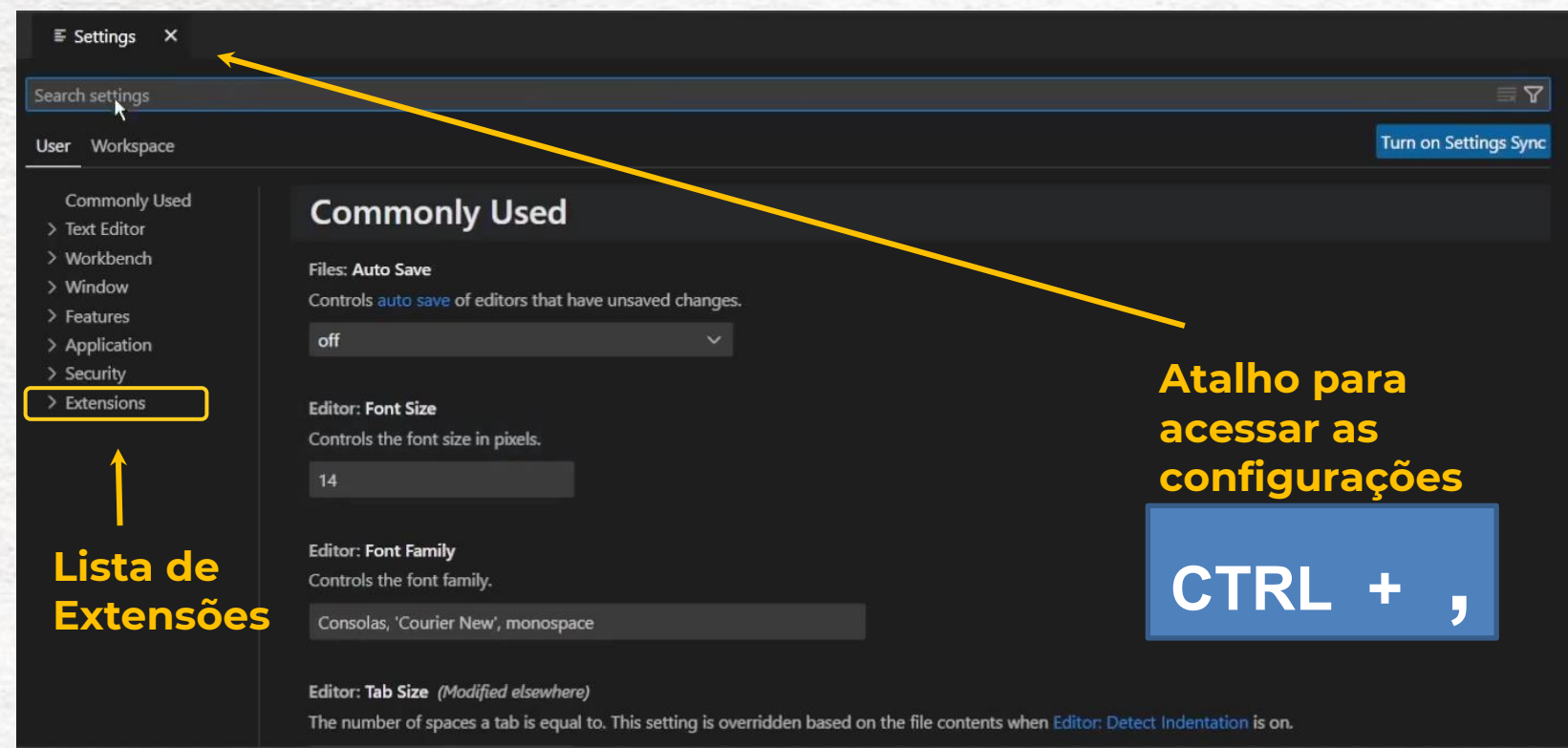
ALT + SHIFT + F

```
1  let variavelCaixinha1 = 'Esse é o meu primeiro texto do arquivo!';
2  const variavelCaixinha2 = 'aqui se encontra o meu segundo texto';
3
4  console.log(variavelCaixinha1);
5  console.log(variavelCaixinha2);
```



Para você acessar as **configurações do VS Code** e verificar qual padronização está sendo utilizada ou configurar o padrão que você quer que o VS Code formate o seu código, você irá acessar através do atalho ' **CTRL + vírgula ( , )** '. Dentro das configurações, clicando em **Extensões**, toda a lista de extensões salvas no seu VS Code irá aparecer.

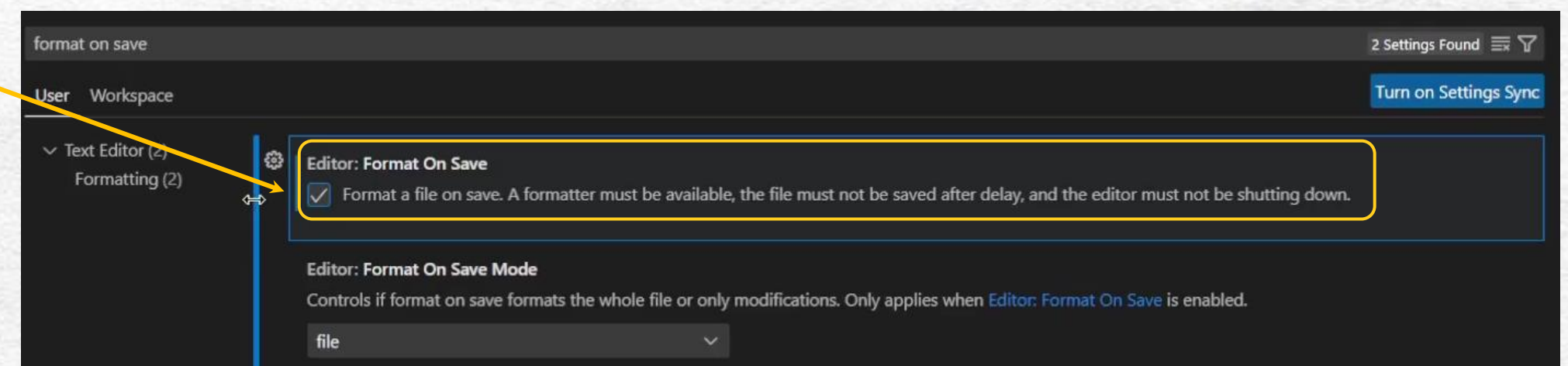
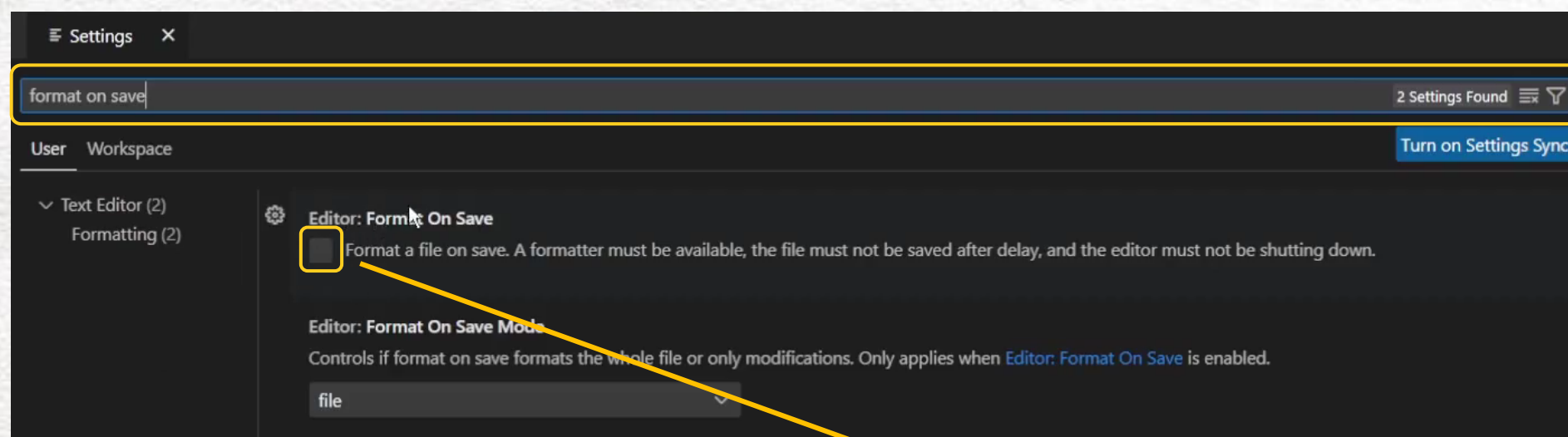
Nesse momento iremos localizar o **Prettier** e clicar nele mostrando ao lado direito todas as configurações e opções que a extensão possui. Não se preocupe se não sabe tudo o que está ali, ou em quais opções você deve mexer, o Prettier já vem como padrão algumas configurações básicas, e ao longo de sua jornada ficará mais fácil entender qual é o padrão de código que irá seguir ( sabendo que será diferente dependendo do ambiente de trabalho ou projeto).





Uma recomendação para facilitar a formatação do seu código, é deixar como padrão a formatação no momento em que salvar o arquivo. Isso facilita o uso da ferramenta, além de economizar tempo e não correr o risco de esquecer de formatar o seu código.

Para isso, ainda nas configurações do VS Code, procuraremos por '**format on save**' na barra de busca, é só verificar se a configuração está ativa, senão basta clicar no **seletor Format On Save** e pronto! Toda vez que **salvar** o seu código (**CTRL + S**), o VS Code irá formatar.





Anteriormente iniciamos o conceito de **Variáveis** em JavaScript e como realizar suas declarações e criações, e utilizamos bastante o comando `console.log()` para visualizarmos elas.

Agora vamos entender um pouco mais sobre os Tipos de Dados que o JavaScript possui. E para isso precisamos entender o que é um **DADO**.



Um **DADO**, em qualquer linguagem de programação, é uma unidade ou elemento de informação que pode ser acessado através de algum identificador. No caso das variáveis, o que elas guardam ( um dado/ uma informação ) que poderá ser acessada quando utilizarmos essas variáveis.

Em JavaScript um Dado é a **palavra de comunicação**, onde ele irá processar o que está no dentro dele e realizar alguma coisa. Alguns dados você já utilizou nas aulas anteriores que são os dados de textos e números. Eles pertencem a tipos de dados diferentes e vamos entender como eles funcionam a seguir.



### TIPO STRING

Trata-se do tipo de dado que são sequências de caracteres alfanuméricos (letras, números e/ou símbolos), ou seja, **Strings** são úteis para guardar dados que podem ser representados em forma de texto. Em JavaScript, uma String sempre estará entre **aspas simples ( ' )**, **aspas duplas ( " )** ou entre **crases ( ` )**.

Até o momento, você percebeu que apenas utilizamos atribuições de variáveis com conteúdo textual com as aspas, e sabemos que não existe diferença para o JavaScript em utilizar aspas simples ou duplas e o retorno do nosso conteúdo será igual, mas o importante é manter um padrão de formatação do seu código.

```
const texto1 = 'Aqui temos uma string normal.';
const texto2 =
  'Já essa, apesar das aspas duplas ao invés das simples, também é uma string normal';

const numeroQueQueroRegistrar = 31472;
```

Agora, a utilização de crases ( ` ) muda um pouco o cenário da string. Quando utilizamos a crase para envolver nossa String, estamos criando uma **string template ( ou template literals)**, que é uma feature do ES6 ( JavaScript mais atual) que permite criar strings complexas mais facilmente, pois ela possui algumas funcionalidades que simplificam essas strings.



## String Template ( Template Literals)

Vamos lembrar um pouco sobre a **Concatenação de strings e variáveis** em JavaScript, que era como estávamos unindo textos diferentes:

```
JS operadores.js × operadores.Javascript.md
aula4 > JS operadores.js > ...
1  const texto1 = 'Oi, meu nome é Daniell! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 + texto2;
5  console.log(textoFinal);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniell! Eu tenho 29 anos.
```

O formato de concatenar não irá apresentar erros na execução do seu código. Mas imagine o cenário no qual você esqueça de colocar um espaço ou pontuação em algumas das frases e ela ter um retorno nesse formato: **Oi, meu nome é DanielEuTenho 29 anos.**

O **String Template**, facilita esse processo de união de strings, de uma forma mais simples as strings mais complexas. A sintaxe pede o uso do sinal de crases ( ` ) no envolto da frase e ao utilizarmos o conteúdo de variáveis colocamos o nome dela dentro de `${ }`.

```
const numeroQueQueroRegistrar = 31472;
const stringTemplate = `O número que decidi registrar foi o ${numeroQueQueroRegistrar}`;

//console.log(stringTemplate) fará aparecer na tela o seguinte texto:
// O número que decidi registrar foi o 31472
```



O formato da **String Template** também pode ser utilizado para adicionar expressão dentro das chaves, como **`${a + b}`**.

```
JS strings.js > ...
1  const a = 10;
2  const b = 20;
3  const stringTemplate = `O resultado da soma é ${a+b} `;
4
5  console.log(stringTemplate);
```

O resultado da soma é 30

Você consegue adicionar quebras de linhas ( colocar textos em diferentes linhas – um em cima do outro) sem a necessidade de concatená-las com o Operador **+** e **`\n`** (codificação de quebra de linha).

```
JS strings.js > ...
1  const nome = 'Daniel';
2  const idade = 29;
3  const stringTemplate = `Oi, meu nome é ${nome}!
4  Minha idade é ${idade}`;
5
6  console.log(stringTemplate);
```

Oi, meu nome é Daniel!  
Minha idade é 29



### TIPO NUMBER (Número)

Trata-se do tipo de dado que representa **informações numéricas**. Sejam números inteiros positivos e negativos (-3, 0 ,1, 12, 187...) ou números decimais (23.456), todos são entendidos como dado do **tipo Number**.

Uma curiosidade sobre o tipo Number no JavaScript, é que todo número é um tipo Number, diferente de outras linguagens de programação. Por exemplo, os números decimais, são conhecidos como números de pontos flutuantes (ou **floats**) e eles não seriam caracterizados como tipo Number, mas no caso do JavaScript não.

Vamos aprender ao longo do curso, que os dados possuem **propriedades e métodos** para trabalharmos ao longo do nosso código, e essa informação é importante, pois independentemente do número, ele possuirá as propriedades que existem no tipo Number.

```
const numero1 = 27;  
const numero2 = 23.25;  
const numero3 = -40;  
const total = numero1 + numero2 + numero3;  
  
console.log(total);  
  
//10.25
```

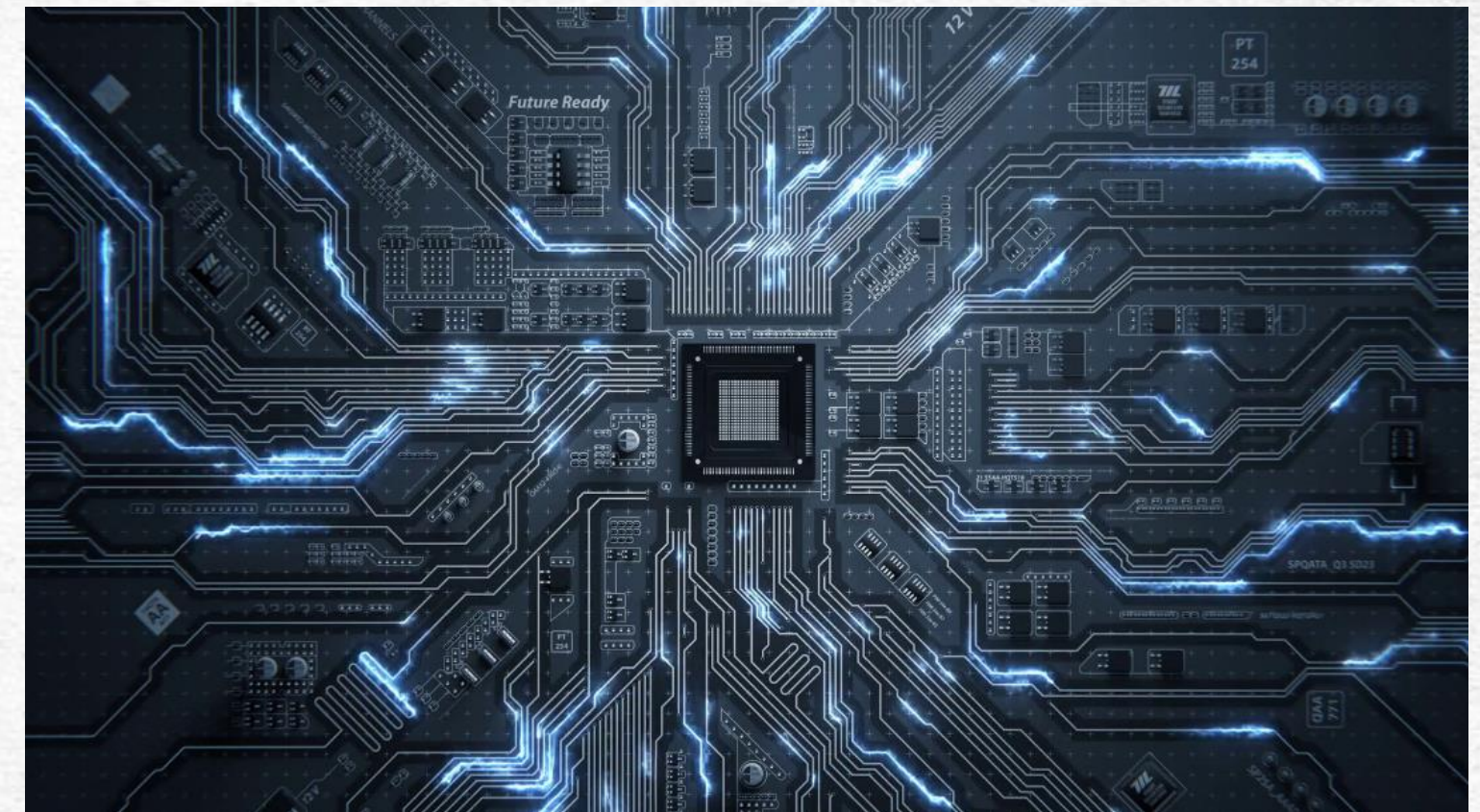


Ao falarmos de Dados, precisamos entender primeiro como um computador irá processá-los. E para isso, vamos falar sobre os conceitos de **Números Binários e Bits**.

O **computador** sendo inspecionado bem a fundo, pode ser entendido como nada mais de que **circuitos elétricos**, ou seja, uma placa eletrônica onde passam vários caminhos de corrente elétrica, podemos entender que o computador nada mais é que um desses circuitos, onde a **inteligência do computador é processar e gerenciar esses caminhos de energia** (pode passar energia ou não).

Para gerenciar esse caminho, o computador utiliza apenas dois números para processar esse funcionamento. São o **número 0**, que indica que **não há energia/corrente** e o **número 1**, que indica que **há sim energia/corrente**. Essa informação binária (descrita por 0 ou 1) é chamada de **BIT**.

**Bit** é a menor unidade de informação que o computador pode utilizar para trabalhar. Pode assumir somente dois valores: 0 ou 1 (verdadeiro ou falso).





### Então como o Computador consegue trabalhar com números maiores do que 1?

Para descrevermos números maiores, precisamos agrupar os Bits, ou seja, um Bit significa 0 ou 1, dois Bits me dão mais possibilidades gerando mais números :

O uso de dois algarismos binários, conseguimos descrever 4 números diferentes ( 0 a 3).

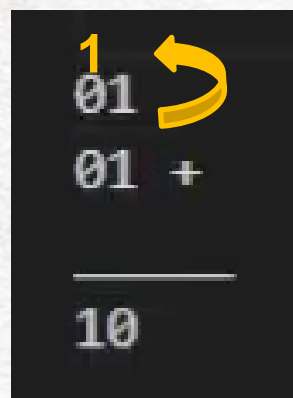
00 => número 0

01 => número 1

10 => número 2

11 => número 3

Isso acontece porque basicamente estamos somando os únicos dois valores que um Bit pode ter, vejamos o exemplo do 2:



É como se quisemos somar os números 1, e para isso aumentamos a casa da dezena dele, como quando fazemos a soma  $7 + 9 = 16$ , onde possuímos dois números na unidade e ao somar, colocamos um número na casa da dezena.

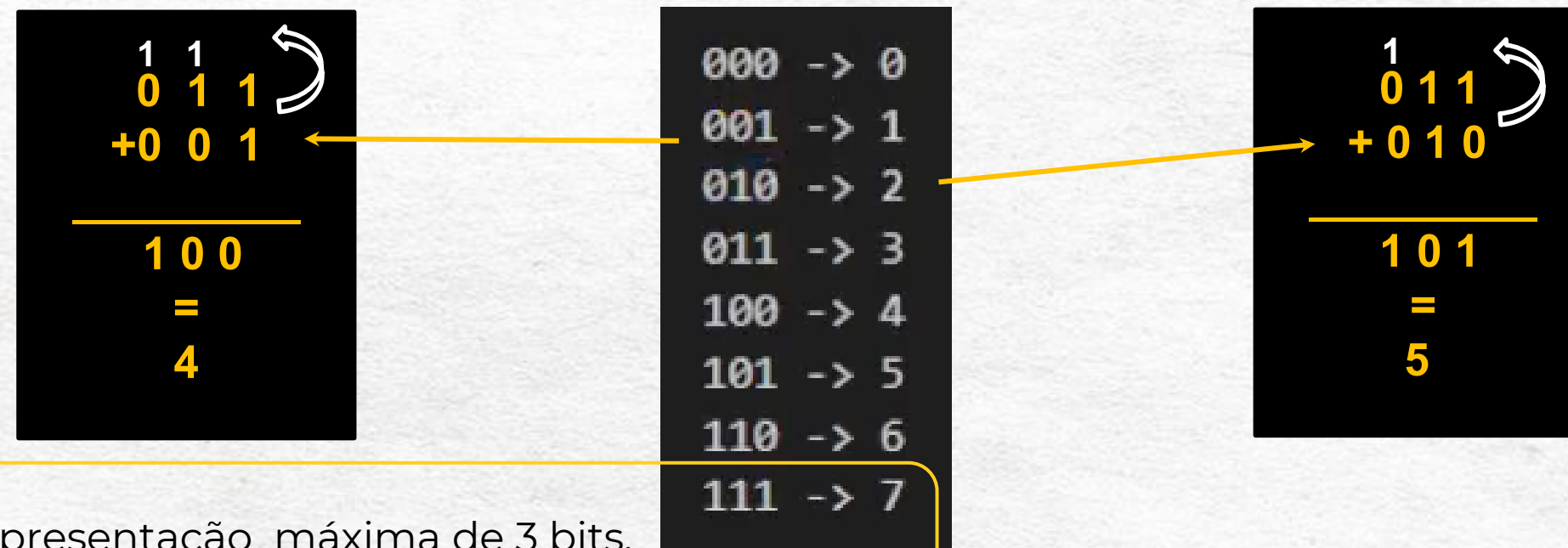
É uma representação diferente do número 2, pois é uma representação binária dele, porém  $1 + 1$  continua sendo 2.



Vamos perceber que existe um padrão para representação binária dos números, onde o agrupamento de mais Bits descreve mais números. Para saber quantos números conseguiremos descrever com o agrupamento, basta elevar o número 2 com o número de Bits que deseja agrupar e o resultado é a quantidade de números que conseguimos representar :

- com 2 bits, tivemos 4 números diferentes  $\Rightarrow 4 = 2^2$
- com 3 bits, tivemos 8 números diferentes  $\Rightarrow 8 = 2^3$
- com 4 bits, tivemos 16 números diferentes  $\Rightarrow 16 = 2^4$
- com 5 bits, tivemos 32 números diferentes  $\Rightarrow 32 = 2^5$
- com n bits, teremos  $2^n$  números diferentes

Uma forma simplificada de entendermos a representação binária é que a soma de  $1 + 1 = 0$  e  $0 + 1 = 1$ . Então quando falamos de um agrupamento de 3 bits, iremos representar os números da seguinte forma:

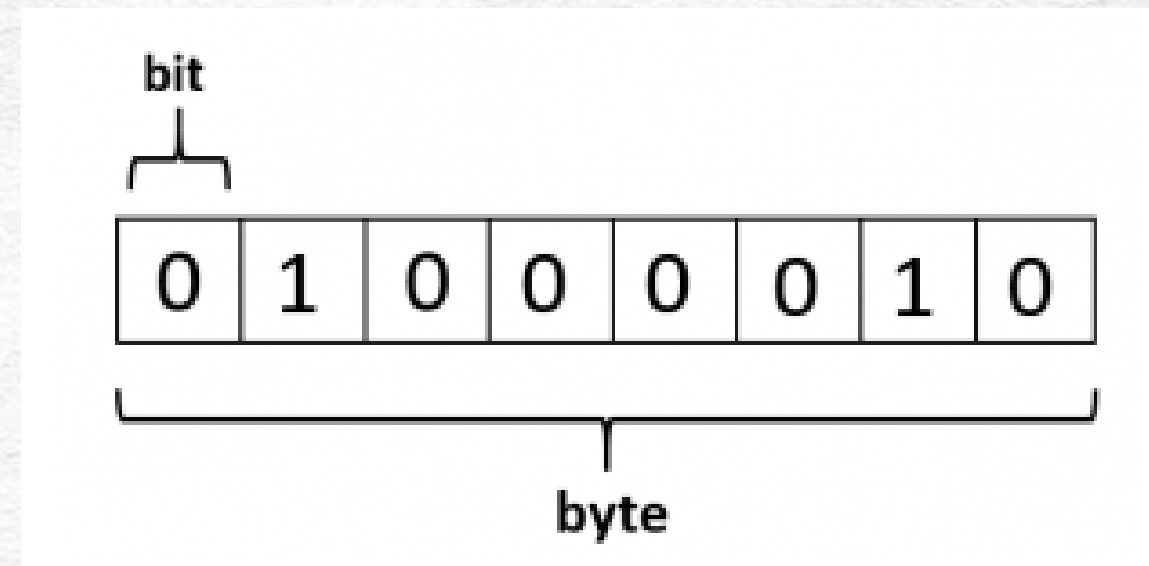


Número 7 é a representação máxima de 3 bits.  
Precisamos inserir um 4º bit para descrever o número 8.



A partir da junção de 8 bits, foi criado uma nova unidade para facilitar o processo, chamado **BYTE**.

Então um **BYTE** representa  $2^8 = 256$  números.



Agora para representarmos números negativos, podemos dedicar 1 bit para o sinal, ou seja, em um agrupamento de 3 bits, podemos dizer que 1 bit será o sinal, e os outros 2 bits irão representar os números, nesse caso ( -1, -2, -3).

111 sendo o número -3  
011 sendo o número +3