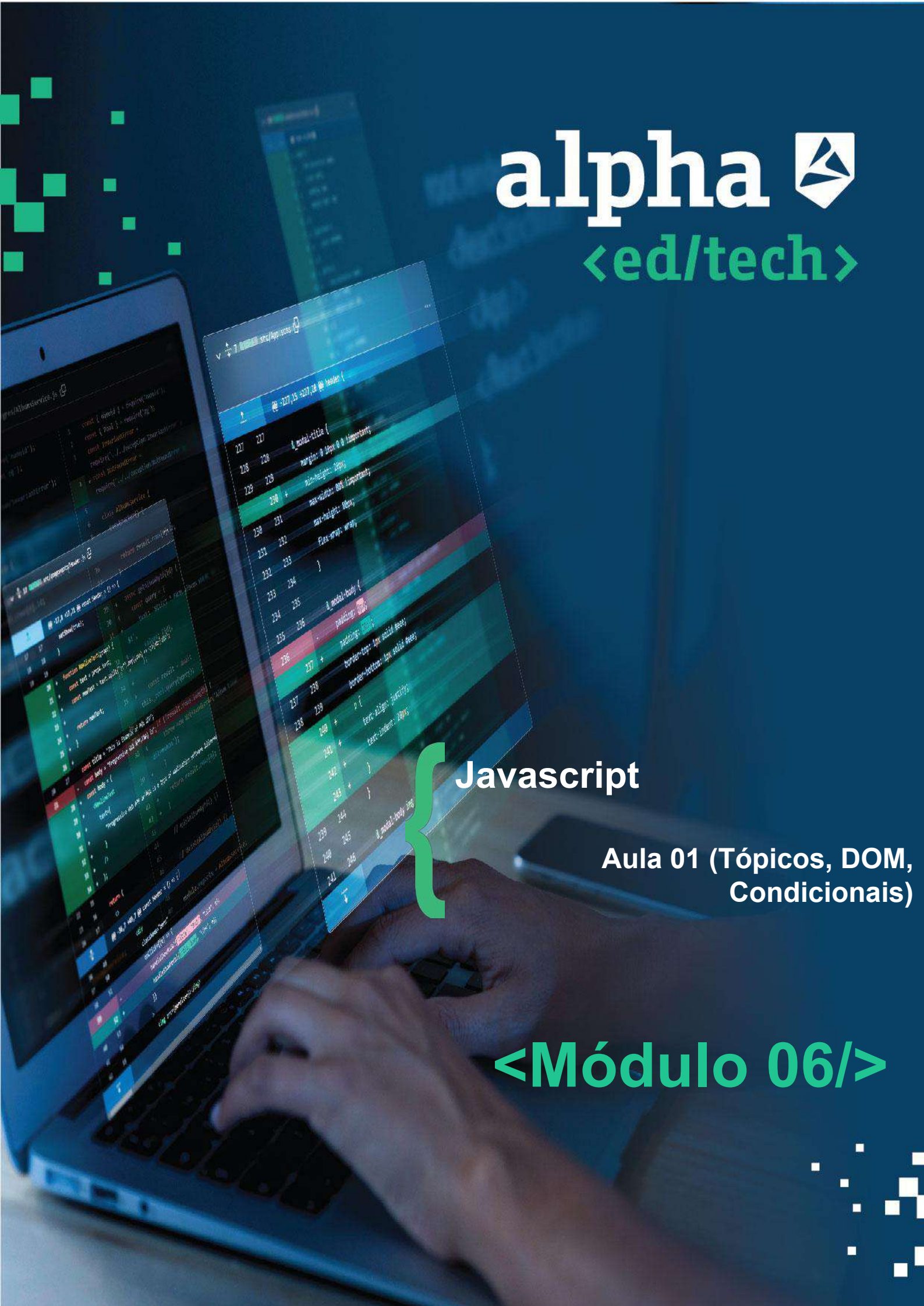


alpha
<ed/tech>

# { Javascript

## Aula 01 (Tópicos, DOM, Condicionais)

## <Módulo 06/>

## Tópicos adicionais

# Introdução

Parabéns ! Você já consegue escrever programas que fazem entrada de dados, armazenam esses dados na memória, fazem cálculos e imprimem resultados na saída para o usuário ver.

Daqui em diante, vamos adicionar detalhes aos conceitos que já introduzimos, e novos conceitos diretamente relacionados.

Segue um sumário do que vamos abordar:

- **Tópicos adicionais sobre números**
  - Formatos numéricos
  - Números especiais Infinity e NaN
- **Tópicos adicionais sobre strings**
  - Caractere de escape e quebra de linha
  - Template literal
  - Indexação de string
  - Propriedade `length` de string
- **Tópicos adicionais sobre variáveis**
  - Formas de declaração de variável
  - Formas de atribuição de variável
  - Escopo global
- **Tópicos adicionais sobre conversão de tipos**
  - Regras da conversão explícita de tipos com Number
  - Regras da conversão implícita de tipos
  - Outros comandos de conversão de tipos para número
- **Tópicos adicionais sobre Funções**
  - Introdução
  - O que é uma função ?
  - Código da função *versus* invocar a função
  - Parâmetros
  - Funções nativas *versus* funções do programador
- **Tópicos adicionais sobre erros no código**
  - Introdução
  - Primeira categoria de erro: Erro de Sintaxe
  - Segunda categoria de erro: Erro de Execução
  - Terceira categoria de erro: Erro de Lógica
  - Depuração (debug) de código
- **Tópicos adicionais sobre a linguagem javascript**
  - Diagrama parcial da sintaxe do javascript
  - História e padronização do javascript
  - Javascript Engine
  - Javascript Runtime
  - Manuais e referências de javascript

# Tópicos adicionais sobre números

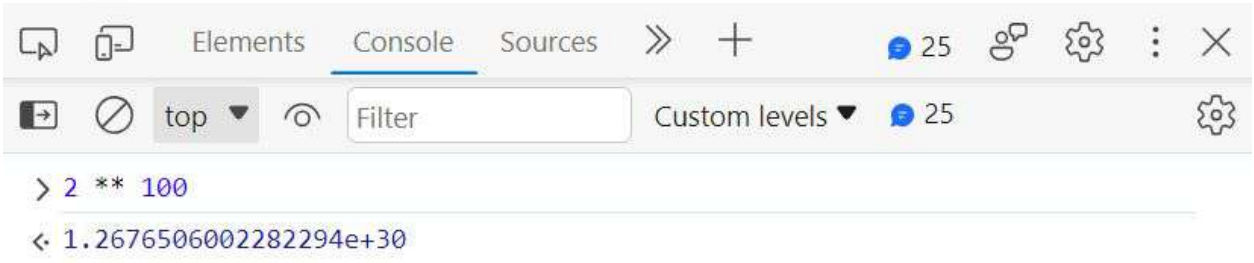
## Formatos numéricos

O javascript suporta números inteiros como `-12`, `0`, `3`.

E também suporta números "quebrados", que na programação são chamados de **float**. Por exemplo `3.141592`, `-2.5`

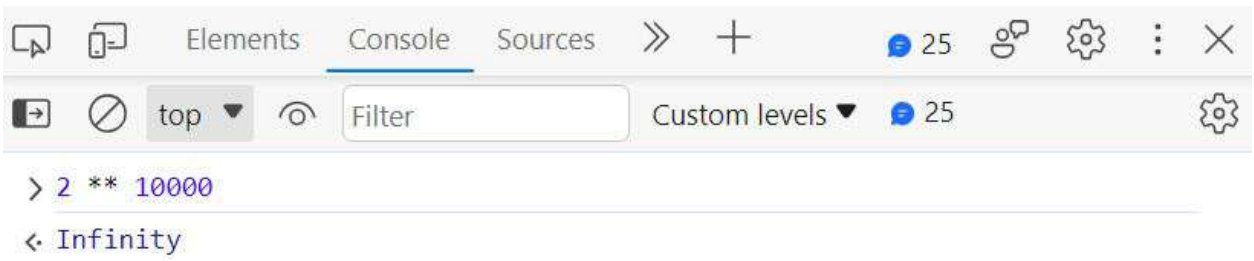
Também é possível usar notação científica, como `-2.5e4`. Isso significa −2.5104, ou seja, é o mesmo que `-25000`.

Quando algum cálculo resulta num número muito grande, o resultado pode aparecer nessa notação, por exemplo:




## Números especiais Infinity e NaN

Os números têm um limite máximo. Se passar desse limite, o resultado é um número especial **Infinity**, por exemplo:




O `Infinity` não é uma string, é um número mesmo.

Divisão de um número não-zero por zero, por exemplo `5/0`, também retorna `Infinity`.

Cálculos com `Infinity` geralmente resultam em `Infinity`, por exemplo `2 * Infinity` retorna `Infinity`.

Quando o javascript detecta algum cálculo inválido, o resultado é outro número especial chamado `NaN` (de novo, não é uma string, é um número).

`NaN` significa "not a number" (não é um número).

O nome é confuso porque parece uma contradição: `NaN` pertence tecnicamente ao tipo de dados numérico, mas a sigla significa "não é um número".

Alguns cálculos que resultam `NaN`:

- `0/0`
- `Infinity - Infinity`
- `Infinity / Infinity`

Cálculos com `NaN` sempre resultam `NaN`, por exemplo:

- `2 * NaN` retorna `NaN`
- `3 / NaN` retorna `NaN`
- `NaN - NaN` retorna `NaN`

# Tópicos adicionais sobre strings

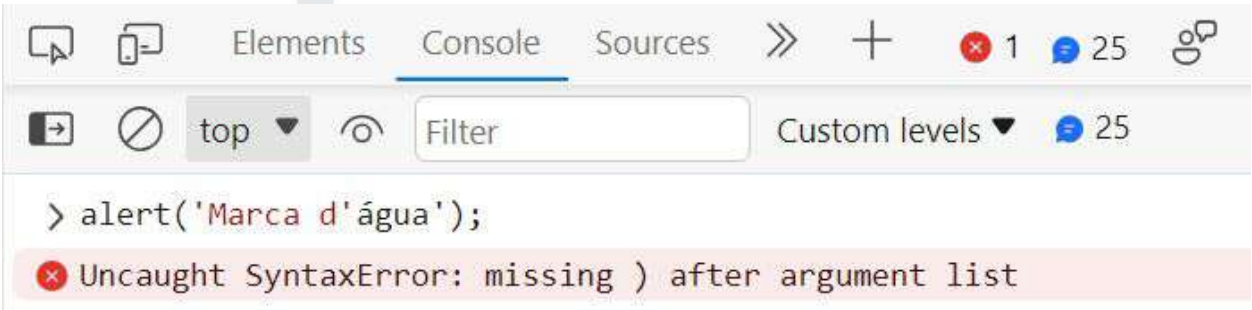
## Caractere de escape e quebra de linha

Suponha que você quer imprimir `Marca d'água` num alerta.

Você tenta o seguinte código:

- `alert('Marca d'água');`

Mas isso vai gerar um erro (e o alerta não vai aparecer):




Como o Console diz, esse é um Erro de Sintaxe (SyntaxError).

O problema é o uso de `'` no meio do texto.

Quando o javascript vê o primeiro `'` ao lado esquerdo do `M`, ele entende que está começando uma string.

Ao ver o próximo `'` ao lado direito do `d`, o javascript entende que a string acaba ali.

Então a string é `'Marca d'`.

O restante do código não é considerado string, e fica inválido.

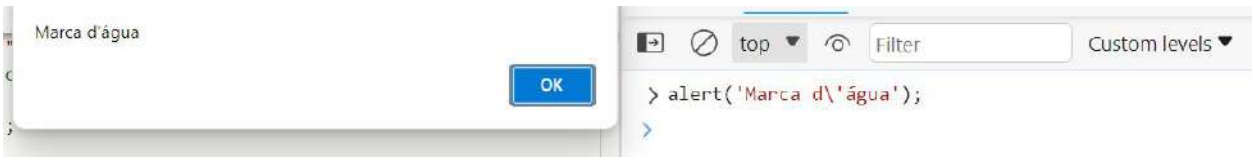
Por isso que o erro é um Erro de Sintaxe: significa que o código não está seguindo as regras de sintaxe do javascript, está escrito errado.

Para resolver isso, tem dois jeitos:

1. Use aspas duplas para delimitar a string.
   Ou seja, `alert("Marca d'água");`
   Assim o primeiro `"` inicia a string, que só vai terminar no próximo `"`.
   Logo a string é `"Marca d'água"` como desejado, não tem nenhum erro.

2. Use o caractere de escape `\`
   Fica assim: `alert('Marca d\'água');`
   A barra invertida é chamada de **Caractere de Escape**. Quando usada, ela simboliza que o próximo caractere, neste caso `'`, não deve ser entendido como fim da string.

Veja que funciona:



# Tópicos adicionais sobre strings

O caractere de escape também pode ser usado seguido de outros caracteres.

Em qualquer caso, ele vai significar que o próximo caractere tem algum significado especial.

Qual é o significado depende do caractere que vem em seguida.

Por exemplo, se você quer mostrar um alerta com duas linhas, assim:


Você poderia tentar quebrar uma string em duas linhas, assim:

```
alert("oi
tchau");
```

Mas isso *não funciona*, porque strings em javascript não podem pular linha (exceção: strings iniciadas com as aspas invertidas ` , mas falaremos sobre isso depois).

Se não pode pular linha, como fazer aquele alerta ?

A resposta é: `alert("oi\ntchau");`

O segredo é a sequência `\n`. Ela significa uma quebra de linha.

Obs: Como a barra invertida é interpretada pelo javascript como caractere de escape, ela não aparece na string.

Então como você faria para mostrar um alerta onde a barra invertida *aparece* ? Ou seja:


A resposta é usar duas vezes o caractere de escape:

- `alert("Uma barra invertida \\");`

Assim a primeira barra invertida, por ser o caractere de escape, faz o javascript entender que o próximo caractere é especial.

Ao ver o próximo caractere, que é outra barra invertida, normalmente o javascript o entenderia como caractere de escape, mas a primeira barra invertida disse que esse caractere deve ser interpretado como especial.

O que significa interpretar a segunda barra invertida como especial ? Significa interpretá-la como uma barra literal, em vez de como um escape.

Por isso que ela aparece no alerta.

# Tópicos adicionais sobre strings

## Template literal

Na discussão anterior sobre strings com aspas simples e duplas, faltou falarmos sobre as strings com aspas invertidas, como `` `Olá mundo !` ``

Essas strings têm duas vantagens.

A primeira é que elas *podem* pular linha, por exemplo o código abaixo é válido e mostra um alerta com duas linhas:

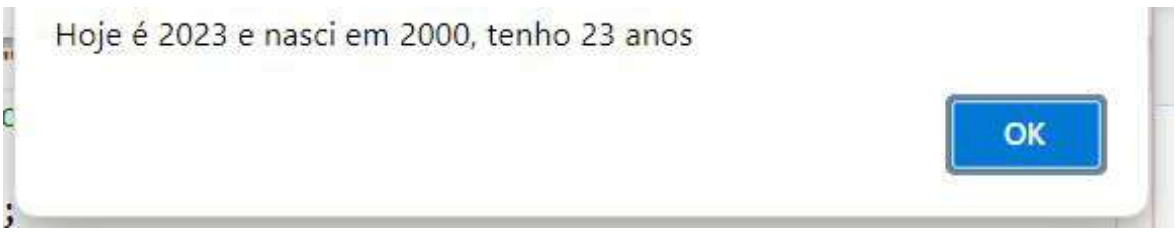
```
alert(`Oi
tchau`);
```

A segunda é que é mais fácil fazer concatenação com essas strings.

Por exemplo, considere o programa abaixo:

```
const hoje = Number(prompt("Ano atual"));
const nascimento = Number(prompt("Ano do seu nascimento"));

const idade = hoje - nascimento;

const mensagem =
  "Hoje é " + hoje + " e nasci em " + nascimento + ", tenho " + idade + " anos";

alert(mensagem);
```

Como você deve imaginar, ele mostra um alerta como este:




O aspecto ruim desse programa é que a construção da string `mensagem` é bagunçada, porque há várias partes sendo concatenadas para formar a mensagem.

Usando strings com aspas invertidas, existe outra maneira de fazer concatenação que é mais fácil de montar:

```
const mensagem = `Hoje é ${hoje} e nasci em ${nascimento}, tenho ${idade} anos`;
```

Isso faz a mesma coisa que antes: concatena as várias partes, substituindo as variáveis pelos valores numéricos delas.

O segredo é que, em strings com aspas invertidas, a sequência `${《Expressão》}` tem significado especial.

Ela significa que o javascript deve calcular a Expressão que está entre `{ }` e colocar o resultado na string.

A Expressão pode ser qualquer javascript válido.

Por exemplo:

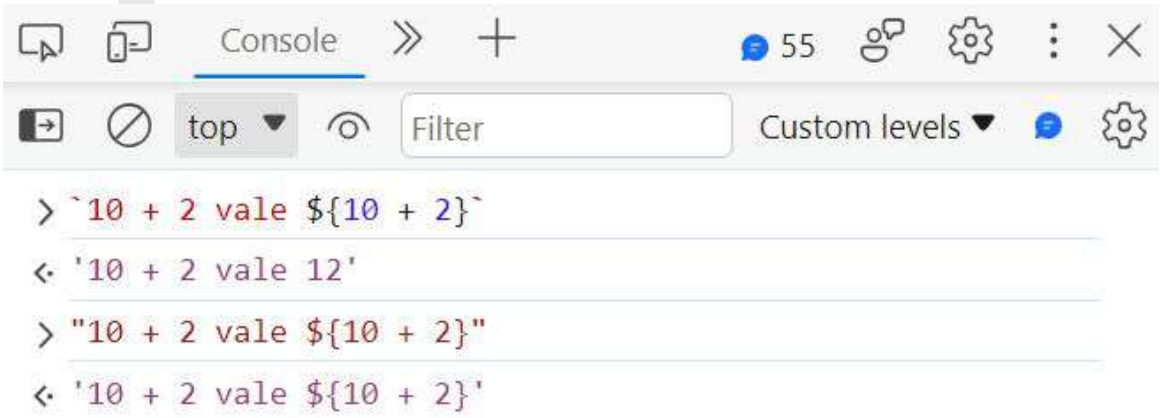
- `` `10 + 2 vale ${10 + 2}` `` ⇒ `` `10 + 2 vale 12` ``
- `` `Se tenho ${anos} anos, daqui a 2 anos terei ${anos + 2}` `` ⇒ `` `Se tenho 23 anos, daqui a 2 anos terei 25` ``

# Tópicos adicionais sobre strings

A sequência `${ ... }` só tem significado especial em strings com aspas invertidas, que também são chamadas de **Template Literal**.

Nas outras strings (aspas simples e duplas), a sequência `${ ... }` seria impressa literalmente.

Veja a diferença:




## Indexação de string

Os caracteres de uma string têm uma posição dentro da string: 1º caractere, 2º caractere, etc.

No javascript, a posição de um caractere é denominada **Índice**.

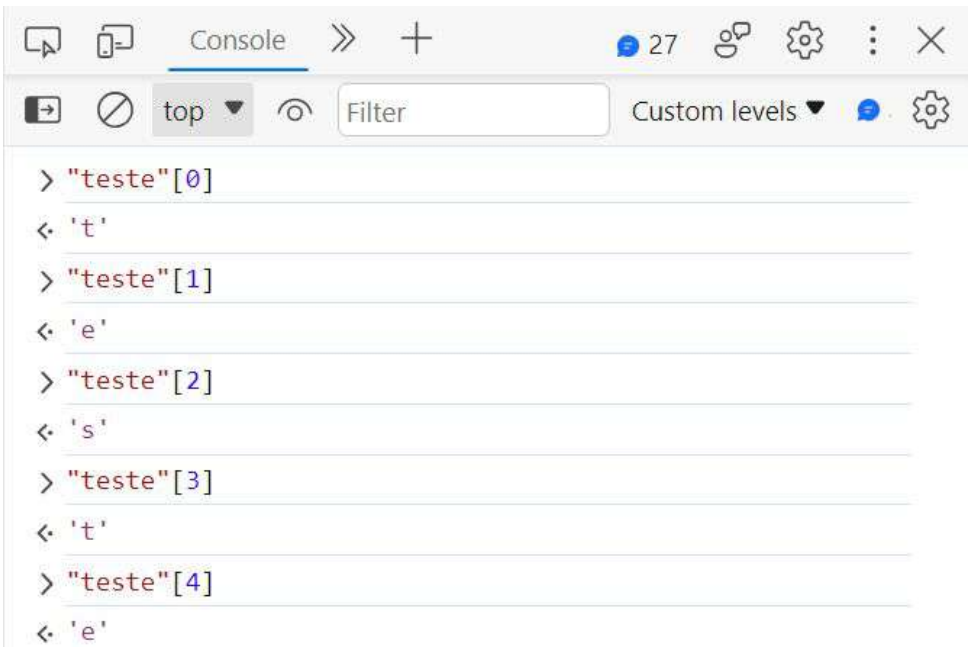
Mas o índice é contado a partir do zero: o 1º caractere está no índice 0, o 2º caractere está no índice 1, etc.

Se você tem uma string, pode "calcular" qual é o caractere em um dado índice.

A sintaxe do comando de acesso a índice (também chamado indexação) é:

- `《string》[《índice》]`

Por exemplo:



# Tópicos adicionais sobre strings

E esse comando também é categorizado como Expressão (assim como cálculos aritméticos), afinal ele "calcula" qual é o caractere em um dado índice.
A string pode estar dentro de uma variável/constante:

```
const mensagem = "Olá";
console.log(mensagem[0]); // ⇒ "O"
```

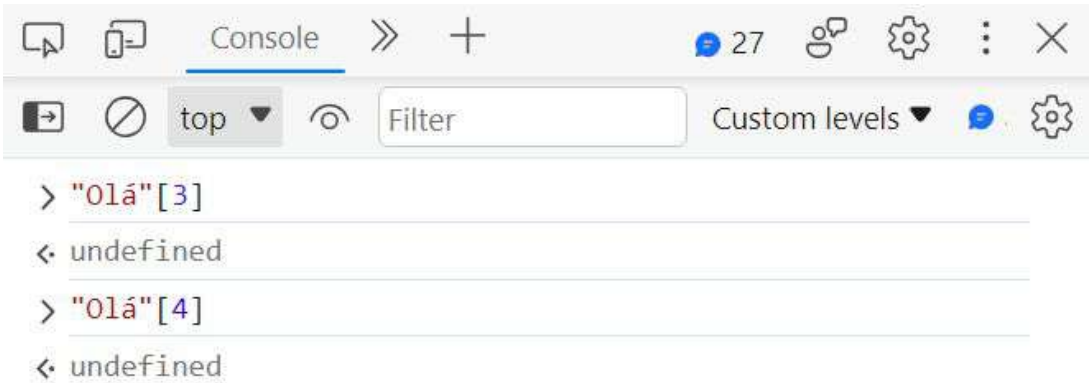
Isso funciona porque o passo a passo da execução da última linha é:

- A instrução é `console.log(mensagem[0]);`
- O navegador web vai "de dentro para fora": a primeira coisa a fazer é calcular a Expressão `mensagem[0]`
- O navegador web sabe que essa é uma Expressão de Indexação.
  Ele "calcula" o valor da variável `mensagem`.
  A instrução se torna: `console.log("Olá"[0]);`
- O navegador web calcula o caractere que está no índice 0.
  A instrução se torna `console.log("O");`
- E aí imprime o "O" como você sabe.

Outra coisa possível é que o valor do índice esteja dentro de uma variável/constante, por exemplo:

```
const mensagem = "Olá";
const indice = 1;
console.log(mensagem[indice]); // ⇒ "l"
```

Vamos deixar para você imaginar como é o passo a passo disso.
Por fim, você pode indexar num índice que não existe.
Por exemplo, na string `"Olá"` existem os índices 0, 1 e 2.
Não existem os índices 3, 4, etc. Mas ainda é possível indexar neles:




Como você pode ver, quando o índice não existe na string, a operação de Indexação retorna **`undefined`**.

# Tópicos adicionais sobre strings

## Propriedade `length` de string

Outro "cálculo" que o javascript consegue fazer sobre strings é calcular quantos caracteres uma string possui.

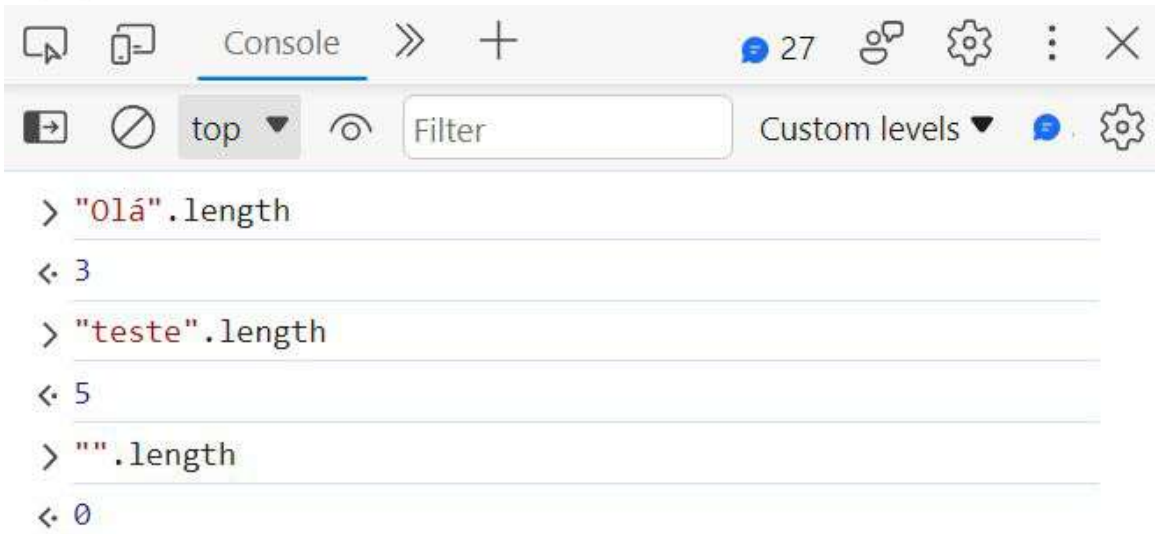
A sintaxe desse comando é:

- `《string》.length`

Isso é classificado como Expressão, porque é um cálculo (cálculo do tamanho da string, "length" significa tamanho em inglês).

Essa operação retorna um número, que é o tamanho (quantidade de caracteres) da string.

Exemplos:




Assim como na operação de Indexação, a string pode estar dentro de uma variável:

```
const mensagem = "oi oi";
console.log(mensagem.length); // ⇒ 5 (o espaço em branco é um caractere)
```

Você já deve imaginar qual é o passo a passo:

1. `console.log(mensagem.length);`
2. `console.log("oi oi".length);`
3. `console.log(5);`

# Tópicos adicionais sobre variáveis

## Formas de declaração de variável

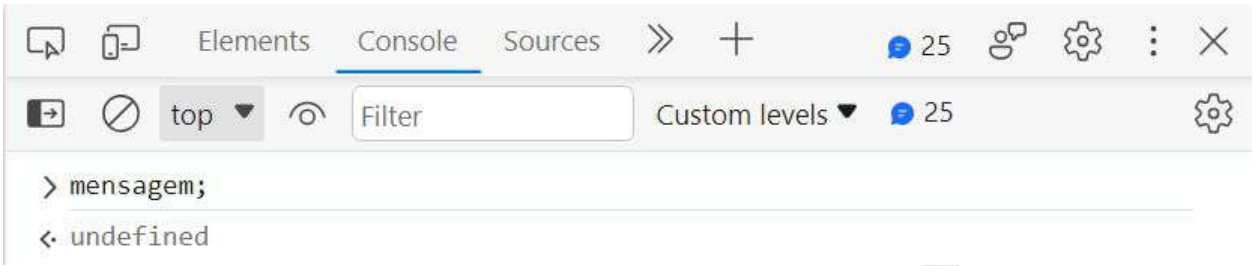
Vimos que o comando `let mensagem = "olá";` declara (cria) uma variável com nome `mensagem` e valor string `"olá"`.

Na verdade há outra forma de declarar uma variável: `let mensagem;`

Ou seja, omitimos o sinal de `=` e o valor.

Nesse caso, a variável é criada sem valor algum.

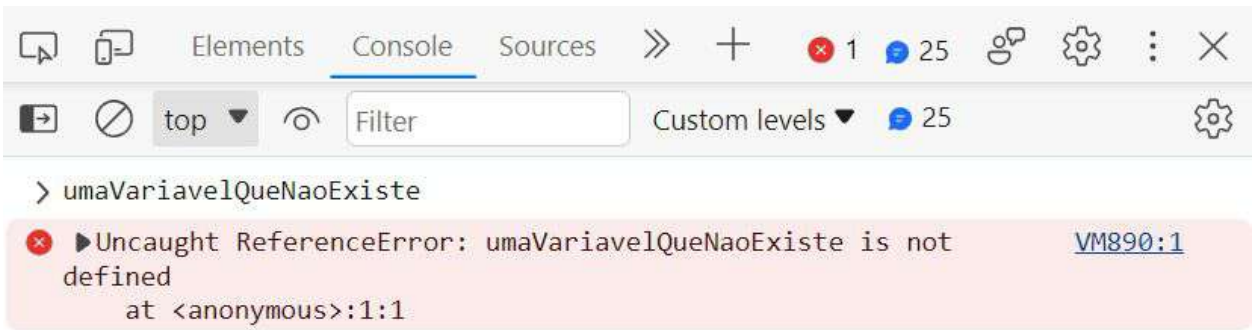
No javascript, uma variável sem valor fica com o valor **`undefined`**. Veja:




Isso não é a mesma coisa que não ter uma variável ! A variável `mensagem` existe, só que ela tem valor vazio.

Já uma variável que não existe é uma que não está declarada no código, ou seja, não tem nenhuma linha de código **`let`** 《nome》 declarando essa variável.

Se você tentar usar uma variável que não foi declarada, o javascript vai dar erro:




Já uma variável que existe mas tem valor **`undefined`** pode ser usada que não vai gerar erro.

# Tópicos adicionais sobre variáveis

## Formas de atribuição de variável

Ao programar em javascript, é comum surgir a necessidade de aumentar o valor de uma variável em 1 unidade (ou 2, ou 3, ...)

Por exemplo, se `idade` é uma variável com valor 40, então daqui a 1 ano ela terá valor 41.

Para aumentar em 1 unidade, podemos digitar o código `idade = idade + 1;`

O passo a passo do javascript é:

- Primeiro, ele avalia a Expressão `idade + 1`, que retorna `41`.
- Então a instrução se traduz em `idade = 41`. Daí ele atribui o valor `41` à variável `idade`.

Para simplificar esse tipo de operação, existe outra sintaxe para o mesmo efeito:

- `idade += 1;`

Esse comando usa um operador que não vimos antes, o `+=` .

Mas o significado é o mesmo que `idade = idade + 41`.

Em geral, a sintaxe do comando é:

- `《variável》 += 《Expressão》`

O importante é que do lado esquerdo precisa ser uma variável e do lado direito uma expressão.

Por exemplo poderia ser `idade += 2` para aumentar em 2 unidades, ou `idade += Number(prompt(...))` para deixar o usuário decidir em quanto a variável será aumentada.

Existem outros operadores de "atalho" para as outras operações aritméticas. Por exemplo:

- `idade -= 5` significa `idade = idade - 5`
- `idade *= 5` significa `idade = idade * 5`
- `idade /= 5` significa `idade = idade / 5`
- `idade %= 5` significa `idade = idade % 5`

Por fim, como as operações mais comuns de todas são `idade += 1` e `idade -= 1` (especificamente com o número 1), existem ainda comandos de atalho especificamente para elas:

- `idade++` significa `idade += 1`
- `idade--` significa `idade -= 1`

Elas também podem ser usadas com o operador do lado esquerdo: `++idade` e `--idade`.

**Note a diferença**

O comando `idade + 1` *não altera o valor da variável*, ele só faz um cálculo.

Esse comando é uma Expressão (cálculo).

Já os comandos abaixo alteram a variável, eles são Comandos de Atribuição (os três comandos são equivalentes):

- `idade = idade + 1;`
- `idade += 1;`
- `idade++;`

# Tópicos adicionais sobre variáveis

## Escopo global

Todas as variáveis/constantes criadas durante a execução do seu script ficam guardadas na memória do computador, numa região chamada de **Escopo Global**.

Na seção desta aula sobre Variáveis, isso era o **Global frame** que aparecia nos diagramas.

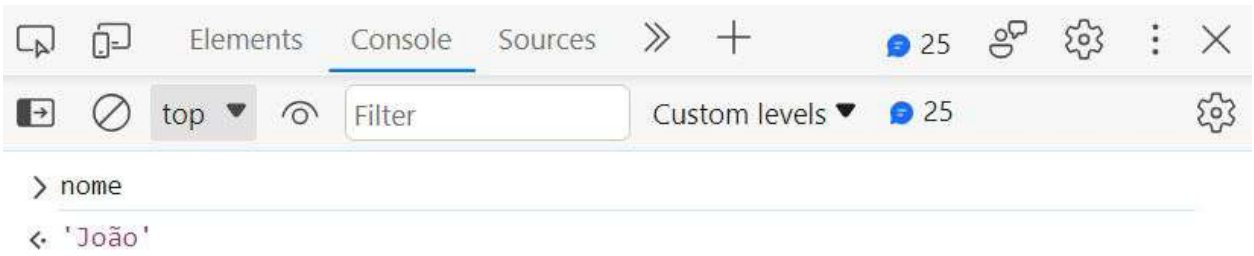
O único efeito prático disso é que as variáveis/constantes não são deletadas depois que o navegador web termina de executar o programa.

Então, se você escrever um programa como:

```
const nome = prompt("Digite o seu nome");
```

E então abrir a página web, o navegador web vai executar esse código e a constante `nome` vai ser mantida na memória do computador.

Aí você pode abrir o Console e digitar `nome`:




A constante ainda existe !

# Tópicos adicionais sobre conversão de tipos

## Regras da conversão explícita de tipos com Number

Já vimos que o javascript tem Conversão Explícita e Conversão Implícita entre tipos de dados.

A conversão explícita é feita com o comando `Number`, que consegue converter qualquer coisa em número.

E às vezes isso gera um resultado surpreendente. Vejamos alguns exemplos:

- **Converter uma string**
  Exemplo (já vimos): `Number("10")` ⇒ `10`
  Mas se a string não representar um número válido, a conversão resulta em **`NaN`**.
  Por exemplo: `Number("batata")` ⇒ **`NaN`**
  Exceção é a string vazia, que converte para 0: `Number("")` ⇒ `0`

- **Converter `null`**
  Resulta em `0`: `Number(null)` ⇒ `0`

- **Converter `undefined`**
  Resulta em **`NaN`**: `Number(undefined)` ⇒ **`NaN`**

A diferença dos comportamentos para **`null`** e **`undefined`** pode ser meio confusa, afinal ambos representam a ausência de valor, então por que eles são convertidos para número de maneira diferente ?

É uma característica do javascript, só aceite…

Claramente o caso mais interessante é converter uma string válida (como "10") em um número (`10`).

Os outros casos geralmente aparecem em situações anômalas, por exemplo na linha de código abaixo:

- `const idade = Number(prompt("Digite sua idade"));`

Como já vimos, se o usuário clicar em Cancel na caixinha do prompt, o prompt retorna **`null`**, daí a execução do código fica assim:

- `const idade = Number(null);` ⇒ `const idade = 0;`

Então a constante `idade` ficará com valor numérico `0`.

# Tópicos adicionais sobre conversão de tipos

## Regras da conversão implícita de tipos

As regras da Conversão Implícita são as seguintes:

- **Operação `+` onde algum operando é uma string**

  Converte o outro operando em string.

  Exemplo 1: `"100" + 10` ⇒ `"100" + "10"` ⇒ `"10010"`

  Exemplo 2: já vimos que quando uma variável é declarada sem valor, por exemplo `let idade;` ela fica com valor `undefined`.

  Nesse caso: `idade + "100"` ⇒ `undefined + "100"` ⇒ `"undefined" + "100"` ⇒ `"undefined100"`

  Exemplo 3: já vimos que o `prompt` retorna `null` caso o usuário clique no botão Cancel, então no código `const nome = prompt(...);` , a constante `nome` ficará com valor `null`.

  Nesse caso: `nome + "100"` ⇒ `null + "100"` ⇒ `"null" + "100"` ⇒ `"null100"`

- **Operação `+` onde nenhum operando é uma string**

  Converte os dois operandos em números usando as mesmas regras da função `Number`.

  Exemplo 1: `undefined + 10` ⇒ `NaN + 10` ⇒ `NaN`

  Exemplo 2: `null + 10` ⇒ `0 + 10` ⇒ `10`

  Exemplo 3: `null + undefined` ⇒ `0 + NaN` ⇒ `NaN`

- **Outras operações aritméticas**

  Converte os dois operandos em números (ou `NaN` se o operando não pode ser convertido num número válido). Não importa se algum operando é string ou não.

  Exemplo 1: `"100" - 10` ⇒ `100 - 10` ⇒ `90`

  Exemplo 2: `"teste" * 10` ⇒ `NaN * 10` ⇒ `NaN`

  Exemplo 3: `10 / undefined` ⇒ `10 / NaN` ⇒ `NaN`

  Exemplo 4: `10 * null` ⇒ `10 * 0` ⇒ `0`

## Outros comandos de conversão de tipos para número

O comando `Number` não é a única forma de converter dados quaisquer para números.

Existem outros dois comandos que também conseguem converter dados para números:

- `parseInt`
- `parseFloat`

Para ver mais sobre eles, você pode consultar a documentação da Mozilla (MDN):

- Documentação sobre parseInt
- Documentação sobre parseFloat

# Tópicos adicionais sobre funções

## Introdução

Você percebeu que a sintaxe dos comandos `console.log`, `alert`, `prompt` e `Number` é parecida ?

Em todos eles, a sintaxe tem a forma:

- `《comando》(《expressão》);`

Por exemplo:

- `console.log(100 * 5);`
- `alert("olá !");`
- `prompt("Qual a distância em km ?");`
- `Number("10");`

Isso não é uma coincidência, é desse jeito porque todos eles são **Funções**.

## O que é uma função ?

Uma Função é um tipo de dados, como número e string. Já listamos anteriormente os principais tipos de dados que o javascript entende.

Mas um número é um número (claro), uma string é um texto. E uma Função, o que é ?

Função é uma sequência de instruções que são executadas somente quando solicitado. Em outras palavras, uma função é um programa completo composto por instruções.

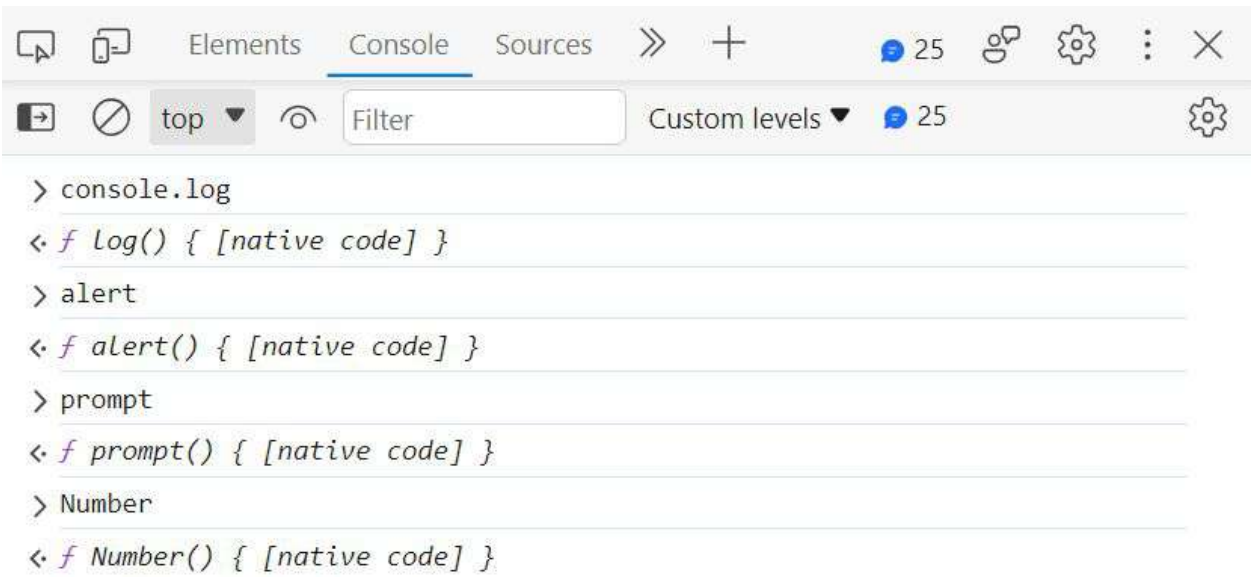
## Código da função *versus* invocar a função

Se uma função é um programa, qual a diferença entre uma função e os programas que vínhamos escrevendo até agora, por exemplo o programa de custo da viagem ?

A diferença é que o programa de custo da viagem é executado assim que a página web carrega, porque o navegador web vê o código javascript e o executa.

Já o código de uma função *não* é automaticamente executado, você precisa *solicitar* que ele seja executado. E você pode decidir *quando* solicitar (pode ser 1 ou várias vezes).

Para ver melhor isso, digite no Console os nomes das funções, mas sem digitar os parênteses:



# Tópicos adicionais sobre funções

Sem os parênteses, o Console mostra o *código* de cada função.

Na verdade não dá para ver o código direito porque só aparece "código nativo" (native code) porque o código dessas 4 funções é parte do navegador web e não pode ser visto.

Mas nosso argumento é: o alert não mostra nenhum alerta e o prompt não mostra nenhuma caixa de digitação.

Em outras palavras, sem os parênteses, o código das funções *não é executado*.

Você está *visualizando* o código em vez de *executar* o código.

É como imprimir a página do livro com a receita de bolo em vez de seguir/operar/executar a receita de bolo.

Por outro lado, quando você digita:

- `console.log(100 * 5);`
- `alert("olá !");`
- `prompt("Qual a distância em km ?");`
- `Number("10");`

Aí está solicitando a execução do código da função, portanto o alerta aparece, a caixa do prompt aparece, a string "10" é convertida em número, etc.

Isso é chamado de **Invocar** ou **Chamar** a função.

É a presença dos parênteses que faz o javascript entender que você quer executar a função.

## Parâmetros

Quando você invoca uma função, você pode passar informações para ela.

Por exemplo a mensagem que o `alert`/`console.log` deve imprimir, ou a string que o `Number` deve converter.

Em outras palavras, a informação que é colocada dentro dos parênteses.

Essa informação é chamada de **Parâmetro** da função.

Nem toda função precisa de parâmetros.

E algumas funções podem precisar de mais do que 1 parâmetro.

# Tópicos adicionais sobre funções

## Funções nativas *versus* funções do programador

As 4 funções que vimos (`console.log`, `alert`, `prompt`, `Number`) são funções nativas, significa que o código delas vem pronto no navegador web para que você as invoque quando precisar.

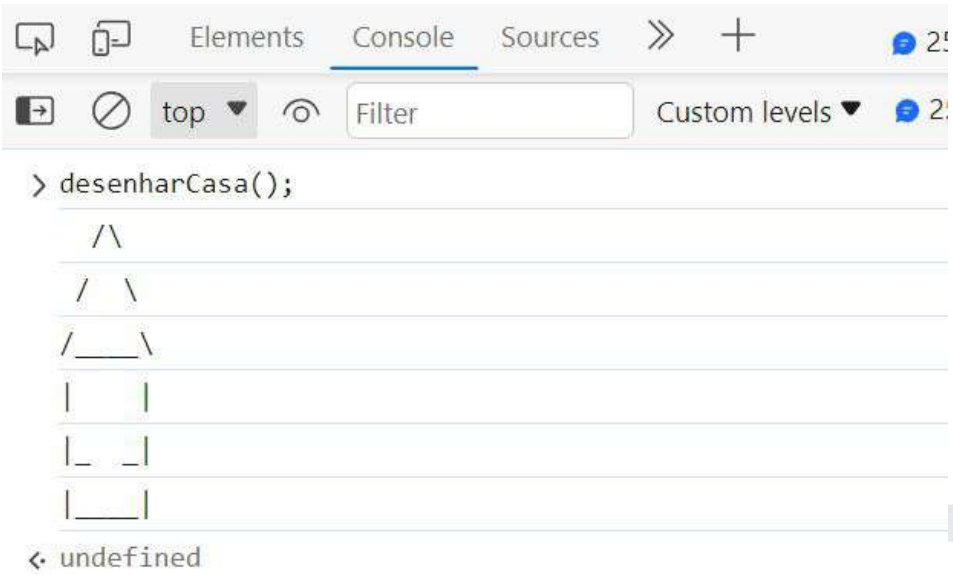
Assim, por exemplo, você não precisa saber *como* que a função `Number` consegue converter uma string em um número.

Você simplesmente invoca essa função e deixa que o código interno dela cuide disso para você.

Mas quando não existe uma função nativa que faça exatamente o que queremos, é possível criar novas funções customizadas que tenham o código que quisermos.

Veremos como fazer isso no futuro. Mas segue um exemplo de uma função customizada com nome `desenharCasa`.

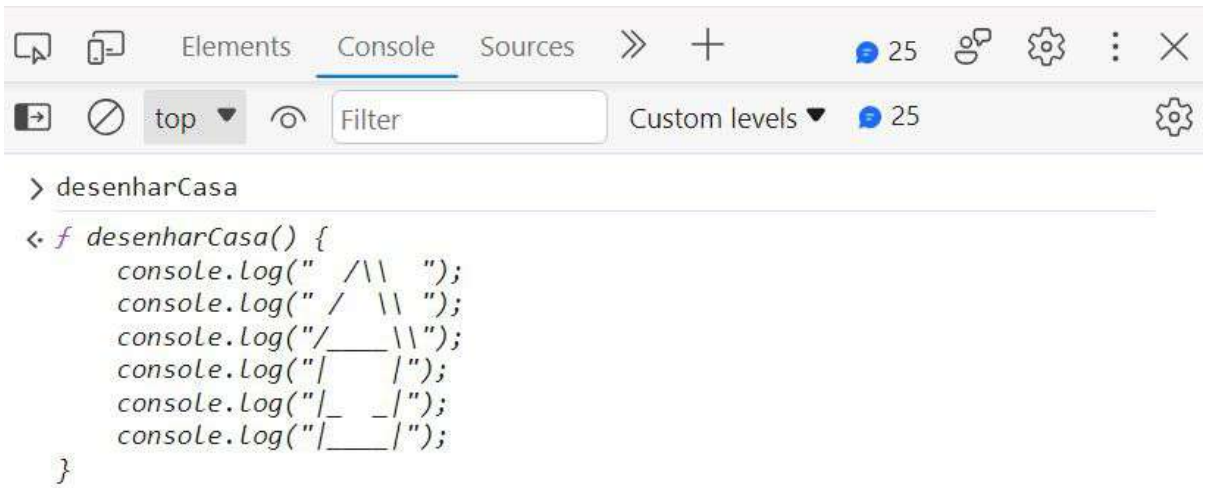
Quando invocada, ela desenha uma casa no Console:




Se você tentar invocar `desenharCasa()` aí no Console do seu navegador, não vai funcionar, porque essa função é customizada, ela não vem pronta no navegador.

Nós tivemos que escrever código javascript para criar essa função, só não mostramos aqui como faz isso.

Mas, como você já sabe, é possível ver o código se omitirmos os parênteses:




É um código simples: várias invocações a `console.log(...)`.

# Tópicos adicionais sobre erros no código

## Introdução

Veremos quais tipos de erro existem no javascript e como consertar problemas em nossos códigos.

## Primeira categoria de erro: Erro de Sintaxe

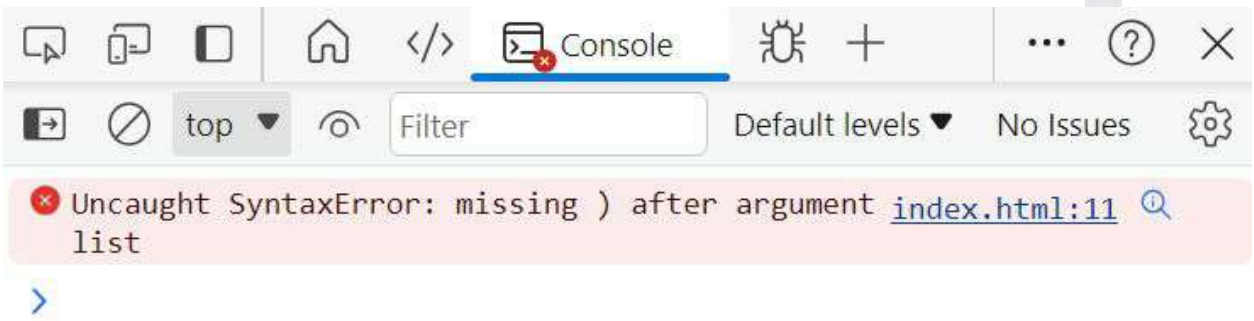
Eles são erros na escrita do código. Quando existe esse tipo de erro, o código nem é executado.

Por exemplo:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Minha página de teste</title>
    <script>
      console.log(10 * 5);
      console.log(10 / 5);
      console.log(10 / (2 * 2) + 5);
      console.log("Olá, " + "tudo bem ?";
    </script>
  </head>
  <body></body>
</html>
```

Note que tem um erro na última linha do javascript (que é a linha 11 do código): falta o fechamento de parênteses após o `console.log`

Se você abrir essa página web num navegador, verá o seguinte:




Primeiro note que, apesar de o erro estar no 4º `console.log` do código, os outros três `console.log` *também não foram executados*.

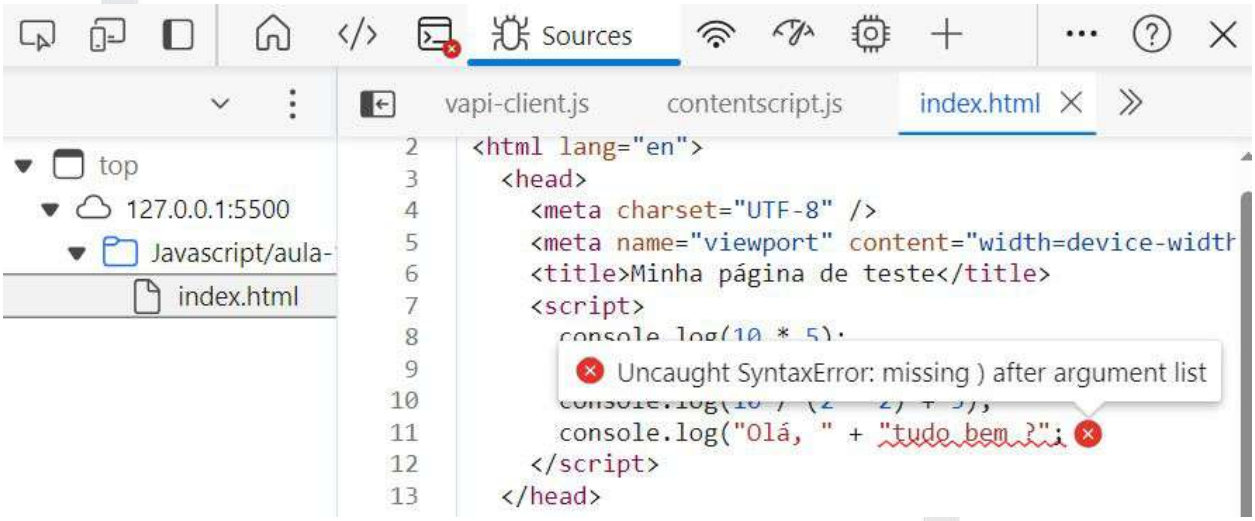
Isso é porque o navegador analisa a sintaxe (escrita) do código javascript inteiro *antes* de começar a executar a primeira linha dele. Se há erro de sintaxe, nada é executado.

Segundo, note que apareceu uma mensagem vermelha dizendo que o erro é um **SyntaxError**, e que ele ocorreu devido à falta do `)`.

Na margem direita aparece `index.html:11` , significa que o erro está na linha 11 do arquivo *index.html*.

# Tópicos adicionais sobre erros no código

Se você clicar nesse link, o Console vai mudar para a guia de *Sources*, aparecendo o código do arquivo e a linha com problema:




## Segunda categoria de erro: Erro de Execução

Esse tipo de erro ocorre quando o código javascript não tem erros de sintaxe (está escrito corretamente) mas, ao ser executado, alguma instrução é inválida.

Por exemplo, suponha que você fez o seguinte programa que pede ao usuário que digite seu nome, e então o programa imprime a quantidade de caracteres do nome:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Minha página de teste</title>
    <script>
      console.log("Início do programa");
      const nome = prompt("Digite seu nome");
      console.log("Seu nome tem tamanho " + nome.length);
      console.log("Fim do programa");
    </script>
  </head>
  <body></body>
</html>
```

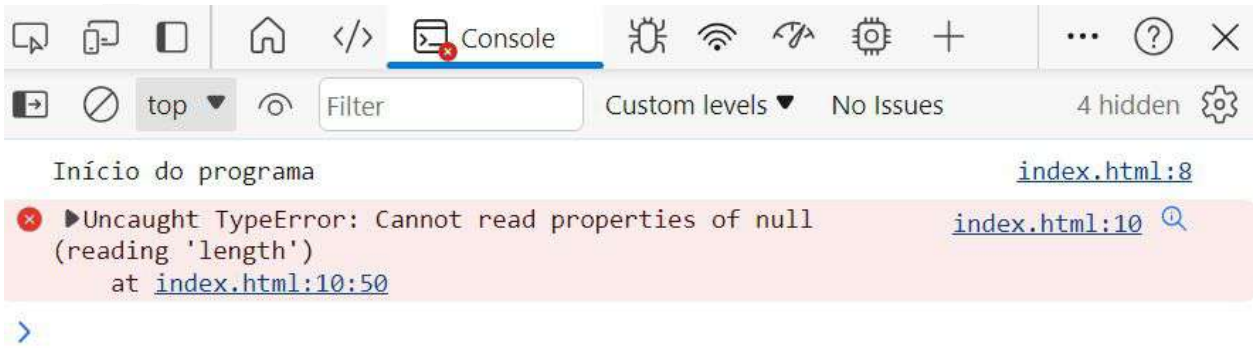
Se o usuário digitar um nome, por exemplo "Marcos", o programa vai funcionar corretamente, imprimindo `"Seu nome tem tamanho 6"`.

Mas se o usuário clicar em Cancel na caixinha do prompt, sabemos que o valor de `nome` será **`null`**.

Aí a terceira linha do javascript (linha 10 do código) tentará executar o comando `nome.length`, mas isso é inválido porque o tipo de dados do `nome` não é string (como era esperado), mas sim **`null`**.

# Tópicos adicionais sobre erros no código

Veja o erro que aparece:




Note que, diferentemente do caso de erro de sintaxe, o início do código *foi sim* executado, afinal apareceu `"Início do programa"` no Console.

Depois, ao executar a linha 10 do código, o javascript se deparou com uma operação inválida `null.length` , e então apareceu no Console a mensagem de erro vermelha.

Ela diz que o erro é um **TypeError**, que é um subtipo de Erro de Execução.

Qualquer erro exceto **SyntaxError** é um subtipo de Erro de Execução.

Existem vários subtipos porque há várias possibilidades de operação inválida no javascript (`null.length` é uma delas).

Por fim, note que o erro ocorreu na linha 10 do código, e depois dela a linha 11 *não foi executada*, afinal não apareceu `"Fim do programa"` no Console.

É sempre assim quando há Erro de Execução: no momento que acontece o erro, o navegador web *aborta* a execução do script.

# Tópicos adicionais sobre erros no código

## Terceira categoria de erro: Erro de Lógica

Esses não são erros no sentido estrito, mas sim um comportamento que o código tem que você programador não estava esperando.

Você deve ter deixado passar despercebido algum detalhe.

Nós já vimos um Erro de Lógica, no seguinte código:

```
const notaProva1 = prompt("Digite a nota da prova 1");
const notaProva2 = prompt("Digite a nota da prova 2");
const soma = notaProva1 + notaProva2;

alert("A soma das notas é " + soma);
```

O objetivo do programador era somar as notas, por exemplo 10 e 5 daria 15.

Mas o programador se esqueceu de converter o valor de retorno dos `prompt` em números.

Por isso o alerta mostra na verdade 105.

Não existe Erro de Sintaxe: o código está escrito corretamente.

Também não há Erro de Execução: o código executa sem abortar, todas as operações são válidas.

Mas há Erro de Lógica: o código não faz aquilo que o programador pretendia.

Erros de Lógica são os mais difíceis de diagnosticar, justamente porque não aparece no Console nenhuma mensagem vermelha dizendo exatamente onde houve erro.

Afinal do ponto de vista do navegador web que está executando o código, *não houve erro nenhum*. Somente um "desencontro" entre o que o programador queria e o que o programador realmente escreveu.

# Tópicos adicionais sobre erros no código

## Depuração (debug) de código

Quando seu código javascript não tem o comportamento que você esperava, a primeira abordagem para diagnosticar o problema é olhar o Console.

Como vimos, se houver mensagens de erro (erro de sintaxe ou erro de execução), elas informarão a linha do código onde aconteceu o problema.

Mas se o programa tiver erro de Lógica, não haverá mensagens de erro no Console.

Nesse caso, você pode usar `console.log` em vários pontos do seu programa, para imprimir os valores das variáveis e assim detectar quando que o valor de alguma(s) dela(s) saiu do esperado.

Se mesmo assim você não consegue identificar a fonte do problema, os navegadores web têm uma ferramenta que permite executar seu código linha por linha pausadamente, vendo todas as variáveis em tempo real.

Esse é o Debugger.

Para exemplificar, vamos mostrar uma página web que inclui um arquivo *index.html* e um *index.js*.

O *index.js* contém o nosso já conhecido código de soma de notas com Erro de Lógica:

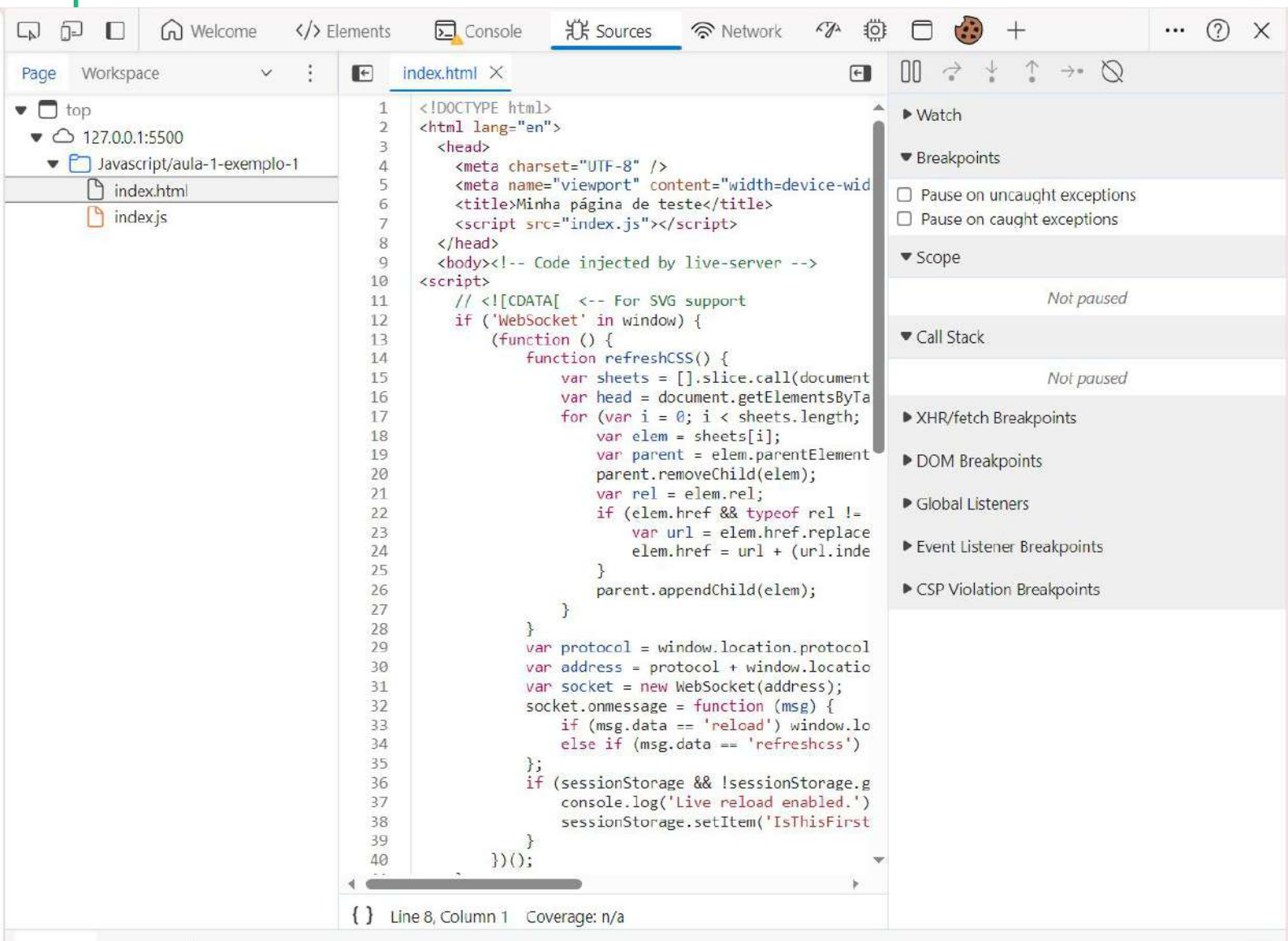
```
const notaProva1 = prompt("Digite a nota da prova 1");
const notaProva2 = prompt("Digite a nota da prova 2");
const soma = notaProva1 + notaProva2;

alert("A soma das notas é " + soma);
```

Vamos mostrar como o Debugger facilita enxergar que o problema desse código é a falta de conversão de string para número nas duas primeiras linhas do código.

Primeiro, ao abrir a página no navegador, podemos acessar o Debugger.

Ele fica na aba *Sources* do Console, e está dividido em 3 áreas:

# Tópicos adicionais sobre erros no código



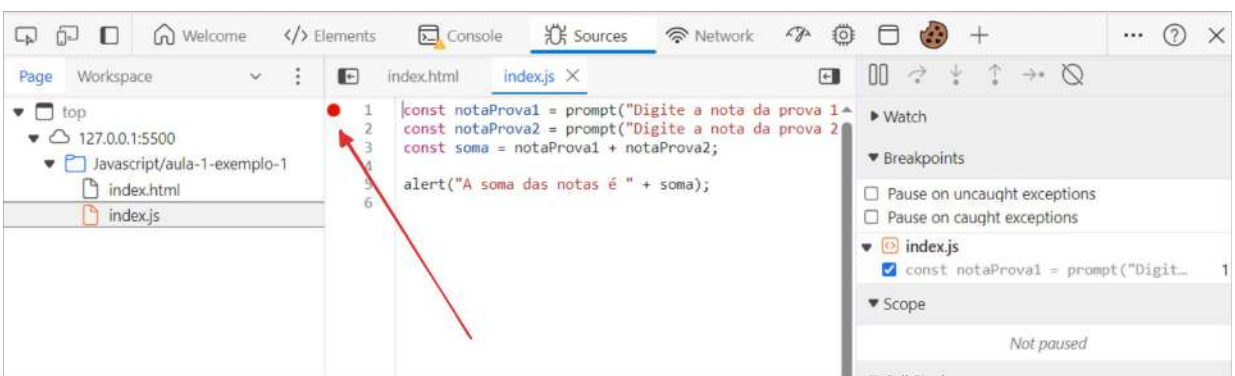

À esquerda, temos a lista de arquivos que o site carregou. Neste caso *index.html* e *index.js*.

No centro, vemos o código-fonte do arquivo que está selecionado à esquerda (*index.html*).

Veja que esse arquivo faz o carregamento do javascript com `<script src="index.js">`. O código da linha 9 em diante do HTML foi adicionado por uma extensão do meu navegador web (ignore).

Por último, no lado direito, vemos o painel de monitoramento onde aparecerão as variáveis em tempo real quando estivermos executando o código nos próximos passos.

Para começar, abra o *index.js* e clique na linha 1 dele para marcá-la com a bolinha vermelha:



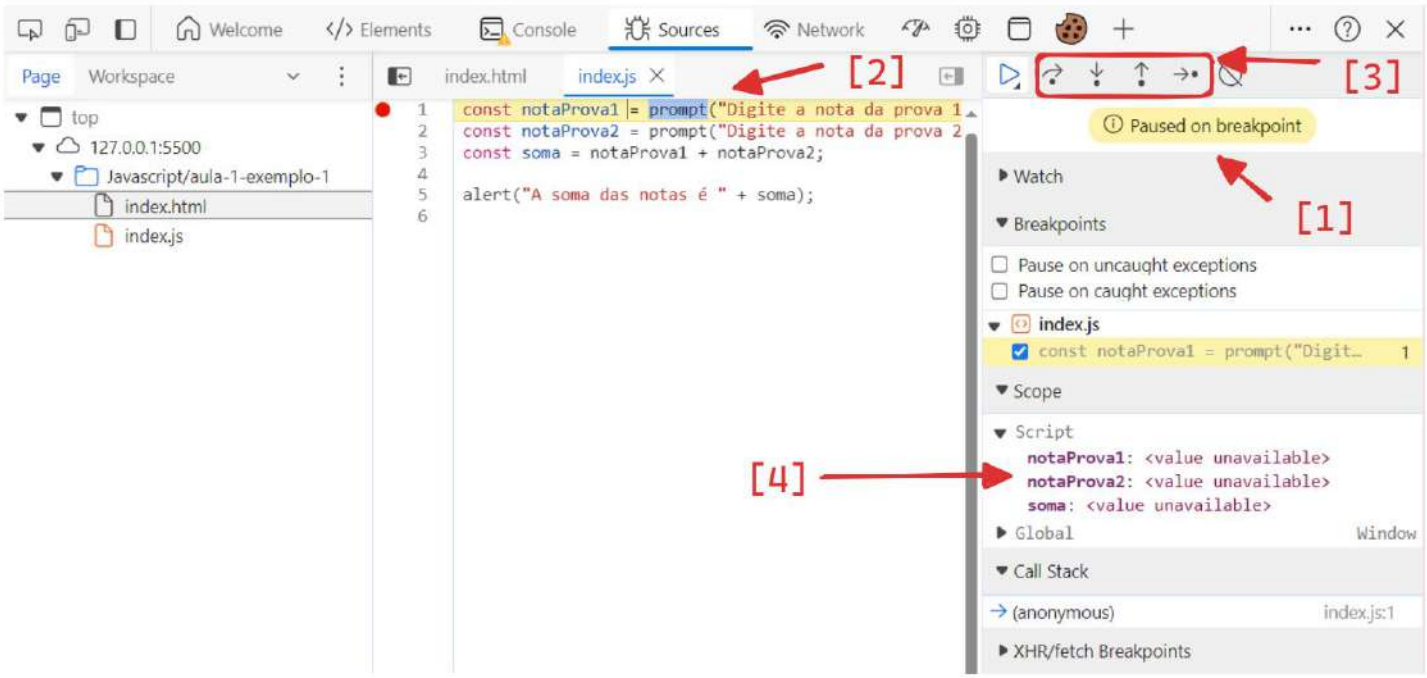
# Tópicos adicionais sobre erros no código

Essa bolinha vermelha é chamada de *Breakpoint* (ponto de quebra).

Quando o navegador web está executando código javascript e chega a uma linha marcada com breakpoint, ele *congela* a execução do código e nos permite avançar linha a linha, observando as variáveis em tempo real.

Então vamos fazer isso. Recarregue a página.

Ao fazer isso, a tela fica assim:




E a página web fica congelada.

Isso acontece porque, ao carregar a página, o navegador começa a executar o javascript, mas a linha 1 tem um breakpoint, portanto ele congela.

No ponto marcado com [1] na imagem, vemos a mensagem em amarelo "Paused on breakpoint" (pausado no breakpoint).

No ponto [2], vemos que a primeira linha do código está marcada em amarelo. Isso significa que o navegador está atualmente nela. Ela não foi executada ainda (por isso que a caixinha do prompt ainda não apareceu).

No ponto [3], vemos quatro opções de botão. Eles servem para avançar no código. Não vamos detalhar a diferença de cada botão.

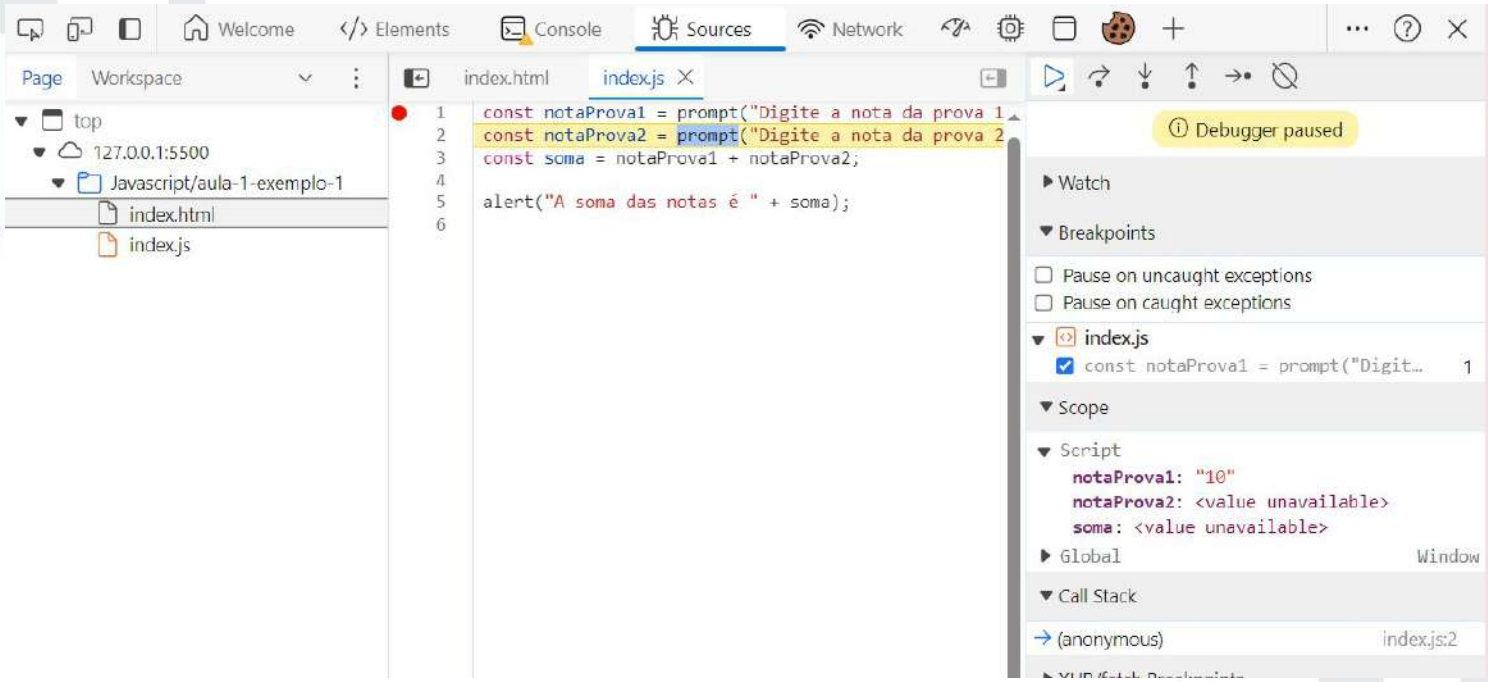
E no ponto [4], vemos as variáveis do código. Elas estão com a mensagem "valor não disponível" (value not available) porque, neste ponto da execução, nenhuma delas foi declarada ainda.

Agora vamos deixar o navegador avançar no código. Clique no botão mais à direita, chamado de Step (passo).

Nesse momento a linha 1 será executada, portanto vai aparecer a caixinha do prompt. Digite 10 e dê OK.

# Tópicos adicionais sobre erros no código

Agora a tela fica assim:

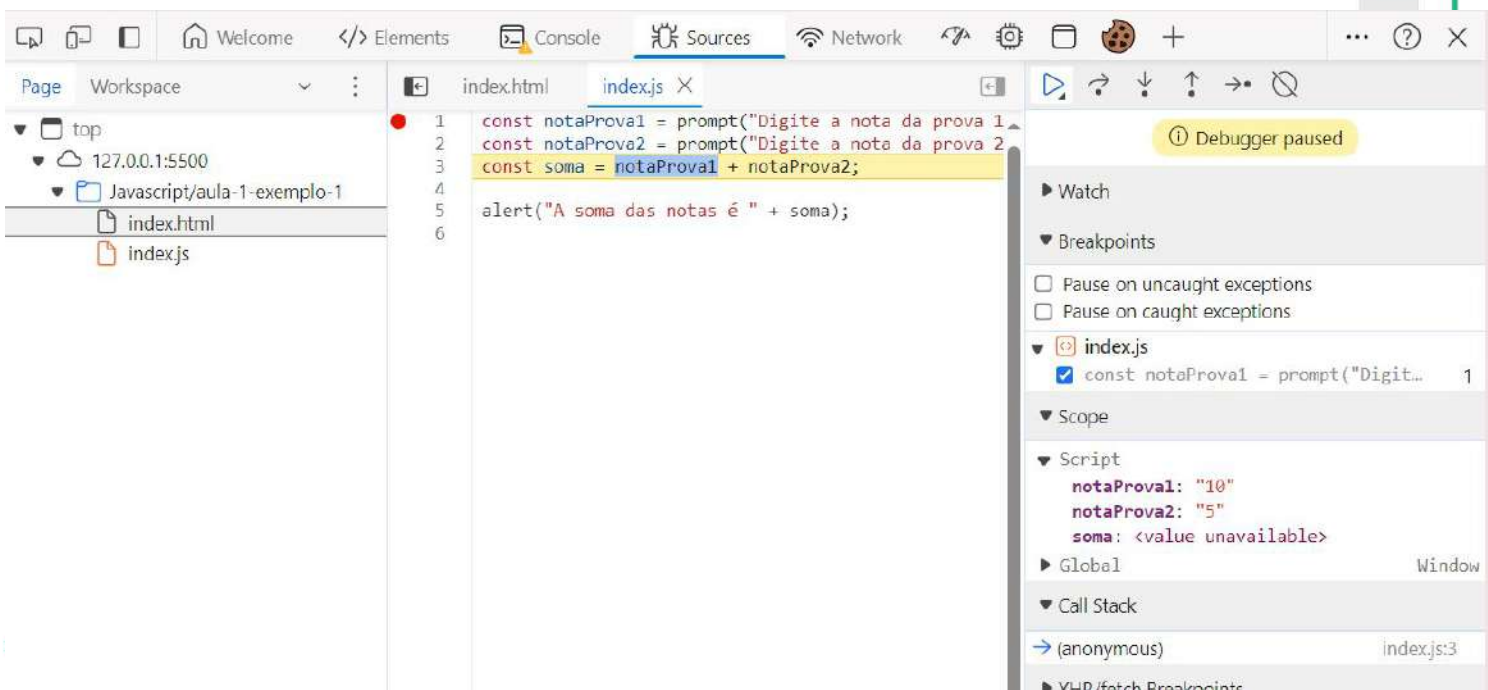



Note que a linha em amarelo agora é a 2. Porque a 1 foi executada e estamos pausados na 2 (ela ainda não foi executada).

E no lado direito, veja que a variável `notaProva1` agora está com o valor string de `"10"`, conforme foi digitado no prompt.

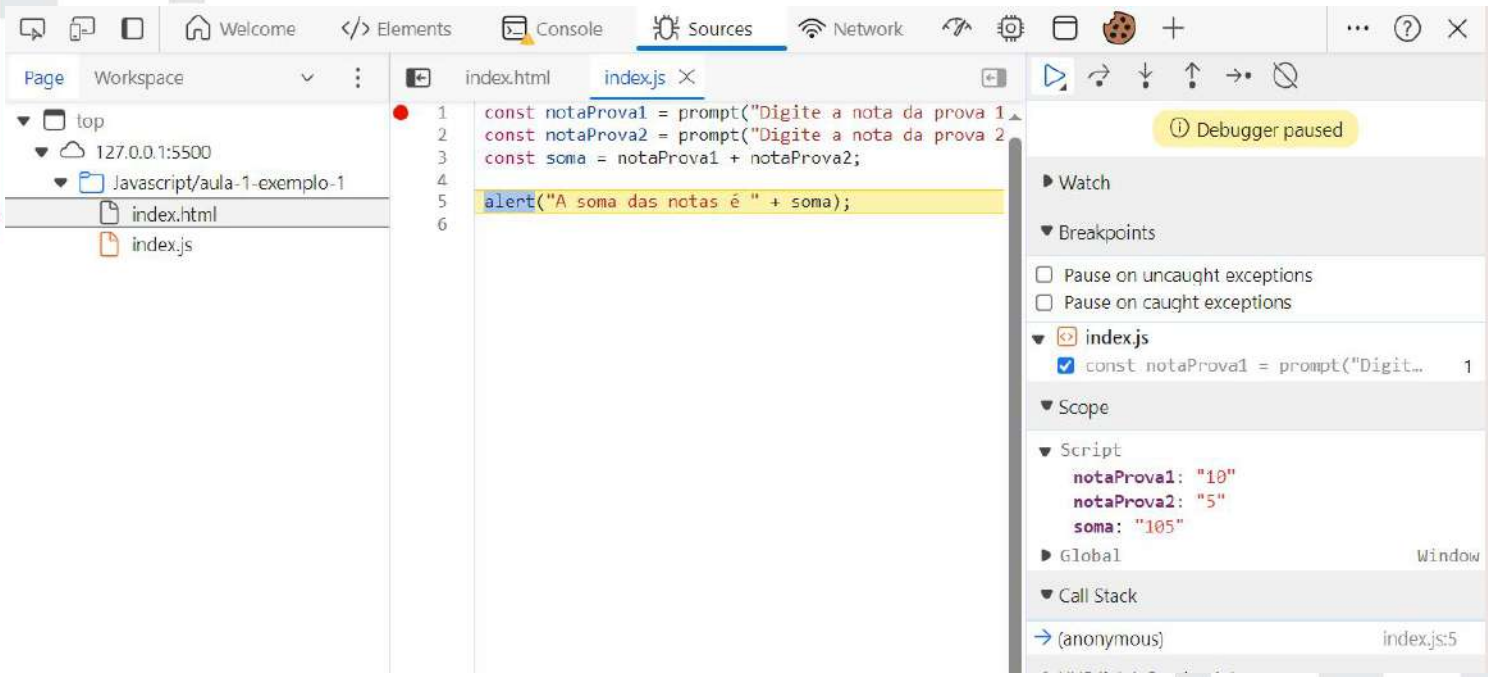
Em seguida, clicando em Step mais uma vez, a linha 2 é executada. Digite 5 no prompt.

Então a tela fica assim:



# Tópicos adicionais sobre erros no código

Portanto estamos pausados na linha 3, e o valor da variável `notaProva2` é a string `"5"`.
Mais uma vez clicando em Step, a tela fica assim:

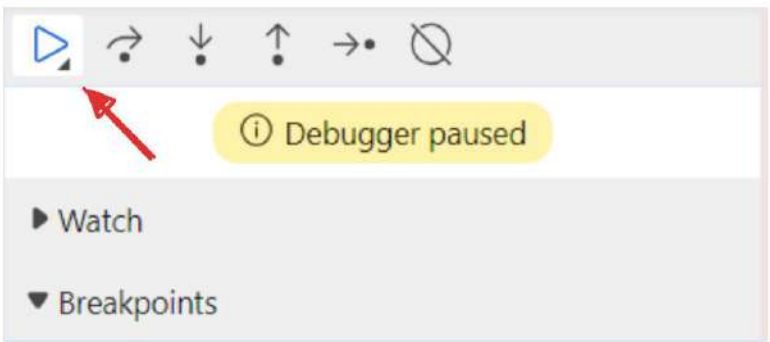



Portanto foi executada a linha 3, logo a variável `soma` tem o valor string `"105"`.

Você deve ter percebido que assim fica óbvio exatamente quais operações estão sendo feitas no programa e quais são os valores das variáveis em cada passo.

Fica evidente que faltou transformar as strings em números antes de fazer a soma.

Não tendo mais necessidade do debugger, clique no botão Resume (imagem abaixo) para desativar o debugger.



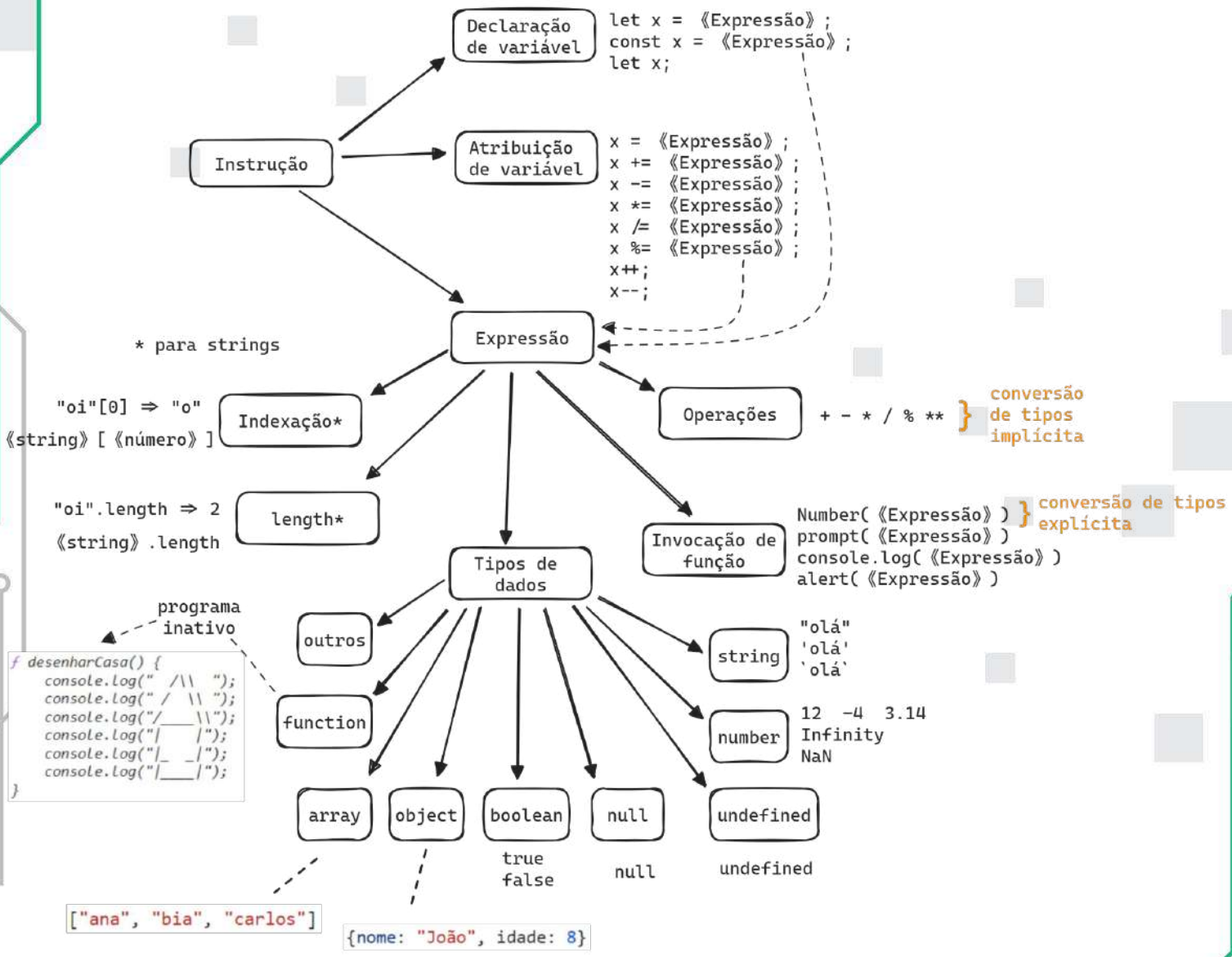

Esse foi o básico sobre o Debugger.

Como última observação, se tivéssemos usado a função `Number` para fazer a conversão de tipos, por exemplo `const notaProva1 = Number(prompt("Digite a nota da prova 1"));` , então o valor da variável `notaProva1` no debugger apareceria como `10` e não como `"10"`, ou seja, seria óbvio que é um número, não uma string.

# Tópicos adicionais sobre a linguagem javascript

## Diagrama parcial da sintaxe do javascript

Com tudo que vimos até agora, o diagrama abaixo organiza os principais comandos do javascript com exemplos:




Evidentemente esse diagrama não está completo, faltam coisas que ainda veremos, por exemplo:

Dentro de "Instrução", ainda faltam instruções condicionais, instruções de repetição, instruções de declaração de função, instruções de captura de erro, instruções de importação/exportação, e instruções de declaração de classe.

Dentro do tipo de dados "object" (objeto), ainda faltam os vários tipos de objeto: objeto "avulso", objeto do DOM, objeto Math, objeto Date, objeto Promise, e outros. E como eles funcionam.

Dentro do tipo de dados "array", ainda falta ver o que eles são, como funcionam, e suas várias ferramentas.

E outros assuntos.

# Tópicos adicionais sobre a linguagem javascript

## História e padronização do javascript

O javascript foi criado em 1995 para adicionar interatividade às página web no antigo navegador web chamado Netscape Navigator.

Desde então, foi adotada pelos outros navegadores web que surgiram, como o Firefox, o Chrome, o Safari e o Edge.

Cada navegador web é um software próprio, feito por uma empresa diferente.

Mas todos devem interpretar o javascript da mesma maneira que os outros navegadores.

Afinal, se houver divergência entre navegadores, o mesmo código javascript pode funcionar em um navegador e não em outro.

Isso seria ruim para os desenvolvedores web que precisam criar aplicações para os sites.

Por causa da necessidade de uniformização, a Ecma International (associação dedicada à padronização de sistemas de informação) criou em 1997 o primeiro documento normativo que define como exatamente deve funcionar a linguagem javascript.

Esse documento é chamado ECMAScript.

Mais concretamente, o que o ECMAScript define é a sintaxe e significado da linguagem.

Exemplos do que o ECMAScript define:

- Números, strings, funções, objetos, e todos os outros tipos de dados
- Regras de conversão de tipos, implícita e explícita
- Sintaxe e funcionamento da declaração e atribuição de variáveis e constantes
- E muito mais...

A Wikipedia tem uma página dedicada às versões do ECMAScript.

Geralmente todo ano é lançada uma nova versão que adiciona funcionalidades à linguagem.

A versão mais atual do ECMAScript pode ser sempre acessada pelo link https://tc39.es/ecma262/.

## Javascript Engine

O javascript é uma linguagem de programação.

E uma linguagem de programação é simplesmente o conjunto de regras de sintaxe.

Para que um código javascript funcione num navegador web, é necessário que esse navegador tenha um *programa* que consiga ler, entender e executar código javascript.

Um programa que faça isso é chamado **Javascript Engine** (tradução literal: motor javascript).

Provavelmente a Engine mais famosa seja a **V8**, que foi desenvolvida com a linguagem de programação C++.

Essa Engine (que, como falamos, é um módulo de software, um programa) é usada atualmente por vários navegadores web, como Google Chrome, Brave, Opera, Vivaldi e Microsoft Edge.

Isso significa que, dentro desses navegadores, é o programa V8 que lê e executa o código javascript das páginas web.

Mas existem outras Engines:

- O navegador Firefox usa uma Engine chamada SpiderMonkey.
- O navegador Safari (Apple) usa a Engine JavaScriptCore.

Seja como for, todas as Engines devem ser compatíveis com as regras do ECMAScript.

Dessa forma, temos a garantia de que um código javascript vai funcionar de maneira idêntica não importa qual seja a Engine que está executando o código.

# Tópicos adicionais sobre a linguagem javascript

## Javascript Runtime

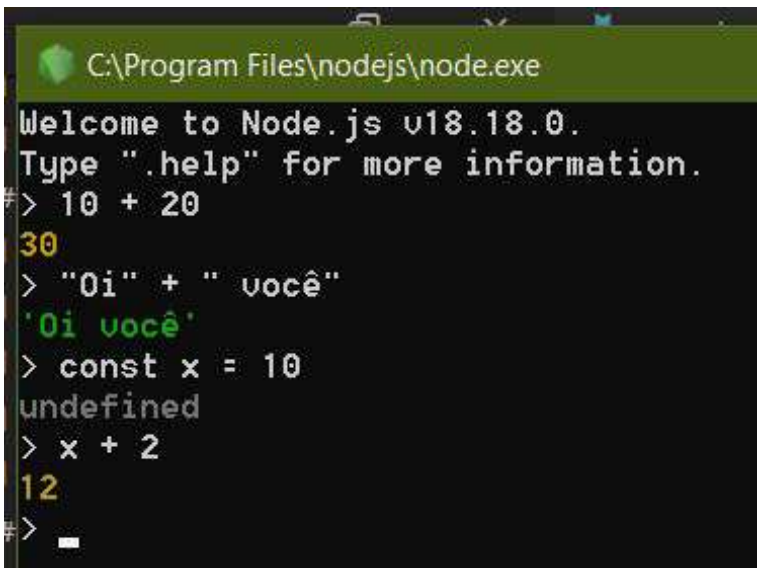
Um Runtime é um ambiente onde o javascript pode ser executado.

Então todo navegador web é um runtime de javascript.

O que talvez você não saiba é que o javascript também pode ser executado *sem nenhum navegador web*.

Por exemplo, o **Node.js** é um outro runtime de javascript que você pode instalar na sua máquina para executar javascript.

Ele não é um navegador web. Tem somente uma tela preta que é análoga ao Console de um navegador:




Para executar javascript, um runtime precisa usar uma engine.

O Node.js usa a engine V8 (aquela mesma que vários browsers utilizam).

Mas várias coisas que funcionam num navegador não existem no Node.js. Por exemplo não existem as funções `alert` e `prompt`.

# Tópicos adicionais sobre a linguagem javascript

## Manuais e referências de javascript

A referência definitiva do javascript é a versão mais atual do já mencionado ECMAScript, que pode ser sempre acessada pelo link https://tc39.es/ecma262/ .

Mas sendo um padrão formal, ele é difícil de entender, não serve para uso cotidiano.

Outra limitação do ECMAScript é que ele define somente o funcionamento interno da linguagem (sintaxe, tipos de dados, cálculos, etc.), mas não define como devem funcionar Entrada de Dados e Saída de Dados.

Ou seja, o ECMAScript não fala por exemplo sobre `console.log` e `alert` (saída de dados) nem `prompt` (entrada de dados).

Essas outras coisas são adições ao básico do ECMAScript, e estão definidas em outros documentos.

No caso do `alert` e `prompt`, eles estão definidos na Especificação do HTML, na seção 8.8.1 (sim, a especificação do HTML fala sobre javascript).

Já o `console.log` está definido na Especificação do Console, na seção 1.1.6.

Outro exemplo: nas próximas aulas, veremos como o javascript pode interagir com o HTML de uma página web, como o caso do botão "⋮⋮⋮" do Google mostrado no começo da aula.

A maneira como o javascript pode interagir com o HTML é definida em outro documento: a Especificação do DOM.

Em resumo, são vários documentos normativos.

E, sendo normativos, eles são de difícil leitura, são pouco úteis para consulta cotidiana.

Para consulta rápida, existem os manuais da Mozilla: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript.

Recomendamos esse site como uma referência confiável que inclui informações não só sobre os internos do javascript, mas também sobre o Console, HTML, etc.

# Fundamentos de objetos e DOM

## Sumário

# Fundamentos do DOM

## Introdução

Até agora nos familiarizamos com as funcionalidades básicas como entrada e saída de dados utilizando `prompt` e `alert`.

Mas essas ferramentas, embora úteis para aprender conceitos iniciais, não refletem a realidade de como as páginas web funcionam.

Em um cenário mais prático e usual, o esperado é que o javascript interaja com os usuários através de elementos como formulários e botões na página, e exiba informações e resultados diretamente na página, sem janelas de pop-up (alert).

Para alcançar essa interação mais integrada, vamos nos aprofundar no estudo do DOM (Document Object Model).

O DOM é uma parte crucial na programação web, pois é ele que nos permite acessar e manipular o conteúdo, a estrutura e o estilo das páginas web (HTML).

Com o conhecimento do DOM, você será capaz de ler dados de formulários, responder a cliques em botões, e alterar o conteúdo da página dinamicamente.

Isso abre um mundo de possibilidades para tornar as páginas web mais interativas e responsivas às ações dos usuários.

## O que é o DOM

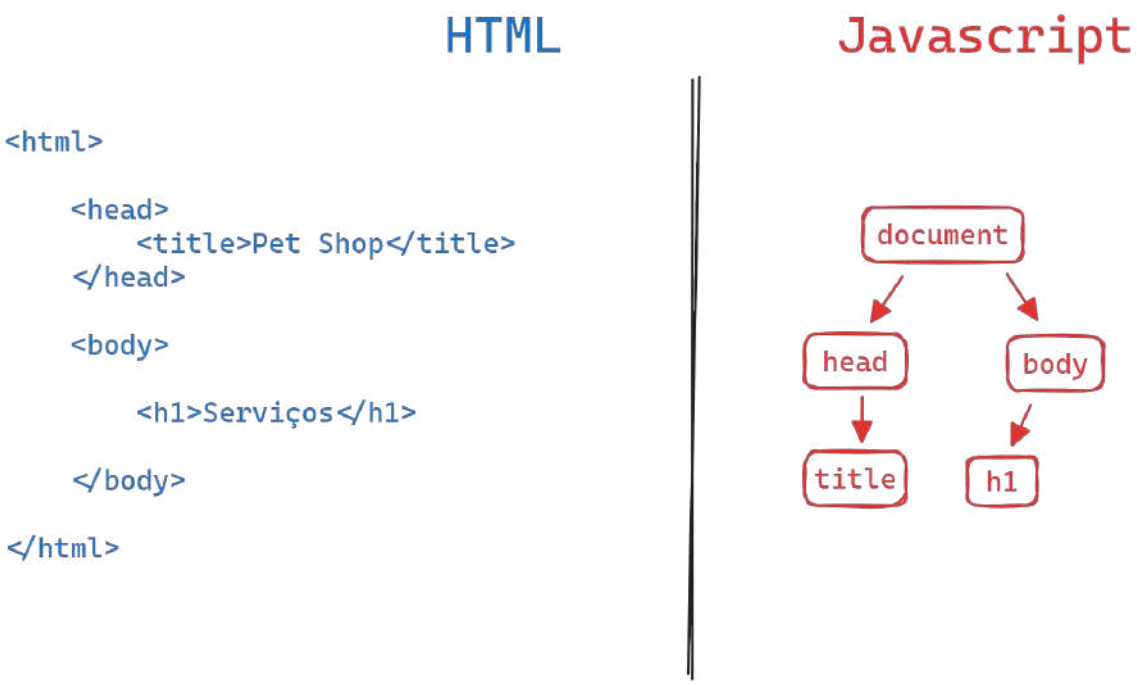
HTML e Javascript vivem em "mundos" separados.

Mas o Javascript precisa ser capaz de interagir com o HTML, afinal sem isso ficamos limitados a prompts e alertas para interagir com o usuário.

Os navegadores Web resolvem isso usando um sistema chamado DOM, sigla em inglês que significa *Modelo de Objetos para o Documento*.

A solução é transformar cada *tag* do HTML (como `<head>`, `<body>`, `<ul>`, etc.) em um *objeto* Javascript, lembrando que *objeto* é um dos tipos de dados da linguagem Javascript (analogamente a números e strings).

A imagem abaixo ilustra uma página HTML e como ela é organizada do lado Javascript com vários objetos (retângulos vermelhos na figura):



# Fundamentos do DOM

## O que são objetos

Para ser capaz de usar os objetos do DOM no Javascript, você precisa entender o que são objetos primeiro.

Vamos começar por isso.

Mas antes de mais nada, atente-se a dois fatos:

- **Objeto é um tipo de dados**

  Significa que é algo que pode ser guardado numa variável assim como números e strings.

  Ou seja, da mesma forma que é possível fazer `const idade = 10;` , é também possível fazer `const meuObjeto = 《 objeto aqui 》;`

  Só falta saber o que exatamente escrever na lacuna "《 objeto aqui 》".

  E assim como é possível fazer `console.log(idade)`, que vai mostrar `10`, também é possível fazer `console.log(meuObjeto)`, que vai mostrar o objeto guardado na constante `meuObjeto`.

- **Nem todo objeto pertence ao DOM**

  Como mostramos no diagrama ilustrativo do DOM (retângulos vermelhos), existem objetos que são "espelhos" ao HTML.

  Mas nem todos os objetos são assim.

  Você pode criar um objeto "avulso" que não está "conectado" ao HTML.

  Em outras palavras:

  Existem os objetos do DOM, que são espelhos para o HTML.

  E também existem outros objetos que não são do DOM, e portanto não têm nenhuma influência sobre o HTML.

# Fundamentos sobre objetos

## Introdução

Não vamos falar aqui sobre objetos do DOM, mas sim sobre objetos "avulsos", que não têm nenhuma conexão com o HTML.

Depois voltamos ao DOM.

Para começar, suponha que você tem um código com informações sobre usuários:

```
const user1Name = "Maria";
const user1Age = 30;
const user1Email = "maria@example.com";

const user2Name = "João";
const user2Age = 40;
const user2Email = "joao@example.com";
```

Esquematicamente, estamos assim:


São 6 variáveis, mas as 3 primeiras se referem uma única "entidade do mundo", que é um usuário.

E o mesmo vale para as outras 3 variáveis.

O problema aqui é que há variáveis demais (6) para algo simples (2 usuários).

Para resolver isso, o javascript tem objetos.

# Fundamentos sobre objetos

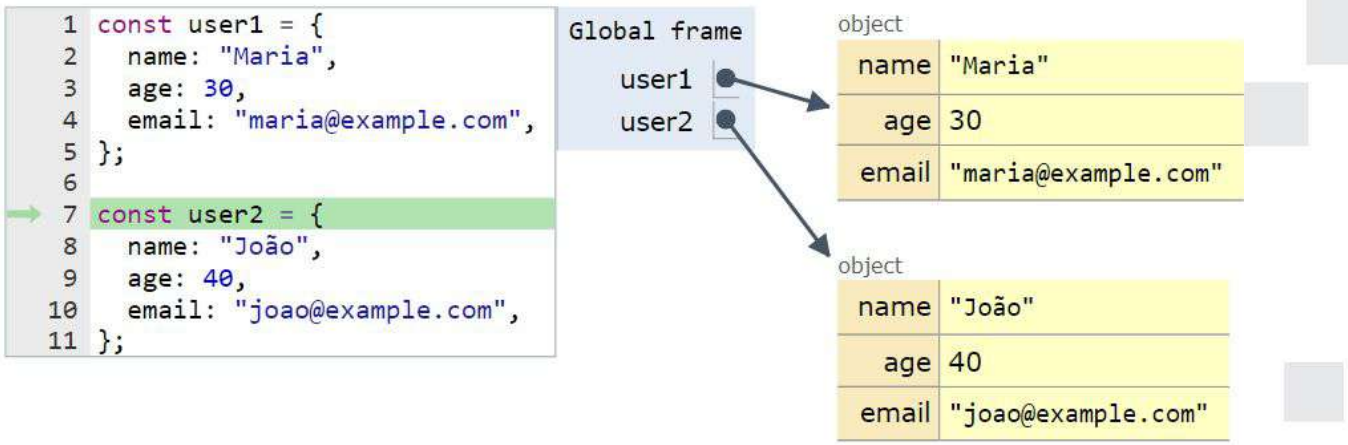
## Sintaxe de criação de objeto

Um objeto é uma forma de aglutinar várias variáveis em uma só.

Convertendo o código anterior para usar objetos, fica assim:

```
const user1 = {
  name: "Maria",
  age: 30,
  email: "maria@example.com",
};

const user2 = {
  name: "João",
  age: 40,
  email: "joao@example.com",
};
```

Esquematicamente, ficamos assim:




Note que agora são somente 2 variáveis: `user1` e `user2`.

Mas o valor de cada variável é uma "tabela" (em amarelo) que contém as 3 informações do usuário.

Essa tabela é o tipo de dados *objeto*.

Pronto ! Com essa ferramenta, quando temos uma "entidade do mundo" com várias informações associadas, podemos criar uma única variável contendo um objeto (tabela) para armazenar essas informações, em vez de várias variáveis separadas.

Dentro do objeto: `name`, `age` e `email` são chamadas **Propriedades** e `"Maria"`, `30`, etc. são chamados **Valores**.

Em geral, a sintaxe para criar um objeto é:

```
{
  《nome da propriedade》: 《valor》,
  《nome da propriedade》: 《valor》,
  ...
}
```

# Fundamentos sobre objetos

As vírgulas são necessárias, mas as quebras de linha não.

Então poderia ser tudo numa mesma linha:

```
const user1 = { name: "Maria", age: 30, email: "maria@example.com" };
```

Por último, é importante notar que as variáveis `user1` e `user2` não **São** objetos, mas sim **Armazenam** objetos.

Em outras palavras, assim como no código `const x = 10;` dizemos que `10` é um valor armazenado na variável `x`, também dizemos no código `const user1 = {.....};` que o objeto (tabela amarela no diagrama) é um valor armazenado na variável `user1`.

Só que `10` é um valor simples (número) enquanto um objeto é um valor complexo, com estrutura interna (é uma tabela).

## Acessando e modificando propriedades

As propriedades de um objeto funcionam como variáveis.

Para acessá-las, usamos a notação de ponto (dot notation):

- `《objeto》.《propriedade》`

Por exemplo:

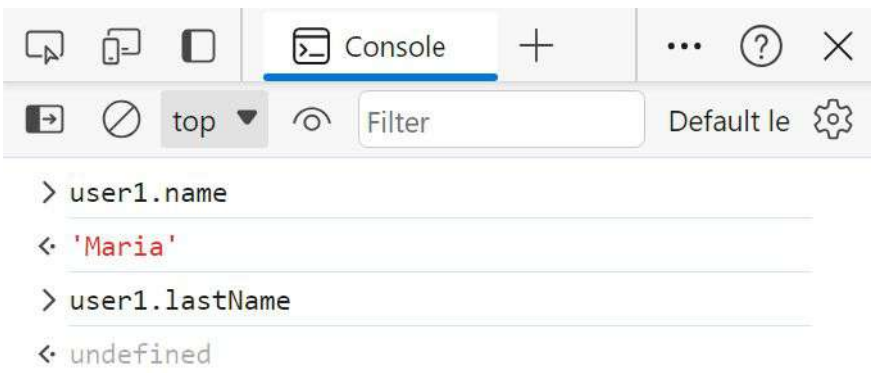
```
console.log(user1.name); // "Maria"
console.log(user1.age); // 30

console.log(user2.name); // "João"
console.log(user2.age); // 40
```

E para modificar o valor de uma propriedade:

```
// Originalmente, `user1.name` tem o valor "Maria"
console.log(user1.name); // "Maria"

// Alteramos o valor da propriedade `name`
user1.name = "Carlos";

console.log(user1.name); // "Carlos"
```

Mesmo se uma propriedade não existe no objeto, ao tentar acessá-la não vai acontecer um erro.

Em vez disso, o acesso terá resultado **`undefined`**. Veja:



# Fundamentos sobre objetos

Como a propriedade `lastName` não existe, o acesso resulta em **`undefined`**.
E é possível adicionar novas propriedades, por exemplo:

```
const user1 = {
  name: "Maria",
  age: 30,
  email: "maria@example.com",
};

// Originalmente, `user1.lastName` não existe.
// Vamos adicionar:
user1.lastName = "Silva";

// Agora `user1` tem 4 propriedades
console.log(user1.lastName); // "Silva"
```

Como `user1` foi declarado como **`const`**, não deveria ser proibido alterá-lo ?
Ou seja, no código abaixo, a última linha não deveria gerar um erro ?

```
const user1 = {
  name: "Maria",
  age: 30,
  email: "maria@example.com",
};

user1.age = 31; // não deveria gerar um erro ?
```

A resposta é: não é um erro, funciona normalmente.
A palavra **`const`** não impede de alterar os valores das propriedades, ela só impede de alterar *a variável em si*.
Ou seja, ela impede de fazer: `user1 = 《qualquer coisa aqui》`
Em outras palavras, impede o comando de atribuição na variável.
Mas não impede o comando de atribuição nas propriedades que estão dentro do objeto.

# Fundamentos sobre objetos

## Objetos dentro de objetos

É possível colocar um objeto dentro de outro.
Por exemplo, a propriedade `address` (endereço) abaixo tem como valor outro objeto:

```
const user1 = {
  name: "Maria",
  age: 30,
  email: "maria@example.com",
  address: {
    street: "Rua ABC",
    number: 720,
    city: "São Paulo",
  },
};
```

O funcionamento disso é análogo ao que já vimos.
Por exemplo, para imprimir a rua (street), fazemos:

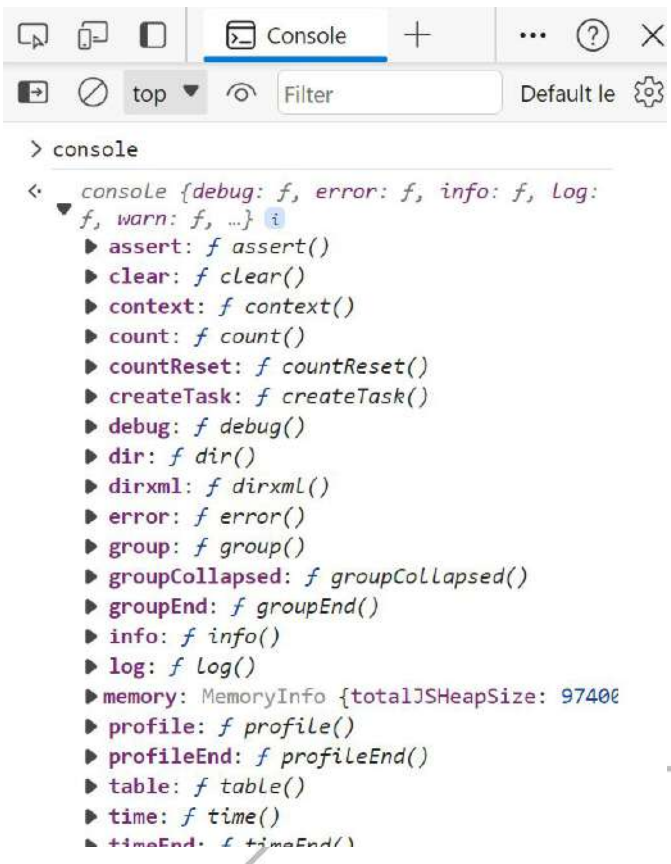
- `console.log(user1.address.street); // "São Paulo"`

Para modificar a rua, fazemos:

- `user1.address.street = "Rua DEF";`

## Objeto console e métodos

Na verdade você já conhecia um objeto, sem saber que era um objeto.
Trata-se do `console`, que é um objeto pré-definido do navegador web.
Para ver melhor, digite `console` no Console:



# Fundamentos sobre objetos

O que aparece são todas as propriedades do objeto `console`.

Note que lá no meio tem a propriedade chamada `log`.

É ela que está sendo acessada quando você faz `console.log(...)`

A única diferença da propriedade `console.log` para as propriedades `user1.name` e `user1.age` do nosso objeto `user1` é:

- Nossas propriedades tinham valores número e string, como `"Maria"` e `30`.
- Já a propriedade `console.log` tem como valor uma função.

Quando uma propriedade de objeto tem como valor uma função, dizemos que essa propriedade é um **Método**.

Ainda não sabemos criar métodos, porque não sabemos criar funções. Veremos sobre isso futuramente.

Por enquanto, só nos importa fazer bom uso de métodos de objetos pré-definidos, como o `console.log`

## Métodos de número e string

Números e strings não são objetos, mas também possuem métodos.

Por exemplo, strings possuem o método `toUpperCase` para transformar a string em letras maiúsculas:

```
> const name = "Maria"
< undefined
> name.toUpperCase()
< 'MARIA'
```

Note que não é `toUpperCase(name)` mas sim `name.toUpperCase()`.

Isso é porque `toUpperCase` é uma propriedade da string.

Como visto na imagem, o `toUpperCase` retorna a string em letras maiúsculas.

Mas é uma cópia da string, ou seja, a constante `name` continua com valor `"Maria"`, não mudou para `"MARIA"`.

Isso vale para todos os métodos de número e string: eles não modificam o número/string original.

Para números, um exemplo de método é o `toFixed`:

```
> const pi = 3.141592
< undefined
> pi.toFixed(2)
< '3.14'
```

Ele arredonda o número para a quantidade solicitada de casas decimais. No exemplo acima, duas casas decimais.

Talvez seja contraintuitivo, mas o `toFixed` retorna uma string. Veja acima que o resultado é `"3.14"` , não `3.14`.

# Fundamentos sobre objetos

Por último, é importante notar que métodos de string só existem em strings, e métodos de número só existem em números.

Por exemplo, se você tentar usar o método `toUpperCase` (que só existe em string) num número, vai dar erro:

```
> const pi = 3.141592
<· undefined
> pi.toUpperCase()
✕ ▶Uncaught TypeError: pi.toUpperCase is not a function    VM598:1
    at <anonymous>:1:4
```

Existem vários métodos de string e número, você pode ver mais na MDN:

- Métodos de string (MDN)
- Métodos de número (MDN)

A documentação da MDN fala sobre vários assuntos, os métodos estão na seção "Instance Methods" de cada página.

# Manipulação básica do DOM

## Introdução

Nesse ponto você sabe o que são objetos e suas propriedades, as quais podem armazenar valores de qualquer tipo de dados, incluindo números, strings, outros objetos e até funções.

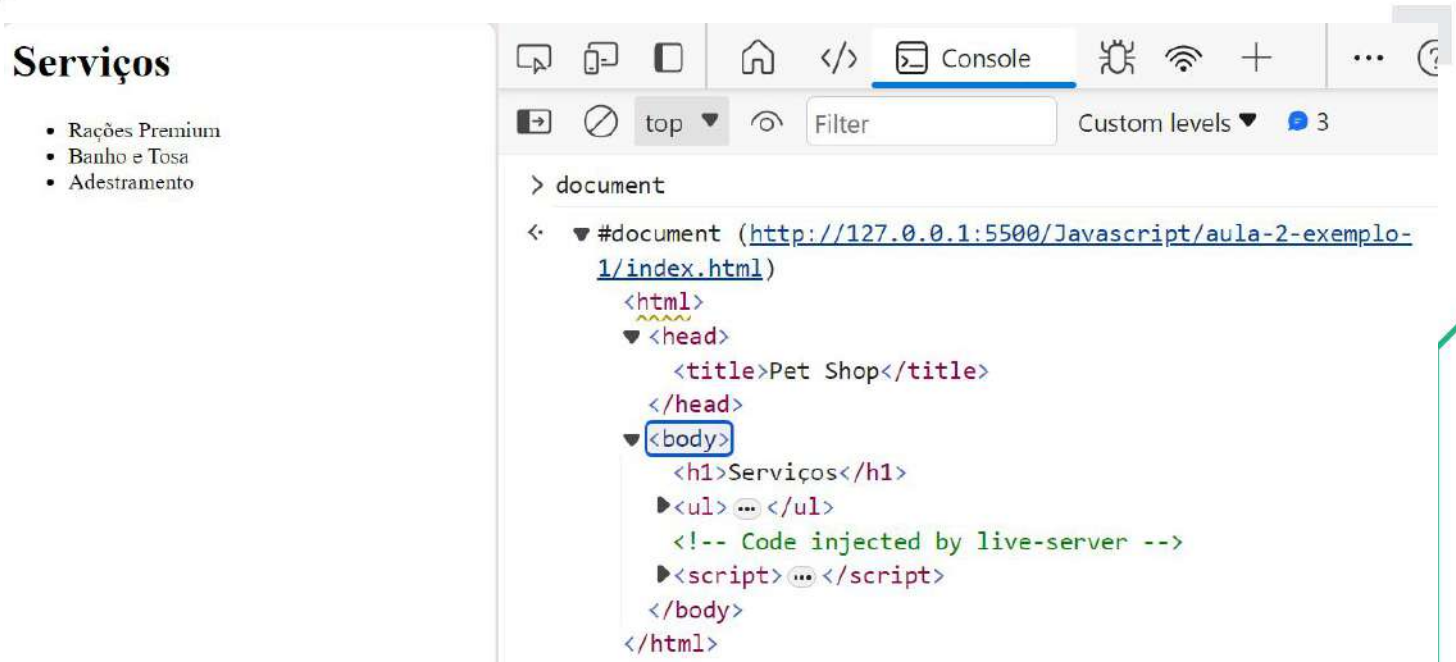
Agora veremos quais são os objetos pré-definidos do navegador web, que conseguem alterar a página HTML.

Usaremos como exemplo a mesma página do Pet Shop do início deste capítulo:

```
<html>
  <head>
    <title>Pet Shop</title>
  </head>

  <body>
    <h1>Serviços</h1>

    <ul>
      <li>Rações Premium</li>
      <li>Banho e Tosa</li>
      <li>Adestramento</li>
    </ul>
  </body>
</html>
```

## Objeto `document`

Depois de abrir a página, digite `document` no Console:



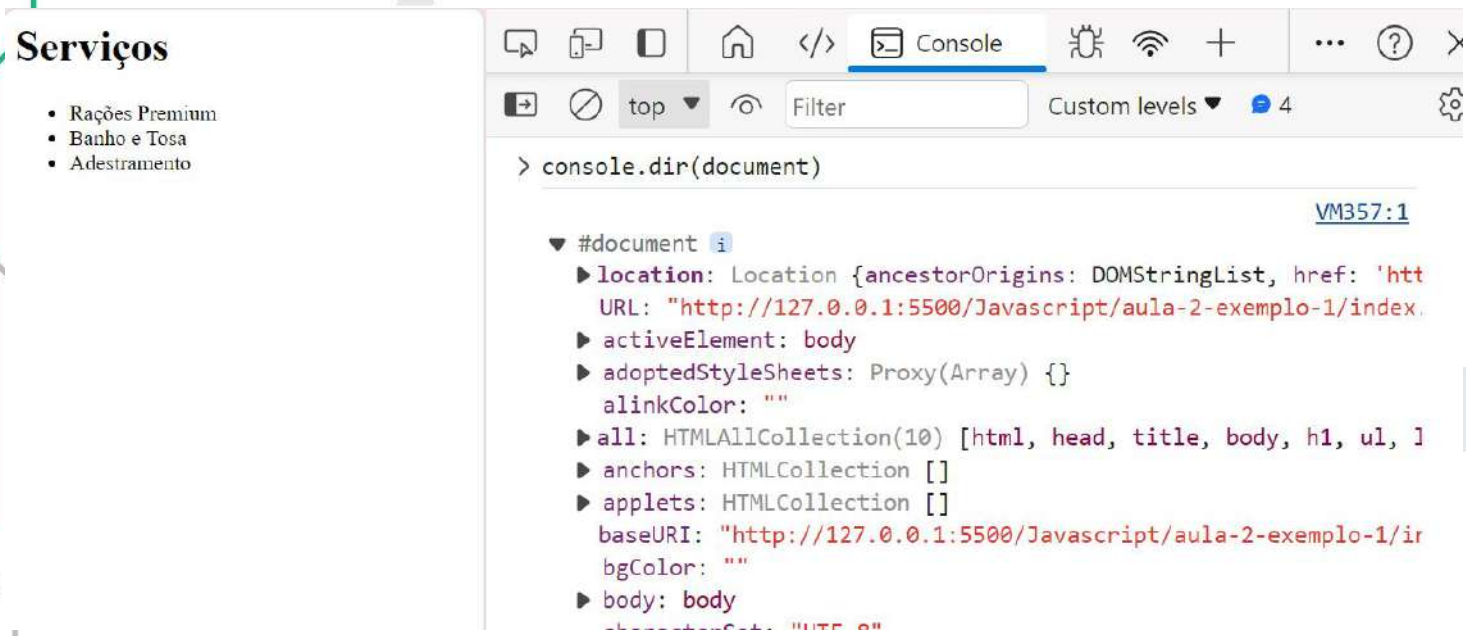
# Manipulação básica do DOM

O resultado parece ser a própria página HTML !

Na verdade `document` é um objeto do DOM, volte no diagrama sobre o DOM (do início do capítulo) e verá que `document` é o retângulo vermelho superior que dá origem a todos os outros.

Esse objeto representa o documento da página como um todo, por isso o Console o exibe como se ele fosse a própria página HTML, em vez de mostrar a tabela de propriedades e valores (como um objeto normal).

Mas você pode forçar a exibição como um objeto normal usando a função `console.dir`:




Agora é possível ver as propriedades desse objeto.

A maioria delas é complexa de entender, mas há algumas que são reconhecíveis, por exemplo `document.URL` é a string `"http://127.0.0.1:5500/Javascript/aula-2-exemplo-1/index.html"` , que é o endereço da página.

O objeto `document` é a origem do DOM. Ele é o pai de todos os outros objetos do DOM.

**Objeto `window`**

Existe um outro objeto relacionado, o `window`.

Enquanto o `document` tem informações sobre o documento HTML, o `window` tem informações sobre a *janela do navegador*.

Por exemplo as dimensões da janela (largura e altura) e o deslocamento da janela em relação à tela do computador (caso a janela não esteja maximizada).

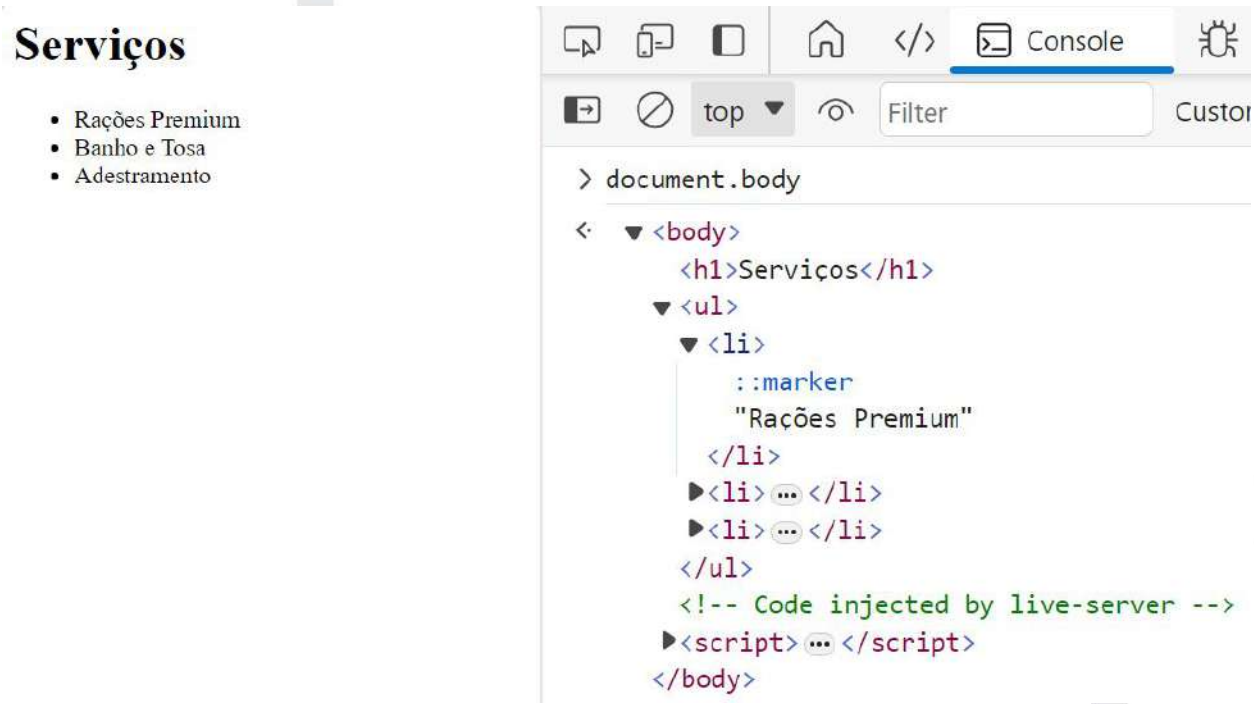
# Manipulação básica do DOM

## Objeto `document.body`

Na figura anterior, note que uma das propriedades do `document` se chama `body`, e ela também aparece no diagrama de retângulos do DOM que fizemos no início deste capítulo.

O `document.body` é outro objeto do DOM (subobjeto dentro do `document`) que representa a tag `<body>` do HTML.

Veja no Console:

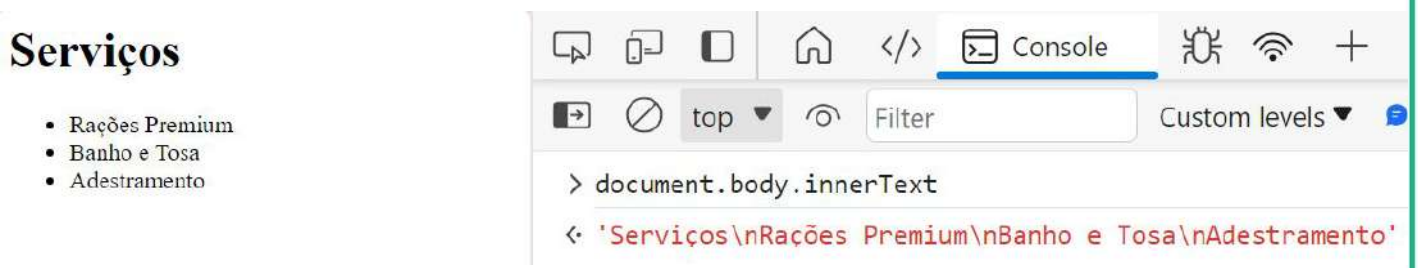



Você pode testar `console.dir(document.body)` para ver todas as propriedades dele (omitimos aqui). Veremos algumas propriedades bastante úteis.

## Propriedade `document.body.innerText`

A propriedade `innerText` do `document.body` tem valor string.

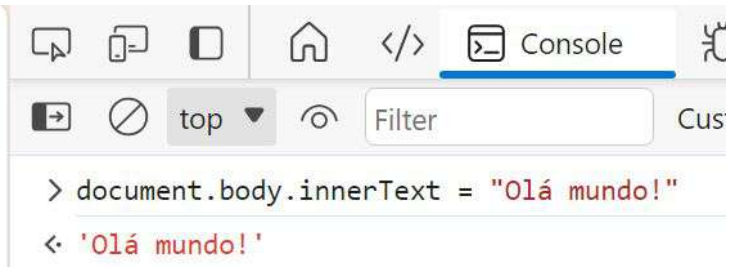
Ela é o texto que está escrito dentro da tag `<body>`. Veja:



# Manipulação básica do DOM

Note que é somente o texto da página sem incluir tags HTML: não aparecem `<h1>`, `<ul>` e `<li>`.
E você pode trocar o texto da página simplesmente trocando o valor dessa propriedade.
Por exemplo, para colocar a mensagem "Olá mundo!" na página, seria assim:

Olá mundo!




Note que isso substituiu totalmente o que havia no `<body>`.
Você pode confirmar isso olhando na aba Elements das ferramentas de desenvolvedor. Só tem a mensagem "Olá mundo!" dentro do `<body>` agora:

Olá mundo!


Existe outra propriedade parecida com `innerText`, chamada `textContent`.
Apesar de similares, existem algumas diferenças entre as duas.
Consulte a documentação da MDN sobre textContent.

# Manipulação básica do DOM

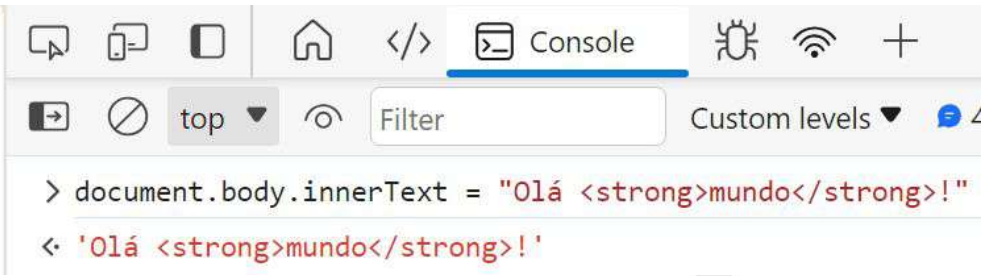
## Propriedade `document.body.innerHTML`

Uma limitação do `innerText` é que não podemos inserir tags HTML na página.

Por exemplo, suponha que você quer escrever "Olá mundo!" onde a palavra "mundo" esteja em negrito (tag `<strong>`).

A tentativa abaixo com `innerText` não funciona, como você pode ver:

Olá mundo!

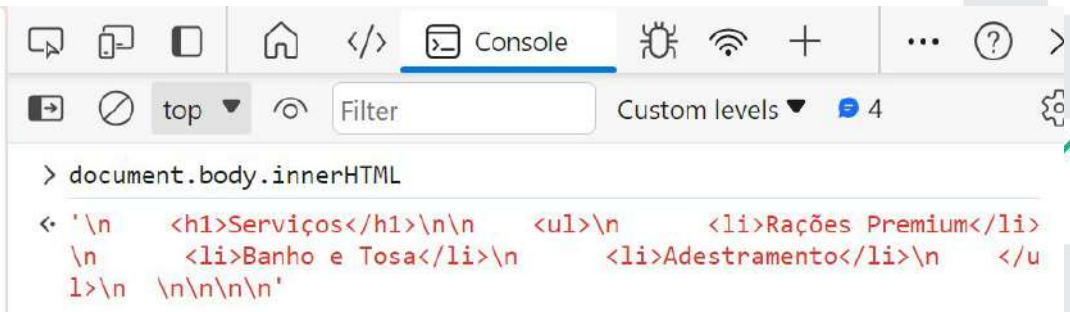



Para que o HTML interprete os "strong" como tags de negrito, em vez de literalmente como a palavra "strong", precisamos da propriedade `innerHTML`.

Ela contém o código do `<body>` incluindo o HTML completo, não só o texto:

**Serviços**

- Rações Premium
- Banho e Tosa
- Adestramento




E com essa propriedade é possível fazer a mensagem com negrito:

Olá **mundo**!


Com a propriedade `innerHTML`, o javascript pode inserir absolutamente qualquer HTML dentro do `<body>`:

**Teste**

Um *parágrafo* aqui

- Item 1
- Item 2

# Manipulação básica do DOM

Ao fazer `document.body.innerHTML = "html aqui"` , você sabe que isso substitui totalmente o conteúdo do `<body>`.

Se você quer *adicionar* conteúdo em vez de *substituir* conteúdo, tem as seguintes opções (que já deve reconhecer, pois são comandos de atribuição já vistos):

- `document.body.innerHTML = document.body.innerHTML + "html adicional aqui";`
- `document.body.innerHTML += "html adicional aqui";`

## Método `document.querySelector`

Usando `document.body.innerHTML`, podemos substituir o HTML do `<body>` da página, mas não é possível adicionar conteúdo *no meio* da página.

Por exemplo, na página do Pet Shop, se quisermos alterar com javascript o conteúdo do `<h1>` sem alterar o restante da página, o `document.body.innerHTML` não vai ajudar.

Felizmente, o DOM não tem somente `document.body`, que é o objeto que representa o `<body>` da página.

Pelo contrário, o DOM tem *um objeto para cada tag da página*: um objeto para o `<h1>`, outro para a `<ul>`, e outros três para cada `<li>`.

Se soubermos como acessar o objeto correspondente ao `<h1>`, poderemos alterar somente o conteúdo do `<h1>` da seguinte forma:

```
const h1Object = 《código para encontrar o objeto <h1>》;

h1Object.innerText = "Novo título da página";
```

Falta acertar a primeira linha desse código.

Para isso existe a função `document.querySelector`.

Ela usa uma ideia emprestada do CSS: os seletores.

Vamos lembrar o que são seletores CSS. Considere o seguinte CSS hipotético:

```
ul {
  background-color: black;
}

.red {
  background-color: red;
}

#mylist {
  color: blue;
}
```

Os seletores são *ul*, *.red* e *#mylist*. Ou seja, são as designações que você usa no CSS para selecionar os elementos da página aos quais cada regra CSS deve ser aplicada.

# Manipulação básica do DOM

No javascript, a função `document.querySelector` usa um seletor CSS para encontrar o elemento solicitado no HTML.

Por exemplo:

- `document.querySelector("ul")` encontra uma `<ul>`
- `document.querySelector(".red")` encontra um elemento com classe CSS *red* (não importa qual a tag do elemento, se é `<p>`, `<h1>` ou qualquer outra coisa)
- `document.querySelector("#mylist")` encontra um elemento com ID *mylist* (não importa qual a tag do elemento)

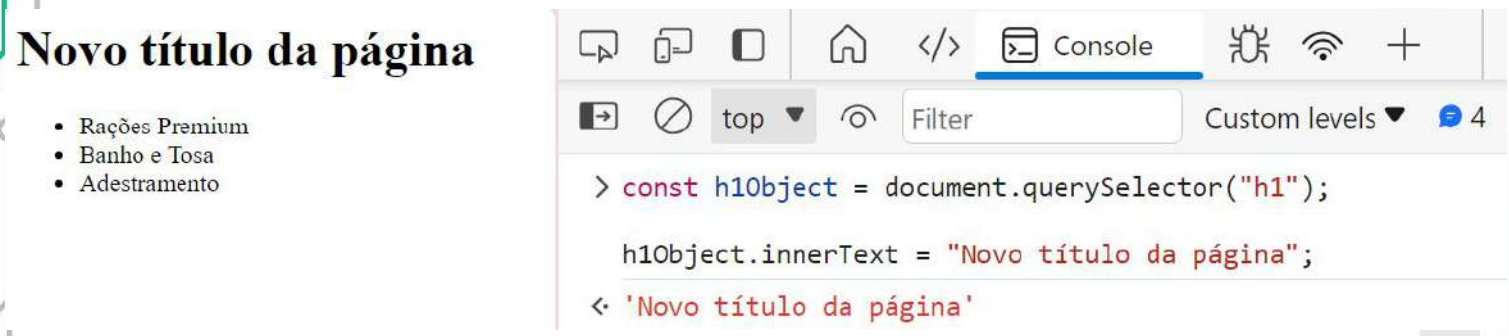
Em qualquer caso, se o `querySelector` encontrar vários elementos, ele retorna o primeiro deles.

Por exemplo, se há várias `<ul>` na página, então `document.querySelector("ul")` vai encontrar a primeira `<ul>`.

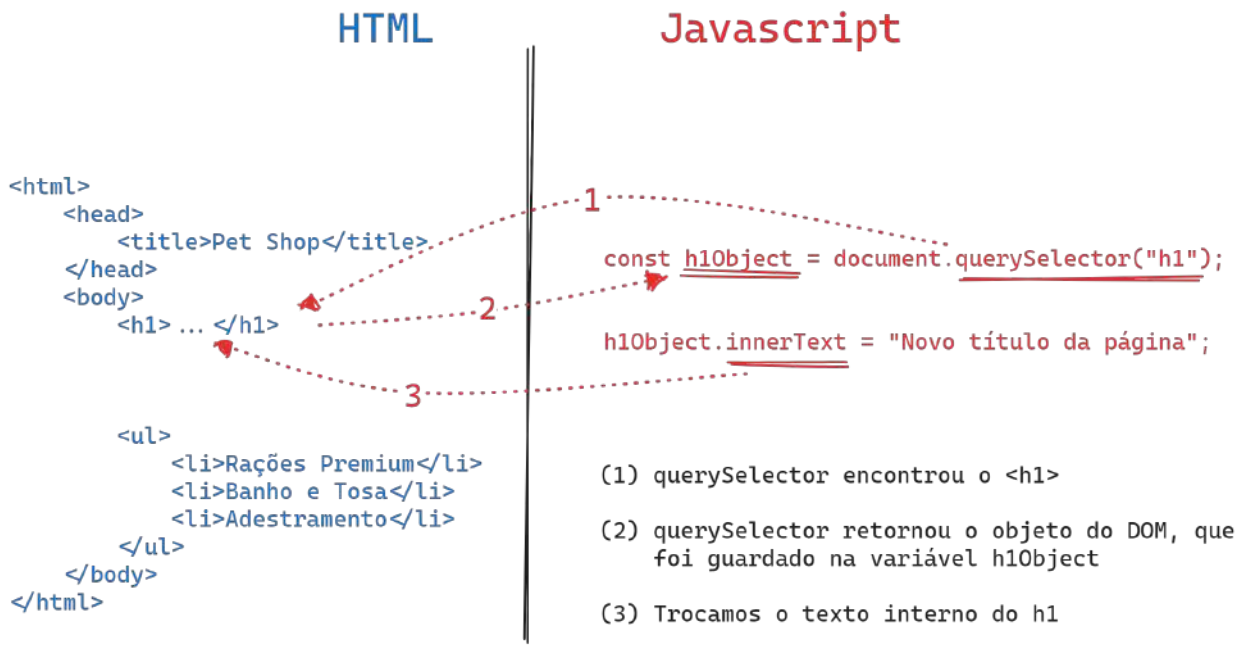
No nosso caso, queremos encontrar o `<h1>`, então o código final fica assim:

```
const h1Object = document.querySelector("h1");

h1Object.innerText = "Novo título da página";
```

Vamos testar para confirmar que funciona:




Esquematicamente, o que fizemos foi o seguinte:



# Manipulação básica do DOM

Se o `querySelector` não encontrar nenhum elemento correspondente na página, ele retornará **`null`**.

Por exemplo, na página do Pet Shop não existe nenhum `<h2>`, logo `document.querySelector("h2")` retornará **`null`**.

Nesse caso, o código pode gerar um erro de execução mais tarde, por exemplo no código abaixo:

```
// querySelector retorna null
// (note que isso não é um erro de execução)
const h2Obj = document.querySelector("h2");

// A variável `h2Obj` tem valor `null`.
// Por isso essa linha produz um erro de execução,
// porque `null.innerText` é um comando inválido
h2Obj.innerText = "Novo título para o h2";
```

Se você tiver erros, fique atento se seus seletores estão escritos corretamente.

Podemos encontrar o objeto do DOM e em seguida alterar o `innerText` dele tudo na mesma linha:

- `document.querySelector("h2").innerText = "Novo título para o h2";`

Assim poupamos 1 passo: não precisamos criar uma variável para armazenar o objeto retornado pelo `querySelector`.

# Manipulação básica do DOM

## Caso especial: seleção por ID

Como o `querySelector` seleciona o *primeiro* elemento que ele encontra, nosso código atual não funcionaria se a página tivesse dois `<h1>` e quiséssemos selecionar o segundo deles.

Para isso, precisaríamos mudar o seletor CSS.

Por exemplo, se o HTML do segundo h1 tiver um ID como `<h1 id="second-title">` , podemos usar:

- `document.querySelector("#second-title");`

Usar IDs no HTML é uma maneira fácil de garantir que o `querySelector` encontre o que queremos.

Além disso, no caso específico de selecionar um elemento pelo ID, existe outra função `document.getElementById` que é otimizada para esse caso.

Ela seria usada assim:

- `document.getElementById("second-title");` , note que não colocamos o # porque ele é implícito

Essa função somente seleciona pelo ID (não suporta outros seletores CSS), por isso o # é omitido.

E o `getElementById` geralmente é *mais rápido* do que o `querySelector` para encontrar um elemento pelo ID, porque é otimizado para isso.

Mas não quer dizer que o `querySelector` seja lento. Os dois são muito rápidos, só que um pode ser mais rápido que o outro.

Na prática, essa diferença de velocidade muito dificilmente será observável.

## Outras propriedades

Qualquer objeto do DOM para o HTML tem as propriedades `innerText` e `innerHTML` que vimos.

Além delas, existem inúmeras outras propriedades que não tratamos aqui. Algumas delas são:

- A propriedade `style` é um subobjeto que permite alterar as regras CSS do elemento.
  Por exemplo: `h1Object.style.color = "red";` torna vermelha a cor da fonte do h1.
  Ou ainda: `h1Object.style.border = "1px solid black";` adiciona um contorno preto ao h1.
  Qualquer regra do CSS pode ser aplicada: margin, padding, border, etc.
  No caso de regras com hífen como "background-color", você deve converter para "camelCase".
  Ou seja, remover o hífen e converter a letra da próxima palavra para maiúscula.
  Ficaria "backgroundColor": `h1Object.style.backgroundColor = "blue";`

- A propriedade `classList` é um subobjeto que permite adicionar e remover classes CSS, usandos os métodos `add` e `remove`.
  Por exemplo `h1Object.classList.add("red")` adiciona a classe "red" ao h1.
  E `h1Object.classList.remove("red")` a remove.
  Existem outros métodos, como o `toggle`, que adiciona a classe se ela não existir, ou remove se ela existir.

Você pode consultar as referência da MDN "Element" e "HTML Element" para ver todas as propriedades disponíveis.

O termo "Element" é um grupo que se refere a todos os objetos do DOM.

Portanto as propriedades listadas nessas referências existem em qualquer objeto do DOM.

Mas alguns objetos são especializados e possuem propriedades adicionais que não estão listadas nos links anteriores.

# Manipulação básica do DOM

Por exemplo, o objeto para um `<button>` tem propriedades adicionais (como `disabled`) que não existem em outros objetos do DOM.

Nesses casos, você deve consultar a referência específica da tag para descobrir quais são as propriedades extras que ela tem.

No caso do `<button>`, você teria que consultar a página "HTML Button Element".

A MDN tem aqui um índice de todas as tags, procure os nomes que começam com "HTML" (HTMLElement, HTMLButtonElement, HTMLImageElement, etc.)

# Reagindo a eventos de clique

## Introdução

Com o que vimos até agora, podemos adicionar um calculador de taxa de serviço ao site do Pet Shop.

Suponhamos que o Pet Shop cobra R$10,00 de taxa fixa e uma taxa adicional de R$2,00 para cada quilograma de peso do animal.

Vamos adicionar um parágrafo no final do código HTML:

```
<html>
  <head>
    <title>Pet Shop</title>
  </head>

  <body>
    <h1>Serviços</h1>

    <ul>
      <li>Rações Premium</li>
      <li>Banho e Tosa</li>
      <li>Adestramento</li>
    </ul>

    <h1>Taxa de serviço</h1>

    <p id="service-fee"></p>
    <script src="index.js"></script>
  </body>
</html>
```

O parágrafo está vazio porque o javascript vai pedir ao cliente que informe o peso do animal, e então escrever dentro do parágrafo o valor da taxa de serviço.

Então um programa (*index.js*) para calcular e mostrar na página a taxa de serviço seria assim:

```
const animalWeight = Number(prompt("Digite o peso do animal (kg)"));

const serviceFee = 10 + animalWeight * 2;

const paragraph = document.getElementById("service-fee");

paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
```

# Reagindo a eventos de clique

Com isso, logo ao abrir a página, o script executa e aparece o prompt:


Digitando "20", a página mostra o resultado como esperado:


Mas tem um problema aqui !
Você só *abriu* a página e já apareceu um prompt. Isso não é *normal*.
Normal seria ter um botão dentro da página, escrito "Calcular".
Ao abrir a página, não deveria aparecer nenhum prompt.
Somente ao clicar no botão. Aí apareceria o prompt e seria feito o cálculo e exibição do resultado.
Veremos como isso pode ser feito.

# Reagindo a eventos de clique

**Você notou que o `<script>` está no final do `<body>`, não na `<head>` ?**

Uma tag `<script>` pode ser colocada em qualquer lugar no HTML, mas há diferenças de um lugar para outro.

No exemplo, colocamos o `<script>` no final do `<body>`.

Mas também seria possível colocá-lo na `<head>`, ou seja, `<head>...<script></script></head>`.

Para perceber a diferença entre as duas opções, saiba que o navegador web carrega o script assim que ele o encontra na página.

Então se o script estiver na `<head>`, o navegador web vai carregá-lo antes de processar o `<body>` da página.

Portanto quando o script executar a linha `document.getElementById("service-fee")`, o resultado será `null` !

Porque nesse momento o navegador web ainda não processou o conteúdo do `<body>`, então ainda não existe o `<p>`.

O script não funcionaria como intencionado.

Já se o script estiver no final do `<body>`, o navegador web já terá finalizado a montagem do conteúdo da página (body) quando começar a executar o script.

Então `document.getElementById("service-fee")` vai funcionar como esperado.

## Funções

Normalmente, quando o navegador web encontra a tag `<script>` na página, ele executa imediatamente todo o código javascript.

Por isso nosso programa executa logo que a página é aberta (mostrando o prompt e tudo mais).

Mas agora queremos evitar isso. O código *não deve* ser executado imediatamente, mas somente quando o usuário clicar no botão.

Então precisamos de uma forma de *desativar* o código para que ele não seja executado imediatamente, mas ainda possa ser *ativado* (executado) depois.

Em javascript, isso é justamente o que chamamos de **Função**: várias instruções de código desativadas, que podem ser ativadas sob demanda.

O código ficará assim:

```
function calculateFee() {
  const animalWeight = Number(prompt("Digite o peso do animal (kg)"));

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");

  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}
```

# Reagindo a eventos de clique

As únicas diferenças são a primeira e a última linhas:

- Na primeira linha, a palavra-chave `function` indica ao navegador web que estamos prestes a criar uma função.
  É necessário começar com essa palavra.

- A palavra `calculateFee` é o nome da função.
  Esse nome é arbitrário, segue as mesmas regras de nomeação de variável.

- Os parênteses são necessários, eles são usados para **Parâmetros**.
  Nesse caso não há nenhum parâmetro (nada escrito no meio dos parênteses).
  Futuramente veremos o que são parâmetros e a utilidade deles. Atualmente não precisamos.

- Finalmente, o abre-chaves { indica que depois dele virão as instruções de código inativadas.

- Da segunda linha em diante, é o mesmo programa que tínhamos antes, sem nenhuma modificação.

- Na última linha, o fecha-chaves } indica que acabou o código da função.

As chaves delimitam o código da função: todas as instruções entre o abre-chaves { e o fecha-chaves } são consideradas instruções inativas.
Significa que, quando a página web for aberta, o navegador *não vai executá-las*, exatamente como queremos.

A primeira linha `function calculateFee()` é chamada de **Cabeçalho** da função.
E o comando completo, desde o cabeçalho até o fecha-chaves, é chamado de **Declaração de Função**.
Pois com esse código estamos declarando (definindo) qual é o nome da função e qual é seu código interno.

Você percebeu que o código interno da função foi escrito com espaçamento em relação à margem esquerda ?
Esse espaçamento é chamado de *indentação*.
Ele é opcional, se você tivesse escrito o código sem indentação, tudo funcionaria da mesma maneira.
Mas a indentação ajuda *visualmente* a identificar quais linhas de código pertencem ao interior da função e quais estão por fora.

# Reagindo a eventos de clique

## Método addEventListener

A função resolve metade do problema: não executar código de imediato.
Falta fazer com que esse código seja executado ao clique de um botão.
Podemos adicionar um botão no código HTML:

```
<html>
  <head>
    <title>Pet Shop</title>
  </head>

  <body>
    <h1>Serviços</h1>

    <ul>
      <li>Rações Premium</li>
      <li>Banho e Tosa</li>
      <li>Adestramento</li>
    </ul>

    <h1>Taxa de serviço</h1>

    <button id="calculate-service-fee">Calcular</button>
    <p id="service-fee"></p>
    <script src="index.js"></script>
  </body>
</html>
```

Agora, no javascript, precisamos instruir o navegador a executar o código da nossa função `calculateFee` quando esse botão for clicado.
O código necessário é o seguinte:

```
function calculateFee() {
  const animalWeight = Number(prompt("Digite o peso do animal (kg)"));

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");

  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", calculateFee);
```

# Reagindo a eventos de clique

Note que somente adicionamos as últimas duas linhas.

Como elas estão *fora* das chaves da função, elas são executadas imediatamente quando a página abre.

E nelas fazemos o seguinte:

Primeiro encontramos o objeto DOM do botão, como você já conhece.

Depois usamos o método `addEventListener` para instruir o navegador a executar a função `calculateFee` quando o botão for clicado ("click").

A string `"click"` é um **Evento**.

O navegador reconhece vários outros eventos além de clique.

Por exemplo: `"mouseenter"` quando o mouse entra em um elemento, `"mouseleave"` quando o mouse sai de um elemento, `"submit"` quando o usuário envia um formulário, etc.

O `addEventListener` serve para associar uma função a um evento: quando acontecer aquele evento, o navegador executará aquela função.

Da maneira como fizemos o código, a função `calculateFee` será executada quando o botão for clicado.

Já se tivéssemos escrito `button.addEventListener("mouseenter", calculateFee);` , a função seria executada quando o mouse entrasse no botão (hover).

O código acima já é a versão final: ao abrir a página, nenhum prompt aparece. Ao clicar no botão, a função é executada, então aparece o prompt e depois o resultado no parágrafo (como antes).

Se o usuário clicar uma segunda vez no botão, a função executa novamente, repetindo o programa (teste aí !)

O nome `addEventListener` tem tradução literal "adicionar ouvinte de evento".

A ideia é que a função `calculateFee` está "ouvindo" a página web (especificamente o botão) na espera de acontecer o evento de clique.

Então é comum dizer que a função `calculateFee` é um "ouvinte de evento" (event listener), ou "gerenciador de evento" (event handler).

O método `addEventListener` não é exclusivo de botões.

Todas as tags HTML podem ser associadas a um evento.

Por exemplo, você pode associar uma função ao clique de um `<p>`, ou de um `<h1>`, etc.

# Reagindo a eventos de clique

Se o HTML tiver mais botões, é possível associar a mesma função `calculateFee` a mais de um botão:

```
button1.addEventListener("click", calculateFee);
button2.addEventListener("click", calculateFee);
...
```

E o contrário também é possível: se você tiver mais de uma função, é possível associar todas ao mesmo botão:

```
function test01() {
  ...
}

function test02() {
  ...
}

button.addEventListener("click", test01);
button.addEventListener("click", test02);
```

Nesse caso, quando o botão for clicado, a `test01` será executada primeiro, depois a `test02`, porque elas foram associadas ao botão nessa ordem.

A associação entre o clique e uma função pode ser feita ainda de duas outras maneiras além de `addEventListener`:

Pode ser usado o atributo `onclick` no próprio HTML, por exemplo:

```
<button onclick="calculateFee()">Calcular</button>
```

Nesse caso precisamos dos parênteses após o nome da função

Outra maneira é pelo javascript mesmo:

```
button.onclick = calculateFee;
```

Essas duas alternativas são desfavorecidas em relação ao `addEventListener`, então vamos seguir usando esse método.

Você pode ler mais sobre as três alternativas aqui

# Reagindo a eventos de clique

## Variáveis locais e globais

Variáveis globais são aquelas que são declaradas fora de uma função.

No nosso caso, a variável `button` é global.

Já as variáveis locais são aquelas que são declaradas dentro de uma função.

Nesse caso as variáveis `animalWeight`, `serviceFee` e `paragraph` são locais.

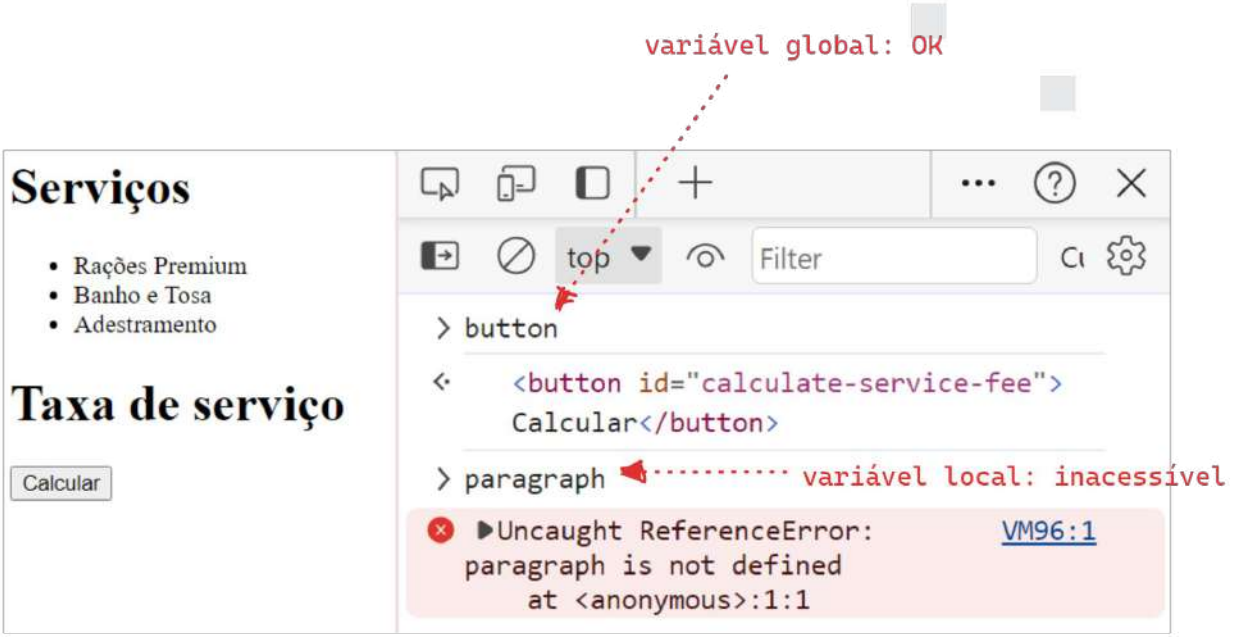
A diferença é que variáveis globais, como você já sabe, existem durante todo o tempo de vida da página web.

Já as variáveis locais só existem enquanto a função está sendo executada.

Quando o código interno da função termina de executar, as variáveis locais são deletadas da memória do computador.

E se a função for executada de novo (porque o usuário clicou outra vez no botão), as variáveis locais são recriadas para a nova execução, depois deletadas novamente (como se fossem "copos descartáveis").

Por causa dessa diferença, você pode acessar a variável global `button` pelo Console, mas não pode acessar as variáveis locais:




O único lugar onde você pode acessar as variáveis locais é dentro do código da função.

Já as variáveis globais podem ser acessadas em qualquer parte do código.

Outra nomenclatura que usamos para isso é **Escopo**.

Existe o **Escopo Global** (Global Frame, ou Global Scope), que se refere a todo código *fora* da função.

No nosso código, as últimas duas linhas estão no escopo global.

E existe o **Escopo Local** (Local Frame, ou Local Scope), que se refere a todo código *dentro* da função.

Na verdade se você tiver mais que uma função, cada função tem o seu próprio Escopo Local.

Significa que cada função tem acesso às suas próprias variáveis locais, mas não às variáveis locais das outras funções.

# Reagindo a eventos de clique

## As duas fases da execução de código

Perceba que existem dois "momentos" de execução de código:

1. Quando a página inicia, executa o código do escopo global, que cria a variável global `button` e associa o ouvinte de evento.

   Isso acontece em frações de segundo, antes de o usuário começar a interagir com a página.

   De fato, enquanto o javascript é executado, a página fica congelada e seria impossível para o usuário interagir com ela mesmo se tentasse.

   Só que esse "congelamento" é imperceptível na prática porque o código termina de executar rapidamente.

2. Mais tarde, quando o usuário começa a interagir com a página, essa interação (clique no botão) dispara o evento "click" no javascript.

   E então o código da função `calculateFee` é executado (e somente ele ! O código global não executa de novo).

## Hoisting

No nosso código, primeiro declaramos a função e depois usamos o `addEventListener`.

Mas você poderia ter feito na ordem contrária, deixando a função por baixo:

```
const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", calculateFee);

function calculateFee() {
  ...
}
```

Nesse código alternativo, a segunda linha faz referência à função que só é definida nas últimas linhas.

Se fosse uma variável (não uma função), você sabe que isso não daria certo, porque é impossível referenciar uma variável antes de declará-la.

Mas para funções *não importa* onde no código a função foi definida. Ela pode ser referenciada pelo `addEventListener` tanto antes quanto depois das linhas onde é declarada.

Isso se chama **Hoisting** ("elevação"): Antes de executar o script, o navegador web analisa o escopo global e encontra todas as declarações de função.

Quando o script começa a executar (escopo global), o navegador já sabe sobre todas as funções, não importa em qual linha foram definidas.

É como se todas as funções tivessem sido declaradas no topo do código, por isso o nome Hoisting.

# Reagindo a eventos de clique

## Método `removeEventListener`

Como dissemos, se o usuário clicar várias vezes no botão, a função será executada várias vezes, porque o `addEventListener` associa o evento "clique" à função `calculateFee` permanentemente.

Mas você pode desfazer essa associação com o método oposto: `removeEventListener`.

Ele precisa ser usado dentro da função. Adicionamos mais uma linha no fim dela:

```
function calculateFee() {
  const animalWeight = Number(prompt("Digite o peso do animal (kg)"));

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");

  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;

  // nova linha
  button.removeEventListener("click", calculateFee);
}

const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", calculateFee);
```

A lógica é: quando o usuário clica no botão, a função é executada e a última linha dela deleta a associação que o `addEventListener` havia criado.

Portanto, depois que a função executar, não existe mais associação entre o evento "clique" e a função `calculateFee`.

Então quando o usuário clicar de novo no botão, nada vai acontecer.

Note que a última linha da função está acessando a variável global `button`.

E isso é possível porque, como já dissemos, uma variável global pode ser acessada em qualquer lugar do código, seja dentro ou fora de uma função.

Você poderia pensar em escrever o `removeEventListener` na última linha do código, no escopo global:

```
function calculateFee() {
  ...
  ...
}

const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", calculateFee);

button.removeEventListener("click", calculateFee);
```

O efeito disso seria desastroso: logo que a página abre, o código associa o evento de clique à função `calculateFee`, mas logo em seguida desfaz essa associação.

Quando o usuário clicar no botão, nada vai acontecer.

# Reagindo a eventos de clique

## Função anônima

O único propósito da função `calculateFee` é ser executada quando o botão for clicado.

No código atual, primeiro definimos a função, depois chamamos o método `addEventListener` para associá-la ao botão.

Mas no javascript é comum fazer as duas coisas ao mesmo tempo: definir a função no mesmo lugar onde chamamos `addEventListener`.

Fica assim:

```
const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", function () {
  const animalWeight = Number(prompt("Digite o peso do animal (kg)"));

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");

  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
});
```

O que nós fizemos: dentro do `addEventListener` onde tinha o nome da função, colocamos diretamente o código dela (incluindo seu cabeçalho `function () { ... }`).

Note que a função não tem mais nome. Como ela não tem nome, é chamada de "anônima".

Mas o comportamento é o mesmo:

- Quando a página inicia, o código global é executado.

  Significa que o código que está fora da função é executado, incluindo o `addEventListener` que cria a associação entre o botão e a função.

  O código dentro das chaves da função *não é executado neste momento*.

- Depois, quando o usuário clica no botão, *somente é executado o código no interior das chaves da função*.

A vantagem da função anônima é que não precisamos de "criatividade" para inventar um nome para ela (ficou sem nome mesmo).

E o código `addEventListener` ficou mais próximo do código da função, portanto é mais rápido enxergar que quando *aquele* botão é clicado, *aquele* código executa.

Mas uma desvantagem é que ficou impossível usar `removeEventListener`, porque a função não tem nome.

Isso raramente é um problema, porque não é comum remover o ouvinte de evento.

# Reagindo a eventos de clique

## Táticas com variáveis globais

Como variáveis globais são permanentes e podem ser acessadas dentro e fora de funções, podemos usar essas variáveis como uma "memória" do programa.

Por exemplo, podemos fazer uma página que conta quantas vezes o usuário clicou num botão.

Toda vez que o botão for clicado, a página mostrará num parágrafo a mensagem "Você clicou X vezes".

O HTML pode ser assim (excerto):

```
<button>Clique em mim</button>
<p></p>
```

E o javascript (completo):

```
let count = 0;

const button = document.querySelector("button");
const p = document.querySelector("p");

button.addEventListener("click", function () {
  count++;
  p.innerText = `Você clicou ${count} vezes`;
});
```

Na primeira vez que o usuário clica no botão, `count` está em `0`, então a função aumenta o valor para `1` e depois escreve "Você clicou 1 vezes" no parágrafo.

Na segunda vez que o usuário clica no botão, `count` está em `1`, então a função aumenta o valor para `2` e depois escreve "Você clicou 2 vezes" no parágrafo.

E assim por diante.

No caso de `count` igual a `1`, a mensagem ficou errada do ponto de vista da gramática portuguesa: "Você clicou 1 vezes".

Para corrigir esse problema, precisaríamos dos comandos condicionais **`if`** e **`else`**.

Como ainda veremos sobre eles, deixaremos a mensagem errada por enquanto.

# Lendo o texto digitado em input

## Introdução

Na página do Pet Shop, a próxima melhoria é permitir que o usuário digite o peso do animal dentro de inputs no HTML, em vez de um prompt.

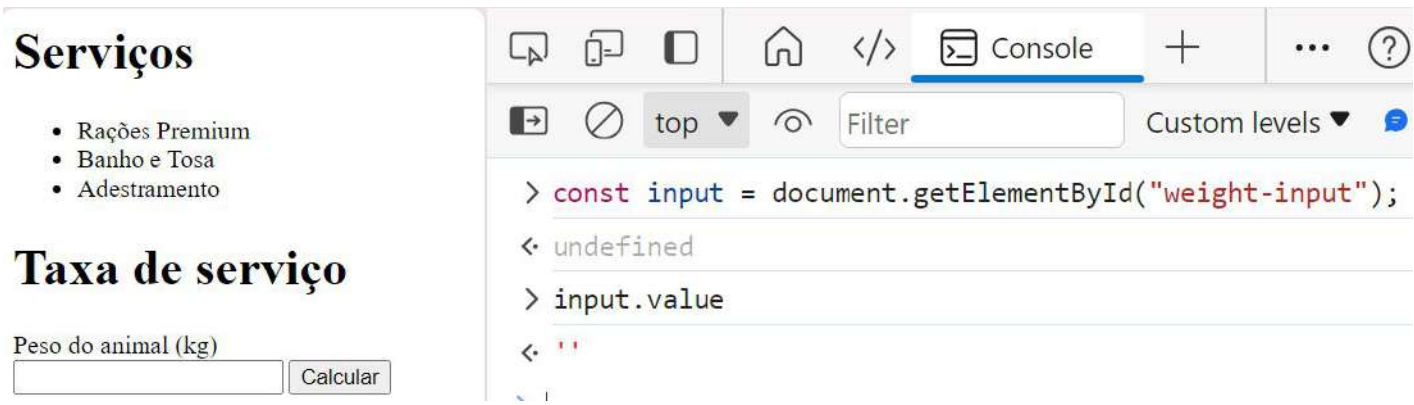
No HTML, colocaremos um `<input>` :

```
<html>
  <head>
    <title>Pet Shop</title>
  </head>

  <body>
    <h1>Serviços</h1>

    <ul>
      <li>Rações Premium</li>
      <li>Banho e Tosa</li>
      <li>Adestramento</li>
    </ul>

    <h1>Taxa de serviço</h1>

    Peso do animal (kg) <input id="weight-input" />
    <button id="calculate-service-fee">Calcular</button>
    <p id="service-fee"></p>
    <script src="index.js"></script>
  </body>
</html>
```

## Propriedade `value`

No javascript, tudo que precisamos fazer é encontrar o objeto DOM do input e acessar sua propriedade `value`, que contém a string digitada no input.

Por exemplo, se você abrir a página e acessar essa propriedade pelo Console, ela terá a string vazia porque não tem nada escrito no input:



# Lendo o texto digitado em input

Mas se você digitar algo no input e depois acessar de novo a propriedade, ela terá a string que foi digitada:


Então o código funcional fica assim (a novidade são as duas linhas no início da função):

```
const button = document.getElementById("calculate-service-fee");

button.addEventListener("click", function () {
  // novas linhas: não usamos mais prompt
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");

  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
});
```

Note que a variável `input` poderia ser global, sem diferença na funcionalidade do código.
Mas o acesso a `input.value` deve ser feito *dentro* da função.
Se esse acesso for feito no escopo global, o resultado será uma string vazia `""` (por quê ?)

A propriedade `value` é específica de algumas tags como `<input>`, `<textarea>` e `<select>`.
Não existe essa propriedade em outros elementos como `<p>`, `<div>`, `<h1>`, etc.

# Outros tipos de input

## Introdução

Existem várias tags HTML para colher dados.

Já vimos o `<input>` básico.

Essa tag tem várias opções para o tipo de dado que queremos colher.

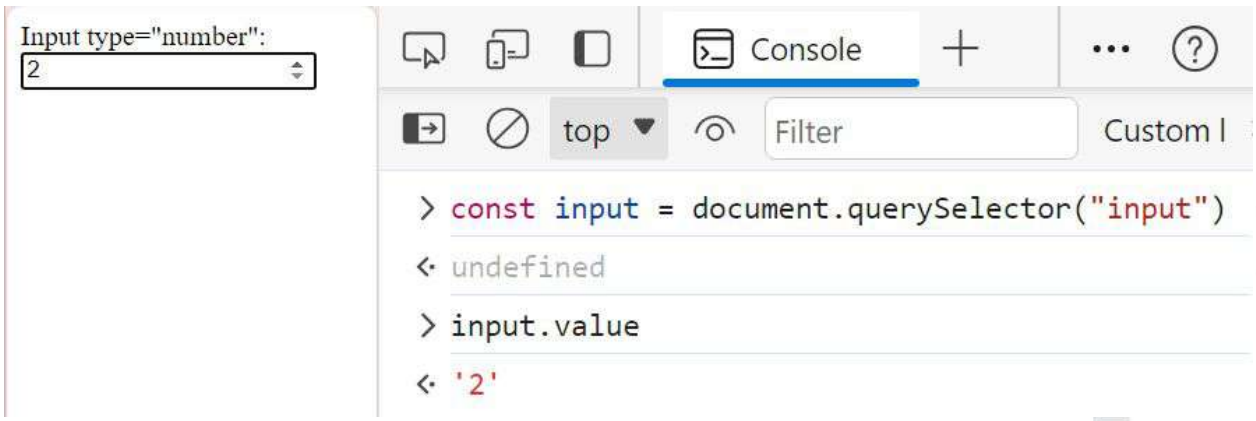
Ao usar a tag `<input>`, o navegador entende como `<input type="text">`, que é um input textual (aceita qualquer texto).

Veremos agora outros tipos de `<input>` e outras tags para colher dados.

## `<input type="number">`

O usuário só consegue digitar números.

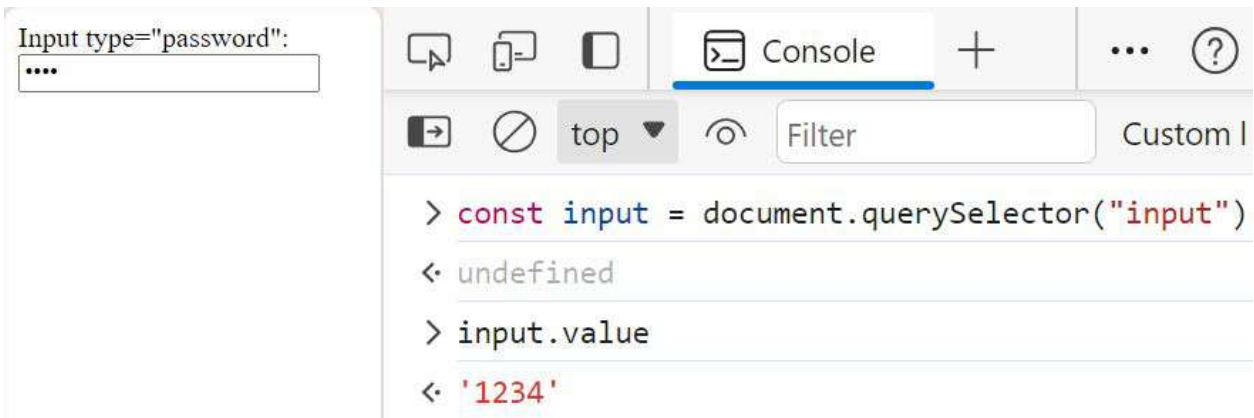
Mesmo assim, o `input.value` no javascript ainda é uma string:




## `<input type="password">`

Usado para senhas.

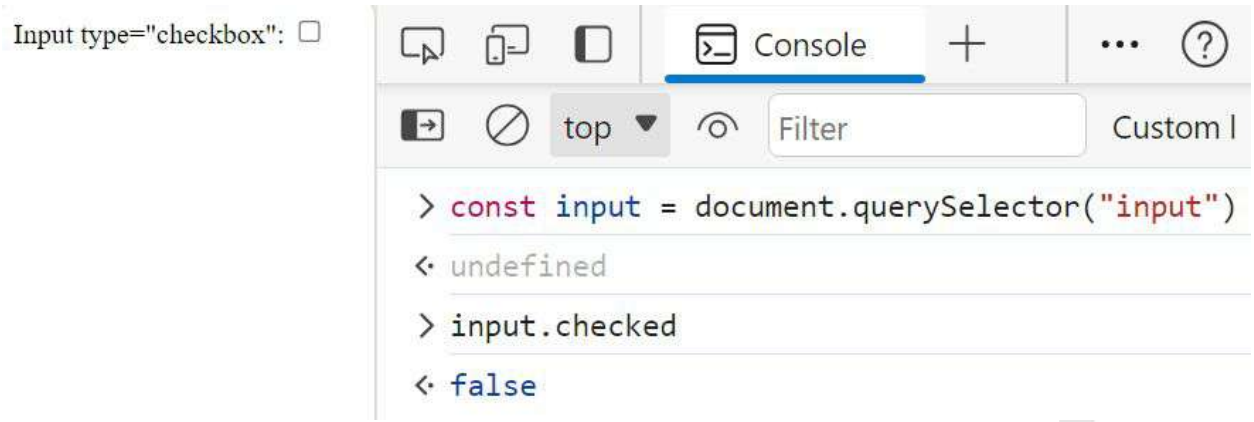
Igual ao `<input type="text">`, mas o texto digitado não é visível:



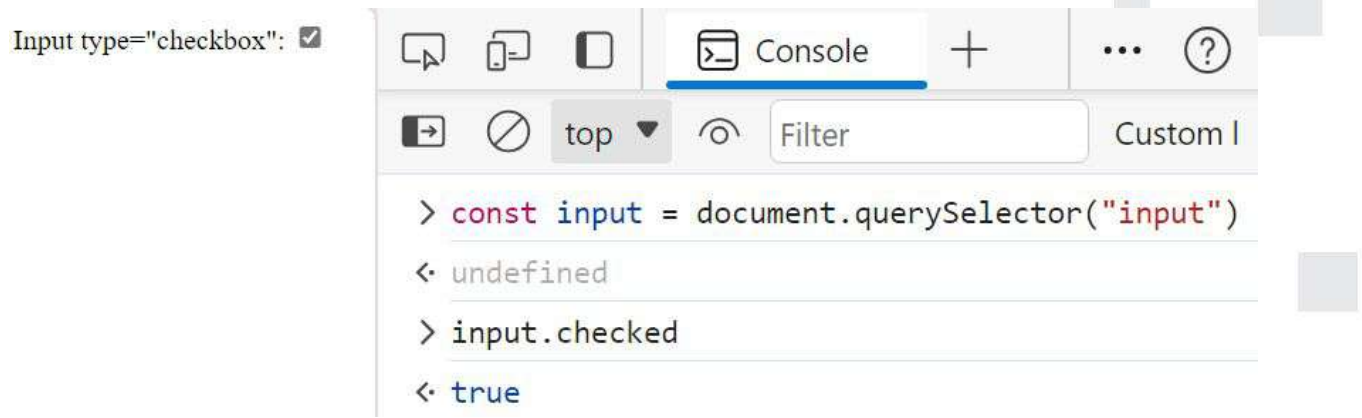
# Outros tipos de input

## `<input type="checkbox">`

É uma caixinha de marcar e desmarcar.
Usado para perguntas de sim ou não, como *"Deseja permanecer conectado ?"*
O valor desse tipo de input é lido com a propriedade `checked`, não `value`.
Se o input estiver desmarcado, o resultado será **`false`** (tipo de dados booleano, não é uma string):



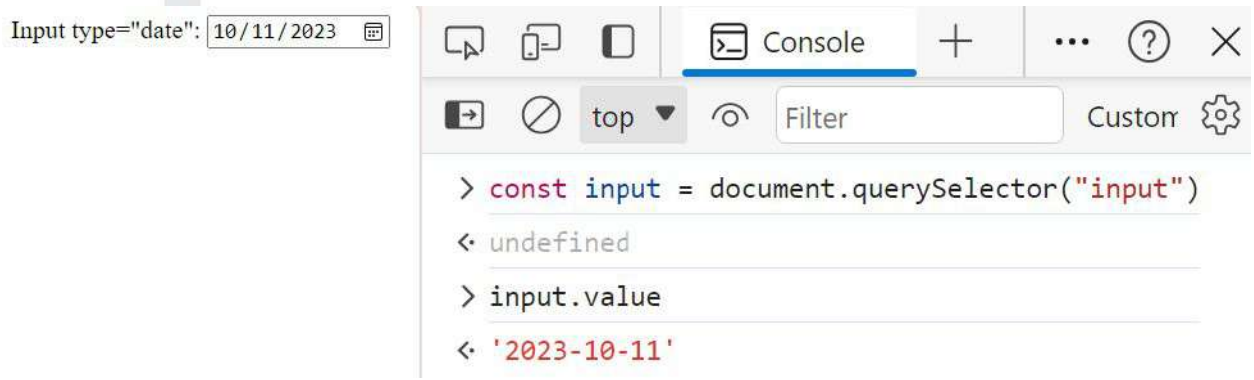

Já se estiver marcado, o resultado será **`true`**:



# Outros tipos de input

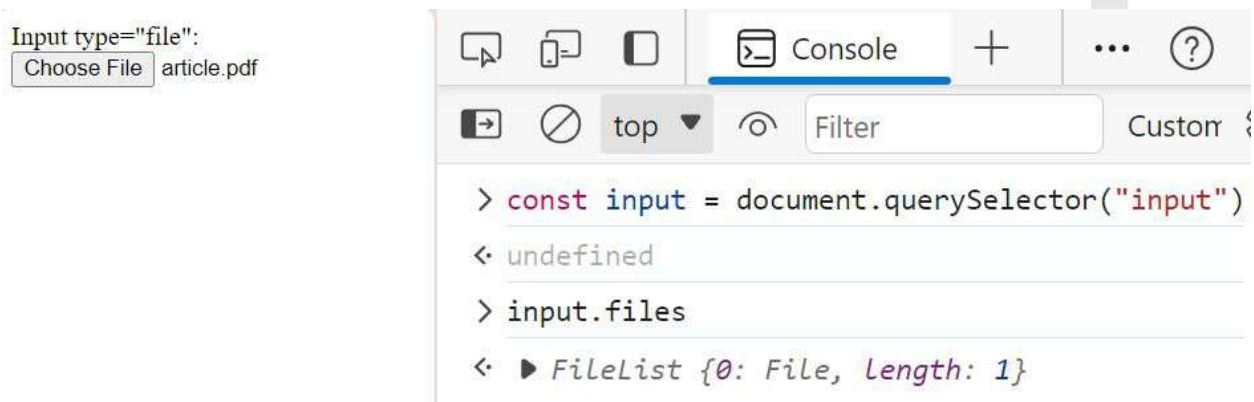
## `<input type="date">`

Usado para selecionar um dia do ano.
O dia escolhido é lido no javascript com `input.value`, que será uma string ano-mês-dia:




## `<input type="file">`

Usado para selecionar um ou mais arquivos.
Abre uma janela para o usuário selecionar arquivos do computador.
Os arquivos escolhidos são acessados no javascript com a propriedade `input.files`:




Mas essa propriedade é complexa: é uma lista de objetos.
Por enquanto fica como curiosidade.

# Outros tipos de input

## Outros tipos de `<input>`

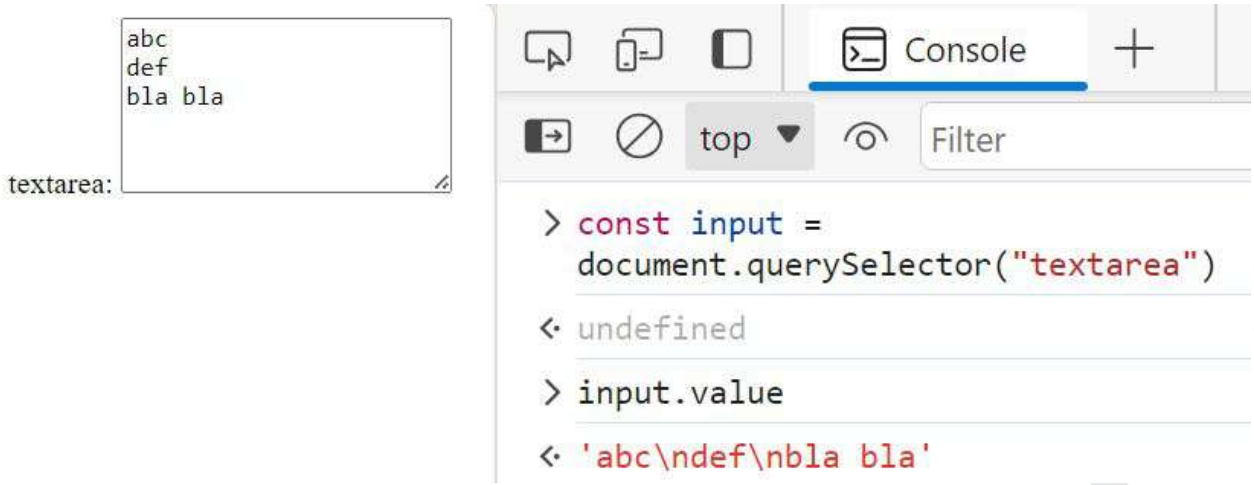
Existem outros tipos de `<input>`, como *reset*, *radio*, *color*, *datetime-local*, *month*, *range*, etc.
Você pode ver mais na referência da MDN sobre input.

## `<textarea>`

Usado para digitar texto com várias linhas (o `<input>` não tem quebra de linha).
No HTML, é usado assim:

- `<textarea></textarea>`

Para ler o conteúdo escrito pelo usuário, o javascript usa a propriedade `value` igual ao `<input>`:



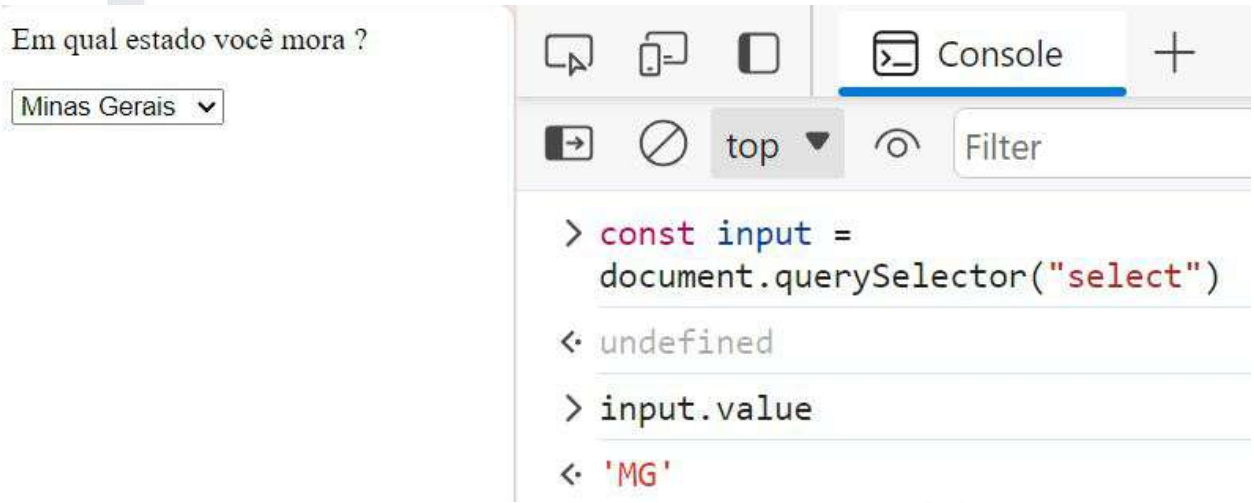

## `<select>`

Usado para selecionar entre opções pré-definidas.
No HTML, é usado assim:

```
<p>Em qual estado você mora ?</p>

<select>
  <option value="SP">São Paulo</option>
  <option value="RJ">Rio de Janeiro</option>
  <option value="MG">Minas Gerais</option>
  <option value="PR">Paraná</option>
  <option value="other">Outro</option>
</select>
```

# Outros tipos de input

O valor escolhido é lido no javascript com `input.value`, que será uma string:




Note que o `input.value` é o que está escrito no atributo `value="MG"` do `<option>` escolhido no HTML. O texto por extenso "São Paulo", "Rio de Janeiro" (etc.) não importa.

# Comando if/else e biblioteca Math

## Sumário

# Comando if/else

## Introdução

Vamos continuar a página web do Pet Shop, onde tínhamos um formulário para calcular o preço do serviço de acordo com o peso do animal (R$10,00 fixos mais R$2,00 por quilograma de peso).

O HTML e javascript eram:

```
<!-- index.html -->
<html>
  <head>
    <title>Pet Shop</title>
  </head>

  <body>
    <h1>Serviços</h1>

    <ul>
      <li>Rações Premium</li>
      <li>Banho e Tosa</li>
      <li>Adestramento</li>
    </ul>

    <h1>Taxa de serviço</h1>

    Peso do animal (kg) <input id="weight-input" />
    <button id="calculate-service-fee">Calcular</button>
    <p id="service-fee"></p>
    <script src="index.js"></script>
  </body>
</html>
```

```
// index.js
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  const serviceFee = 10 + animalWeight * 2;

  const paragraph = document.getElementById("service-fee");
  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

const button = document.getElementById("calculate-service-fee");
button.addEventListener("click", calculateFee);
```

A lógica era: o usuário preenche o peso do animal no `<input>` e clica no botão. Isso faz a função `calculateFee` ser executada, e ela exibe num parágrafo a taxa de serviço calculada.

Agora imagine que o Pet Shop inventou um sistema de desconto: para os clientes que possuem cadastro no Pet Shop, o desconto é de 10% sobre o valor total do serviço.

A página terá mais um input, da forma `<input type="checkbox" id="account-checkbox">` , para o usuário marcar se ele tem cadastro ou não.

Quando o usuário clicar no botão, o javascript precisa calcular o preço do serviço considerando o desconto caso o checkbox esteja marcado.

Como fazer isso é o que veremos agora.

# Comando if/else

## Comando `if`

Para calcular o preço do serviço com desconto, basta adicionar uma linha:

```
...

let serviceFee = 10 + animalWeight * 2; // cálculo original
serviceFee *= 0.9; // nova lógica: 10% de desconto

...
```

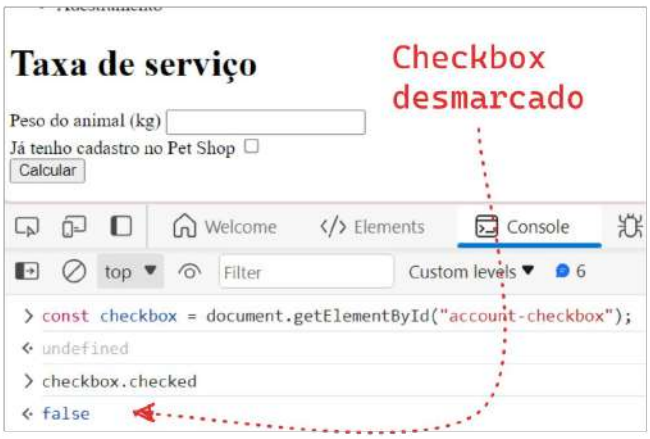
Note que precisamos mudar para `let` porque a segunda linha muda o valor da variável.

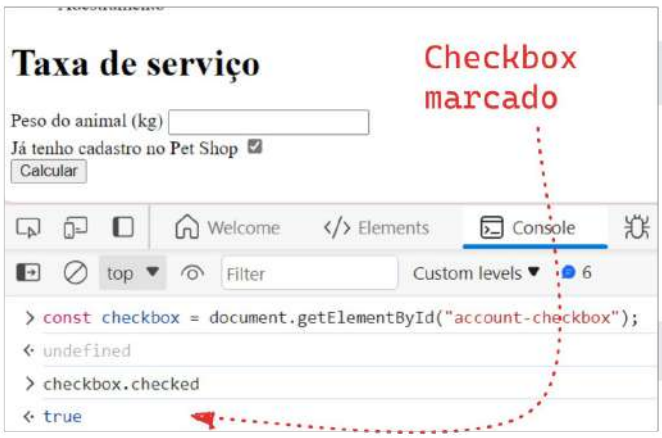
Mas com essa segunda linha, todos os clientes receberão desconto, não importa se ele marcou ou não o input checkbox.

Precisamos fazer com que essa segunda linha de código seja *opcional*: o navegador web deve executar essa linha se o input checkbox estiver marcado, mas não deve executar se não estiver marcado.

Para isso, vamos usar duas ferramentas:

A primeira é que podemos saber se o checkbox está marcado ou não usando a propriedade `checked` do objeto DOM (já vimos isso). Ou seja:




Lembre-se que a propriedade `checked` é um booleano, ou seja, `true` ou `false`.

Já a segunda ferramenta é um novo comando do javascript, que tem a capacidade de, baseado numa *condição*, decidir se certas linhas de código devem ser *executadas* ou *puladas*.

Precisamos desse comando porque nossa *condição* é o checkbox estar marcado, e a linha que deve ser executada nesse caso (ou pulada caso contrário) é a linha `serviceFee *= 0.9;`.

# Comando if/else

Trata-se do comando **if**, cuja sintaxe é a seguinte:

```
if (《Expressão》) {
    《Comando》;
    《Comando》;
    ...
}
```

Na lacuna da 《Expressão》, você deve colocar algo que resulte num booleano (**true** ou **false**).

Essa expressão é chamada **Condição** do **if**. Os parênteses fazem parte do comando, eles não têm nenhuma relação com funções, o comando **if** não é uma função.

Se o resultado dessa expressão for **true**, todos os comandos que estão entre o abre-chaves e o fecha-chaves serão executados (um de cada vez, de cima para baixo, como usual).

Esses comandos são chamados **Corpo** do **if**, ou **Bloco** do **if**. Podem ser quaisquer comandos que você já conhece.

Caso contrário (i.e. a condição é **false**), os comandos do corpo serão pulados como se não existissem.

No nosso caso, preencheremos a lacuna da 《Expressão》 com `checkbox.checked`, porque sabemos que essa propriedade tem ou valor **true** ou valor **false** (conforme o que o **if** precisa).

No corpo do **if**, colocaremos o comando `serviceFee *= 0.9;`.

Portanto, se `checkbox.checked` tiver valor **true**, a linha de desconto será executada. Caso contrário, não será executada. Exatamente como queremos.

O javascript completo fica assim:

```
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  let serviceFee = 10 + animalWeight * 2;

  const checkbox = document.getElementById("account-checkbox");

  // "if" significa "se" (de condição) em inglês.
  // Então esse comando pode ser lido assim:
  // Se (checkbox está marcado) {
  //   dar desconto;
  // }
  if (checkbox.checked) {
    serviceFee *= 0.9;
  }

  const paragraph = document.getElementById("service-fee");
  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

const button = document.getElementById("calculate-service-fee");
button.addEventListener("click", calculateFee);
```

# Comando if/else

Você pode testar para garantir que funciona:



**Múltiplos comandos no corpo do `if`**

Dentro das "chaves" do `if` (corpo/bloco), você pode colocar mais de um comando se quiser/precisar.

Por exemplo:

```
if (《condição》) {
  alert("Teste 01");
  alert("Teste 02");
}
```

Se a condição do `if` for `true`, os dois `alert` serão executados.

Se a condição for `false`, os dois `alert` serão pulados como se não existissem.

Caso você tenha um único comando para executar dentro do bloco do `if`, as chaves se tornam opcionais.

Por exemplo:

```
if (《condição》)
  alert("único comando aqui");
```

Mas sugerimos usar sempre as chaves, para deixar mais visualmente claro que o código está dentro do bloco do `if`.

# Comando if/else

## Operador de comparação “Menor que” (<)

O cenário do checkbox foi “fácil” porque a propriedade `checked` é um booleano (`true` ou `false`), exatamente como o `if` precisa.

Mas e se a condição que você quer expressar não for um booleano ?

Por exemplo, vamos mudar a regra de precificação do Pet Shop:

Em vez de o desconto ser baseado no cadastro do usuário, ele será baseado na idade do animal, colhido com um `<input id="age-input">`.

Se o animal tiver menos de 5 anos de idade, o cliente ganha 10% de desconto. Se tiver 5 anos ou mais, não há desconto.

E não tem mais o checkbox de cadastro.

Primeiro, precisamos ler a idade no javascript:

```
...

const ageInput = document.getElementById("age-input");
const age = Number(ageInput.value);

...
```

Agora vem a novidade: `age` não é um booleano, então não podemos colocar diretamente `if(age)` (não faz sentido).

Mas podemos fazer um *cálculo que resulta num booleano*.

Para isso, usamos o operador `<` (“menor que”), que é classificado como **Operador de Comparação**, porque ele compara dois números (veja abaixo).

Veja os testes no Console com duas idades diferentes preenchidas no input:


# Comando if/else

O comando `age < 5` é um *cálculo* (Expressão) assim como `10 + 3` e `10 * 5` são cálculos (Expressões).
Mas não é um *cálculo aritmético*, é um *cálculo lógico*.
Num *cálculo aritmético*, o resultado é um número. Já num *cálculo lógico*, o resultado é um booleano.
Na imagem, o teste da esquerda tem `age` igual a `3`. Logo o cálculo `age < 5` é feito assim:

1. `age` vale `3`
2. Logo o cálculo é `3 < 5`. Leia isso como uma pergunta: "3 é menor que 5 ?"
3. A resposta é "sim"/"verdade". Então o resultado do cálculo é **`true`**, como você vê no Console.

Na imagem da direita, o passo a passo é análogo: `age < 5` ⇒ `10 < 5` ⇒ **`false`**.
O sinal `<` é chamado de *operador* porque ele tem papel similar aos operadores `+`, `-`, `*` etc.
Afinal, assim como o `+` pega dois números e dá um resultado, como `10 + 3` com resultado `13`, o operador `<` pega dois números e dá um resultado, como `3 < 5` com resultado **`true`**.
Mas os operadores aritméticos `+`, `-`, `*` (etc.) dão resultado numérico, enquanto o operador `<` dá um resultado booleano (**`true`** ou **`false`**).
Com isso, fica da seguinte forma o código javascript que resolve o problema de dar desconto somente se o animal tem menos que 5 anos:

```
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  let serviceFee = 10 + animalWeight * 2;

  const ageInput = document.getElementById("age-input");

  const age = Number(ageInput.value);

  if (age < 5) {
    serviceFee *= 0.9;
  }

  const paragraph = document.getElementById("service-fee");
  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

const button = document.getElementById("calculate-service-fee");
button.addEventListener("click", calculateFee);
```

# Comando if/else

## Outros operadores de comparação

Existem vários operadores de comparação, além do `<`.

Todos eles pegam dois números e fazem alguma comparação entre eles, dando como resultado um booleano (`true` ou `false`).

O que muda é *qual comparação* cada operador realiza.

Todos devem ser lidos como perguntas: *"o número da esquerda é menor/maior/igual ao número da direita ? sim ou não ?"*

Seguem todos os operadores de comparação do javascript com exemplos:

- `<` : Menor que: O número da esquerda é menor que o da direita ?
  - `3 < 4` ⇒ `true`
  - `4 < 4` ⇒ `false`
  - `5 > 4` ⇒ `false`
- `>` : Maior que: O número da esquerda é maior que o da direita ?
  - `3 > 4` ⇒ `false`
  - `4 > 4` ⇒ `false`
  - `5 > 4` ⇒ `true`
- `<=` : Menor ou igual: O número da esquerda é menor ou igual ao da direita ?
  - `3 <= 4` ⇒ `true`, porque 3 é menor que 4
  - `4 <= 4` ⇒ `true`, porque 4 é igual a 4
  - `5 <= 4` ⇒ `false`, porque 5 não é nem menor nem igual a 4
- `>=` : Maior ou igual: O número da esquerda é maior ou igual ao da direita ?
  - `3 >= 4` ⇒ `false`, porque 3 não é nem maior nem igual a 4
  - `4 >= 4` ⇒ `true`, porque 4 é igual a 4
  - `5 >= 4` ⇒ `true`, porque 5 é maior que 4
- `==` ou `===` : Igual: O número da esquerda é igual ao da direita ?
  - `3 === 4` ⇒ `false`
  - `4 === 4` ⇒ `true`
  - `5 === 4` ⇒ `false`
- `!=` ou `!==` : Diferente: O número da esquerda é diferente do da direita ?
  - `3 !== 4` ⇒ `true`
  - `4 !== 4` ⇒ `false`
  - `5 !== 4` ⇒ `true`

Note que os operadores de "Igual" e "Diferente" têm duas versões cada um: `==`/ `===` e `!=`/`!==`.
Falaremos sobre a diferença mais para frente.

# Comando if/else

Se algum dos dois números sendo comparados for `NaN` (ou os dois números forem `NaN`), qualquer comparação dará resultado `false`.

Por exemplo:

- `NaN < 10` ⇒ `false`
- `NaN > 10` ⇒ `false`
- `NaN === 10` ⇒ `false`
- `NaN < NaN` ⇒ `false`
- `NaN > NaN` ⇒ `false`
- `NaN === NaN` ⇒ `false`

Talvez o mais "estranho" seja o último caso: `NaN` *não é igual a* `NaN` !

## Comparação entre strings

Os operadores de comparação podem também trabalhar com strings, em vez de números.

Por exemplo, abaixo segue um programa que pergunta ao usuário qual a senha.

Se ele acertar a senha (que é a string "SEGREDO"), aparecerá uma mensagem de sucesso. Se ele errar, não aparecerá nada.

```
const password = prompt("Digite sua senha");

// se a string na variável `password` for igual à string "PASSWORD" (todos os
caracteres iguais, sem nenhum caractere a mais ou a menos):
if (password === "SEGREDO") {
  alert("Você acertou a senha");
}
```

Note que, quando dizemos "trabalhar com strings", queremos dizer que os dois lados da comparação são strings.

No exemplo acima, a variável `password` é uma string (pois é o resultado do prompt), e do outro lado do `===` há a string `"SEGREDO"`. Ou seja, são strings dos dois lados.

Na próxima seção falaremos sobre situações onde os tipos de dados são diferentes (por exemplo comparar uma string com um número).

Por enquanto, continuando no cenário onde os dois lados da comparação são strings, saiba que os operadores `<` e `>` também funcionam.

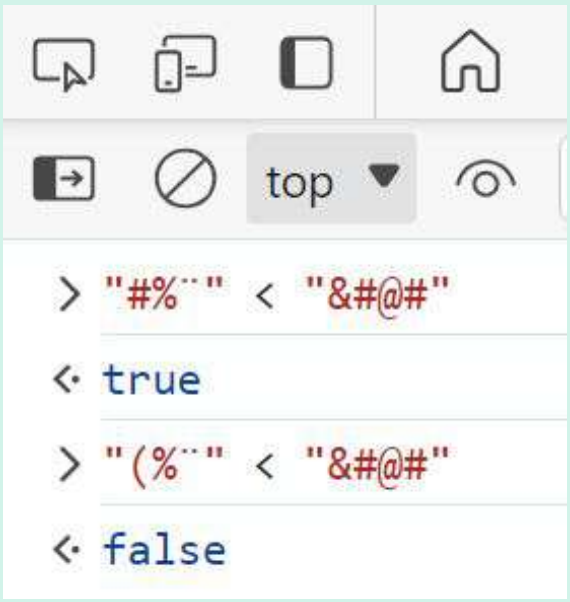
# Comando if/else

Mas < para strings significa: *"A string esquerda aparece antes da string direita na ordem alfabética ?"*
Por exemplo:

- `"analista" < "bob"` ⇒ **`true`**
  - porque na ordem alfabética (ordem do dicionário), "analista" aparece antes de "bob"
- `"analista" < "analista"` ⇒ **`false`**
  - porque as strings são idênticas, elas aparecem "no mesmo lugar" na ordem alfabética
- `"carlos" < "bob"` ⇒ **`false`**
  - porque "carlos" aparece depois de "bob" na ordem alfabética, não antes

Já o > significa o contrário: *"A string esquerda aparece depois da string direita na ordem alfabética ?"*
Porém é muito raro precisarmos de comparações < ou > para strings, então considere isso como uma curiosidade.

Na verdade a comparação < e > é mais complicada do que simplesmente "ordem alfabética".
Afinal ela funciona também para strings com caracteres que não são letras:




A verdade sobre as comparações < e > é que elas se baseiam em "code points": cada caractere tem um código numérico, e a comparação se baseia nesses códigos.
Mas é tão raro precisarmos de comparações < ou > entre strings que vamos deixar isso como curiosidade.
Se quiser, você pode começar a ler sobre Unicode (que é o nome da tabela de códigos numéricos para caracteres) na MDN.

# Comando if/else

## Operadores lógicos

Vamos mudar de novo a regra de desconto do Pet Shop:

Haverá desconto de 10% se o cliente tiver cadastro no Pet Shop (`<input type="checkbox" id="account-checkbox">`) e o animal tiver idade inferior a 5 anos (`<input id="age-input">`).

Para ganhar desconto, o cliente tem que satisfazer *as duas condições*.

Se tiver cadastro mas o animal for velho (> 5 anos), não há desconto. Se o animal for jovem mas o cliente não tiver cadastro, não há desconto.

Ou seja, não basta uma das duas coisas, precisa de ambas.

Uma opção seria usar dois `if`, um dentro do outro:

```
...

const accountCheckbox = document.getElementById("account-checkbox");
const ageInput = document.getElementById("age-input");

const hasAccount = accountCheckbox.checked; // true ou false
const age = Number(ageInput.value);

if (hasAccount) {
  if (age < 5) {
    // dar desconto aqui
  }
}

...
```

Mas isso não é muito legível.

O javascript tem uma ferramenta que permite expressar a dupla condição usando um único comando `if`.

Trata-se dos **Operadores Lógicos**.

Nesse caso, precisamos do operador `&&` (lido como "e", de "isso *e* aquilo")

Assim como os operadores de comparação (`<` , `>`, `===`, etc.), o operador `&&` também produz um resultado booleano.

Mas os operadores de comparação geralmente são usados para comparar dois números, ou duas strings.

Por outro lado, o operador `&&` *trabalha com dois valores booleanos*.

No nosso caso, os valores booleanos serão:

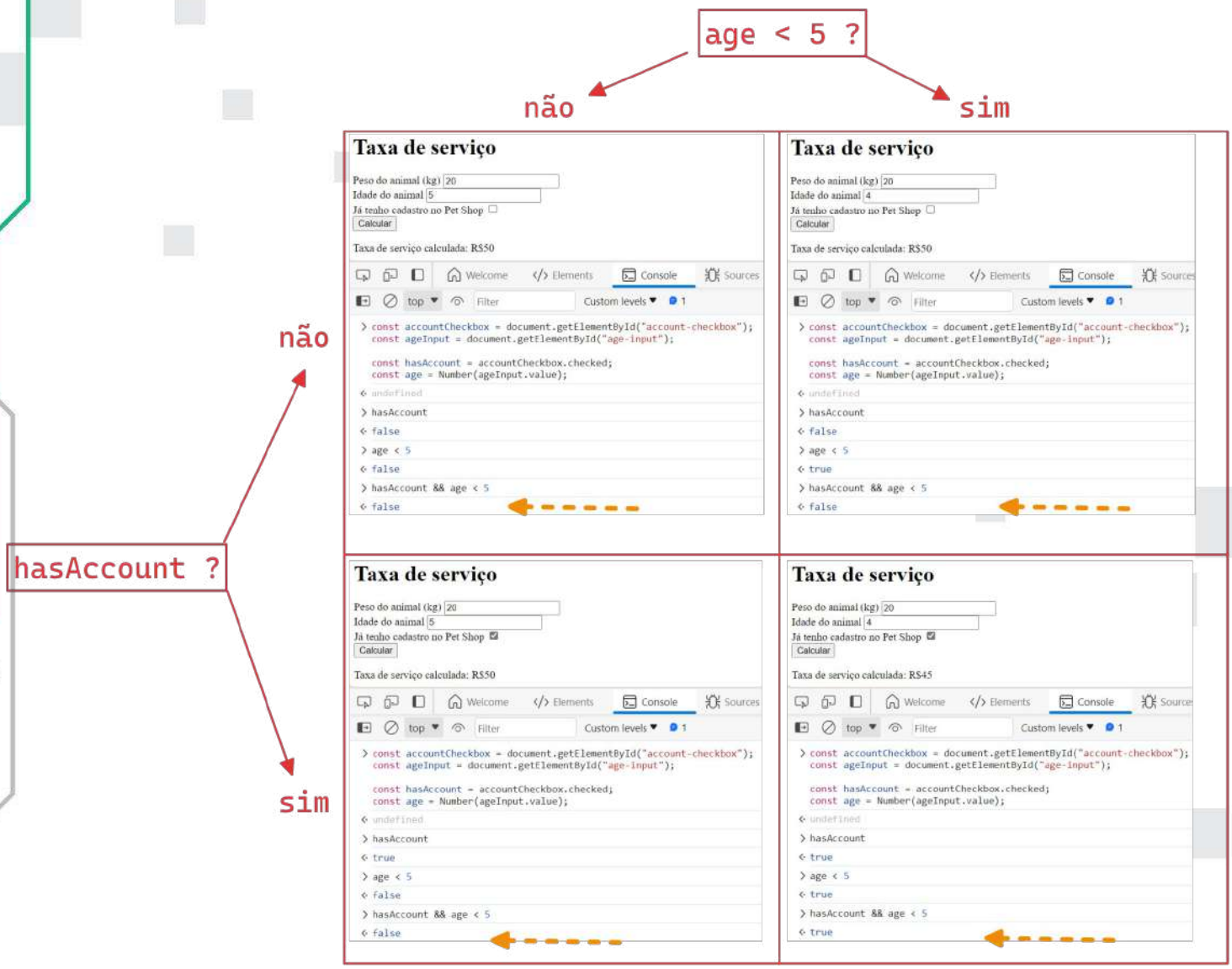
- `hasAccount`
- `age < 5`

O cálculo `hasAccount && age < 5` verifica se ambos os valores booleanos são `true`.

Lido em português, esse cálculo fica "tem cadastro ***e*** a idade é menor que 5".

# Comando if/else

Veja os testes abaixo com esse cálculo (dê zoom se a figura estiver pequena):




Veja que o caso da direita inferior, quando `hasAccount` é **`true`** e `age < 5` é **`true`**, é o único caso em que o cálculo `hasAccount && age < 5` resulta em **`true`**.

Nos outros três casos, esse cálculo dá **`false`**.

Ou seja, o operador `&&` exprime exatamente a ideia que estamos procurando: só pode haver desconto se o usuário tem cadastro ***e*** o animal tem menos de 5 anos, ambas as condições devem ser satisfeitas.

# Comando if/else

Com isso, já podemos escrever o código completo que resolve o problema do desconto:

```
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  let serviceFee = 10 + animalWeight * 2;

  const checkbox = document.getElementById("account-checkbox");
  const hasAccount = checkbox.checked;

  const ageInput = document.getElementById("age-input");
  const age = Number(ageInput.value);

  // se tem cadastro E a idade é menor que 5
  if (hasAccount && age < 5) {
    serviceFee *= 0.9;
  }

  const paragraph = document.getElementById("service-fee");
  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

const button = document.getElementById("calculate-service-fee");
button.addEventListener("click", calculateFee);
```

Para completar, existem dois outros operadores lógicos: `||` e `!`.

O `||`, lido como "ou" (de "isso *ou* aquilo") seria usado se o critério de desconto fosse: "tem cadastro ***ou*** a idade é menor que 5".

Ou seja, não precisa atender aos dois critérios, basta algum deles. Se o cliente tiver cadastro e o animal for velho, ganha desconto. Se o cliente não tiver cadastro mas o animal for novo, ganha desconto.

O único jeito de perder o desconto é se o cliente não tem cadastro e o animal é velho, ou seja, não atendeu nenhuma das duas condições.

O código seria simplesmente `if(hasAccount || age < 5)`, lido como "se tem cadastro ou a idade é menor que 5".

Por último, o operador `!`, lido como "não" (de "não isso"), é usado para inverter um valor booleano.

Ou seja:

- `!true` ⇒ `false`
- `!false` ⇒ `true`

Ele seria usado se o critério de desconto fosse: "não tem cadastro".

Não importa a idade do animal. O que importa é o cadastro: se não houver cadastro, ganha desconto.

Nesse caso, o código seria `if(!hasAccount)`, lido como "se não tem cadastro".

Funciona porque se a variável `hasAccount` for `true`, o `!` vai inverter para `false`, fazendo a condição do `if` ser `false`, logo o desconto não será dado.

E se a variável `hasAccount` for `false`, o `!` vai inverter para `true`, fazendo a condição do `if` ser `true`, logo o desconto será dado.

# Comando if/else

## Comando `else`

Suponha que você quer escrever um programa de login:

O programa pergunta ao usuário qual a senha de login (com `prompt` para facilitar). Se o usuário acertar, o programa mostra a mensagem "Login bem sucedido" (com `alert` mesmo), senão o programa mostra "Login falhou".

Usando somente o comando **`if`**, você precisaria de dois deles:

```
const password = prompt("Digite a senha de login");

// supondo que a senha é fixa, é a string "SEGREDO"
if (password === "SEGREDO") {
  alert("Login bem sucedido");
}

if (password !== "SEGREDO") {
  alert("Login falhou");
}
```

Assim, se o usuário digitar exatamente a string `"SEGREDO"`, o primeiro **`if`** será executado e o segundo **`if`** não (porque se `===` deu **`true`**, o `!==` vai dar **`false`**).

E se o usuário errar a senha, o primeiro **`if`** não será executado e o segundo **`if`** será (porque se o `===` deu **`false`**, o `!==` vai dar **`true`**).

Então funciona.

Mas está repetitivo: no segundo **`if`** você teve que escrever praticamente a mesma condição do primeiro **`if`**, cuidando para inverter a lógica de `===` para `!==`.

O javascript tem um jeito de escrever a condição somente uma vez:

```
const password = prompt("Digite a senha de login");

// se a senha digitada for "SEGREDO"
if (password === "SEGREDO") {
  alert("Login bem sucedido");
}
// caso contrário
else {
  alert("Login falhou");
}
```

O comando **`else`** (tradução literal: "senão", "caso contrário") é o "antônimo" do **`if`**.

No código acima, o navegador web avalia a condição do **`if`** e, se ela for **`true`**, executa o bloco do **`if`**. Ao chegar no **`else`**, ele *pula* o bloco do **`else`**.

Já se a condição do **`if`** for **`false`**, o navegador web pula o bloco do **`if`** (como você já sabe). Ao chegar no **`else`**, ele executa o bloco.

Em outras palavras, o bloco do **`else`**:

- É pulado se o bloco do **`if`** for executado.
- É executado se o bloco do **`if`** for pulado.

O **`else`** é uma ferramenta útil para casos como o do exemplo, onde você quer realizar uma certa ação *X* caso a condição seja verdadeira ou uma outra ação *Y* caso a mesma condição seja falsa.

# Comando if/else

O comando `else` não pode ser usado sozinho:

```
// isso não funciona
else {
  // ...
}
```

Ele somente pode ser usado após um `if`.

## Cadeia de condicionais

Quando exemplificamos o comando `else`, ele foi motivado por uma regra de precificação do Pet Shop que falava sobre dois casos:

- Animais com menos de 5 anos: preço R$20,00
- Caso contrário: preço R$30,00

Mas as regras poderiam ser mais complicadas, incluindo *mais que dois* casos.

Por exemplo:

- Animais com menos de 5 anos: preço R$20,00
- Caso contrário:
  - Se o cliente tem cadastro: preço R$25,00
  - Caso contrário: preço R$30,00

Agora são 3 casos, que devem ser analisados de cima para baixo (i.e. se o animal tem menos de 5 anos, o preço é R$20,00 e não devemos olhar para os outros critérios).

# Comando if/else

No javascript, é possível lidar com esse cenário usando uma cadeia de condicionais:

```
...

let serviceFee;

if (age < 5) {
  serviceFee = 20;
}
else if (hasAccount) {
  serviceFee = 25;
}
else {
  serviceFee = 30;
}

paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
```

Essa construção funciona exatamente como intencionado:

- Primeiro o navegador web analisa a condição `age < 5`.
  Se ela for **`true`**, o preço é R$20,00 e os outros dois blocos são pulados

- Se a condição deu **`false`**, o navegador analisa a segunda condição `hasAccount`.
  Se ela for **`true`**, o preço é R$25,00 e o terceiro bloco é pulado

- Se a segunda condição deu **`false`**, o navegador executa o terceiro bloco (preço de R$30,00).

Em geral, essa construção em cadeia pode ter vários **`else if`** no meio:

```
if (《condição 1》) {
  《comandos no bloco 1》;
}
else if (《condição 2》) {
  《comandos no bloco 2》;
}
else if (《condição 3》) {
  《comandos no bloco 3》;
}
else if (《condição 4》) {
  《comandos no bloco 4》;
}
else {
  《comandos no último bloco》;
}
```

O navegador web vai analisar cada condição da cadeia, de cima para baixo.
Na primeira condição que der **`true`**, o bloco respectivo é executado e os seguintes são pulados.
Mas se nenhuma condição der **`true`**, o último bloco é executado.

# Tópicos adicionais sobre if/else

## Comparação entre tipos de dados diferentes

Você pode comparar tipos de dados diferentes, por exemplo um número e uma string, como `"3" < 5`.

Mas isso não é tão comum, afinal que tipo de situação te motivaria a comparar um número e uma string ?

Se houver alguma comparação desse jeito no seu código, provavelmente é um *bug* 🐛

Por exemplo, voltando ao cenário do Pet Shop (com desconto se a idade do animal é menor que 5 anos) você poderia ter esquecido de converter para número o valor lido do `<input>`:

```
const ageInput = document.getElementById("age-input");

// ops ! Esqueci de converter para número
const age = ageInput.value;

// a variável `age` tem uma string, por exemplo "3".
// Então a comparação será "3" < 5.
// É uma comparação entre um número e uma string
if (age < 5) {
  ...
}

...
```

Então o correto seria lembrar de converter para número: `const age = Number(ageInput.value)`.

Daí a comparação `age < 5` se tornaria uma comparação "normal" entre dois números.

Mas mesmo sendo raro comparar tipos de dados diferentes, pelo menos saiba como o navegador web opera nesses casos, para que você não se confunda caso aconteça uma comparação assim.

Para entender como ocorre uma comparação entre tipos de dados diferentes, existem 3 casos no javascript:

### Caso 1: operadores <, >, <=, >=

Quando dois tipos de dados diferentes são comparados por um desses quatro operadores de comparação, o navegador web vai converter os dois valores para número, usando as mesmas regras da função `Number`.

Com isso, vai se tornar uma comparação entre dois números, que você já sabe como funciona.

Exemplos:

- `"4" < 5` ⇒ `4 < 5` ⇒ **`true`**
- `"" < 5` ⇒ `0 < 5` ⇒ **`true`**
  - pois `Number("")` ⇒ `0`
- **`null`** `< 5` ⇒ `0 < 5` ⇒ **`true`**
  - pois `Number(null)` ⇒ `0`
- **`undefined`** `< 5` ⇒ **`NaN`** `< 5` ⇒ **`false`**
  - pois `Number(undefined)` ⇒ **`NaN`**, e já vimos que qualquer comparação com **`NaN`** resulta **`false`**
- `""` `<` **`null`** ⇒ `0 < 0` ⇒ **`false`**
- `""` `>` **`null`** ⇒ `0 > 0` ⇒ **`false`**

# Tópicos adicionais sobre if/else

## Caso 2: operadores `===` e `!==`

Esses operadores são chamados de *strict* (firme/estrito).

Porque eles nunca fazem nenhuma conversão de tipo implícita (diferente do caso anterior).

O operador `===` só dá resultado **`true`** se os dois valores comparados tiverem o mesmo tipo de dados, e o mesmo valor-em-si.

Se os tipos forem diferentes, o resultado é sempre **`false`**, sem fazer nenhuma conversão.

Exemplos:

- `"4" === 4` ⇒ **`false`**
  - pois os tipos de dados são diferentes (não é feita conversão da string para número)
- `4 === 4` ⇒ **`true`**
  - pois os tipos são iguais (número) e o valor-em-si é igual (número 4)
- `4 === 5` ⇒ **`false`**
  - pois os tipos são iguais, mas os valores-em-si não são iguais (4 e 5)
- `"olá" === "olá"` ⇒ **`true`**
- `"olá" === "tchau"` ⇒ **`false`**

E o operador `!==` é o contrário do `===`. Ou seja, sempre que o `===` resulta **`true`**, o `!==` resultaria **`false`** (e vice-versa).

## Caso 3: operadores `==` e `!=`

Esses operadores são chamados de *loose* (frouxo/flexível).

Quando usados com tipos de dados diferentes, eles fazem conversão implícita de tipos antes de comparar o valor-em-si.

Por exemplo:

- `"4" == 4` ⇒ `4 == 4` ⇒ **`true`**

Mas a regra de conversão de tipos não é a mesma da função `Number`. Nem sempre os operandos serão convertidos para números.

São regras mais complicadas, não nos importa saber exatamente como funciona.

Se quiser, você pode consultar a referência da MDN.

Recomendamos não usar os operadores *loose* `==` e `!=`, que fazem conversão implícita de tipos por você (com regras de conversão que provavelmente você não entende completamente).

Prefira usar os operadores *strict* `===` e `!==`, e faça conversões de tipo explícitas, como converter o texto de um `<input>` num número usando a função `Number`.

# Tópicos adicionais sobre if/else

## Escopo de bloco

Ao usar os comandos `if`/`else`, cuidado com o *lugar* onde você cria variáveis, porque o efeito pode ser diferente do que você imagina.

Por exemplo, suponha que o Pet Shop tenha outra regra de precificação: para animais de menos de 5 anos, o preço é fixo em R$20,00 ; já para os outros animais, o preço é fixo em R$30,00.

Você poderia pensar em codificar da seguinte forma:

```
...

if (age < 5) {
  const serviceFee = 20;
}
else {
  const serviceFee = 30;
}

paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;

...
```

A intenção aqui foi: para animais com menos de 5 anos, declaramos a constante `serviceFee` com o valor 20, senão a declaramos com valor 30.

Mas esse código *não funciona !*

A última linha dará um Erro de Execução, dizendo que "`serviceFee` não está definido".

Ou seja, ele reclama que na última linha do código não existe a constante `serviceFee`.

Isso acontece porque a região dentro das chaves do `if` constitui um escopo próprio.

Significa que variáveis declaradas dentro do bloco do `if` somente podem ser usadas lá dentro (e o mesmo vale para o bloco do `else`).

No exemplo, como a constante `serviceFee` foi declarada dentro do bloco do `if`, ela só existe lá dentro e é apagada assim que o código sai do bloco do `if`.

Já a última linha do código está *fora* do bloco do `if`, portanto nessa linha a constante `serviceFee` não existe mais.

Para corrigir esse erro, você poderia declarar a variável *antes* do bloco do `if` e do `else` (precisaria ser `let` em vez de `const`):

```
...

let serviceFee;

if (age < 5) {
  serviceFee = 20;
}
else {
  serviceFee = 30;
}

paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
```

# Tópicos adicionais sobre if/else

## Validação de input numérico

A maioria dos programas úteis realiza Entrada de Dados.

Por exemplo, o programa de precificação do Pet Shop precisa saber qual o peso do animal (etc.) para calcular o preço do serviço.

E quando há entrada de dados, há a chance de os dados serem inválidos.

Por exemplo, o usuário pode digitar um peso negativo.

Comandos condicionais podem ser usados para validar os dados:

```
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  // se o peso é negativo, mostrar um alerta
  if (animalWeight < 0) {
    alert("Peso não pode ser negativo");
  }
  // caso contrário, fazer os cálculos
  else {
    const serviceFee = 10 + animalWeight * 2;
    const paragraph = document.getElementById("service-fee");
    paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
  }
}

const button = document.getElementById("calculate-service-fee");
button.addEventListener("click", calculateFee);
```

A validação acima lida com peso negativo, mas não lida com peso "vazio" (se o usuário não digitou nada no input de peso).

Se é desejável considerar como inválido um input vazio, vejamos:

Primeiro lembramos que nesse caso o `input.value` é uma string vazia `""`.

A função `Number` (no código) converte a string vazia no número `0`, como você já sabe.

Então podemos evitar a string vazia trocando a condição do **`if`** para **`if`** `(animalWeight <= 0)`.

# Tópicos adicionais sobre if/else

Outra maneira de fazer a mesma coisa é checar se `input.value` é a string vazia sem converter para número:

```
...

const input = document.getElementById("weight-input");
const animalWeight = Number(input.value);

// se o peso não foi preenchido ou é negativo, mostrar um alerta
if (input.value === "" || animalWeight < 0) {
  alert("Peso não pode ser vazio/negativo");
}

...
```

Por último, se você não está usando `<input type="number">`, o usuário pode digitar um texto não-numérico como `"abcd"` para o peso do animal.

Nesse caso, a função `Number` retorna **`NaN`** (como você já sabe).

Para que o **`if`** também detecte esse valor inválido, podemos usar a função `isNaN`.

Ela retorna **`true`** ou **`false`**, por exemplo:

- `isNaN(10)` ⇒ **`false`**
- `isNaN(NaN)` ⇒ **`true`**

Usando essa função, o código de validação fica assim:

```
...

const input = document.getElementById("weight-input");
const animalWeight = Number(input.value);

// se o peso não foi preenchido, ou é negativo, ou não é um número, mostrar um alerta
if (input.value === "" || animalWeight < 0 || isNaN(animalWeight)) {
  alert("Peso não pode ser vazio/negativo");
}

...
```

# Tópicos adicionais sobre if/else

No caso do `NaN`, antes de conhecer a função `isNaN`, você poderia ter pensado no seguinte código:

- `if(animalWeight === NaN)`

Mas isso não funciona !

A comparação `animalWeight === NaN` sempre terá resultado `false` , até mesmo quando o valor de `animalWeight` for `NaN` ! Ou seja, `NaN === NaN` ⇒ `false` !

Isso acontece porque, como já vimos nesta aula, comparação `===` que envolve `NaN` sempre resulta `false`.

Se você não lembra disso, reveja a seção sobre Operadores de Comparação.

## Validação de input textual

Além de números, formulários também frequentemente pedem dados textuais, como emails, nomes e senhas.

Nesse caso precisamos usar ferramentas de string para validação.

### 1: Limpeza (pré-processamento)

Um caso de uso bastante comum é "limpar" (formatar corretamente) a string antes de processá-la.

Por exemplo, num campo de nome, espaços em branco no início e no final da string não são relevantes.

Por exemplo se o usuário digitar `"  João da Silva  "` (note os espaços em branco no início e fim, digitados por engano pelo usuário).

Nesse caso podemos limpar a string com o método de string `trim()`:

- `document.querySelector("input").value.trim()`

O `trim` retira os espaços no início e fim:

- `"  João da Silva  "` ⇒ `"João da Silva"`

Outro requerimento comum é converter os caracteres para letra minúscula, sendo útil o método `toLowerCase()`:

- `document.querySelector("input").value.trim().toLowerCase()` primeiro limpa os espaços das beiradas, depois converte para minúsculo.

O passo a passo é:

- `document.querySelector("input").value.trim().toLowerCase()`
- ⇒ `" João da Silva ".trim().toLowerCase()`
- ⇒ `"João da Silva".toLowerCase()`
- ⇒ `"joão da silva"`

# Tópicos adicionais sobre if/else

## 2: Validação de comprimento

Por exemplo para validar que uma senha tem entre 8 e 12 caracteres:

```
function validate() {
  const password = document.querySelector("input").value;

  if (password.length < 8 || password.length > 12) {
    alert("Senha deve ter entre 8 e 12 caracteres");
  }
  else {
    // senha é válida
    ...
  }
}
```

## 3: Restrições de formato

É comum a aplicação exigir que a string tenha um formato específico (quais caracteres são válidos e em qual ordem).

Por exemplo: uma senha que não pode ter outros caracteres além de letras e números.

Outro exemplo: a string deve ter formato de telefone: (XX) XXXX-XXXX.

Ou ainda formato de email, que é mais complicado que telefone (precisa ter o @ no meio, não pode ter espaços, etc.)

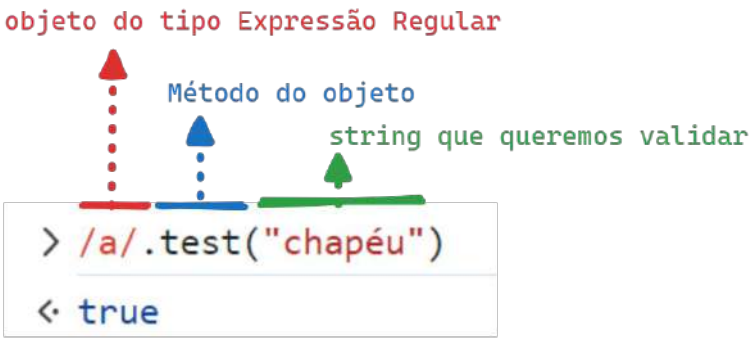
Para esses casos que exigem uma "estrutura" para a string, as linguagens de programação (incluindo javascript) possuem uma ferramenta chamada **Expressão Regular**.

A Expressão Regular é uma ferramenta para validar strings.

Em javascript, uma expressão regular é um tipo especial de objeto.

Ele especifica um formato que a string deve ter, e então usamos o método `test()` do objeto para verificar se a string atende ou não ao formato.

Por exemplo:




Pode parecer estranho, mas `/a/` é um objeto (especial) do tipo Expressão Regular.

As duas barras são somente delimitadores (como se fossem aspas numa string), e dentro das barras está presente a regra que queremos testar.

Nesse caso a regra é: "deve ter uma letra *a*".

Invocamos o método `test` desse objeto e passamos para ele a string a ser testada.

O método retorna um booleano: `true` nesse caso porque a string obedece à regra (`"chapéu"` tem letra *a*).

# Tópicos adicionais sobre if/else

Vamos ver outros exemplos:


A regra é: "deve ter a sequência *at*".

A primeira string não tem essa sequência, por isso `false`.

A segunda string tem a sequência, por isso `true` (note que não precisa necessariamente *começar* com "at", só precisa ter essa sequência em algum lugar dentro da string).

Com isso você pode testar muitas regras.

Basta saber *como* escrever a regra dentro da expressão regular.

Por exemplo vamos ver como fica a seguinte regra: "deve começar com letra *a*, deve ter uma letra *t* no meio, e deve terminar com letra *o*".

A expressão regular para isso é `/^a.*t.*o$/`:

- `^a` significa "começa com *a*" (o `^` significa "inicia com")
- `.` significa "qualquer caractere". E `*` significa "zero ou mais vezes". Juntos, significa "qualquer caractere zero ou mais vezes".
- `o$` significa "termina com *o*" (o `$` significa "termina com")

Então lida por inteiro, essa expressão é regular fica assim: "começa com *a*, depois vem qualquer caractere zero ou mais vezes, depois vem a letra *t*, depois vem qualquer caractere zero ou mais vezes, depois termina com *o*."

Veja alguns testes:

```
> /^a.*t.*o$/.test("patins")
< false
> /^a.*t.*o$/.test("patinação")
< false
> /^a.*t.*o$/.test("apartamento")
< true
> /^a.*t.*o$/.test("departamento")
< false
> /^a.*t.*o$/.test("ato")
< true
> /^a.*t.*o$/.test("a, tem aqui um pato")
< true
```

# Tópicos adicionais sobre if/else

Não vamos entrar em detalhes (qual regra para validar um número de telefone ? etc.), você pode ver mais nas referências:

- FreeCodeCamp: Tutorial sobre Expressão Regular
- MDN: Uso de Expressão Regular no javascript
- regexr.com: site para interpretar o significado de uma expressão regular

## Comando `return`

Uma desvantagem de usar **`if + else`** para fazer validação é que o código "útil" (cálculo de preço) fica todo dentro do **`else`**.

Isso torna o código mais difícil de entender, porque ele está indentado dentro do bloco do **`else`**.

Em outras palavras, quanto mais níveis de indentação, mais bagunçado fica.

Uma alternativa é usar o comando **`return`**, cujo efeito é *interromper* a execução da função:

```
function calculateFee() {
  const input = document.getElementById("weight-input");
  const animalWeight = Number(input.value);

  // se o peso é negativo, mostrar um alerta
  if (input.value === "" || animalWeight < 0) {
    alert("Peso não pode ser vazio/negativo");
    // o `return` interrompe a execução da função `calculateFee`.
    // Portanto se o navegador web entrar no bloco do `if`,
    // o return impede de continuar para o código posterior.
    return;
  }

  // caso contrário, fazer os cálculos
  const serviceFee = 10 + animalWeight * 2;
  const paragraph = document.getElementById("service-fee");
  paragraph.innerText = `Taxa de serviço calculada: R$${serviceFee}`;
}

...
```

O **`return`** serve para evitar que o código "útil" seja executado caso tenhamos detectado input inválido.

Usando essa técnica, o comando **`if`** atua como um "cão de guarda": se entrar no **`if`**, o comando **`return`** mata a função, impedindo de executar o código depois do **`if`**.

Com isso, o código depois do **`if`** não precisa mais ser colocado dentro de um comando **`else`**.

Fica mais legível.

O comando **`return`** não está relacionado diretamente com **`if`**/`else`, mas sim com a função.

O **`return`** só pode ser usado dentro de uma função, porque o efeito dele é abortar a execução da função.

Nós aproveitamos esse comando aqui porque, quando usado dentro do **`if`**, ele torna desnecessário o **`else`**, simplificando o código.

# Tópicos adicionais sobre if/else

## Variáveis para melhorar a legibilidade das condições

A condição de um comando `if` pode ficar complicada de entender.

Por exemplo suponha um cenário onde temos algumas variáveis contendo informações sobre um usuário do sistema (não importa de onde elas vieram).

E queremos mostrar um alerta dizendo se o usuário tem ou não acesso a uma certa funcionalidade do sistema:

```
...
// explicação das variáveis:
// - userType: "admin" ou "user" (ou é administrador ou é usuário normal)
// - accountStatus: "active" ou "suspended" ou "deleted"
// - hasPaidSubscription: true se o usuário assina o serviço, false caso contrário
// - daysSinceLastLogin: número de dias desde o último login do usuário

if (
  (userType === "admin" && accountStatus === "active") ||
  (userType === "user" &&
    accountStatus === "active" &&
    hasPaidSubscription &&
    daysSinceLastLogin < 30)
) {
  alert("Acesso concedido");
} else {
  alert("Acesso negado");
}
```

Você consegue entender quais são as regras para permitir o acesso ?

Fazendo algum esforço, dá para ver que o `||` define duas possibilidades de acesso:

1. É um administrador que tem conta ativa (status é "active").
2. Ou é um usuário comum que tem assinatura e fez login recentemente (nos últimos 30 dias).

Não é tão difícil de entender, mas leva alguns segundos.

Isso pode ser melhorado, porque código complexo só atrapalha quem está lendo (precisa gastar mais esforço para entender, no fim do dia faz diferença).

# Tópicos adicionais sobre if/else

Veja como fazer melhor:

```
...

const isAdminWithActiveAccount = userType === "admin" && accountStatus === "active";

const isUserWithSubscription = userType === "user" && accountStatus === "active" &&
hasPaidSubscription;

const isRecentlyLoggedIn = daysSinceLastLogin < 30;

if (isAdminWithActiveAccount || (isUserWithSubscription && isRecentlyLoggedIn)) {
  alert("Acesso concedido");
}
else {
  alert("Acesso negado");
}
```

O segredo é decompor a condição em etapas menores e armazenar os resultados dessas etapas em constantes com nomes descritivos.

Aí usar essas constantes dentro da condição do `if`.

Com isso o comando `if` fica muito fácil de ler (traduzindo): *"se é um administrador com conta ativa ou é um usuário assinante que fez login recentemente"*.

# Tópicos adicionais sobre if/else

## Operador ternário

Considere a seguinte situação hipotética em um programa de cálculo de imposto de renda (simplificado):

- Se a renda é de até R$1900, não há imposto.
- Caso contrário o imposto é 7.5% da renda.

Um programa que calcula o imposto poderia ser assim:

```
const income = Number(prompt("Renda (em R$): "));

let tax; // imposto
if (income <= 1900) {
  tax = 0;
} else {
  tax = 0.075 * income;
}

alert(`Valor devido de imposto: ${tax}`);
```

Mas como o comando `if/else` está sendo usado com o único propósito de atribuir o valor para a variável `tax`, você poderia pensar que o seguinte código também funcionaria:

```
const income = Number(prompt("Renda (em R$): "));

const tax = if (income <= 1900) {
  0;
} else {
  0.075 * income;
}

alert(`Valor devido de imposto: ${tax}`);
```

Ou seja, estamos tentando escrever um código que corresponde ao seguinte pensamento: *"O imposto é: se a renda é ≤ 1900, então 0, senão 0.075 * renda"*

Porém esse código *não funciona* porque o comando `if/else` *não é uma Expressão*. Ele não tem um "valor de retorno".

Esse código não vai nem executar, é um erro de sintaxe tentar colocar o `if` do lado direito de uma declaração de variável (`const tax = ...`).

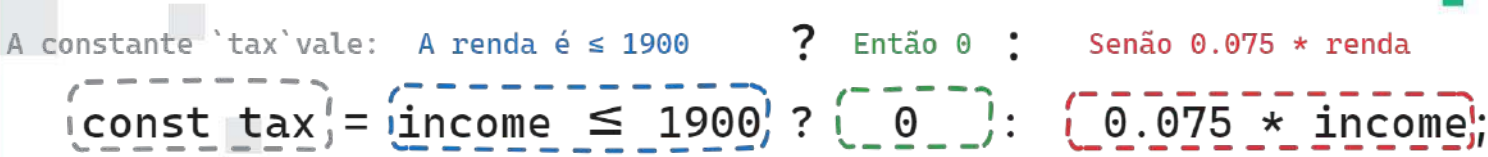
Mas se você entendeu a ideia, ela faz sentido, e tem um jeito de fazer isso no javascript, só que não é usando `if` + `else`, e sim `?` + `:` .

Ou seja, o código que funciona é o seguinte:

```
const income = Number(prompt("Renda (em R$): "));

const tax = income <= 1900 ? 0 : 0.075 * income;

alert(`Valor devido de imposto: ${tax}`);
```

# Tópicos adicionais sobre if/else

Leia da seguinte forma:




Isso é chamado **Operador Ternário**, e ele é uma expressão (tem "valor de retorno").
O mecanismo é:

- Primeiro avalia a condição (azul).
- Se deu `true`, o valor de retorno é a expressão em verde, depois do `?`.
- Se deu `false`, o valor de retorno é a expressão em vermelho, depois do `:`.

Fica bem mais curto que o comando `if/else` original.
Porém é mais limitado: nas lacunas verde e vermelha, você só pode colocar uma Expressão (cálculo, número, string, outro tipo de dados, ou invocação de uma função como `prompt`).
Não dá para declarar uma variável ali, por exemplo.
É possível fazer uma cascata de condicionais usando vários operadores ternários sucessivos.
Por exemplo, se a regra de imposto fosse um pouco mais complicada:

- Se a renda é de até R$1900, não há imposto.
- Caso contrário, se a renda é de até R$2800, o imposto é 7.5% da renda.
- Caso contrário, se a renda é de até R$3700, o imposto é 15% da renda.
- Caso contrário, o imposto é 22.5% da renda.

Pense fazer isso com `if/else`, ficaria bem grande (cascata de condicionais).
Com operador ternário, fica assim:

```
const income = Number(prompt("Renda (em R$): "));

// a quebra de linha não faz diferença,
// é para melhorar a legibilidade
const tax =
  income <= 1900
    ? 0
    : income <= 2800
    ? 0.075 * income
    : income <= 3700
    ? 0.15 * income
    : 0.225 * income;

alert(`Valor devido de imposto: ${tax}`);
```

# Tópicos adicionais sobre if/else

## Truthy e Falsy

Você já sabe que o javascript faz conversões de tipo implícitas quando você usa operações aritméticas, como `+ - * /`.

Nesses casos, ou ele converte os operandos para números, ou para strings (dependendo de algumas regras, que já vimos e não importam agora).

O que talvez você não saiba é que existe um contexto onde o javascript converte para *booleano*.

Esse contexto é a condição de um comando **`if`**.

Por exemplo, considere o programa abaixo:

```
const text = prompt("Digite um texto");

if (text) {
  alert("Seu texto não é a string vazia");
} else {
  alert("Seu texto é a string vazia (ou null)");
}
```

Veja que a condição do **`if`** é a variável `text`, que é uma string (ou **`null`** se o usuário cancelar o prompt), não um booleano.

Isso não gera um erro de execução ! Pelo contrário, o javascript simplesmente converte a string para booleano.

As regras são:

- String vazia `""` converte para **`false`**.
- Todas as outras strings convertem para **`true`**.

Então se o usuário digitar uma string não-vazia como `"hello world !"`, o passo a passo é:

- **`if`**`(text)` ⇒ **`if`**`("hello world !")` ⇒ **`if`**`(true)` ⇒ executa o bloco do **`if`**

Já se o usuário não digitar nada:

- **`if`**`(text)` ⇒ **`if`**`("")` ⇒ **`if`**`(false)` ⇒ executa o bloco do **`else`**

Já se o usuário cancelar o prompt, o `text` vai valer **`null`** (como você sabe), e o **`null`** converte para **`false`**.

Pode parecer estranho esse tipo de **`if`**, mas em javascript é bem comum encontrar código desse jeito.

Por exemplo para validar um input que não pode ser deixado em branco:

```
function validate() {
  const name = document.querySelector("input").value;

  // se `name` for vazio: !name ⇒ !"" ⇒ !false ⇒ true ⇒ executa o if
  if (!name) {
    alert("Seu nome precisa ser preenchido");
    return;
  }

  alert(`Olá, ${name}`);
}
```

# Tópicos adicionais sobre if/else

Valores que convertem para `true` são chamados informalmente de "truthy", e valores que convertem para `false`, de "falsy".

Segue abaixo uma tabela das regras de conversão:

| Valor | Truthy | Falsy |
|---|---|---|
| `null` | | ✅ |
| `undefined` | | ✅ |
| 0 | | ✅ |
| `NaN` | | ✅ |
| Outros números | ✅ | |
| `""` | | ✅ |
| Outras strings | ✅ | |
| Objetos | ✅ | |

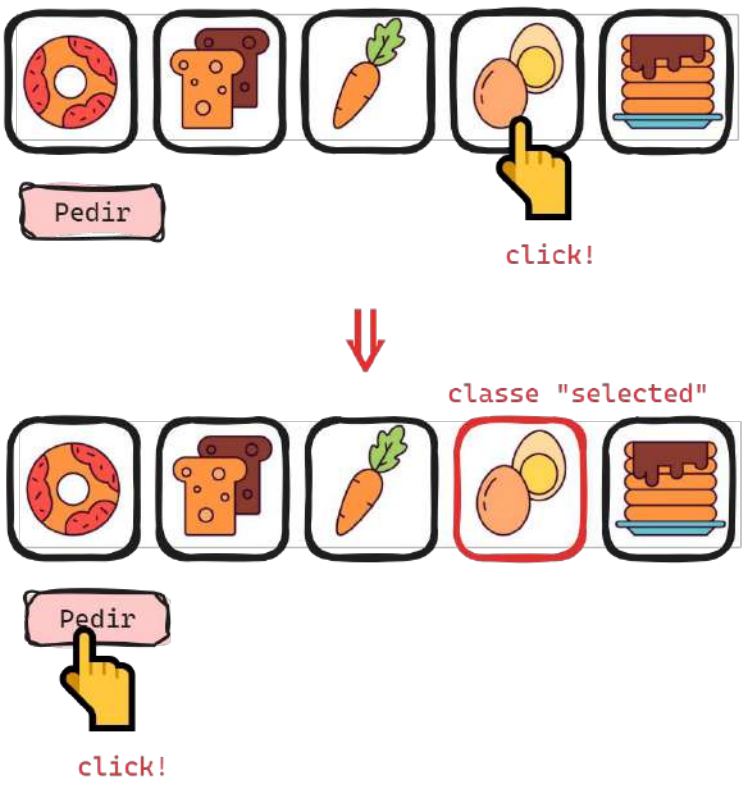
Essa última regra (todos os objetos convertem para `true`) é frequentemente usada para diferenciar entre um objeto e o valor `null` (que converte para `false`).

Por exemplo, imagine um site de delivery, onde o cliente primeiro seleciona um prato depois clica no botão "Pedir".

Nesse site, cada prato fica em uma `<div>` com borda preta.

O usuário precisa primeiro selecionar um prato clicando nele, o que faz o javascript marcá-lo com uma classe CSS "selected" que deixa a borda vermelha.

Depois o cliente clica no botão "Pedir":



# Tópicos adicionais sobre if/else

Nesse momento, o javascript processa o pedido.

A primeira coisa que o javascript precisa fazer é checar se realmente o cliente selecionou algum prato (porque ele poderia não ter selecionado nada e clicado direto no botão).

Isso pode ser feito assim:

```
function onClick() {
  // encontrar o objeto do DOM com classe "selected".
  // Se o usuário não tiver selecionado nada, o
  // querySelector vai retornar null porque
  // não existe no HTML um elemento com classe "selected"
  const selectedDish = document.querySelector(".selected");

  // Se selectedDish é null:
  // - if (!selectedDish) ⇒ if (!null) ⇒ if(!false) ⇒ if(true) ⇒ entra no if
  // Se selectedDish é um objeto:
  // - if (!selectedDish) ⇒ if (!{...}) ⇒ if (!true) ⇒ if(false) ⇒ não entra no if
  if (!selectedDish) {
    alert("Você precisa selecionar um prato");
    return;
  }

  alert("Pedido aceito, seu prato já vai chegar !");
}
```

Usar a conversão implícita pode deixar o código mais curto, e em códigos de outros desenvolvedores você encontrará muito disso.

Mas nunca é necessário, é sempre possível usar booleanos explicitamente.

Por exemplo o último caso do objeto poderia ser: `if(selectedDish !== null)`.

# Biblioteca Math

A biblioteca `Math` não tem relação com comandos condicionais.
Ela é uma coleção de funções para facilitar certos cálculos matemáticos.
O `Math` é um objeto pré-definido, digite no Console:

```
> Math
< ▼ Math {abs: ƒ, acos: ƒ, acosh: ƒ, asin: ƒ, asinh: ƒ, …} i
      E: 2.718281828459045
      LN2: 0.6931471805599453
      LN10: 2.302585092994046
      LOG2E: 1.4426950408889634
      LOG10E: 0.4342944819032518
      PI: 3.141592653589793
      SQRT1_2: 0.7071067811865476
      SQRT2: 1.4142135623730951
    ▶ abs: ƒ abs()
    ▶ acos: ƒ acos()
    ▶ acosh: ƒ acosh()
    ▶ asin: ƒ asin()
    ▶ asinh: ƒ asinh()
    ▶ atan: ƒ atan()
    ▶ atan2: ƒ atan2()
    ▶ atanh: ƒ atanh()
    ▶ cbrt: ƒ cbrt()
    ▶ ceil: ƒ ceil()
    ▶ clz32: ƒ clz32()
    ▶ cos: ƒ cos()
    ▶ cosh: ƒ cosh()
    ▶ exp: ƒ exp()
    ▶ expm1: ƒ expm1()
    ▶ floor: ƒ floor()
    ▶ fround: ƒ fround()
```

Como você pode ver, `Math` é um objeto contendo propriedades e métodos relacionados a cálculos matemáticos.
As propriedades são números famosos como π , e (número de Euler) e √2 (raiz quadrada de 2):

```
> Math.PI
< 3.141592653589793
> Math.E
< 2.718281828459045
> Math.SQRT2
< 1.4142135623730951
```

# Biblioteca Math

Já os métodos são funções matemáticas comuns, por exemplo:

- `Math.abs` é a função módulo, que retorna o valor absoluto de um número (valor sem o sinal):
  - `Math.abs(0)` ⇒ `0`
  - `Math.abs(-3)` ⇒ `3`
  - `Math.abs(3)` ⇒ `3`
- `Math.floor` é a função piso, que arredonda o número para baixo:
  - `Math.floor(3.14)` ⇒ `3`
  - `Math.floor(4.99)` ⇒ `4`
  - `Math.floor(5)` ⇒ `5`
- `Math.random` é uma função que retorna um número aleatório diferente (entre 0 e 1) a cada vez que é chamada:

```
> Math.random()
< 0.9194675526350076
> Math.random()
< 0.4384506578785765
> Math.random()
< 0.11977154293959114
> Math.random()
< 0.13388279677206838
```