
Escopos, funções, vetores e objetos em JavaScript

UFJF - DCC202 - Desenvolvimento Web

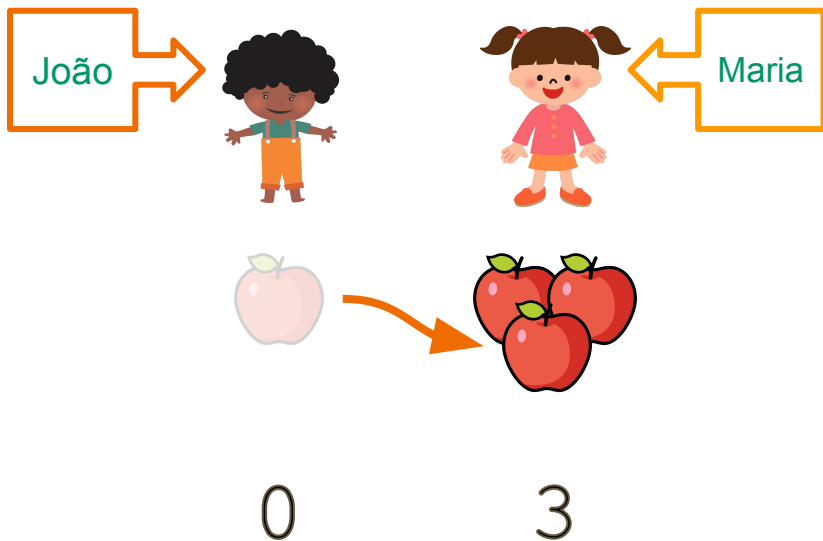
Prof. Igor Knop igor.knop@ufjf.br



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

Funções

Nós usamos funções para dar nomes a sequências de operações simples, mas que em conjunto têm um significado para nosso problema. Entretanto, há muito mais do que isso.



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

Funções

Nós usamos funções para:

- Diminuir a repetição de código igual;
- Reduzir programas grandes em pequenas partes;
- Isolar as partes do programa umas das outras;
- Melhorar a manutenibilidade.

As funções também ajudam a aumentar o nível de abstração de nossos programas, criando um vocabulário próprio para o problema que estamos resolvendo.

Blocos de código e Escopo de Variáveis

Escopo é conhecido como o programa organiza a ligação de valores a identificadores de suas variáveis, constantes e funções.

Uma variável que não esteja dentro de nenhum escopo, é considerada no escopo global.

Funções possuem seus próprios escopos, onde os parâmetros e variáveis declaradas dentro delas podem ser acessadas. Esse é o chamado escopo local e permite que uma função não interfira com o escopo da outra. Variáveis e constantes declaradas com `let` e `const` pertencem ao escopo no qual foram criadas.

Blocos de código em JavaScript também possuem seu próprio escopo e enxergam as variáveis do escopo “pai”. Se uma variável é declarada, você só enxerga a variável mais interna.

Blocos de código e Escopo de Variáveis (2)

Observe que não é possível declarar duas variáveis com o mesmo identificador em um mesmo escopo.

É possível criar novos escopos, com funções e blocos de código, e neles utilizar declarar usando o mesmo identificador.

```
let x = 3; // x vale 3 no escopo global
```

```
function exemplo(){  
  let x = 4 // x vale 4 no escopo local  
  {  
    let x = 5 // x vale 5 no escopo de bloco  
  }  
  // aqui é x de valor 4  
}
```

```
//aqui é x de valor 3
```

Blocos de código e Escopo de Variáveis (2)

Observe que não é possível declarar duas variáveis com o mesmo identificador em um mesmo escopo.

É possível criar novos escopos, com funções e blocos de código, e neles utilizar declarar usando o mesmo identificador.

```
let x = 3; // x vale 3 no escopo global
```

```
function exemplo(){
```

```
  let x = 4 // x vale 4 no escopo local
```

```
  {
```

```
    let x = 5 // x vale 5 no escopo de bloco
```

```
  }
```

```
  // aqui é x de valor 4
```

```
}
```

```
//aqui é x de valor 3
```

Declaração de Funções

Funções são declaradas em nossos programas para posteriormente serem chamadas. Isto é, devemos criar uma ligação entre um trecho de código a um identificador para que possamos acessá-la depois (igual como fizemos com os dados). Funções declaradas são içadas (*hoisting*) no escopo.

```
function identificador (parametros){  
}
```

As funções podem retornar um dado quando terminam sua execução através da operação **return**. Normalmente isso é associado a algum processamento de dados interno, mas isso não é obrigatório. Uma função sem um **return**, retorna **undefined**.

Elas também receber parâmetros, para alterar a sua operação interna e deixá-las mais versáteis. Mas isso também é opcional.

E elas podem fazer qualquer combinação, como receber parâmetros para realizar seu trabalho e, por fim, retornar um dado resultante da operação.

Declaração de Funções (2)

```
function calculaSoma(){  
  let soma = 2 + 3;  
}
```



Só calcula internamente.

```
function calculaERetornaSoma(){  
  let soma = 2 + 3;  
  return soma;  
}
```



Calcula internamente e retorna um valor.

```
function recebeECalculaSoma(a, b){  
  let soma = a + b;  
}
```



Recebe parâmetros e calcula internamente.

```
function recebeCalculaERetornaSoma(a, b){  
  let soma = a + b;  
  return soma;  
}
```



Recebe parâmetros, calcula internamente e retorna um valor.

Funções como valores

As funções em JavaScript podem ser tratadas como valores. Dessa forma, é possível omitir o identificador pois o nome das variáveis e constantes atribuídas que permitirão encontrar a função.

```
const produto = function(a, b){  
  return a * b;  
};  
const subtracao = function(a, b) {  
  return b - a;  
};  
let operacao = produto;  
operacao(2, 3);  
operacao = subtracao;  
operacao(2, 3);
```

Acima, é importante destacar que funções como valores não são içadas!

Funções seta

Uma terceira forma de criar funções é através da notação seta (*Arrow Functions*). Ela define uma lista de parâmetros a um bloco de código.

```
const cubo = (x) => {  
  return x**3;  
};
```

Funções com apenas um parâmetro podem omitir os parênteses e as com apenas um retorno, podem omitir as chaves e o **return**.

```
const cuboAlternativo = x => x**3;
```

Funções seta também não são içadas como as funções como valor, e se diferem em não sobrescrever a referência a **this**, usam a do escopo que são criadas, como será visto mais adiante.

Funções puras

Uma função é dita pura se ela não causa nenhum efeito colateral, e não lê nenhum dado que possa ser efeito colateral de outro código.

```
let a = 0
```

```
let b = 1
```

```
function incrementaNaoPura() {
```

```
  a = a + b;
```

```
}
```

Essa função não pura porque:

- Ela causa um efeito colateral ao alterar o valor de **a** fora do escopo local;
- Ela usa diretamente o valor de **a** e **b**, fora do escopo local;

Funções puras (2)

Uma função é dita pura se ela não causa nenhum efeito colateral, e não lê nenhum dado que possa ser efeito colateral de outro código.

```
let a = 0
let b = 1
function incrementaPura(x, y) {
  x = x + y;
  return x;
}
a = incrementaPura(a, b);
```

Essa função é pura porque:

- Ela não causa um efeito colateral pois não altera o valor de **a** diretamente;
- Ela não usa diretamente o valor de **a** e **b**, do escopo que foi criada;

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 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");
```


Objetos: métodos e this

Seria muito difícil e pouco prático usar as referências para os objetos nas próprias funções. Por isso existe a referência `this`, que muda bastante de comportamento em Javascript.

Mas, quando se usa uma função por declaração para um método, o **this** passa a se referir ao objeto que ao qual está ligada em uma chamada.

Cuidado: funções seta como método não ligam o **this** ao objeto!

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



Vetores (*Arrays*)

Nós organizamos a memória do computador em variáveis, que são capazes de associar a um identificador um único valor, de um determinado tipo, para modelar nossos problemas.

Entretanto, existem problemas que descrevem um conjunto de valores ao invés de um único valor como os números da loteria do mês ou quais cartas um jogador tem uma mão de pôquer.

O JavaScript tem uma estrutura especial para isso, os vetores ou *arrays*. Um literal de array pode ser declarado usando colchetes:

```
let sorteados = [13, 23, 26, 29, 45, 59];  
const maoDePoker = ["2♠", "K♥", "Q♦", "A♣", "7♥"];
```

Os valores podem ser acessados individualmente por um índice, por exemplo, sorteados

Vetores: acesso direto aos itens

Os valores podem ser acessados individualmente por um índice, por exemplo, sorteados:

```
let sorteados = [13, 23, 26, 29, 45, 59];  
const maoDePoker = ["2♠", "K♥", "Q♦", "A♣", "7♥"];  
  
sorteados[0] === 13;  
sorteados[1] === 23;  
maoDePoker[3] === "A♣";  
maoDePoker[4] === "7♥";  
maoDePoker[5] === undefined;
```

Vetores: propriedades

Os *arrays*, como quase todos os tipos em JavaScript, possuem propriedades. Usando a notação de objetos, podemos acessar o tamanho, número de itens em cada *array*:

```
let sorteados = [13, 23, 26, 29, 45, 59];  
const maoDePoker = ["2♠", "K♥", "Q♦", "A♣", "7♥"];  
  
sorteados.length === 6;  
sorteados.maoDePoker === 5;
```

Vetores: métodos

Os *arrays* possuem uma grande quantidade de métodos para percorrer e manipular seus itens:

```
let sorteados = [13, 23, 26, 29, 45, 59];
const maoDePoker = ["2♠", "K♥", "Q♦", "A♣", "7♥"];

sorteados.includes(42) === false;
sorteados.indexOf(26) === 2;
maoDePoker.pop() === "7♥";
maoDePoker.length === 5;
maoDePoker.shift() == "2♠";
maoDePoker.length === 4;
maoDePoker.push("K♦");
maoDePoker.push("A♠"); // estado final: ['K♥', 'Q♦', 'A♣', 'K♦', 'A♠']
```

Alguns modificam o conteúdo do *array* e outros não. Modificar os itens do array não é mudar o array!

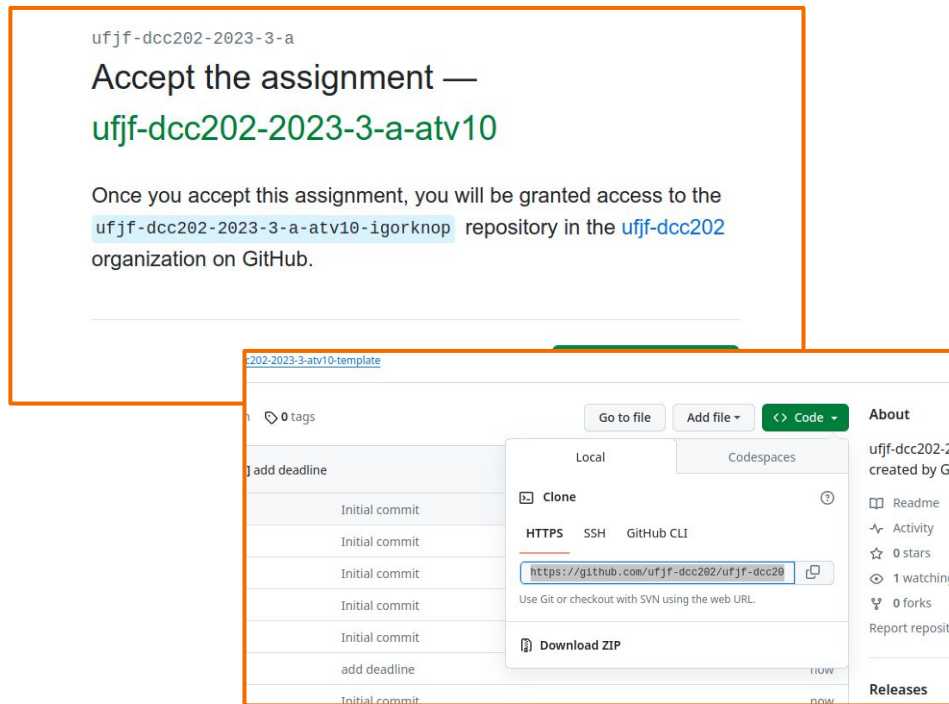
Exercícios: Faça o clone da atv10

Crie um clone da atividade 10 para experimentarmos os escopos.

Após clonar, abra a pasta e lembre-se de verificar se está logado no VSCode e também configure seu nome e email no github no terminal com:

```
git config user.name "Seu Nome"
```

```
git config user.email "seu@email.com"
```



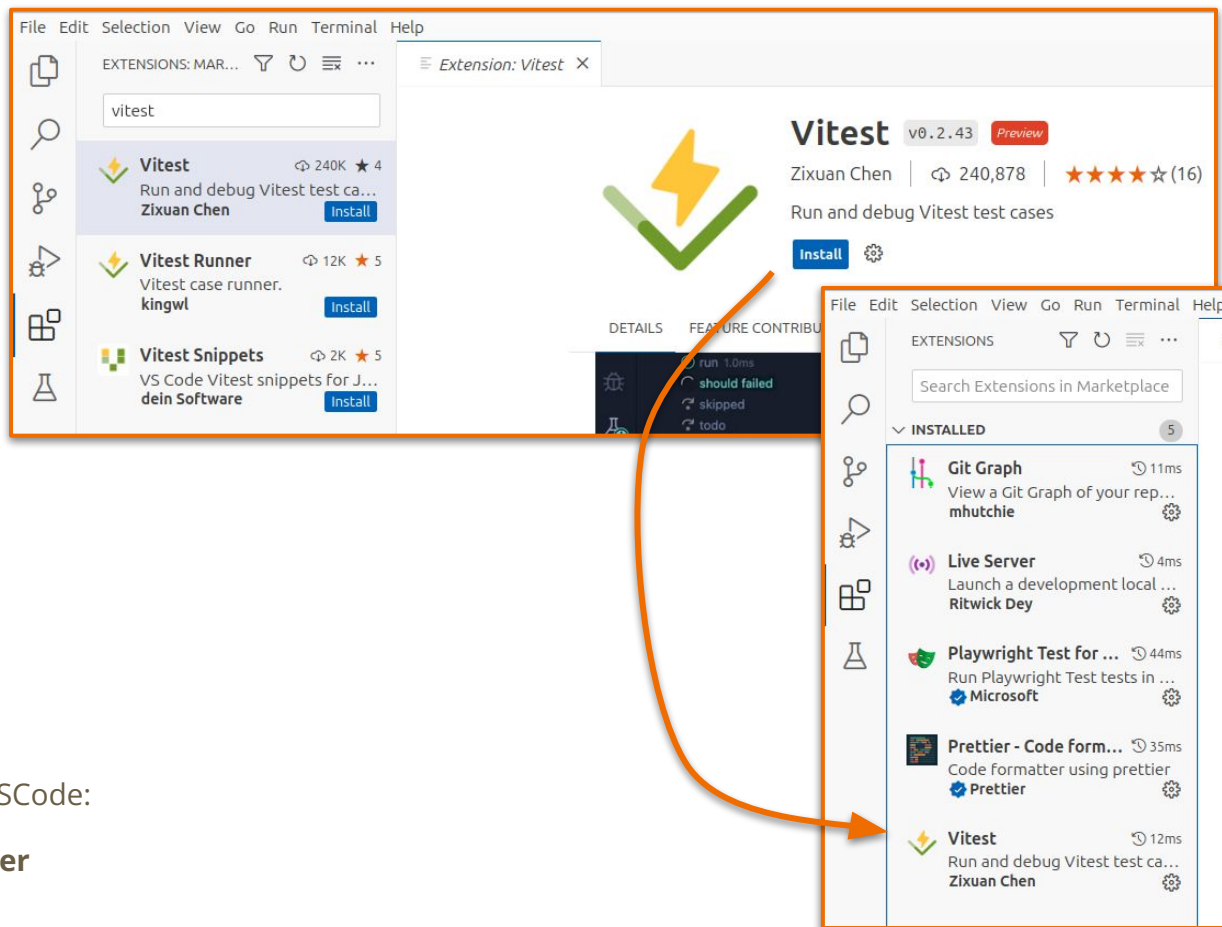
Instalar a extensão do vitest

Neste projeto, além de depurar, nós vamos utilizar um outro sistema de testes, o **vitest** em paralelo com o playwright.

Para tentar deixar integrado no VSCode, procure e instale a extensão Vitest de Zixuan Chen.

Ou pelo comando (CTRL+Shift+P) do VSCode:

ext install ZixuanChen.vitest-explorer



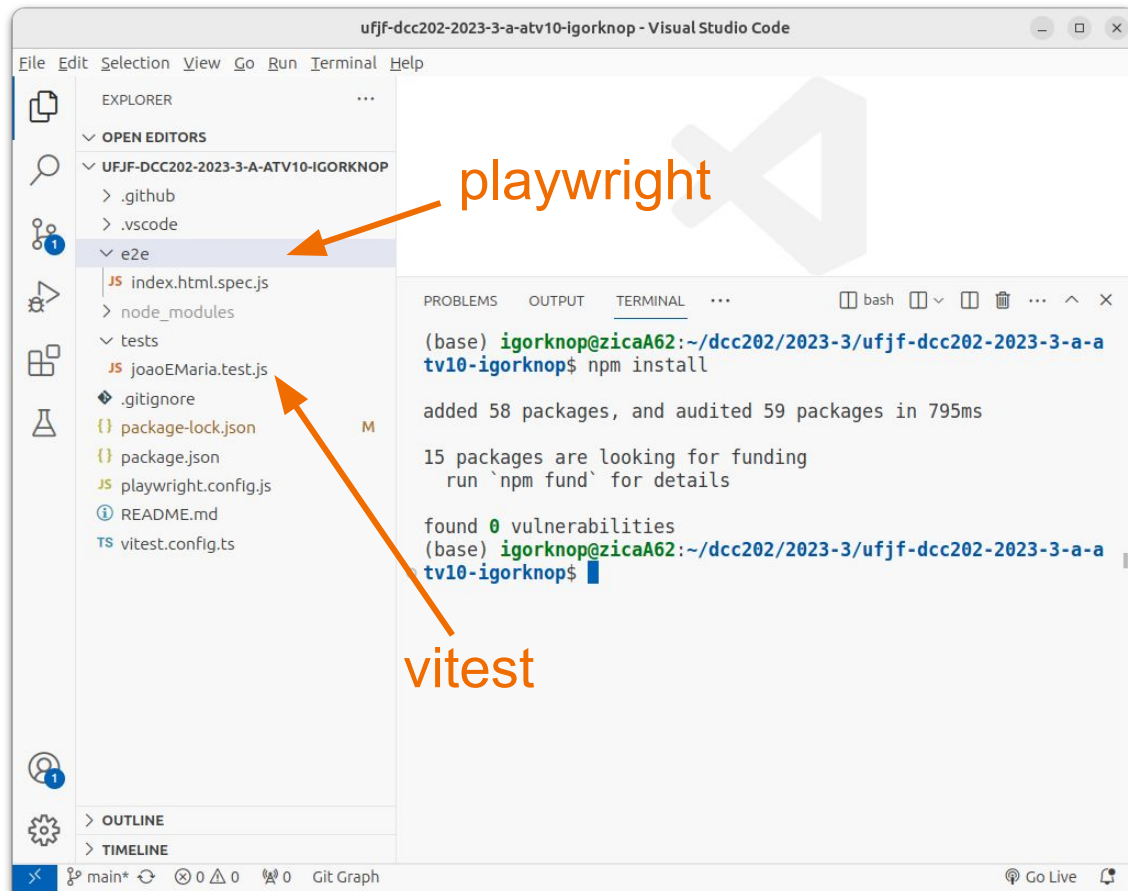
Instalando as dependências

No terminal, instale as dependências de teste com:

npm install

A partir de agora, os testes do playwright vão ficar dentro da pasta **e2e** (testes *end-to-end*)

Já os testes do vitest vão ficar dentro da pasta **tests**.



Adicione um documento index.html

Adicione um documento index.html na raiz do projeto.

Na cabeça do documento, adicione um elemento **script** com o tipo **module** e fonte externa para um arquivo **main.js**.

Mesmo que esse elemento não tem conteúdo, é necessário fechar com a etiqueta final **</script>**.

Atenção: Os testes do playwright agora vão rodar em cima do <http://localhost:5500>, portanto é preciso carregar o live server para rodar os testes!

```
<!DOCTYPE html>
<html lang="pt">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DCC202 - Atividade 10: Igor Knop</title>
  <script src="main.js" type="module"></script>
</head>

<body>
  <main>
    <h1>DCC202 - Atividade 10: Igor Knop</h1>
    <p>Esta é a página principal tem código JavaScript nela. Você só verá o
    resultado no console e durante a depuração.
    </p>
  </main>
</body>

</html>
```

Adicione um módulo main.js e um joaoEMaria.js

Adicione um arquivo **main.js** e outro **joaoEMaria.js** na raiz do projeto.

Por enquanto, só adicione uma impressão no terminal.

Como ele está na mesma pasta do index.html, confira se o atributo src está correto com o **<script>**.

Fique atento para as letras maiúsculas e minúsculas nos nomes de arquivo!

```
// main.js  
console.log('módulo main');
```

```
// joaoEMaria.js  
console.log('módulo joaoEMaria');
```


Duas ferramentas de testes

Resumindo, agora na tela de testes, nós temos dois executores de testes:

playwright é utilizado para rodar nossos testes na tela final, como um usuário faria.

vitest é utilizado para rodar nossos testes em módulos JavaScript isolados.

É importante agora não deixar um executar os testes do outro, pois não vão funcionar!

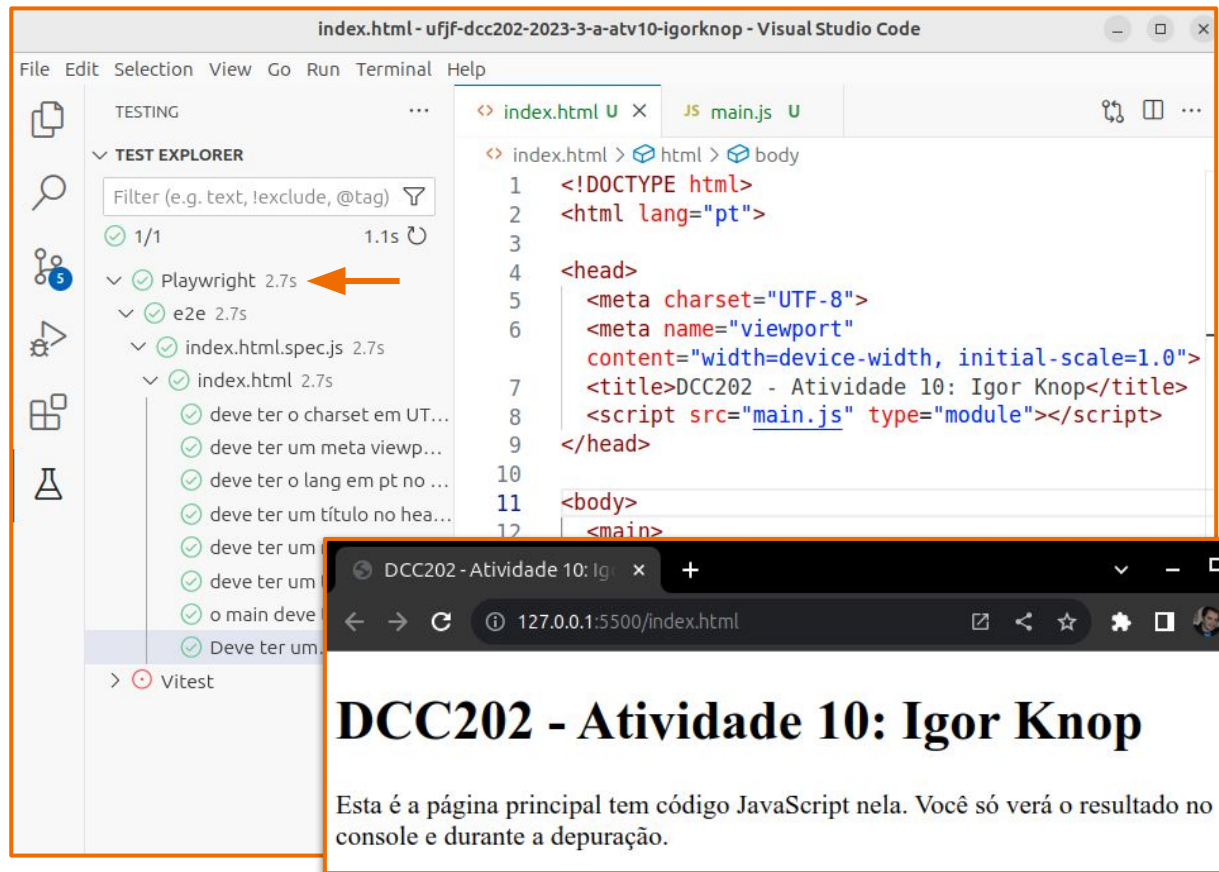


Execute os testes de e2e no playwright

O playwright vai usar o arquivo `e2e/index.html.spec.js` para executar os testes.

Ele vai acessar o servidor executado pelo live server na porta 5500 e procurar pelo que é pedido.

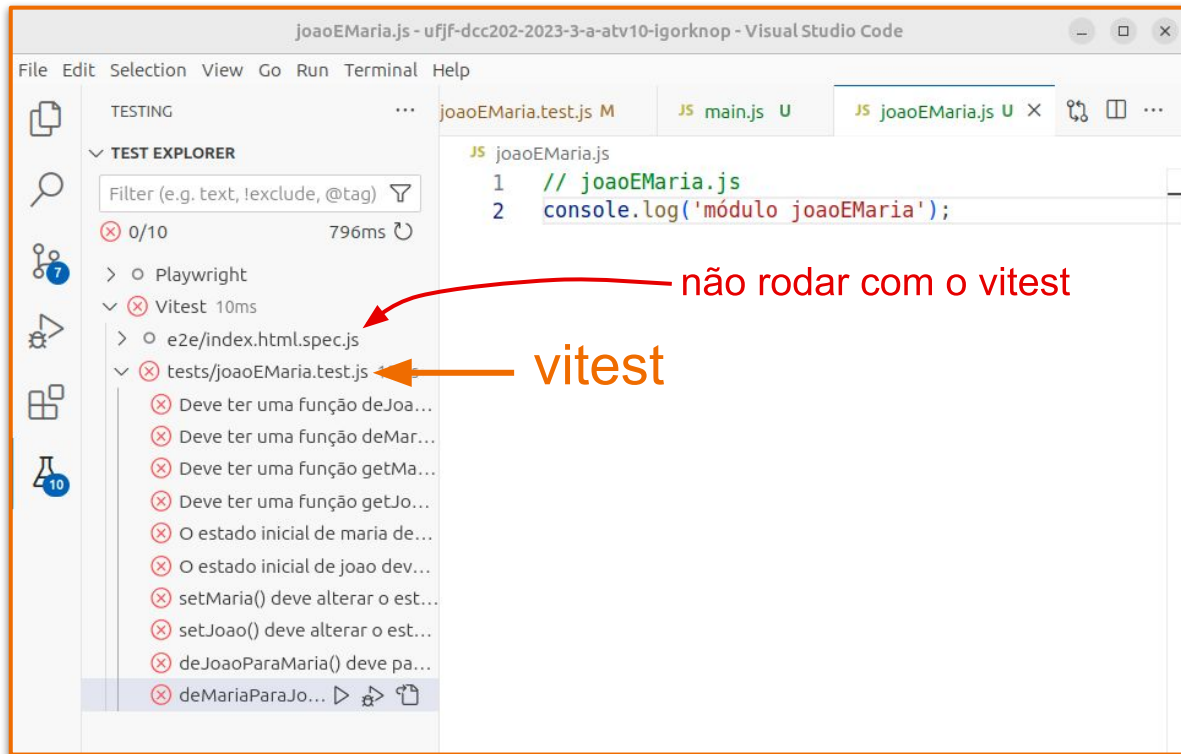
Se todos os teste continuam falhando, verifique se o servidor está rodando e se o `index.html` pode ser acessado no endereço e porta.



Não execute o e2e com o vitest

Por um problema na extensão do vitest, ele está ignorando o arquivo de configuração e está mostrando também os testes e2e:

- não pode executar testes e2e com o vitest!
- Quando for testar a tela, tem que ir na opção do playwright.

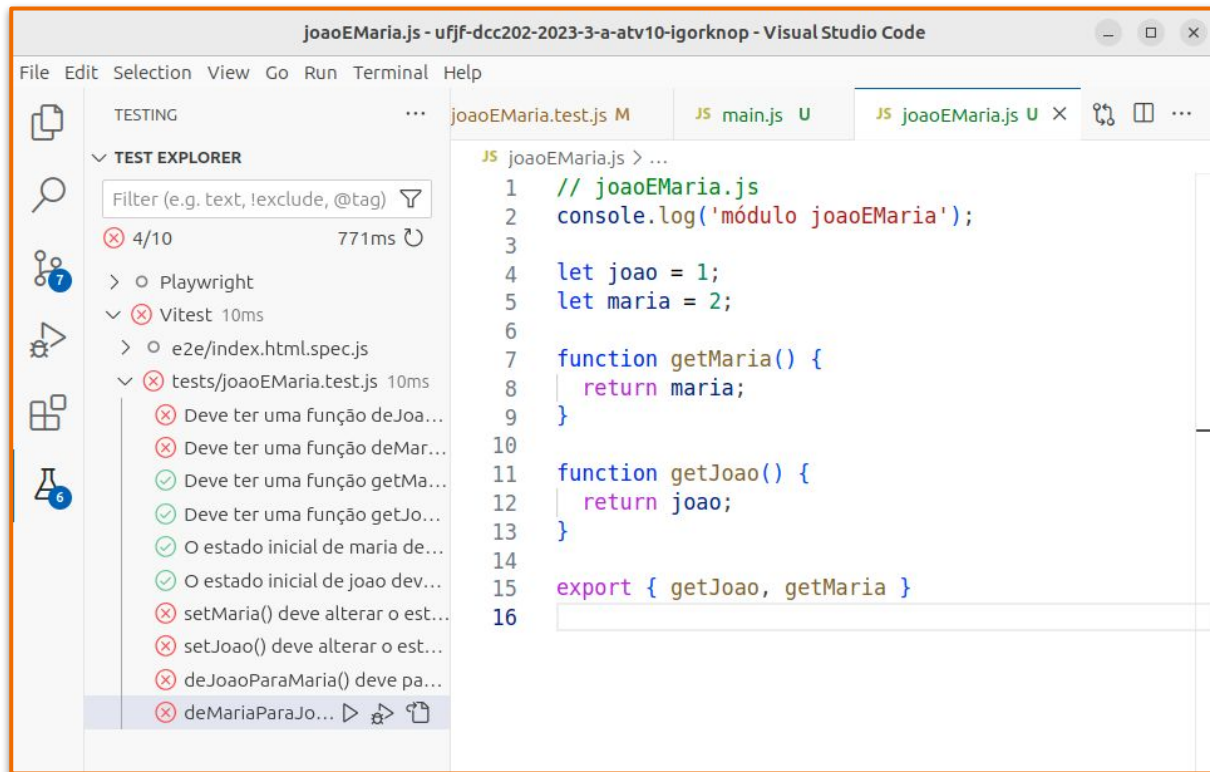


Escondendo o estado no módulo joaoEMaria.js

Nosso objetivo é manter um estado no escopo de módulo do joaoEMaria.

A única forma de acessar e modificar esse estado será via funções que iremos exportar para uso em outros módulos.

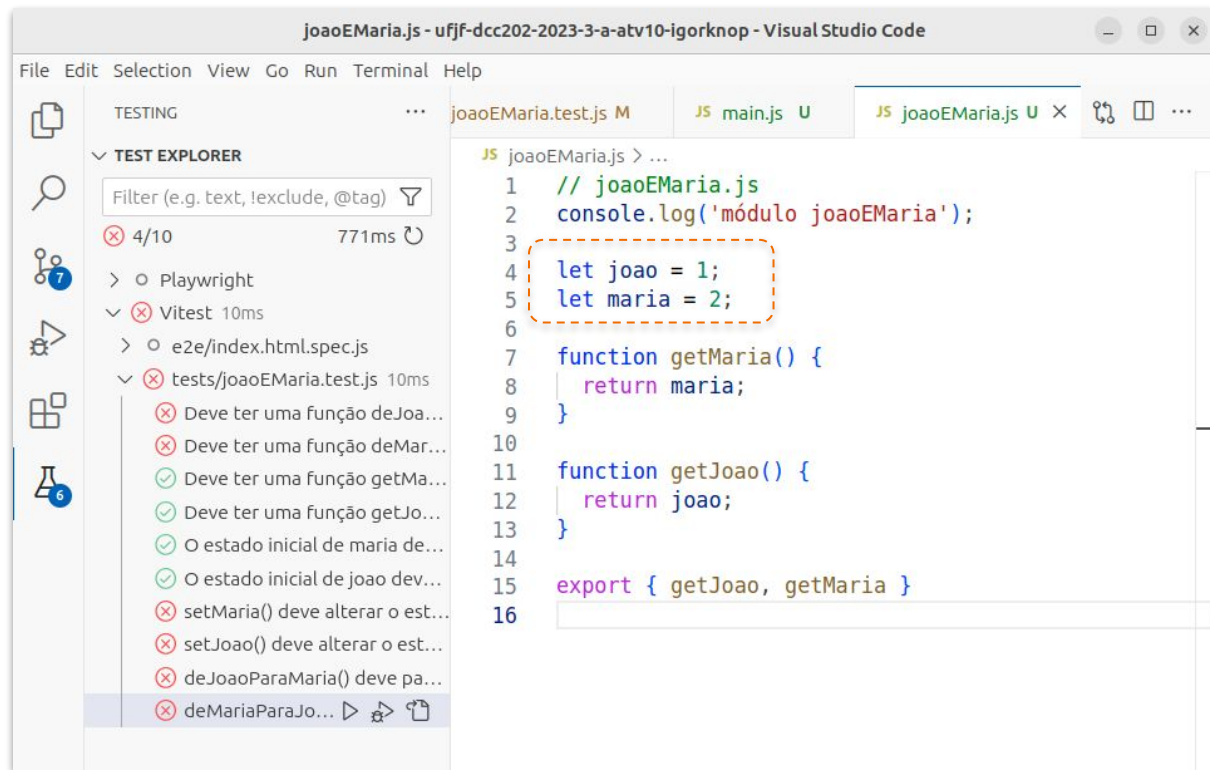
Ao definir métodos para alterar um estado isolado, estamos criando uma interface de programação de aplicações (API) bem rudimentar.



Escondendo o estado no módulo joaoEMaria.js (2)

O estado pode ser representado apenas como duas variáveis comuns.

Como uma variável definida em um escopo só é vista no escopo e escopos mais internos, não há chance de conflito com outras partes da aplicação.



The screenshot shows the Visual Studio Code editor with the file `joaoEMaria.js` open. The file content is as follows:

```
1 // joaoEMaria.js
2 console.log('módulo joaoEMaria');
3
4 let joao = 1;
5 let maria = 2;
6
7 function getMaria() {
8   return maria;
9 }
10
11 function getJoao() {
12   return joao;
13 }
14
15 export { getJoao, getMaria }
16
```

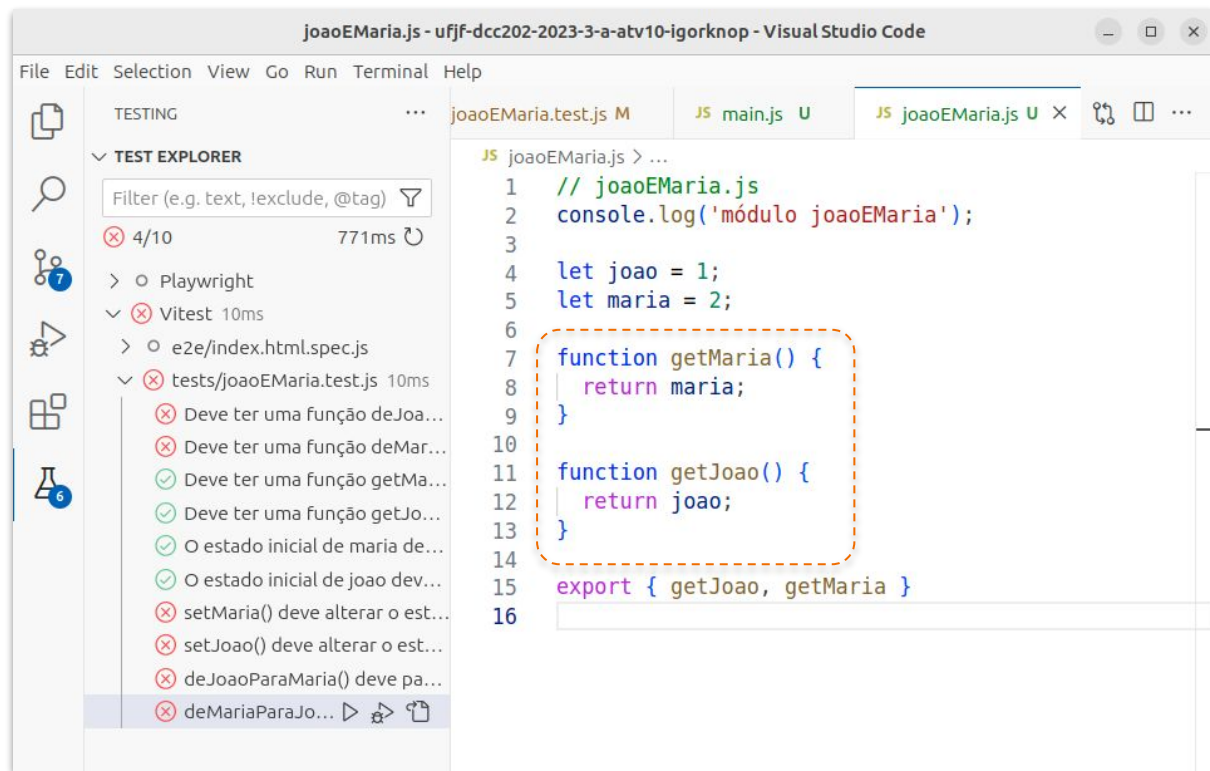
The Test Explorer on the left shows a list of tests under the `tests/joaoEMaria.test.js` file. The tests are:

- Deve ter uma função deJoa... (Failed)
- Deve ter uma função deMar... (Failed)
- Deve ter uma função getMa... (Passed)
- Deve ter uma função getJo... (Passed)
- O estado inicial de maria de... (Passed)
- O estado inicial de joao dev... (Passed)
- setMaria() deve alterar o est... (Failed)
- setJoao() deve alterar o est... (Failed)
- deJoaoParaMaria() deve pa... (Failed)
- deMariaParaJo... (Failed)

Escondendo o estado no módulo joaoEMaria.js (3)

Duas funções de acesso vão retornar os valores. Uma para cada variável (apesar que futuramente usarmos uma só para tipos mais complexos).

Essas funções, declaradas no escopo do módulo conseguem enxergar o valor dessas variáveis mesmo quando chamadas futuramente. Efeito que chamamos de ***closure***.



The screenshot shows the Visual Studio Code editor with the file `joaoEMaria.js` open. The file contains the following code:

```
1 // joaoEMaria.js
2 console.log('módulo joaoEMaria');
3
4 let joao = 1;
5 let maria = 2;
6
7 function getMaria() {
8   return maria;
9 }
10
11 function getJoao() {
12   return joao;
13 }
14
15 export { getJoao, getMaria }
```

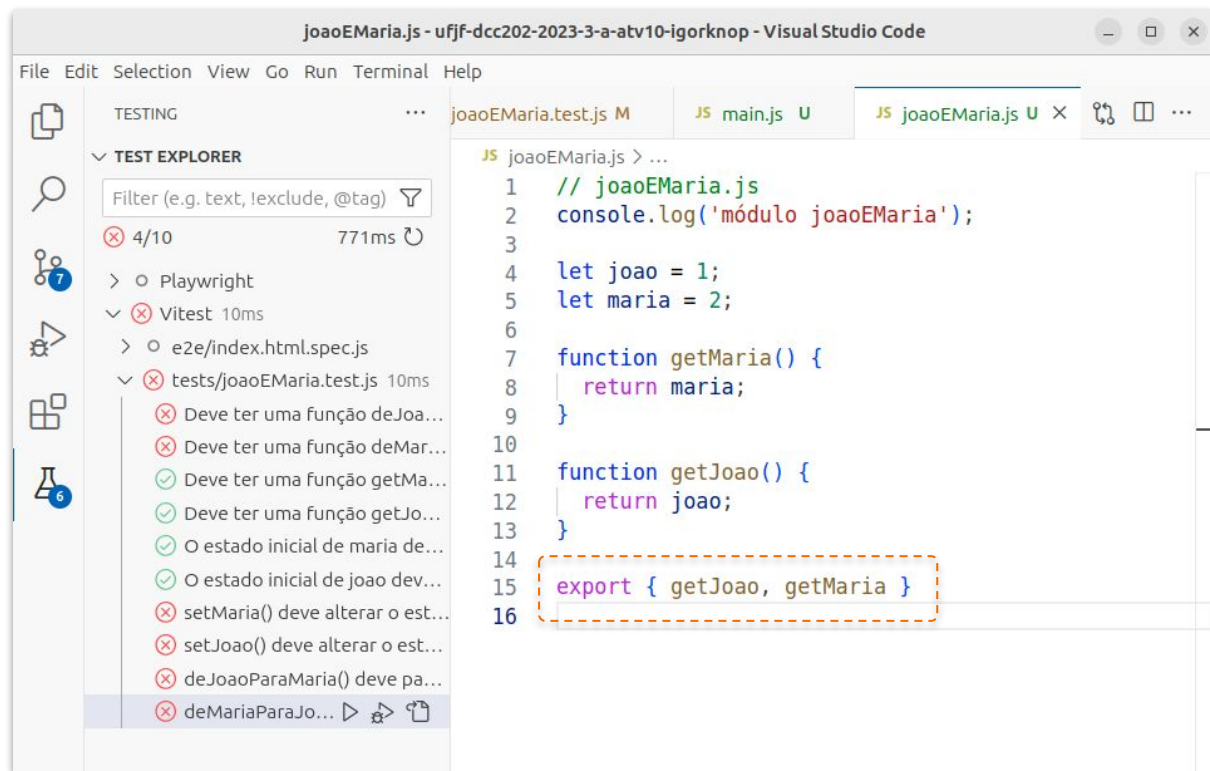
The `getMaria()` and `getJoao()` functions are highlighted with a dashed orange box, indicating they are the focus of the discussion. The left sidebar shows the **TEST EXPLORER** panel with a filter set to `4/10` and a total time of `771ms`. The test results list shows several failed tests (marked with red X) and several passed tests (marked with green checkmarks).

Escondendo o estado no módulo joaoEMaria.js (4)

Para que essas funções sejam utilizadas fora do módulo, precisamos exportá-las.

Os módulos ECMAScript nos permitem exportar com a lista de identificadores de função no módulo que elas foram declaradas e importar no módulo que vai fazer o uso.

A palavra **export** pode ser colocada diante de cada função ou ter uma lista de exportações em alguma parte do código. Mas só pode haver um **export**.

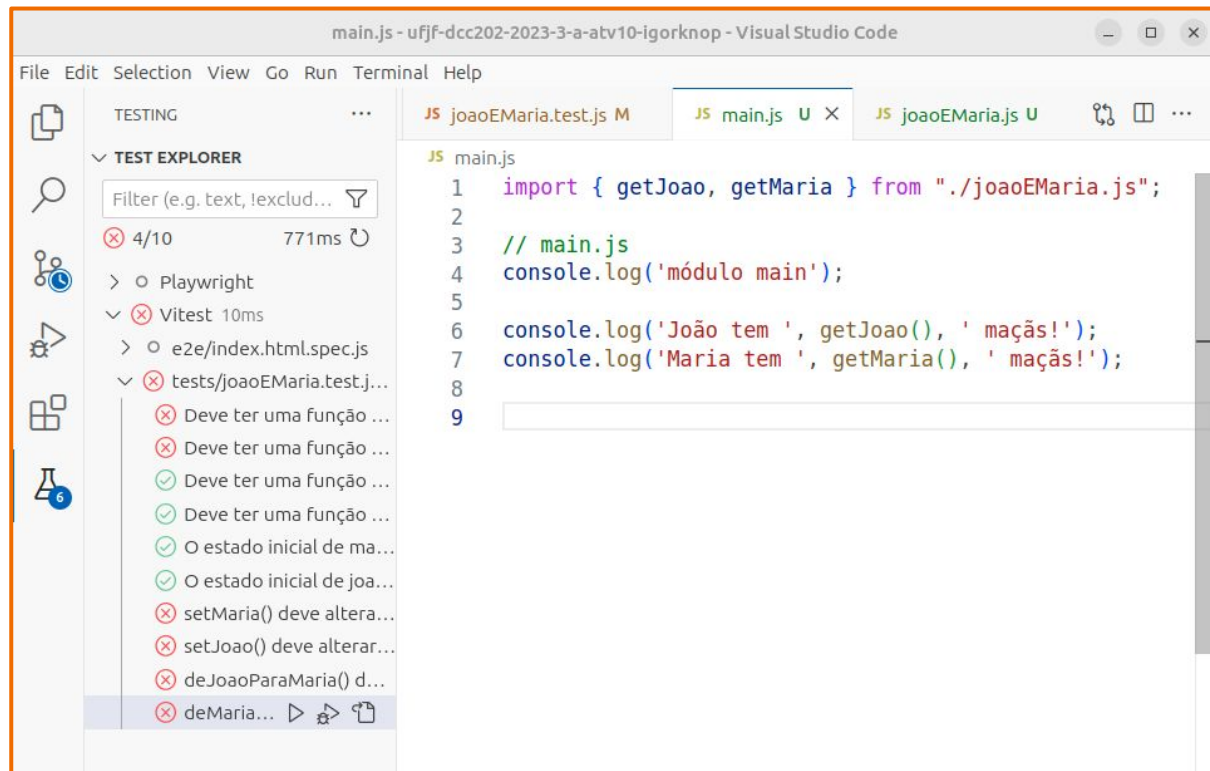


Acessando o estado no módulo main.js

Para enxergarmos os estados no módulo main.js, precisamos primeiro importar as funções de acesso.

O comando **import** especifica quais funções temos interesse e também onde encontrá-las. Importante:

- No navegador, sempre devemos explicitar a extensão do arquivo **.js** (mesmo que o vscode diga que não precisa)
- O **./** significa pasta atual. Se não colocar, ele vai pensar que é um módulo externo, instalado via **npm**.

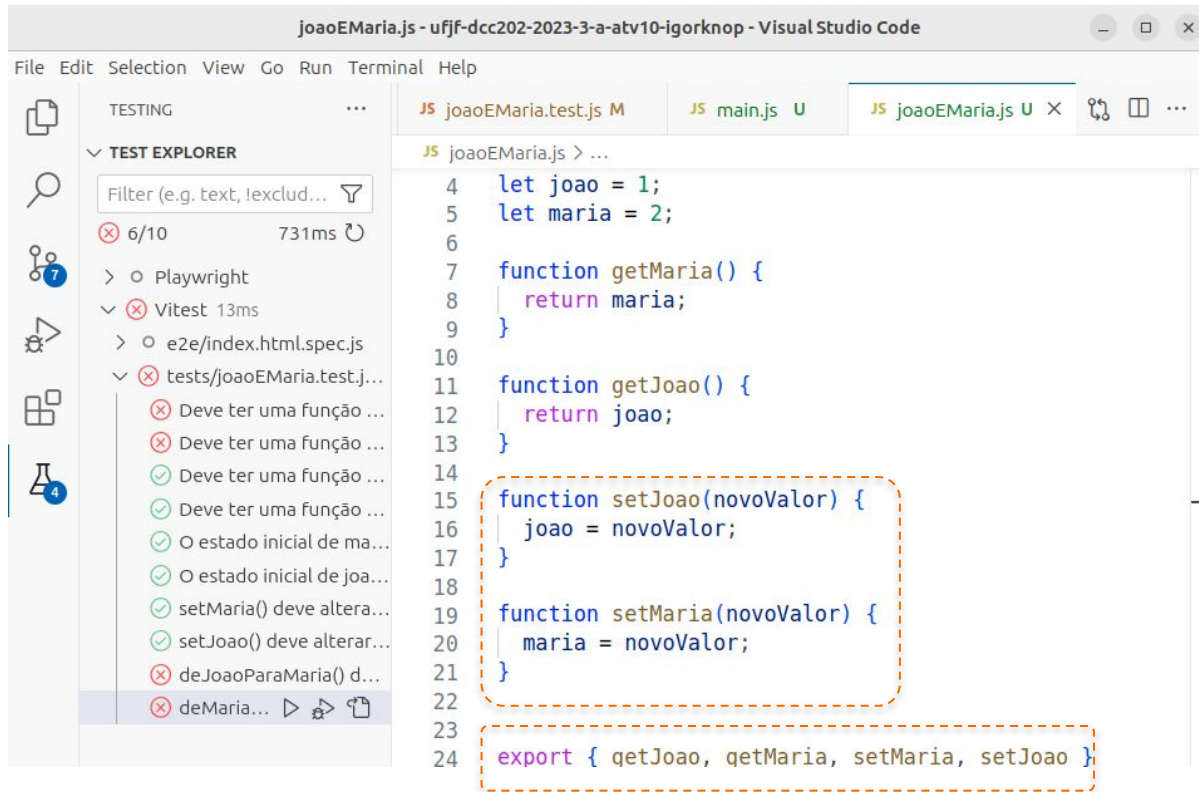


Alterando o estado

O estado só pode ser alterado por funções que enxerguem as variáveis. Portanto, precisamos de mais funções no módulo **joaoEMaria.js**.

As funções **setMaria()** e **setJoao()** serão a única forma de alterar externamente os valores.

Elas devem ser exportadas e importadas para uso em outro módulo.



The screenshot shows the Visual Studio Code interface with the file `joaoEMaria.js` open. The left sidebar displays the **TEST EXPLORER** panel, showing a list of tests for `joaoEMaria.test.js`. The tests are categorized by status: failed (red X) and passed (green checkmark). The failed tests are "Deve ter uma função ..." (three instances), and the passed tests are "Deve ter uma função ..." (three instances), "O estado inicial de ma...", "O estado inicial de joa...", "setMaria() deve altera...", "setJoao() deve alterar...", "deJoaoParaMaria() d...", and "deMaria...". The main editor shows the code for `joaoEMaria.js`, which includes variables `joao` and `maria`, and functions `getMaria()`, `getJoao()`, `setJoao(novoValor)`, and `setMaria(novoValor)`. The `setJoao` and `setMaria` functions are highlighted with dashed orange boxes. The `export` statement at the bottom is also highlighted with a dashed orange box.

```
joaoEMaria.js - utf8-dcc202-2023-3-a-atv10-igorknop - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TESTING
...
JS joaoEMaria.test.js M JS main.js U JS joaoEMaria.js U x

TEST EXPLORER
Filter (e.g. text, !exclud...
6/10 731ms
> o Playwright
v x Vitest 13ms
> o e2e/index.html.spec.js
v x tests/joaoEMaria.test.j...
  x Deve ter uma função ...
  x Deve ter uma função ...
  x Deve ter uma função ...
  x Deve ter uma função ...
  x O estado inicial de ma...
  x O estado inicial de joa...
  x setMaria() deve altera...
  x setJoao() deve alterar...
  x deJoaoParaMaria() d...
  x deMaria...

4 let joao = 1;
5 let maria = 2;
6
7 function getMaria() {
8   return maria;
9 }
10
11 function getJoao() {
12   return joao;
13 }
14
15 function setJoao(novoValor) {
16   joao = novoValor;
17 }
18
19 function setMaria(novoValor) {
20   maria = novoValor;
21 }
22
23
24 export { getJoao, getMaria, setMaria, setJoao }
```

Alterando o estado (2)

Como a única entrada de dados para o módulo é através das funções **setMaria()** e **setJoao()**, elas são um ótimo lugar para garantirmos que valores com significado errado para o problema sejam ignorados (ou lance um erro, como faremos futuramente).

Podemos, apenas para exemplo, falar que valores negativos serão registrados como zero.



joaoEMaria.js - uffj-dcc202-2023-3-a-atv10-igorknop - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TESTING

TEST EXPLORER

Filter (e.g. text, !exclud...)

6/10 731ms

- Playwright
- Vitest 13ms
 - e2e/index.html.spec.js
 - tests/joaoEMaria.test.j...
 - Deve ter uma função ...
 - Deve ter uma função ...
 - Deve ter uma função ...
 - Deve ter uma função ...
 - O estado inicial de ma...
 - O estado inicial de joa...
 - setMaria() deve altera...
 - setJoao() deve alterar...
 - deJoaoParaMaria() d...
 - deMaria...

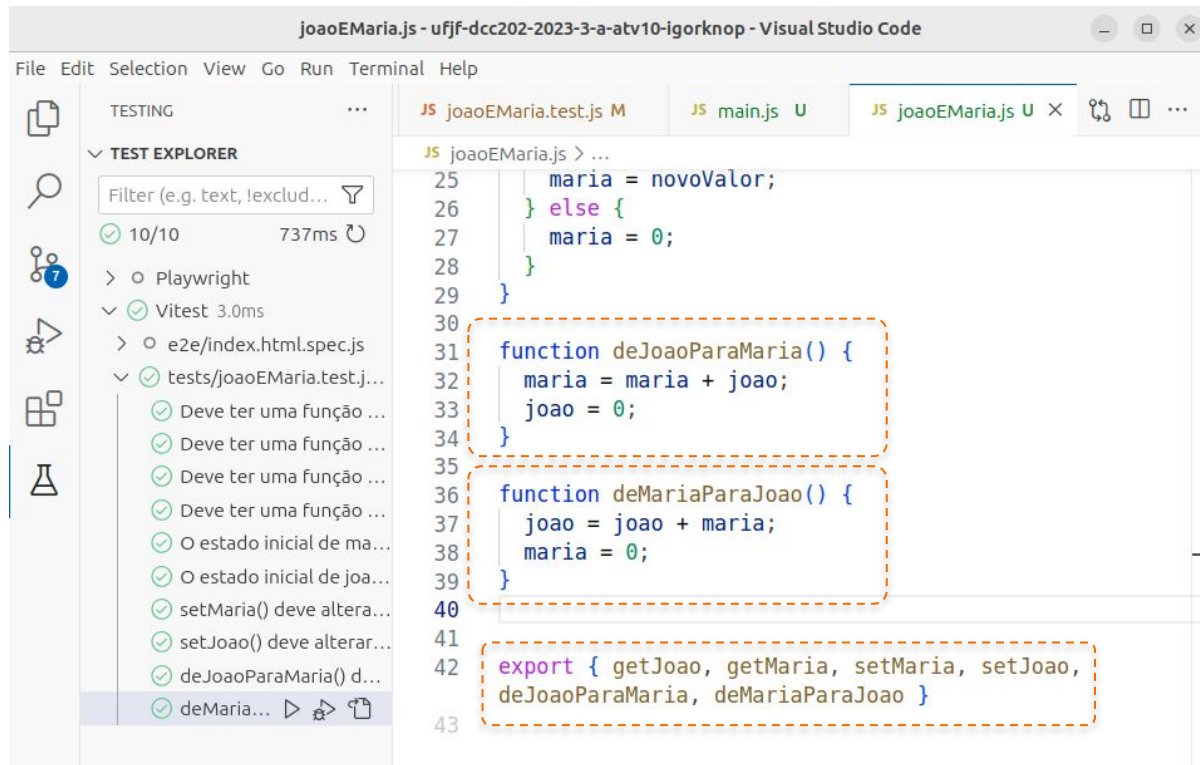
```
14
15 function setJoao(novoValor) {
16     if (novoValor > 0) {
17         joao = novoValor;
18     } else {
19         joao = 0;
20     }
21 }
22
23 function setMaria(novoValor) {
24     if (novoValor > 0) {
25         maria = novoValor;
26     } else {
27         maria = 0;
28     }
29 }
30
31
32 export { getJoao, getMaria, setMaria, setJoao }
33
```

Adicionando operações mais complexas

As operações sobre o estado que fazem mais sentido para o seu problema aumentam o nível de abstração.

Por exemplo, passar todas as maçãs de uma pessoa para a outra. Podemos criar essas operações também.

Nesse momento, todos os testes estão passando no **vitest**!



The screenshot shows the Visual Studio Code interface with the file `joaoEMaria.js` open. The left sidebar displays the **TEST EXPLORER** with a filter set to `10/10` and a duration of `737ms`. The test suite `tests/joaoEMaria.test.js` is expanded, showing a list of 10 tests, all of which are passing (indicated by green checkmarks). The main editor shows the code for `joaoEMaria.js`, which includes a `deJoaoParaMaria` function, a `deMariaParaJoao` function, and an `export` statement. The code is as follows:

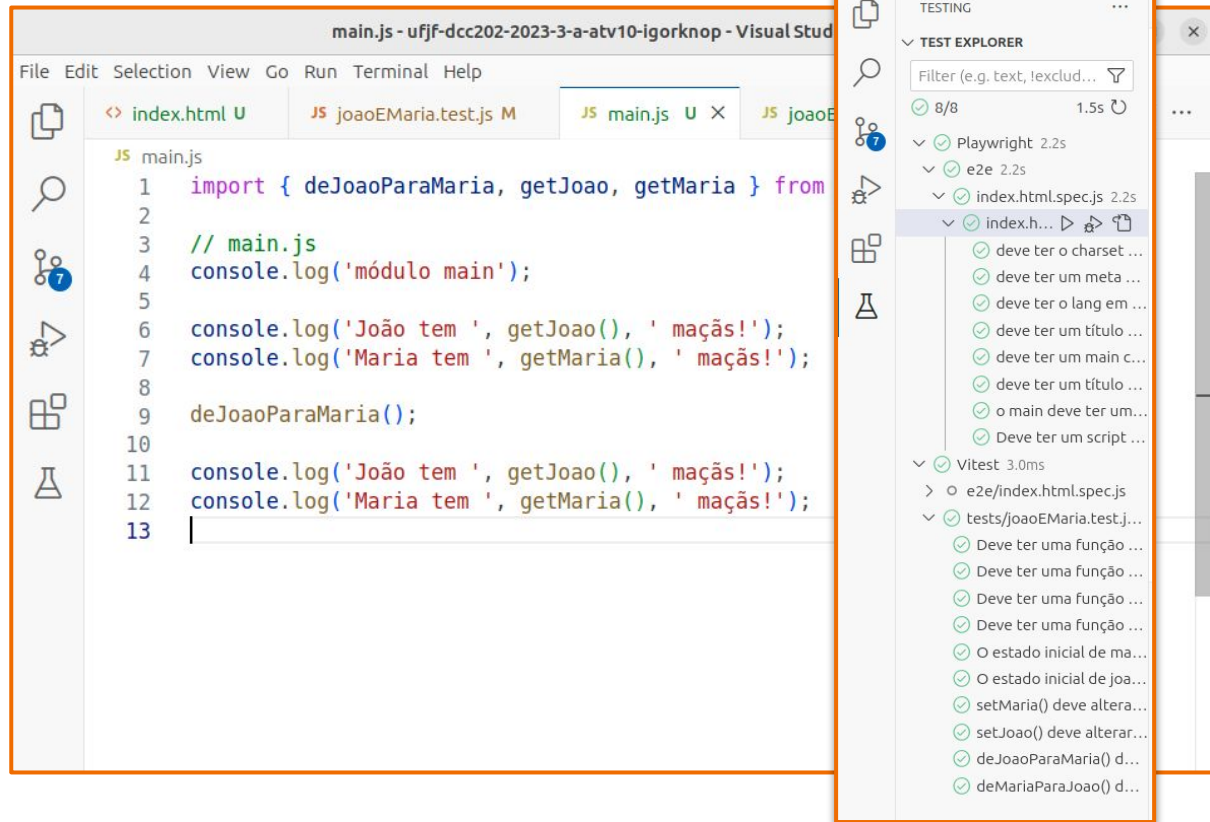
```
25     maria = novoValor;
26   } else {
27     maria = 0;
28   }
29 }
30
31 function deJoaoParaMaria() {
32   maria = maria + joao;
33   joao = 0;
34 }
35
36 function deMariaParaJoao() {
37   joao = joao + maria;
38   maria = 0;
39 }
40
41
42 export { getJoao, getMaria, setMaria, setJoao,
43         deJoaoParaMaria, deMariaParaJoao }
```

Acessando as operações

Se seu módulo funciona sozinho, grandes chances do código externo que faz uso dele funcionar também!

Dessa forma nós:

- Dividimos problemas maiores em problemas menores;
- Garantimos que as partes menores estão se comportando como deveriam;
- Evitamos que um código interfira acidentalmente no outro, pois cada módulo é isolado e só pode ser acessado pela sua interface.



Vitest no terminal

A extensão do vitest ainda está bem instável. Se você tiver problemas para rodá-la, você pode usar a versão modo texto, executando no terminal:

npm run test

O projeto da atividade já vem configurado para rodar o vitest agora.

```
(base) igorknop@zicaA62:~/dcc202/2023-3/ufjf-dcc202-2023-3-a-atv10
-igorknop$ npm run test

> ufjf-dcc202-2023-3-atv10-template@1.0.0 test
> vitest

DEV v0.34.6 /home/igorknop/dcc202/2023-3/ufjf-dcc202-2023-3-a-atv10-igorknop

stdout | unknown test
módulo joaoEMaria

✓ tests/joaoEMaria.test.js (10)
  ✓ Deve ter uma função deJoaoParaMaria() exportada no módulo
  ✓ Deve ter uma função deMariaParaJoao() exportada no módulo
  ✓ Deve ter uma função getMaria() exportada no módulo
  ✓ Deve ter uma função getJoao() exportada no módulo
  ✓ O estado inicial de maria deve ser 2
  ✓ O estado inicial de joao deve ser 1
  ✓ setMaria() deve alterar o estado de maria
  ✓ setJoao() deve alterar o estado de joao
  ✓ deJoaoParaMaria() deve passar todas as maçãs de joao para maria
  ✓ deMariaParaJoao() deve passar todas as maçãs de maria para joao

Test Files  1 passed (1)
Tests       10 passed (10)
Start at    15:16:31
Duration    349ms (transform 27ms, setup 0ms, collect 16ms, tests 4ms, environment 0ms, prepare 67ms)

PASS Waiting for file changes...
press h to show help, press q to quit
```

Para saber mais...

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