

---

# Dados, Tipos, Variáveis e Depuração em JavaScript

UFJF - DCC202 - Desenvolvimento Web

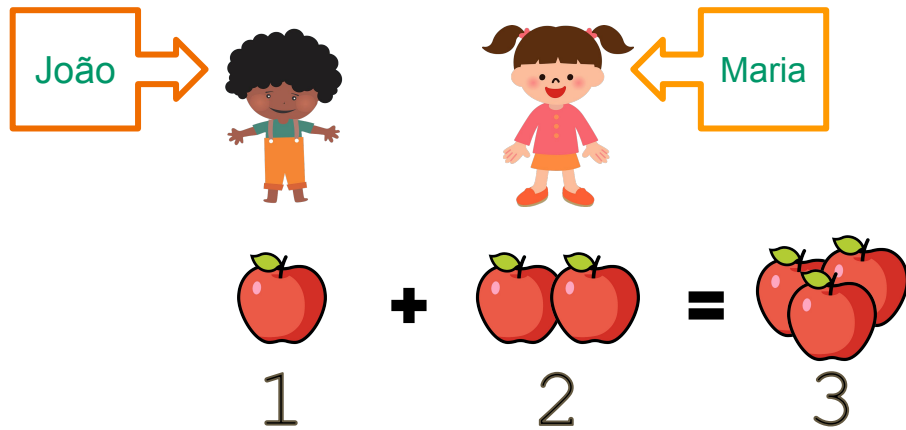
Prof. Igor Knop [igor.knop@ufjf.br](mailto:igor.knop@ufjf.br)



<https://bit.ly/3PhBQQg>

# Programação de computadores

Em um nível mais básico, o computador só trabalha com dados. Dados são representações de quantidades, na forma de valores discretos.



Ao utilizar uma linguagem de programação para representar problemas do mundo real na forma de dados, e descrever quais operações sobre eles fazem sentido, que vamos buscar realizar diversos tipos de trabalho com um computador.

# JavaScript

Para trocar o comportamento de um computador, é necessário descrever quais operações ele realiza em uma linguagem de programação. Existem centenas de linguagens diferentes, com propósitos diferentes.

Para a programação web, uma das linguagens de programação mais populares é o JavaScript, hoje normatizado como ECMAScript.

O JavaScript pode ser executado tanto do lado do cliente, nos navegadores quanto no lado do servidor, com ambientes de execução dedicados.



*client side*  
*"front end"*



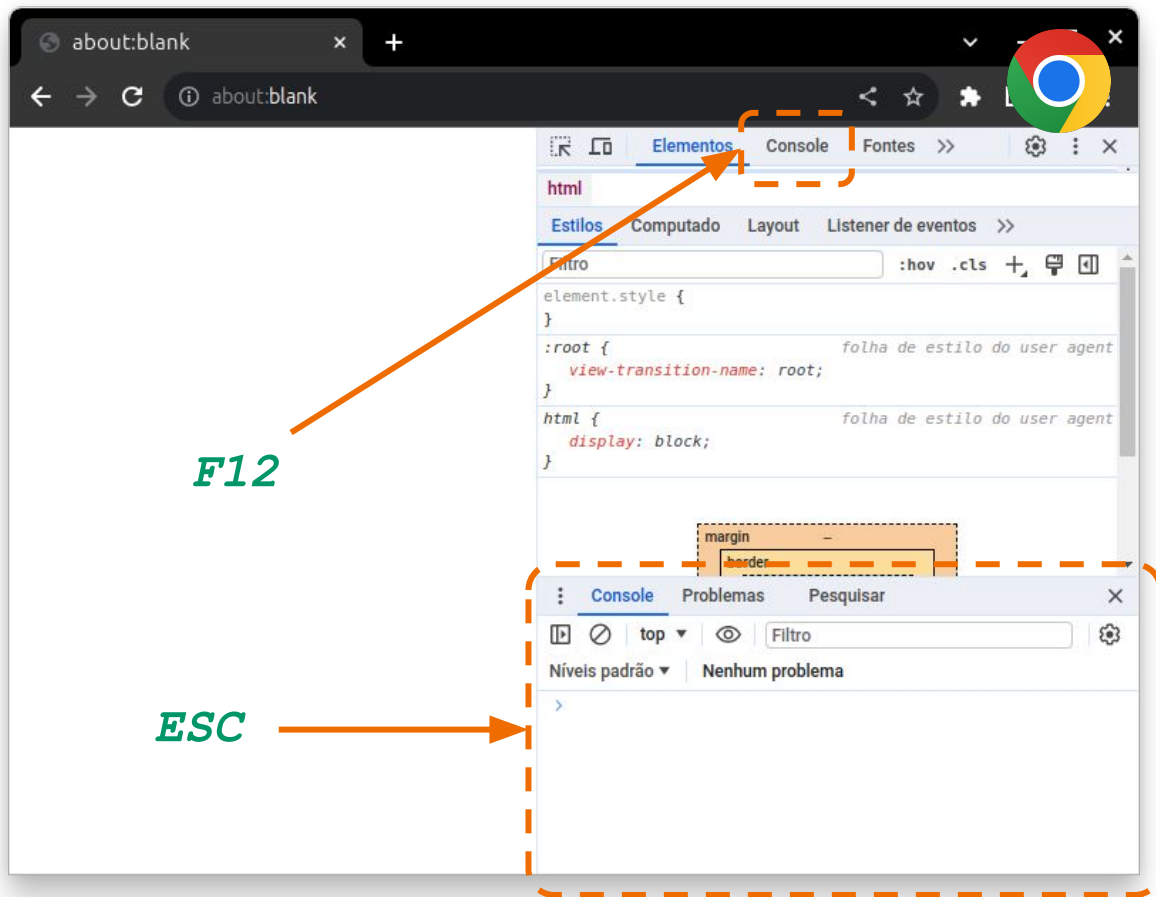
*server side*  
*"back end"*

## Ambiente de execução: Navegador

Os navegadores modernos fornecem um ambiente isolado para executar código JavaScript.

No Chrome, você pode ter acesso às ferramentas de desenvolvedor apertando a tecla **F12**.

Estamos particularmente interessados no Console, que tem uma aba própria ou pode ser aberto com a tecla **ESC**.



## Ambiente de execução: Console do Navegador

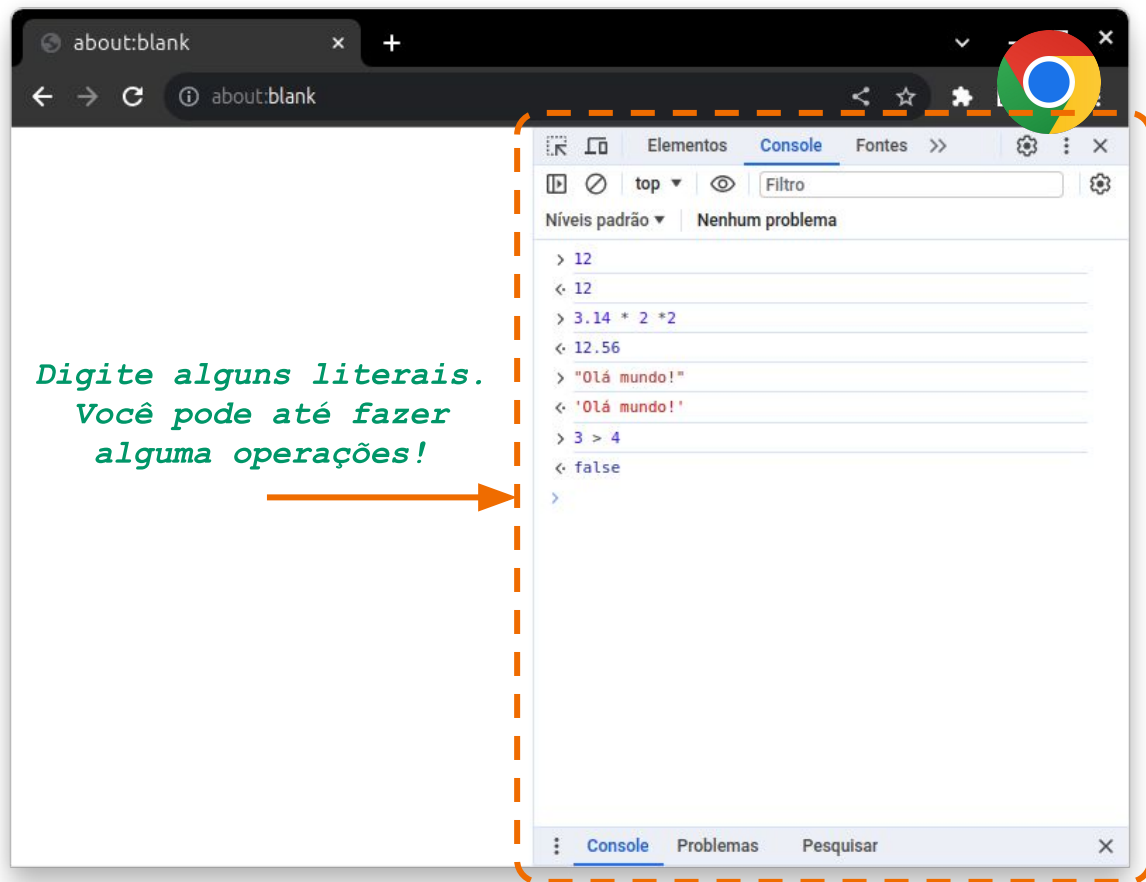
O console do navegador é a primeira opção que temos de começar a interagir com o JavaScript direto no navegador.

Ele funciona no formato *read-eval-print loop* (**REPL**). O que pode ser entendido como laço de leitura, interpretação e impressão.

Você digita um texto na linguagem JavaScript, o navegador interpreta e imprime o resultado ou uma mensagem de erro.

Importante destacar que o navegador que está executando o código, isolado para cada página.

*Digite alguns literais.  
Você pode até fazer  
alguma operações!*



## Ambiente de execução: NodeJS

O NodeJS, quando instalado no seu computador, também tem uma interface REPL para experimentar.

Você pode abrir um terminal e, se a instalação estiver correta, o comando **node** vai colocá-lo no console.

Para sair do console, basta apertar **CTRL+C** duas vezes, **CTRL+D** ou o comando **.exit**



```
igorknop@zicaA62: ~  
(base) igorknop@zicaA62:~$ node  
Welcome to Node.js v19.9.0.  
Type ".help" for more information.  
> 2  
2  
> 5 + 3  
8  
> 3.14 * 7 * 7  
153.86  
> "Olá mundo! 🌞  
'Olá mundo! 🌞  
> 5 < 10  
true  
>  
(To exit, press Ctrl+C again or Ctrl+D or type .exit)  
>  
(base) igorknop@zicaA62:~$
```

# Dados na forma de literais

Um valor literal é a primeira forma de representar os dados de forma escrita. Internamente esses valores são guardados na forma binária, mas as linguagens de programação possuem representações para cada uma delas.

Em JavaScript temos os seguintes tipos de dados, que chamamos de tipos primitivos:

- Números: 1 5 3.14 12e4 7e-9
- Textos (*strings*) 'Bom dia' "Boa tarde" `Boa noite` '4' '#' `😊`
- Lógicos (*booleanos*) true false
- Nulo null
- Não definido undefined

Nota: Ainda temos os tipos *BigInt* e *Symbol*, mas deixa para depois...

# Tipo: Números

Os números diretamente representam quantidades da forma que as entendemos. Essas quantidades podem ser discretas ou contínuas.

São discretos na forma de valores inteiros, quando representamos coisas que não têm sentido ser divididas. Por exemplo: Ana tem 2 irmãos e mora na casa 34. Não é possível ter 2 irmãos e meio, se ela ganhar mais um irmão, passará direto de 2 para 3 irmãos.

Para representar coisas e conceitos contínuos, valores reais, que há uma quantidade de valores infinita entre dois valores devido à sua parte fracionária. Por exemplo, Maria mediu 1,45m e pesa 36,3Kg. Pode ser que se usarmos instrumentos mais precisos, podemos ver que Maria mede na realidade 1,446679m hoje, mas amanhã vai medir 1,446834m em função de seu crescimento.

Em JavaScript utiliza-se o mesmo tipo para representar números inteiros e reais. Nos números reais, é importante lembrar que se usa o ponto para a parte fracionária pois segue o padrão dos países de língua inglesa.



# Expressões com Números

Um dos principais usos de números é para realizar cálculos aritméticos. Uma expressão é um arranjo de números e operadores que resulta em um novo número.

Em nosso exemplo anterior, se eu quero saber quantas maçãs João e Maria ter em conjunto, eu posso escrever uma expressão `1 + 2` que são as quantidades de cada um e esperar que o próprio JavaScript calcule e me dê o resultado.

Há uma grande quantidade de operadores sobre números, mas as operações de soma (+), subtração (-), produto (\*), divisão (/), resto da divisão (%) e exponenciação (\*\*) são as básicas da aritmética e devemos nos acostumar com seu símbolo na linguagem. O uso de parênteses também auxilia na leitura e garantia de ordem na interpretação.

<code>1 + 2</code>	<code>resulta em 3</code>
<code>3 * 7</code>	<code>resulta em 21</code>
<code>3.14 * (5 ** 2)</code>	<code>resulta em 78.5</code>

# Tipo: Texto (*string*)

Os textos representam símbolos diversos, sozinhos ou colocados em uma “fita”. São associados com nossas letras não só usadas para formar palavras e frases, como também pontuação, emojis e até símbolos que não possuem uma impressão visível.

Internamente são armazenados como uma série de números, que podem ser acessados individualmente, mas esse trabalho é todo deixado a cargo do JavaScript na maioria das vezes.

Um literal de texto deve ser sempre representado entre aspas simples, duplas ou crases. Dessa forma, `'A'`, `"A"` ou ``A`` são três formas de representar o mesmo literal de texto do símbolo A, uma única letra. Analogamente, `'Juiz de Fora'`, `"Juiz de Fora"` e ``Juiz de Fora`` representam o mesmo literal de um texto com 12 letras (espaços em branco são símbolos também!).

# Expressões com textos

Textos também podem ser utilizados em expressões. O resultado de uma expressão de texto, quase sempre, é um novo texto.

O operador + quando tem um de seus operandos como texto, se comporta como operador de concatenação. Concatenar dois textos é gerar um terceiro que é junção dos dois. Por exemplo:

"Bom dia, " + "Juiz de Fora!"                      resulta em "Bom dia, Juiz de Fora!"

'Boa noite, ' + "UFJF" + '! '                      resulta em "Boa noite, UFJF!"

'A soma de ' + 2 + ' e ' + 3 + ' é ' + (2 + 3) + ' .'                      resulta em "A soma de 2 e 3 é 5."

Avalie os exemplos com cuidado. É possível concatenar vários textos em uma mesma expressão, independente das aspas de suas partes.

Também é possível concatenar textos com outros tipos, que serão convertidos em texto em um processo que chamamos de *coerção de tipo*, que será visto em detalhes posteriormente.

## Expressões com textos (2)

A concatenação permite inserir valores no texto em posições importantes para servir como resultado de cálculos e informações para a pessoa que estiver usando nosso programa. Mas ela pode acabar ficando complexa demais como no último exemplo acima.

```
'A soma de ' + 2 + ' e ' + 3 + ' é ' + (2 + 3) + '.'
```

resulta em "A soma de 2 e 3 é 5."

Para auxiliar em situações assim, o literal de texto quando criado com crases é chamado de modelo ou *template*, permite usar quebras de linha e também colocar expressões inteiras dentro do texto para serem resolvidas. Esses pontos de interpolação devem estar dentro de `{}`.

Por exemplo:

```
`Um círculo de raio ${3} tem ${3.14 * (3 ** 2)} de área.`
```

resulta em "Um círculo de raio 3 tem 28.26 de área."

## Expressões com textos (3)

Os textos podem conter símbolos especiais como os espaços em branco, quebras de linha, as próprias aspas utilizadas para a representação do literal e código de codificação de caracteres. Para esses casos, utilizamos uma barra invertida (\) para realizar o *escapamento* de símbolos especiais. Por exemplo:

"Esse texto está \"entre aspas\"." resulta em "Esse texto está "entre aspas"."

'Matriz:\n\tA\tB\n\tC\tD' resulta em "Matriz:

A B

C D"

'Estou feliz! \u{1F600}' resulta em "Estou feliz! 😊"

Avalie os exemplos com cuidado. É possível concatenar vários textos em uma mesma expressão, independente das aspas de suas partes.

Também é possível concatenar textos com outros tipos, que serão convertidos em texto em um processo que chamamos de *coerção de tipo*, que será visto em detalhes posteriormente.

# Tipo: lógico (boolean)

Os tipos lógicos são os baseados em lógica de Boole e representam apenas dois valores: `true` (verdadeiro) ou `false` (falso). Atenção: são literais sem aspas!

Podemos utilizá-los para representar situações que só podem ter dois valores. Por exemplo: Se mercearia que vai comprar as maçãs de João e Maria está aberta ou não. Se a mercearia não estiver aberta, nem adianta juntar as maçãs...

Esse tipo é muito utilizado em expressões lógicas como igual a (`==`), diferente de (`!=`), maior que (`>`), menor ou igual a (`<=`) e outros, com operadores relacionais que comparam outros valores. Serão nossa principal forma de deixar nossos programas capazes de tomar decisões no futuro. Por exemplo:

<code>2 &gt; 3</code>	resulta em <code>false</code>
<code>3.14 &lt;= 5</code>	resulta em <code>true</code>
<code>5 == 10 / 2</code>	resulta em <code>true</code>
<code>'UFJF' != 'UFJF'</code>	resulta em <code>false</code>

# Expressões *versus* Afirmações (*statements*)

Um dado é representado por um valor literal seguindo as regras de uma linguagem de programação para um determinado tipo.

Um fragmento de código que produz um valor é uma *expressão*. As expressões podem ser combinadas para criar uma expressão maior, mas que no final, ainda produz um valor de um tipo específico.

Uma afirmação (*statement*) é uma sentença completa em uma linguagem de programação.

Uma afirmação pode ser composta por apenas uma expressão (e produzir um valor apenas) ou usar de algum outro componente da linguagem de programação que tenha ou não um *efeito colateral*.

Efeitos colaterais podem ser efeitos simples, como escrever na memória do computador (seja na volátil ou na de massa), escrever algo na tela ou ler o estado de algum periférico externo ao programa (como o mouse ou teclado).

Uma sequência de afirmações são separadas por ponto e vírgula ( ;) em JavaScript. É possível evitar o uso deles, mas é recomendado começar na linguagem usando-os como separador de afirmações até ficar mais seguro.

Uma sequência de afirmações em uma linguagem de programação é um programa de computador.

# Estado

O primeiro efeito colateral que vamos explorar é exatamente armazenar os valores na memória volátil, que só existirá durante a execução do programa.

Ao conjunto dos dados que é mantido e alterado durante a execução de um programa, nós chamamos de *estado*.

O estado descreve de forma completa todos os dados que representam o problema em um determinado momento do tempo. Como se fosse uma foto.

No nosso exemplo original das maçãs, o estado é formado por quantas maçãs João tem e quantas Maria tem.

Se não houver nenhuma ação ou operação sobre o estado, ele vai se manter: Se João e Maria não ganharem ou perderem maçãs, eles vão continuar indefinidamente com as que começaram.



# Definição de Variáveis e Constantes

No JavaScript o estado é descrito por ligações (*bindings*) entre valores e um identificador. Através desse identificador podemos acessar o valor durante a execução do programa.

Existem quatro palavras chave que criam ligações em JavaScript, **let**, **const**, **yield** e **var**. Vamos primeiro nos concentrar nas duas primeiras, começando pelo **let** e criar nossa primeira afirmação:

```
let joao = 1;  
let maria = 2;
```

Essas duas afirmações, compostas pela palavra reservada **let** seguida de um nome (ou identificador) dizem que eu vou querer guardar e ler valores da memória em duas posições diferentes e independentes.

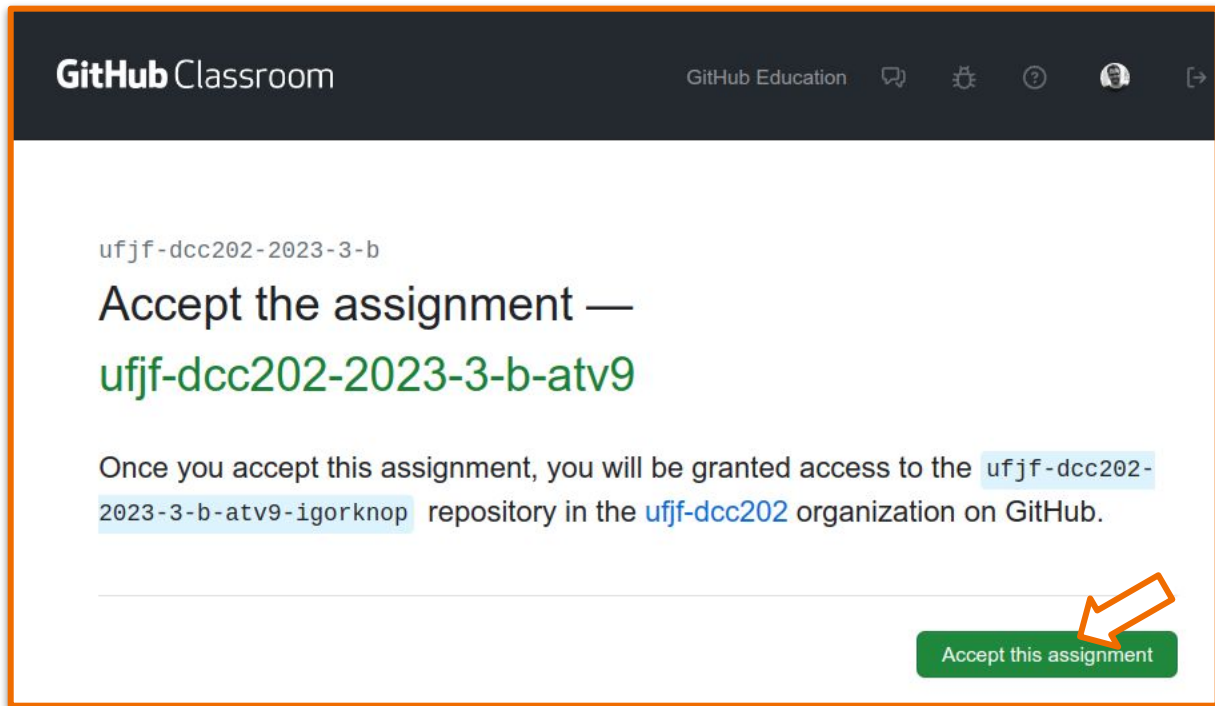
## Exercícios: Depuração JavaScript (atv9)

Vamos experimentar criar um projeto com código JavaScript na forma de uma aplicação que roda no cliente no Google Chrome e outra para no ambiente NodeJS e ver como acompanhar o seu estado usando o VSCode.

Aceite a atv9 no Google Classroom.

Novamente, esta atividade vai ter uma série de arquivos diferentes que são responsáveis por rodar os testes.

Só edite o arquivo README.md e os arquivos html que você vai criar.



## Faça o clone no VSCode

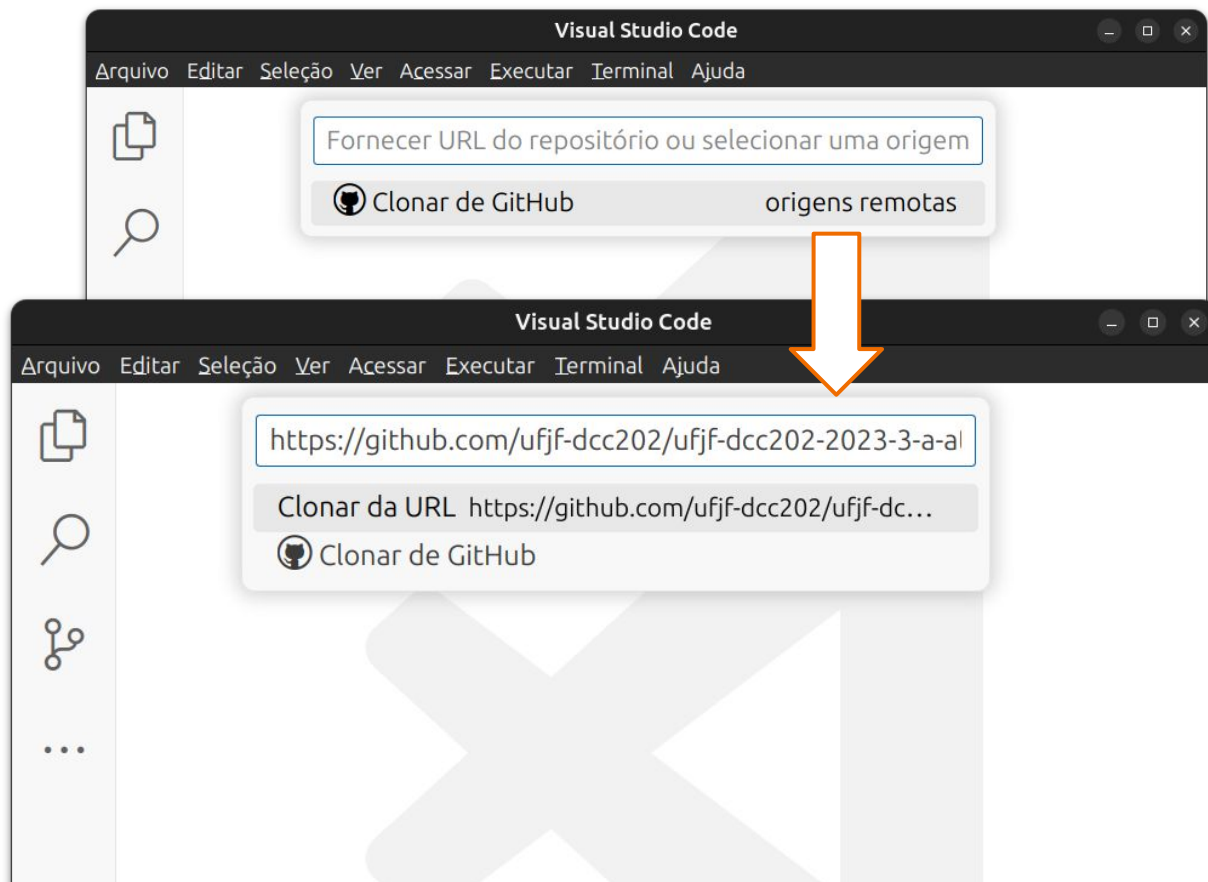
Dentro do VSCode, use o CTRL+SHIFT+P para abrir o terminal de comandos e digite:

> Git: Clone <ENTER>

Cole o endereço do repositório e dê <ENTER> novamente.

Um diálogo abrirá para escolher o destino para o clone.

Abra o repositório clonado para começar os trabalhos.



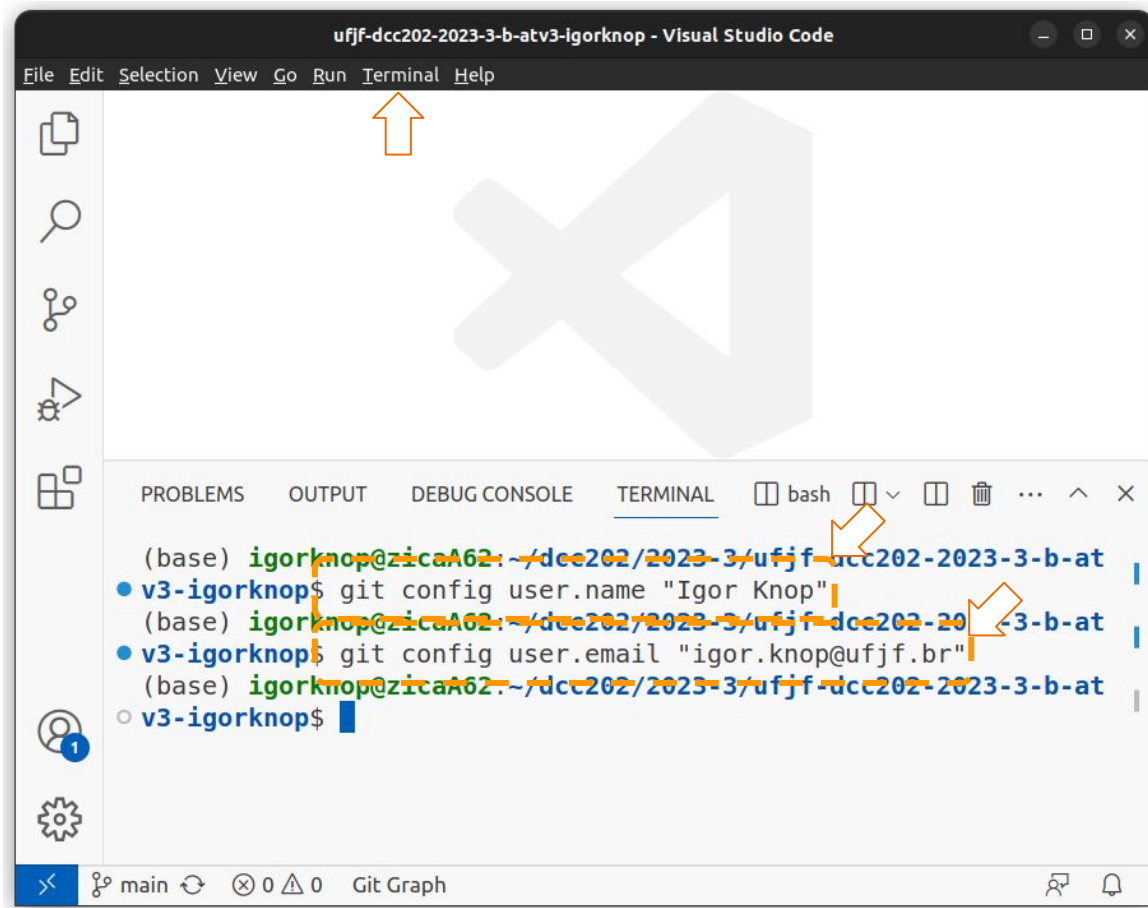
## Exercício: Configure seu nome de usuário e email no git

Abra um novo terminal e configure seu nome de usuário e email:

```
git config user.name "SEU NOME"
```

```
git config user.email "SEU EMAIL NO GITHUB"
```

Confira se funcionou!



## Execute os testes

No terminal, execute o comando para instalar as dependências de teste:

```
npm install
```

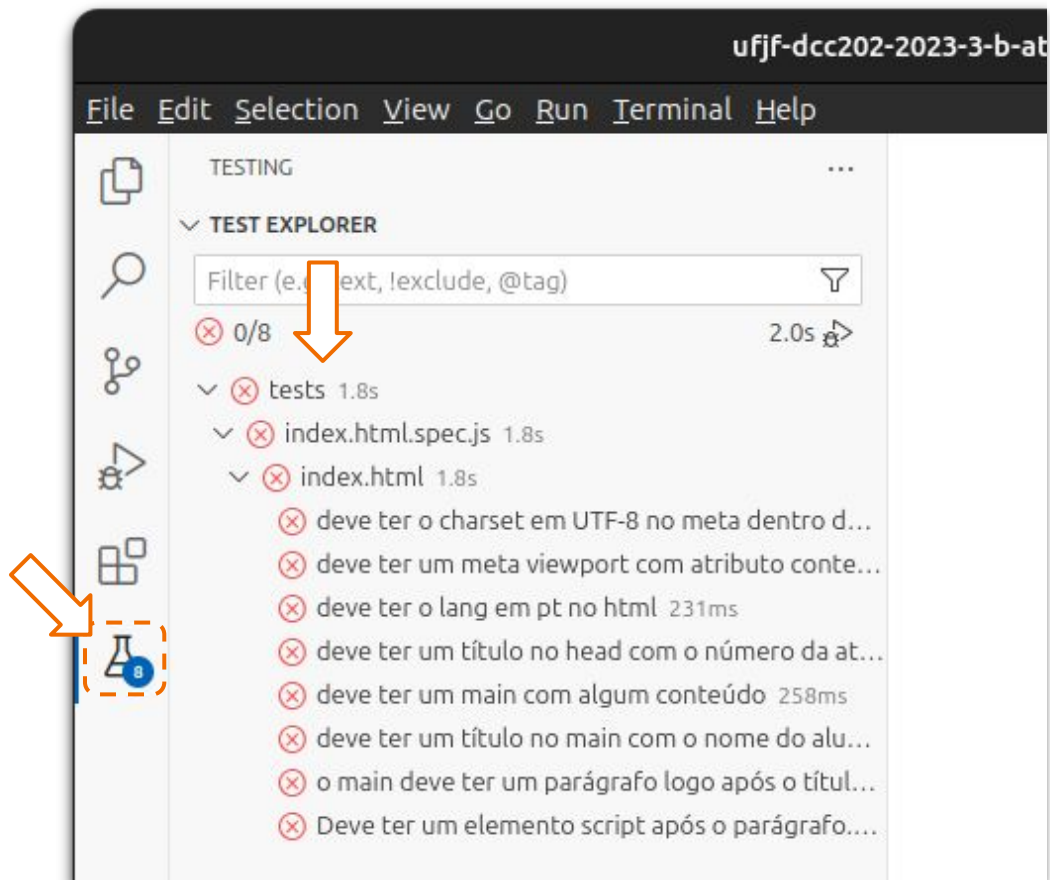
Depois vá no executor de testes e clique em um dos plays para executar.

São três conjuntos diferentes de testes, com um total de 30 casos de testes.

Todos devem falhar pois os arquivos não existem (FILE NOT FOUND)

Se a mensagem for referente ao playwright não ter os navegadores necessários, execute o comando:

```
npx playwright install
```



**Atenção:** É importante ver os testes falhando e entender o porquê deles terem falhado!

# Criar e depurar o programa no NodeJS

Na raiz do repositório, crie um arquivo **index.js**.

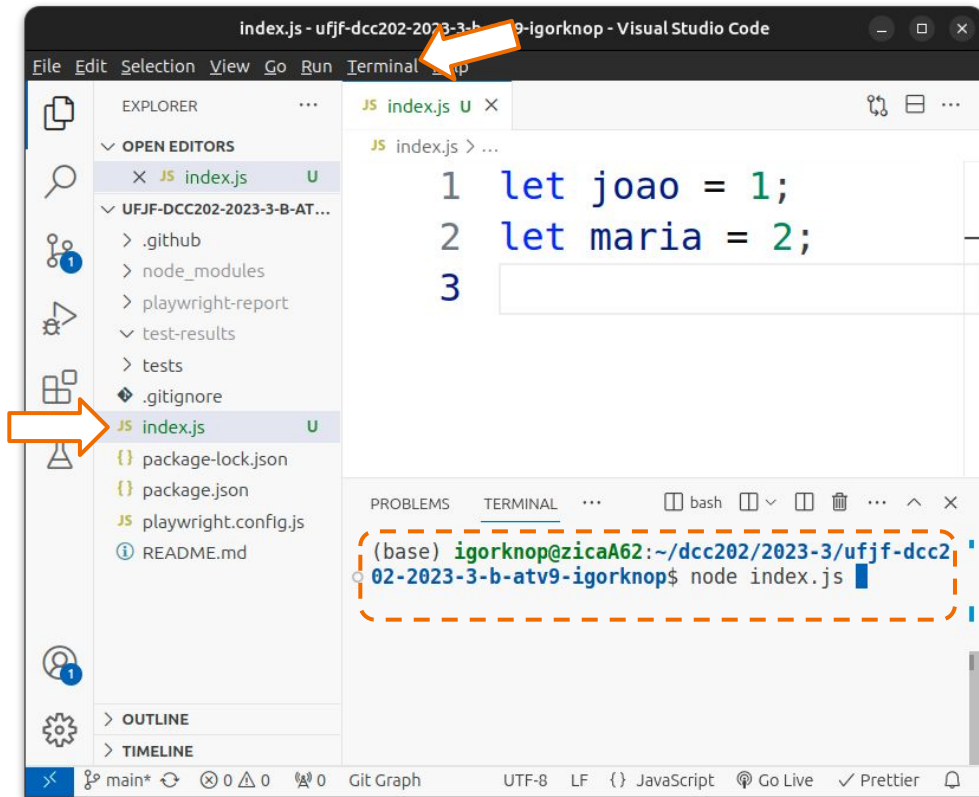
Esse arquivo vai ter duas linhas que serão responsáveis por definir o estado de um programa.

Vamos começar utilizando o ambiente NodeJS e a linha de comando em um terminal para executar esse código.

Abra um novo terminal com **Terminal > Novo Terminal**

No terminal, execute o comando:  
`node index.js`

O programa deve executar sem erro, mas nada visível vai acontecer: precisamos investigar a execução do *script* pela depuração!



# Criar e depurar o programa no NodeJS

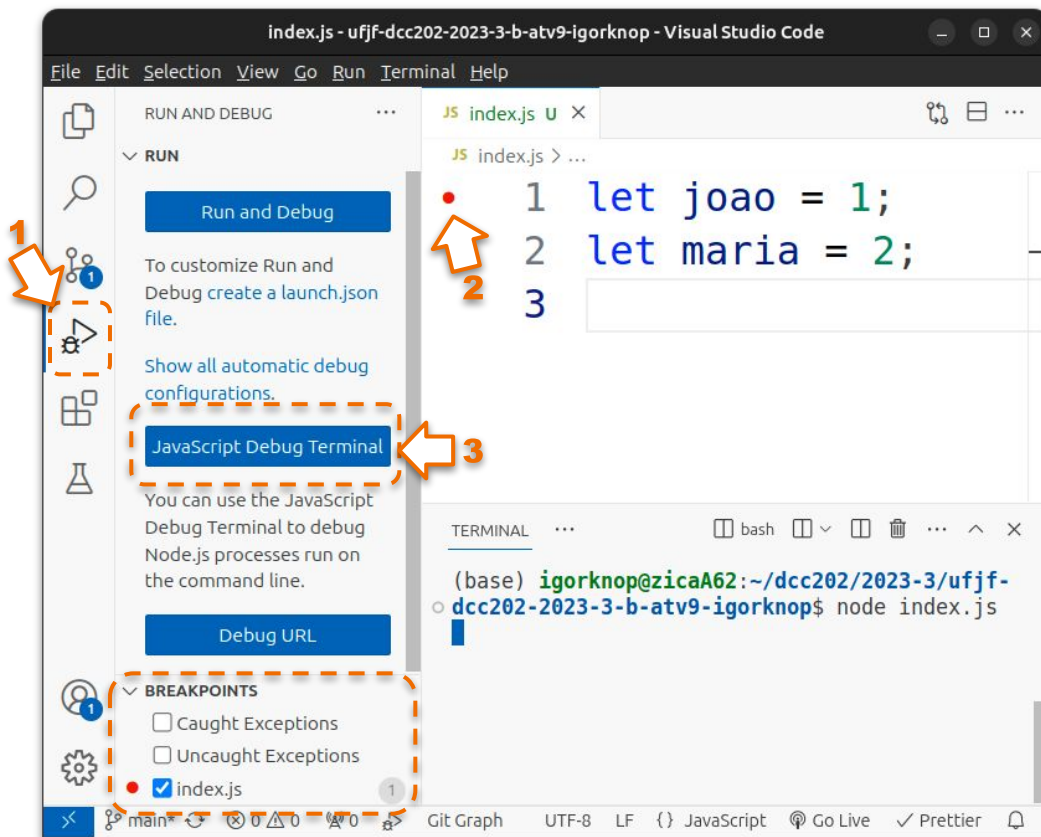
Na barra lateral do VSCode, selecione a aba de Debug (ou CTRL+SHIFT+D). Isso vai abrir interface de depuração.

No código fonte, dê um clique na primeira linha, no espaço vazio à esquerda da numeração. Isso vai criar um chamado **ponto de parada** ou *breakpoint*.

Os *breakpoints* também podem ser gerenciados na parte inferior da tela de depuração. Eles vão representar pontos nos quais o nosso programa vai parar para que possamos visualizar o seu estado atual.

Quando executamos nosso script com o NodeJS, ele começou na primeira afirmação com o chamado contador de programa. Esse contador de programa, pula de afirmação a afirmação até passar por todas, que é quando o programa termina.

Vamos executar novamente o programa, mas agora utilizando o terminal de depuração.





# Criar e depurar o programa no NodeJS

Na barra lateral do VSCode, selecione a aba de Debug (ou CTRL+SHIFT+D). Isso vai abrir interface de depuração.

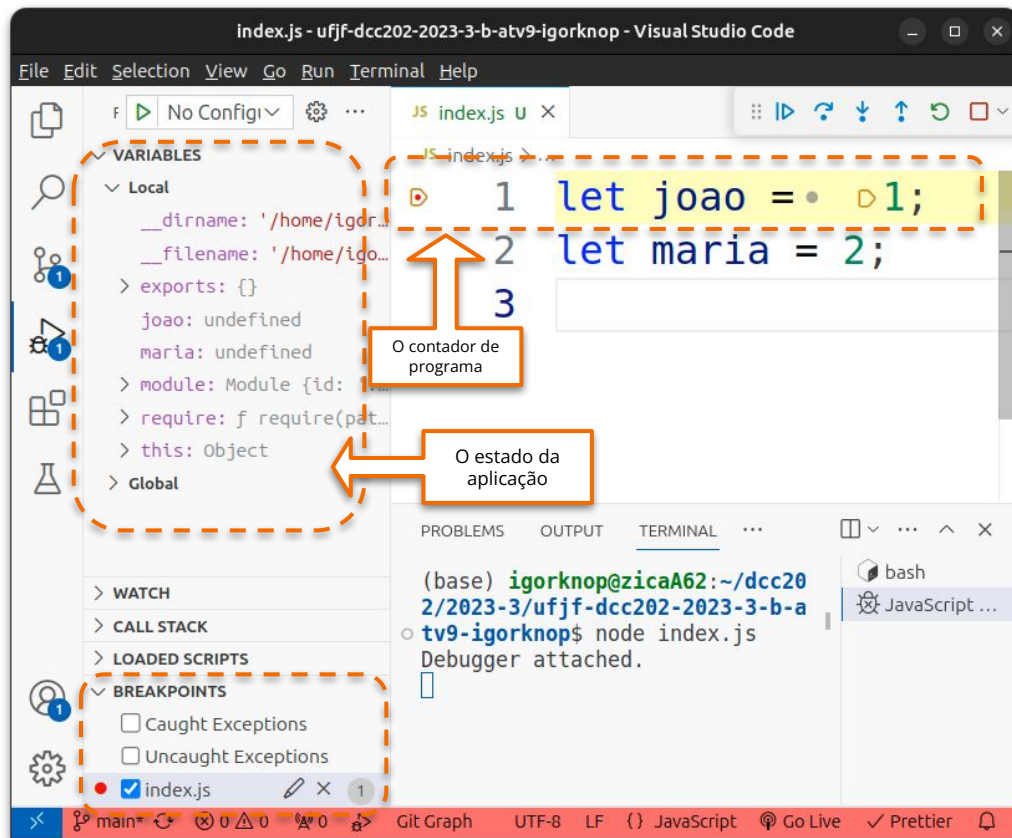
No código fonte, dê um clique na primeira linha, no espaço vazio à esquerda da numeração. Isso vai criar um chamado **ponto de parada** ou *breakpoint*.

Os *breakpoints* também podem ser gerenciados na parte inferior da tela de depuração. Eles vão representar pontos nos quais o nosso programa vai parar para que possamos visualizar o seu estado atual.

Quando executamos nosso script com o NodeJS, ele vai posicionar sobre a próxima afirmação a ser resolvida o chamado contador de programa.

Após executar a afirmação, o contador de programa vai se mover para a afirmação a ser resolvida. Quando não houverem mais afirmações, o programa terminou.

Vamos executar novamente o programa, mas agora utilizando o terminal de depuração.





# Controles de depuração

O contador de programa é controlado pelos botões no canto superior direito da tela.

## Continue (F5)

Vai executar todas as afirmações até encontrar um outro ponto de parada ou o programa terminar.

## Step Over (F10)

Executa a afirmação atual e depois posiciona na próxima.

## Step Into (F11)

Executa a afirmação atual, se ela for uma função (que veremos já já), segue com o contador de programa para dentro dela.

## Step Out (Shift+F11)

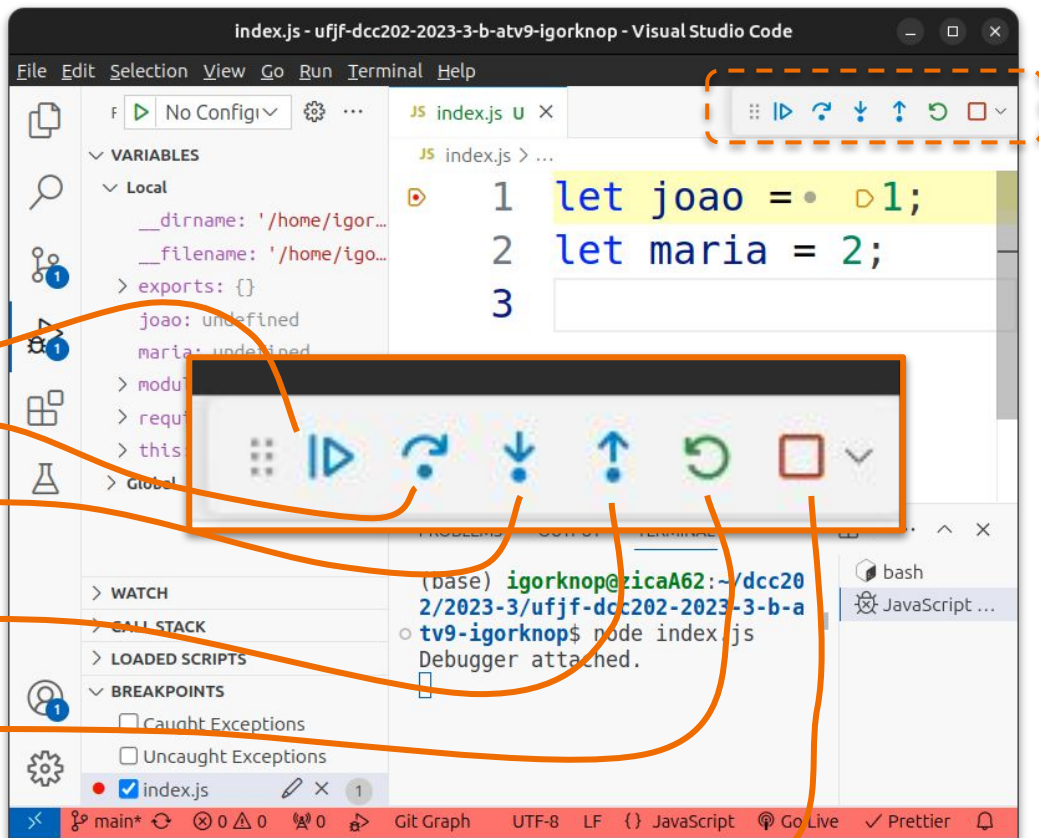
O contador de programa já está em uma função e você quer retornar dela imediatamente, sem ver os próximos passos.

## Restart (CTRL+Shift+F5)

Reinicia todo o processo de depuração, colocando o contador de programa de novo na primeira afirmação.

## Disconnect/Stop (Shift+F5)

Interrompe todo o processo de depuração.



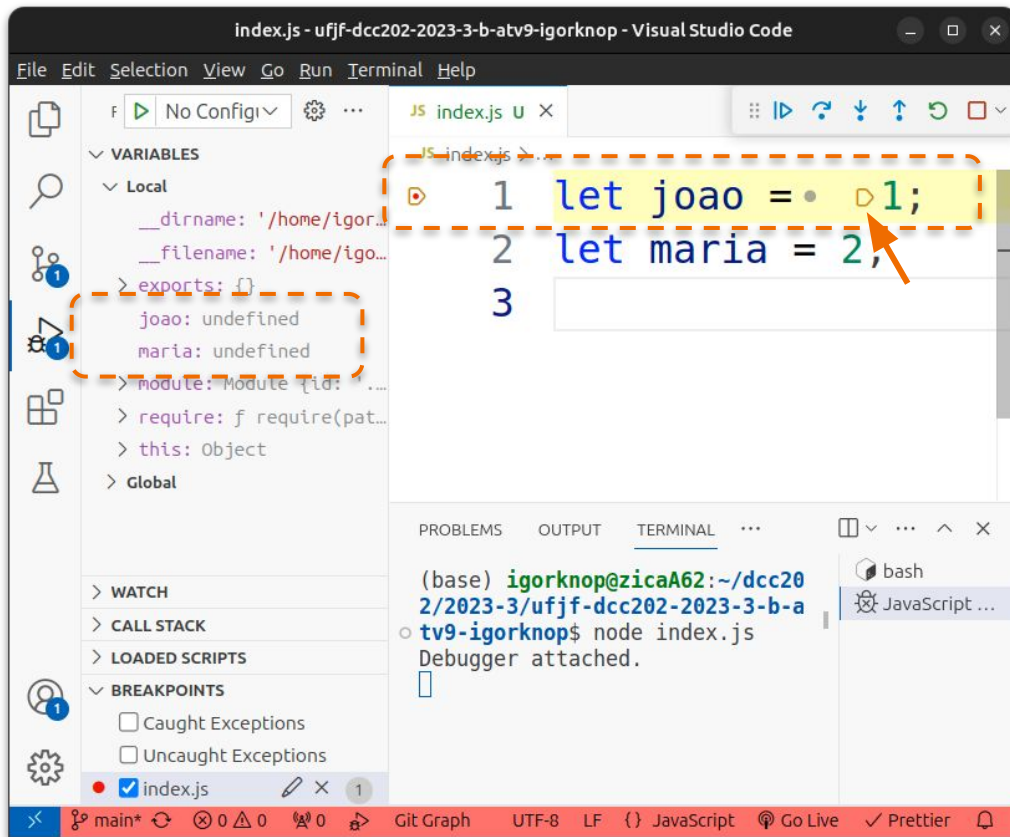
## Depuração: passo 1

Com o contador de programa na primeira linha, perceba que, antes de executar a primeira afirmação. Há o que parece ser um segundo contador de programa, mas isso é para indicar que a atribuição será realizada.

O estado já mostra a ligação a dois espaços de memória para **joao** e **maria**.

Isso nós chamamos de variáveis, um espaço de memória, identificado para ser usado posteriormente.

Após observar isso, dê um step over (F10).

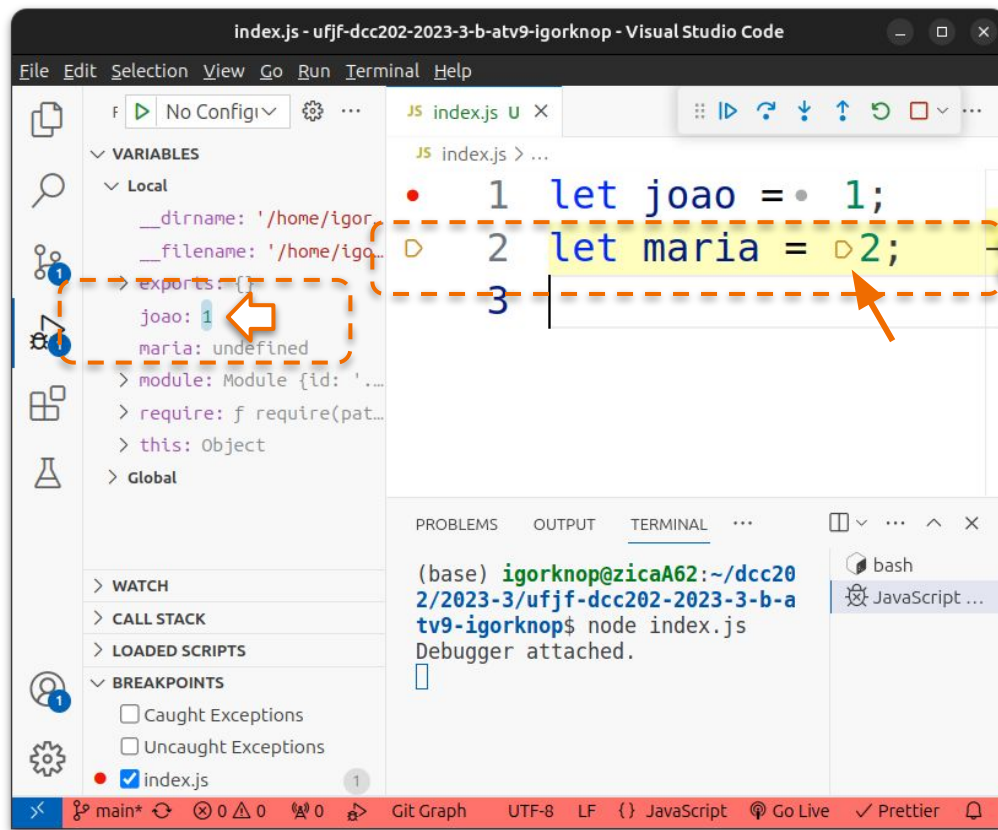


## Depuração: passo 2

Após um *step over*, a afirmação é executada, causando como efeito colateral que a variável **joao** agora esteja ligada ao valor **1**.

O contador de programa agora está na linha 2, também na atribuição, esperando a confirmação para executar a segunda afirmação.

Após observar isso, dê mais um *step over* (F10).



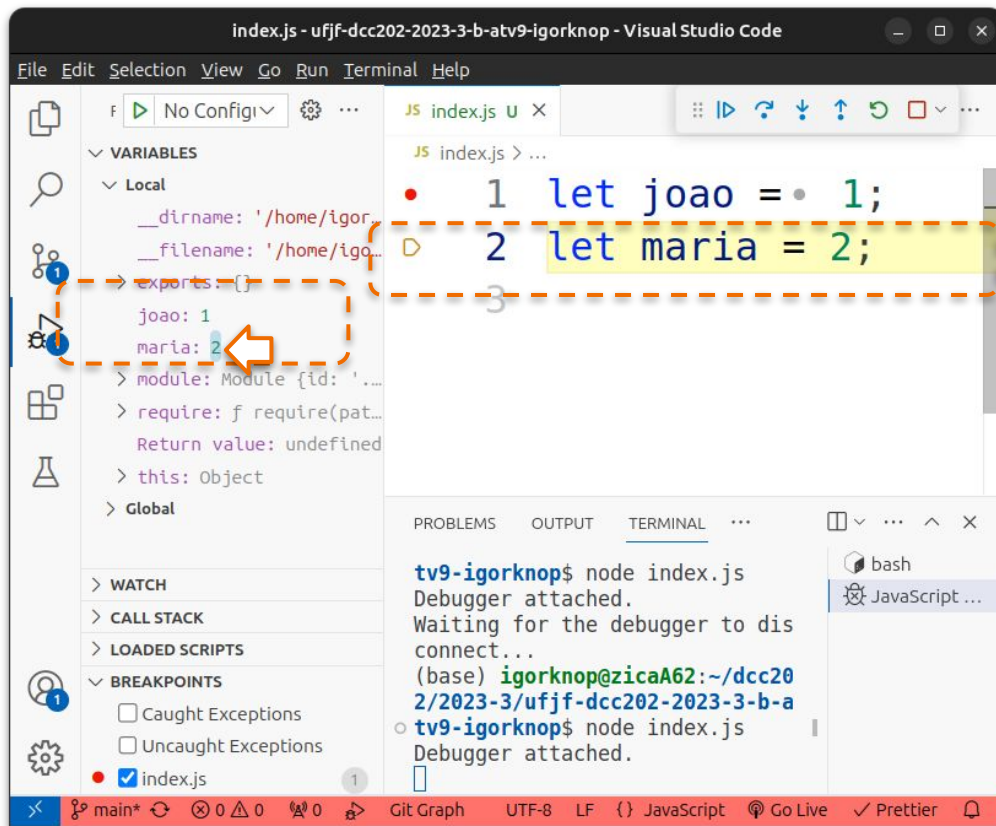
## Depuração: passo 3

Após um *step over*, a afirmação é executada, causando como efeito colateral que a variável **maria** agora esteja ligada ao valor **2**.

O contador de programa agora ainda está na linha 2, mas como essa é a última afirmação, ele está esperando para encerrar o programa.

Após observar isso, dê mais um *step over* (F10).

Isso fará com que a atribuição termine.



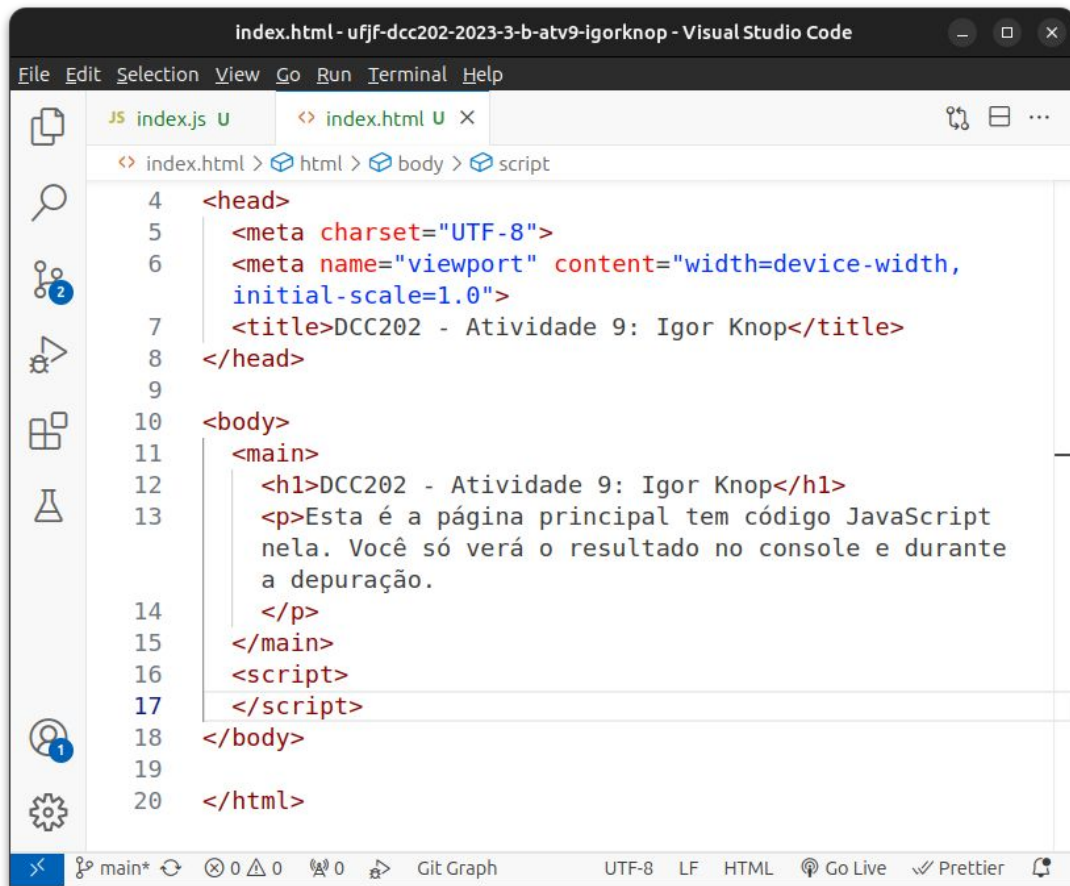
## Criar e depurar o JavaScript no Navegador

Outra forma de depurar nosso código é com ele sendo executado no Google Chrome.

Você pode fazer isso de dentro do VSCode ou pelo próprio navegador. Vamos primeiro repetir o processo anterior dentro do nosso editor.

Mas antes, precisamos criar um documento index.html para colocarmos nosso script.

Crie na raiz do projeto um arquivo **index.html**.



```
index.html - ujf-f-dcc202-2023-3-b-atv9-igorknop - Visual Studio Code
File Edit Selection View Go Run Terminal Help
JS index.js U index.html U X
index.html > html > body > script
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width,
   initial-scale=1.0">
7   <title>DCC202 - Atividade 9: Igor Knop</title>
8 </head>
9
10 <body>
11   <main>
12     <h1>DCC202 - Atividade 9: Igor Knop</h1>
13     <p>Esta é a página principal tem código JavaScript
        nela. Você só verá o resultado no console e durante
        a depuração.
14     </p>
15   </main>
16   <script>
17   </script>
18 </body>
19
20 </html>
```

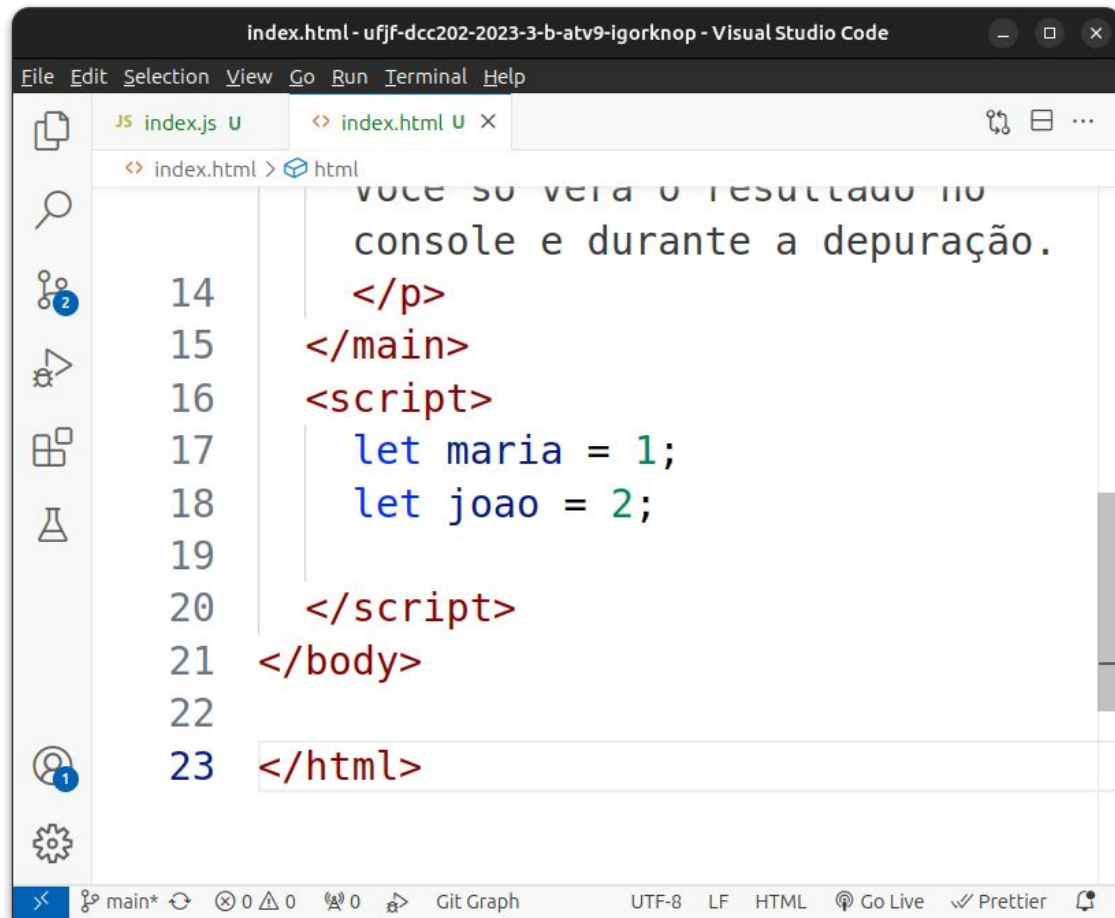
## Elemento script no HTML

A forma mais simples de inserir um trecho de código JavaScript é através do elemento `<script></script>`.

Isso vai criar no documento uma área com código JavaScript embutido, nos permitindo usar a linguagem para manipular a página futuramente.

Dentro do elemento, adicione nosso código com duas atribuições.

Vamos agora configurar a depuração...



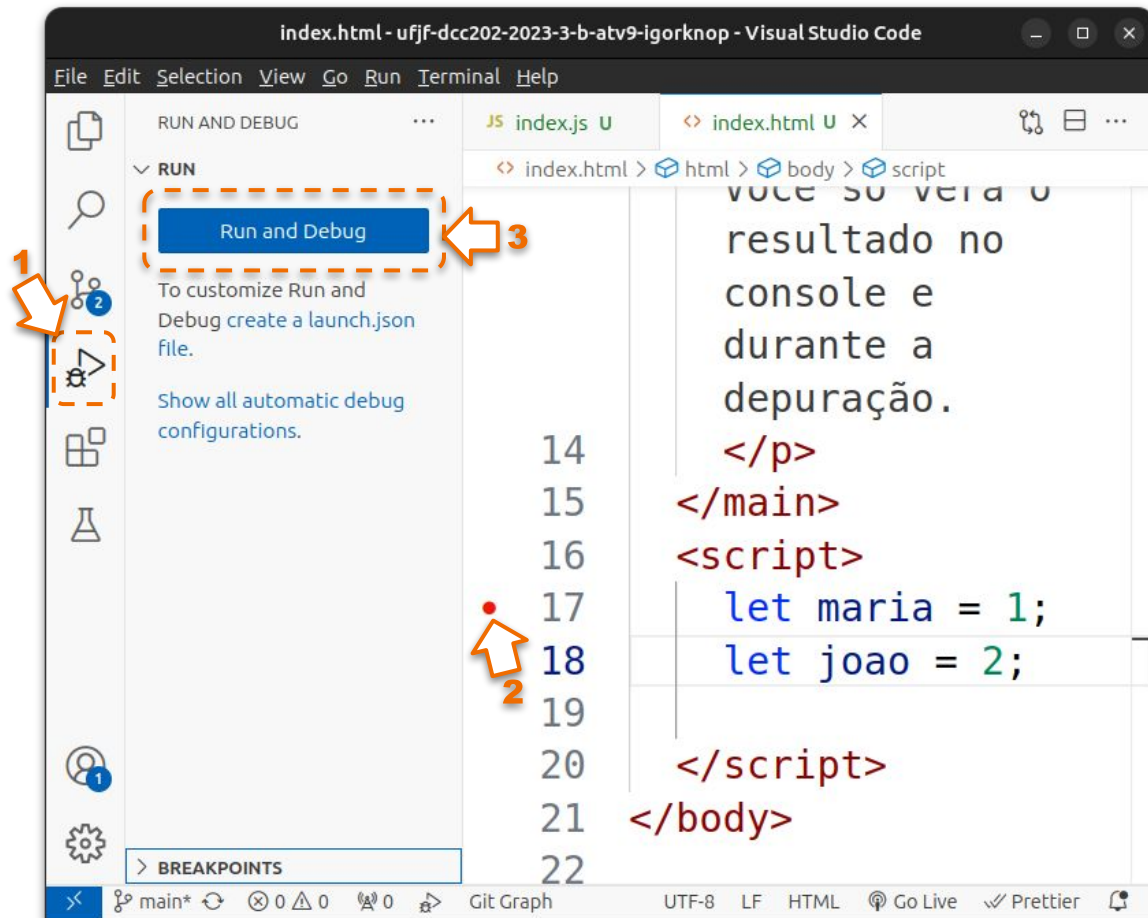
```
index.html - ufff-dcc202-2023-3-b-atv9-igorknop - Visual Studio Code
File Edit Selection View Go Run Terminal Help
JS index.js U index.html U x
index.html > html
você só verá o resultado no
console e durante a depuração.
14 </p>
15 </main>
16 <script>
17   let maria = 1;
18   let joao = 2;
19
20 </script>
21 </body>
22
23 </html>
```



## Elemento script no HTML

Novamente adicione um ponto de parada no código na primeira linha do script.

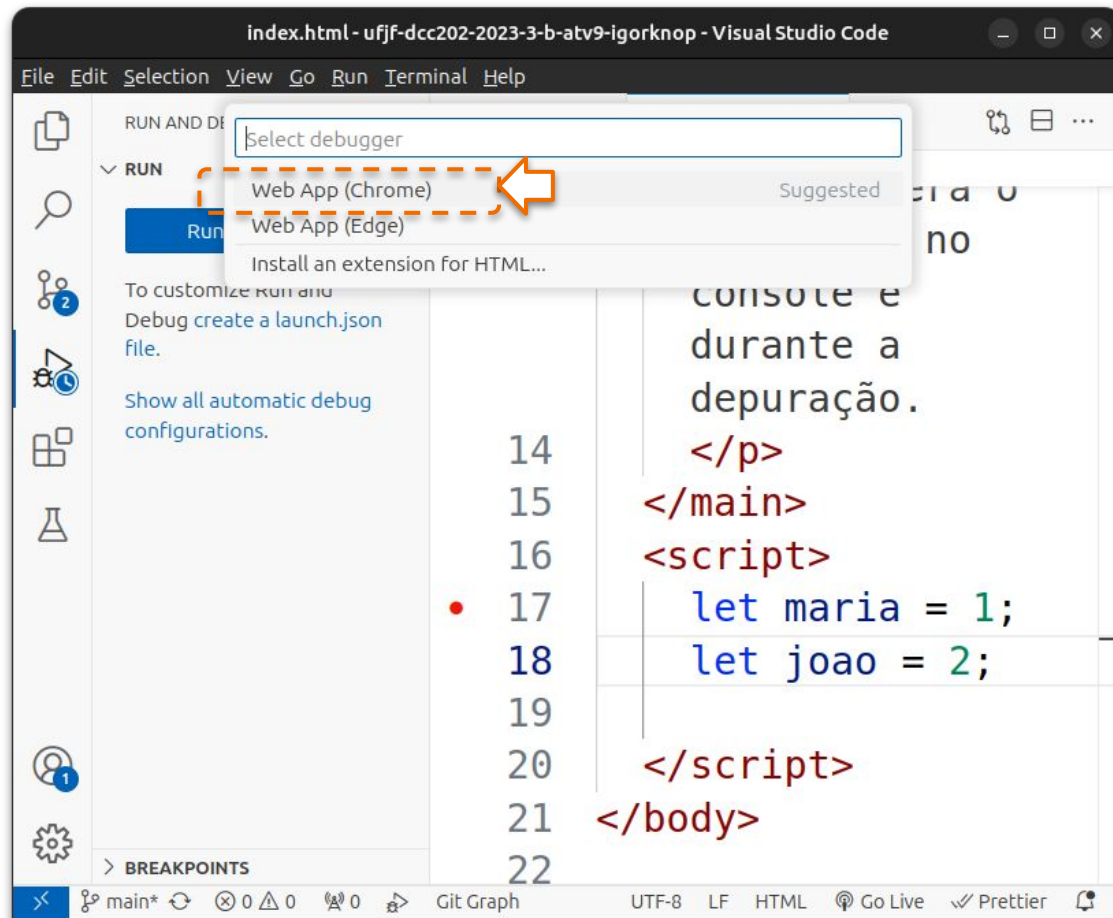
Vá na área de depuração e pressione o botão **Run and Debug**.



## Elemento script no HTML

Novamente adicione um ponto de parada no código na primeira linha do script.

Vá na área de depuração e pressione o botão **Run and Debug**.

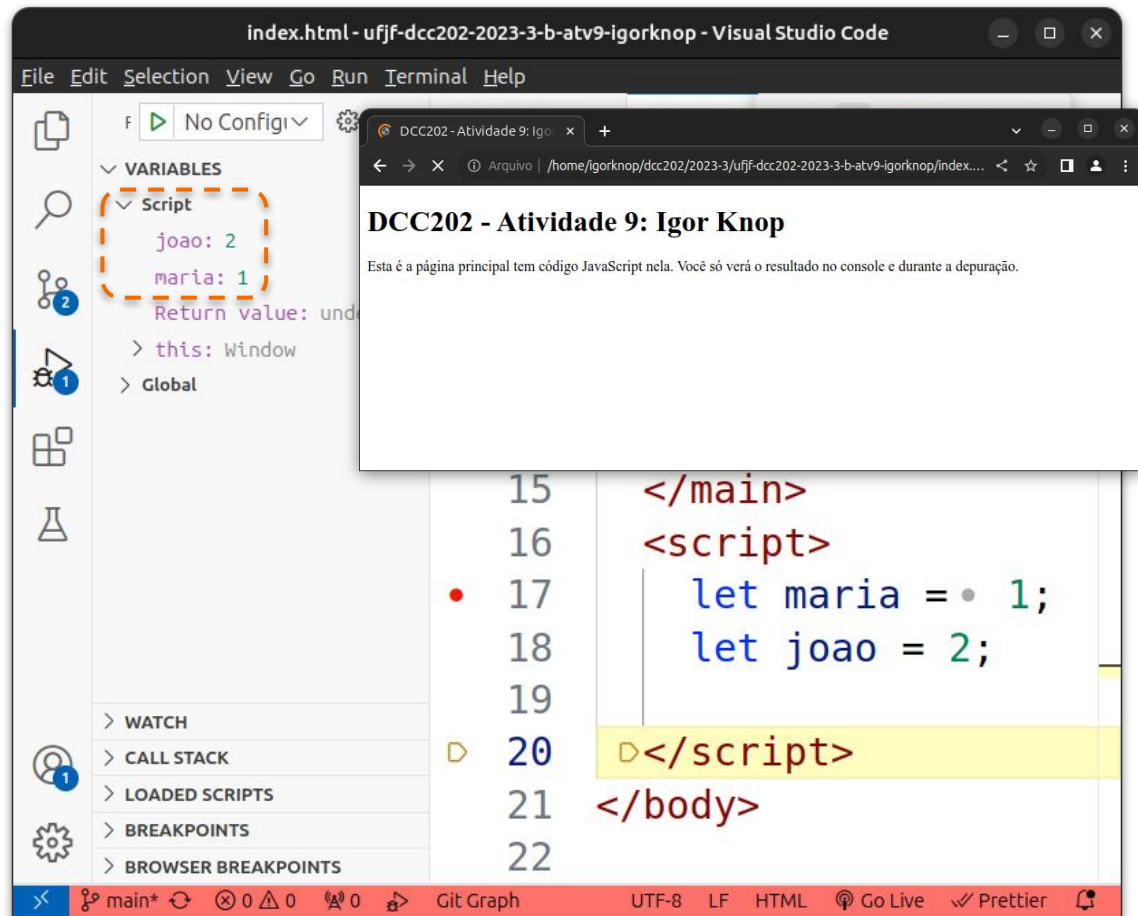




## Elemento script no HTML

Uma nova janela do navegador vai abrir e vai ficar como se estivesse ainda em carregamento Isso mostra que tem algo em depuração.

As variáveis só vão aparecer durante a depuração. Observe que vão aparecer dentro do escopo **Script**. Ainda temos o escopo **Global**.



# Funções: sintaxe básica

Funções nos permitem agrupar um conjunto de afirmações e dar um nome para elas.

Ao dar um nome para uma função, nós começamos a aumentar a abstração de nosso código e criamos as operações que vão agir sobre nossos dados.

Por exemplo: podemos falar que temos duas operações, uma chamada **deJoaoParaMaria()** que representa os passos necessários para indicar que João deu todas as suas maçãs para Maria e ficou sem nenhuma e uma **deMariaParaJoao()** que faz o contrário.

```
function deJoaoParaMaria()  
{  
    maria = maria + joao;  
    joao = 0;  
}  
function deMariaParaJoao()  
{  
    joao = joao + maria;  
    maria = 0;  
}
```

# Funções: sintaxe básica

Funções nos permitem agrupar um conjunto de afirmações e dar um nome para elas.

Ao dar um nome para uma função, nós começamos a aumentar a abstração de nosso código e criamos as operações que vão agir sobre nossos dados.

Por exemplo: podemos falar que temos duas operações, uma chamada **deJoaoParaMaria()** que representa os passos necessários para indicar que João deu todas as suas maçãs para Maria e ficou sem nenhuma e uma **deMariaParaJoao()** que faz o contrário.

```
function deJoaoParaMaria()  
{  
    maria = maria + joao;  
    joao = 0;  
}  
function deMariaParaJoao()  
{  
    joao = joao + maria;  
    maria = 0;  
}
```

Funções são muito importantes em JavaScript e nós vamos ainda discuti-las em de detalhes.

# Funções: sintaxe básica

As funções que são definidas com a palavra reservada `function`, são içadas (*hoisting*) no script.

Portanto, você pode usá-las mesmo que sejam definidas posteriormente no código.

```
let maria = 1;  
let joao = 2;  
deJoaoParaMaria();  
deMariaParaJoao();
```

```
function deJoaoParaMaria() {  
  maria = maria + joao;  
  joao = 0;  
}
```

```
function deMariaParaJoao() {  
  joao = joao + maria;  
  maria = 0;  
}
```

# Objetos: propriedades

Outro conceito importante para iniciar no JavaScript é o de objetos.

Se as funções nos permitem agrupar afirmações criar novas operações, mais complexas, mais perto do nosso problema, os objetos nos permitem agrupar dados e operações.

Exemplo: Podemos usar os objetos para agrupar variáveis relacionadas. Essas são chamadas de propriedades ou atributos.

Ao lado temos um exemplo do uso de literal de objetos onde acessamos as propriedades por um ponto.

Objetos também são um assunto complexo, que voltaremos posteriormente.

```
let oJoao = {  
  tipo: "maçã",  
  qtd: 1  
}
```

```
let oMaria = {  
  tipo: "maçã",  
  qtd: 2  
}
```

```
oMaria.qtd = oMaria.qtd + oJoao.qtd;  
oJoao.qtd = 0;
```

# Objetos: métodos

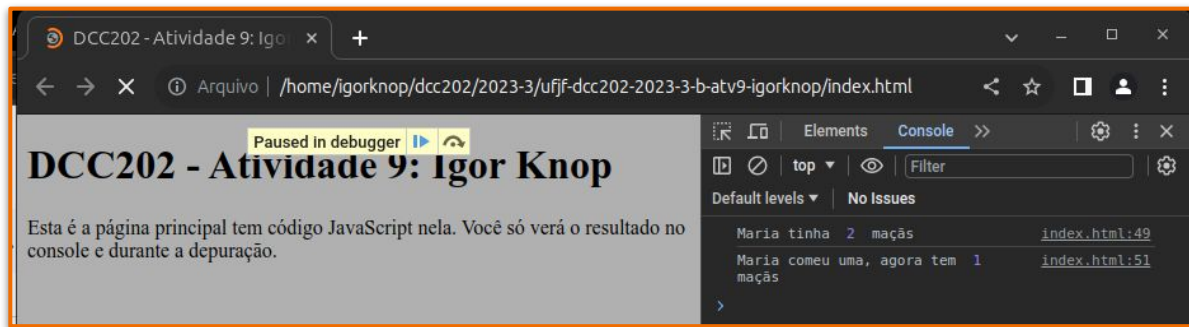
Objetos também podem ter funções como campos membro. Neste caso, são chamados de **métodos**.

Os métodos podem acessar as propriedades dos objetos ao qual estão ligados pela palavra reservada **this**.

Os métodos são acessados também pelo ponto, mas devem ser chamados usando os parênteses como nas funções.

Um objeto que vem configurado para nós é o **console**. Ele vai permitir acessar nosso primeiro efeito colateral de saída de dados.

```
let oMaria = {  
  tipo: "maçã",  
  qtd: 2,  
  comeu: function () {  
    oMaria.qtd--;  
  }  
}  
  
console.log("Maria tinha ", oMaria.qtd, " maçãs");  
oMaria.comeu();  
console.log("Maria comeu uma, agora tem ", oMaria.qtd, " maçãs");
```



# Tipo: Nulo (*null*) e Não definido (*undefined*)

Dois tipos primitivos são os tipos nulo e vazio, respectivamente com os literais `null` e `undefined`. Atenção: são literais sem aspas! Caso contrário seriam textos.

Um `undefined` é um tipo que aparece quando algo não foi definido, não existe ou não foi encontrado. Muitas vezes externa à vontade do programador.

Já um valor `null` é normalmente usando para deliberadamente representar a ausência de alguma coisa, não quantidade, um vazio.

Eles só vão fazer mais sentido adiante no curso. Portanto não se preocupe com eles no momento, mas saiba que não pode usar esses nomes pois são reservados e fique de olho quando algum desses valores aparecer.;

# Para saber mais...

- **Values, Types, and Operators.** In: Eloquent JavaScript. Available on Internet: [https://eloquentjavascript.net/01\\_values.html](https://eloquentjavascript.net/01_values.html)
- **Functions.** In: Eloquent JavaScript. Available on Internet: [https://eloquentjavascript.net/03\\_functions.html](https://eloquentjavascript.net/03_functions.html)
- **Objects and Arrays.** In: Eloquent JavaScript. Available on Internet: [https://eloquentjavascript.net/04\\_data.html](https://eloquentjavascript.net/04_data.html)