



Digital
College

FORMAÇÃO EM
**DESENVOLVEDOR
FULL STACK**

UNIDADE 1:

CONCEITOS E FUNDAMENTOS WEB

MÓDULO 4:

> LÓGICA DE PROGRAMAÇÃO EM JAVASCRIPT >

1.1 Tag Script

O elemento HTML **<script>** é usado para incluir ou referenciar um script executável.

• O elemento

Scripts inseridos dinamicamente (usando **document.createElement**) são executados assincronamente por padrão, então para torná-lo uma execução síncrona (ex. executar scripts na ordem que eles foram carregados) atribua `async=false`.

• Crossorigin

Elementos script passam o mínimo de informação para `window.onerror` em scripts que não passem na checagem do CORS. Para permitir logs de erro para sites que usam domínios diferentes para arquivos estáticos, use esse atributo.

• Defer

Esse atributo Booleano é usado para indicar ao navegador que o script deve ser executado depois que o documento tenha sido parcelado, mas antes de disparar o evento `DOMContentLoaded` (en-US)

Scripts com o atributo defer vão impedir que o evento `DOMContentLoaded` seja disparado até que o script seja carregado e tenha terminado de ser avaliado.

• Integrity

Contém metadados embutidos que um agente de usuário pode usar para verificar se um recurso buscado foi entregue sem manipulação inesperada.

Consulte Integridade do Sub-recurso.

• Nomodule

Esse atributo booleano é definido para indicar que o script não deve ser executado em navegadores que suportam módulos ES6 — na verdade, isso pode ser usado para servir scripts de fallback para navegadores mais antigos que não suportam **código JavaScript modular**.

• Src

Este atributo especifica o URI de um script externo; isso pode ser usado como uma alternativa para incorporar um script diretamente em um documento. Se um script elemento tem um src atributo especificado, ele não deve ter um script embutido em suas tags.

• Text

Assim como o `textContent` atributo, este atributo define o conteúdo de texto do elemento. Ao contrário do `textContent` atributo, no entanto, esse atributo é avaliado como código executável depois que o nó é inserido no DOM.

- **Type**

Indica o tipo de script representado. O valor deste atributo estará em uma das seguintes categorias:

Omitido ou um tipo MIME JavaScript: para navegadores compatíveis com HTML5, isso indica que o script é JavaScript. A especificação HTML5 insta os autores a omitir o atributo em vez de fornecer um tipo MIME redundante. Em navegadores anteriores, isso identificava a linguagem de script do src código incorporado ou importado (por meio do atributo). Os tipos MIME JavaScript estão listados na especificação .

Module: HTML5 Para navegadores compatíveis com HTML5, o código é tratado como um módulo JavaScript. O processamento do conteúdo do script não é afetado pelos atributos charset, defer. Para obter informações sobre como usar module, consulte ES6 em Profundidade: Módulos.

Qualquer outro valor ou tipo MIME: o conteúdo incorporado é tratado como um bloco de dados que não será processado pelo navegador. O src atributo será ignorado.

Observe que no Firefox você pode usar recursos avançados, como instruções let e outros recursos em versões JS posteriores, usando. No entanto, esteja ciente de que, como esse é um recurso não padrão, isso provavelmente interrompe o suporte para outros navegadores, em particular navegadores baseados em Chromium.

Exemplo de uso:

```
<!-- HTML5 -->
<script src="javascript.js"></script>
```

1.2 Alert

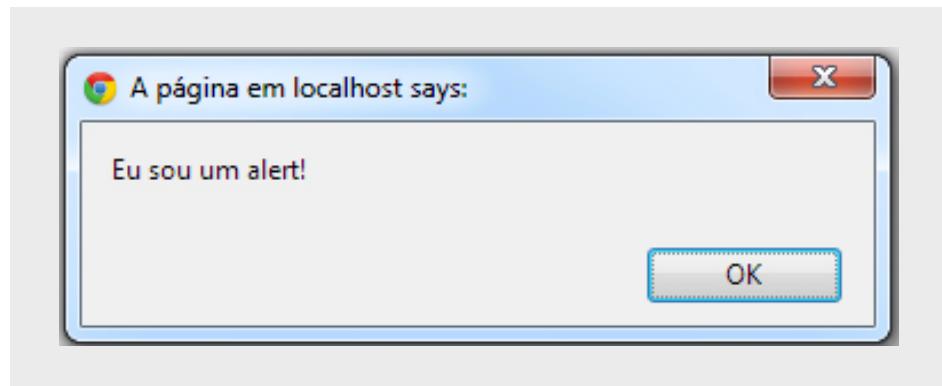
O **alert** é uma das mais simples caixas de diálogo, com uma aparência simples e intuitiva elas são muito usadas em validações de formulários e/ou bloqueio de ações do browser.

Sua principal função é mostrar ao usuário uma mensagem e um botão de confirmação de que o usuário tenha visto a mensagem. Para chamar essa função, basta utilizarmos o código alert(), que receberá uma string (mensagem que será exibida ao usuário).

Exemplo de uso:

```
<!DOCTYPE html> <html> <head> <title>Tutorial de Alert em JavaScript -  
Linha de Código</title> <script> function funcao1() { alert("Eu sou um  
alert!"); } </script> </head> <body> <input type="button"  
onclick="funcao1()" value="Exibir Alert" /> </body> </html></div>
```

O resultado do código acima deverá ser o mesmo que o apresentado:



1.3 Hello World

Neste exemplo, você aprenderá a imprimir 'Hello World' em JavaScript de três maneiras diferentes.

Um "Olá, Mundo!" é um programa simples que imprime Hello, World! na tela. Por ser um programa muito simples, este programa é frequentemente usado para introduzir uma nova linguagem de programação para iniciantes.

Usaremos essas três maneiras de imprimir 'Hello, World!'.

```
console.log()  
alert()  
document.write()
```

1. Usando console.log()

console.log() é usado na depuração do código.

```
Código fonte  
// the hello world program  
console.log('Hello World');
```

Saída

```
Olá Mundo!  
Aqui, a primeira linha é um comentário.  
// the hello world program  
A segunda linha  
console.log('Hello, World!');  
imprime o 'Olá Mundo!' cadeia para o console.
```

2. Usando alert()

O alert() método exibe uma caixa de alerta sobre a janela atual com a mensagem especificada.

```
Código fonte  
// the hello world program  
alert("Hello, World!");
```

3. Usando document.write()

document.write() é usado quando você deseja imprimir o conteúdo no documento HTML.

```
Código fonte  
// the hello world program  
document.write('Hello, World!');
```

1.4 HTML vs JavaScript

• JavaScript:

JavaScript é uma linguagem de programação que está em conformidade com a especificação ECMAScript. É uma linguagem de script de alto nível introduzida pela Netscape para ser executada no lado do cliente do navegador da web. Ele pode inserir texto dinâmico em HTML. JavaScript também é conhecido como a linguagem do navegador.

• HTML:

HTML (HyperText Markup Language) é o bloco de construção mais básico da Web. Ele define o significado e a estrutura do conteúdo da web. É a combinação de hipertexto e linguagem de marcação. O hipertexto define o link entre as páginas da web.

Vamos ver a diferença entre JavaScript e HTML:

S.Nº.	JAVASCRIPT	HTML
1	JavaScript é uma linguagem de script de alto nível introduzida pela Netscape para ser executada no lado do cliente do navegador da web.	HTML é o bloco de construção mais básico da Web. Ele define o significado e a estrutura do conteúdo da web.
2	JavaScript é uma linguagem de programação avançada que torna as páginas da web mais interativas e dinâmicas.	HTML é uma linguagem de marcação padrão que fornece a estrutura primária de um site.
3	JavaScript simplesmente adiciona conteúdo dinâmico aos sites para torná-los mais bonitos.	HTML funciona na aparência do site sem os efeitos interativos e tudo mais.
4	Manipula o conteúdo para criar páginas da web dinâmicas	As páginas HTML são estáticas, o que significa que o conteúdo não pode ser alterado.
5	Ele adiciona interatividade às páginas da Web para torná-las boas.	Ele define a estrutura básica de uma página da web.
6	JavaScript é uma linguagem de programação avançada usada para criar páginas da web dinâmicas.	O HTML é renderizado de todo o lado do servidor, o que é diferente do script do lado do cliente.
7	JavaScript não é compatível com vários navegadores (algumas funções podem não funcionar em todos os navegadores).	HTML é compatível com vários navegadores (funciona bem com todas as versões do navegador).
8	JavaScript pode ser incorporado dentro de HTML.	HTML não pode ser embutido em JavaScript.

1.5 Tipagem

Ouvimos muito por aí que o JavaScript não possui tipagem, mas não é bem assim.

Ele é dinamicamente tipado, o que significa que nós não precisamos declarar seu tipo. Mas isso só é possível porque ele mesmo identifica o tipo da variável que estamos declarando e o atribui, mesmo que não explicitamente.

Por exemplo:

```
const cliente = "Juliana"
```

Nós não precisamos dizer para o JavaScript que cliente é uma String, ele já sabe que é!

E aí você pensa: "Poxa, como ele é bonzinho! Economiza o nosso tempo, né?". Sim, eu também acho. Mas é muito normal nos depararmos com uns erros e bugs bizarros graças a essa tipagem dinâmica, quando não entendemos muito bem como ela funciona.

Imagine a seguinte situação: trabalhamos em um sistema importantíssimo de finanças pessoais. Nele, a pessoa digita quanto ganha e quanto gasta e nós vamos adicionando e subtraindo esses valores de seu saldo.

Após uma atualização no nosso sistema, recebemos uma ligação desesperada do comercial da empresa: tem uns valores muito loucos aparecendo para os clientes e as contas não batem.

Nessa atualização permitimos que os números sejam retirados diretamente da conta corrente do cliente.

O que não havíamos percebido é que esses valores vinham como String e não estávamos tratando isso. Sendo assim, um valor de R\$ 20,00 que entrou em uma conta com mais R\$ 20,00 fez com que essa conta totalizasse R\$ 2020,00!

E é exatamente para entender como tratar e prevenir esse tipo de problema que esse artigo foi feito.

• Conhecendo os tipos existentes no JavaScript

Os tipos mais conhecidos hoje no JavaScript são: **string, boolean, string, function, undefined e object**.

Acho que a primeira coisa que vem à mente é se realmente só existem esses tipos. Cadê o null e o array? Bem, eles não existem!

Vamos pegar algumas variáveis do nosso sistema de finanças para analisarmos um pouco os tipos das variáveis que existem.

```
const cliente = 'Joaquina';
const premium = false;
let saldo = 20.00;
const atualizaSaldo = (saldo, novoValor) => {
    saldo += novoValor;
}
const transacao = {
    descricao: 'Almoço Juliano',
    tipo: 'crédito',
    valor: '20.00',
    data: `08/05/2019`
}
const contasBancarias = [
{agência: 1234, conta: 16452-2},
{agência: 1234, conta: 16458-5}
];
```

Para vermos o tipo dessas variáveis vamos usar o operador unário `typeof`.

`typeof cliente //string`

// o js reconhece string pelo uso de aspas, simples ou duplas, ou crase:

`typeof premium //boolean`

`typeof saldoConta //number`

`typeof atualizaSaldo //function`

A variável `atualizaSaldo` é uma função e isso é definido pela sintaxe que, no caso, é arrow function.

`typeof transacao //object`

A variável `transacao` é um objeto e isso é definido pelas chaves {}.

No nosso sistema, também temos a funcionalidade de pagamento agendado. Você consegue salvar a data de um pagamento para ser debitado do seu saldo.

Por conta dessa funcionalidade, também precisamos avisar o(a) usuário(a), caso esta pessoa esteja tentando realizar uma transação que já foi feita. Para isso, temos essa função:

```
const verificaTransacao = (transacao, novaTransacao) => {
    return transacao === novaTransacao;
}
```

Legal! Agora vamos chamar a nossa função:

```
const novaTransacao = {
    descricao: 'Almoço Juliano',
    tipo: 'crédito',
    valor: '20.00',
    data: `08/05/2019`
}

verificaTransacao(transacao, novaTransacao) //false
```

Acabamos de achar um problema no nosso sistema! Ele nunca vai avisar aos nossos usuários que as transações são iguais dessa forma. Mesmo os nossos objetos tendo exatamente os mesmos valores, eles não são iguais. E o que isso significa?

• Um pouco sobre objetos:

No JavaScript, o objeto é um valor de referência, ou seja, aponta para algum lugar específico na memória e, em comparações, é utilizada essa referência.

Com isso em mente, agora sabemos que ao fazer `(transacao === novaTransacao)` o JavaScript vai verificar se essas duas referências apontam para o mesmo local na memória e não vai verificar os valores internos dos nossos objetos.

Então, cuidado com comparações com objetos e tente sempre comparar os valores existentes no objeto.

`typeof contasBancarias //object`

A variável `contasBancarias` é um objeto e isso é definido pelo... pera, essa variável deveria ser um Array!

Pois é, aqui é onde começa a ficar um pouco estranho. Arrays são do tipo object para o JavaScript, o que significa que não podemos saber se uma variável é um array ou não apenas com o `typeof`. Para sabermos se uma variável é um array, precisamos fazer da seguinte forma:

`Array.isArray(contasBancarias) //true`

`Array.isArray(transacao) //false`

Agora sim! Mas temos que lembrar que, por ser do tipo objeto, um array também é um valor de referência e um array nunca vai ser igual ao outro (a menos que eles apontem para o mesmo lugar). Então, assim como no objeto, é bom sempre compararmos os valores do array e não o array em si.

- **Resultados numéricos inesperados:**

Agora vamos tentar resolver o nosso problema inicial e ajustar a nossa função atualizaSaldo() para que ela só nos retorne o valor se for um número, usando o operador typeof:

```
const atualizaSaldo = (saldo, novoValor) => {
  const novoSaldo = saldo + novoValor;
  if(typeof novoSaldo == number) {
    saldo = novoSaldo;
  } else {
    return 'Não foi possível calcular o resultado';
  }
}
```

Pronto! Com essas alterações estamos garantindo que nossa aplicação não apresente erros inesperados, certo? **Vamos testar:**

```
atualizaSaldo (20.00, 20) //20
atualizaSaldo ('vinte', 20) //NaN
```

No segundo exemplo ele me retornou o valor NaN, que significa Not a Number. Mas, se não é um número, não deveria ter retornado o texto que eu pedi?

Nesse caso não, porque o tipo de NaN também é number, por mais confuso que isso possa parecer. Isso acontece porque ele é o resultado de uma operação numérica e, na matemática, podemos associar com o \perp (não existe).

E ainda temos mais uma complicação com esse valor:

```
NaN == NaN //false
```

Um NaN nunca vai ser igual ao outro, visto que ele pode ser qualquer coisa que não seja um número. Se precisarmos saber se esse valor é NaN, podemos usar a seguinte função:

```
const resultado =
  atualizaSaldo('vinte', 20);
isNaN(resultado) //true
```

- **Usando o booleano false ao seu favor:**

Sabemos que undefined é um valor não definido e tem seu próprio tipo. Mas e o null? **Vamos testar:**

```
typeof null //object
```

Ele é do tipo objeto (????), mas um tipo que precisamos tomar cuidado ao usar.

É importante também saber quais valores são considerados falsos pelo JavaScript e usarmos isso ao nosso favor. Os valores considerados falsos são: 0, "", undefined, NaN e null. Isso significa que, se na nossa regra de negócio formos considerar o 0 e string vazia como valores falsos, podemos fazer da seguinte forma:

```
const atualizaSaldo= (saldo, novoValor) => {
    const novoSaldo = saldo + novoValor;
    if(novoSaldo) {
        saldo = novoSaldo;
    } else {
        return 'Não foi possível calcular o resultado';
    }
}
```

Pronto, dessa forma garantimos que o saldo só seja atualizado se o valor realmente for um número. Como o if é uma operação condicional, ele sempre vai converter as expressões para valores booleanos, ou seja, vai verificar se a variável novoSaldo representa um valor verdadeiro ou falso.

Entendendo a coerção automática de tipos

Isso só acontece porque também temos a coerção automática de dados no JavaScript. O que isso significa? Que sempre que fizermos operações com valores de tipos diferentes, ele vai converter um dos dois para poder realizar a operação. **Por exemplo:**

```
1 + '1' //11
1 - '1' //0
1 == '1' //true
```

- **Dicas finais para escapar desses problemas:**

A primeira dica (provavelmente a mais falada também) é sempre que possível usar === ao invés de ==. Isso é importante porque o === retorna verdadeiro apenas se o valor E o tipo forem iguais, cancelando a coerção automática do JS.

```
1 == '1' //true
1 === '1' //false
```

Também é melhor sempre converter os valores antes de fazermos as operações, para evitarmos valores indesejados.

Por exemplo, nós sempre recebemos o valor de cursos finalizados como string, mas precisamos calcular esses valores como números. **Podemos fazer:**

```
const atualizaSaldo= (saldo, novoValor) => {
  const novoSaldo = saldo + Number(novoValor);
  if(novoSaldo) {
    saldo = novoSaldo;
  } else {
    return 'Não foi possível calcular o resultado';
  }
}
```

1.6 document.write

- **Sintaxe**

document.write(markup);

- **Copy to Clipboard**

Parametros

- **Markup**

Uma string contendo o texto a ser gravado no documento.

Exemplo:

```
<html>

<head>
    <title>Escreva exemplo</title>

    <script>
        function newContent() {
            document.open();
            document.write("<h1>Sair com o velho - entrar com o novo!</h1>");
            document.close();
        }
    </script>
</head>

<body onload="newContent();">
    <p>Algum conteúdo do documento original.</p>
</body>

</html>
```

- **Copy to Clipboard**

Notas

Escrevendo em um documento que já foi carregado sem chamar `document.open()` (en-US) automaticamente vai chamar `document.open`. Ao término da escrita, é recomendável chamar `document.close()` para dizer ao navegador para encerrar o carregamento da página. O texto que você escreve é analisado no modelo de estrutura do documento. No exemplo acima, o elemento `h1` se torna um nó (node) no documento.

Se chamar `document.write()` incorporada em uma tag HTML `<script>` embutida, então `document.open()` não será chamada. Por exemplo:

```
<script>
  document.write("<h1>Título principal</h1>")
</script>
```

1.7 Operações Matemáticas

• Todo mundo ama matemática

Ok, talvez não. Alguns de nós gostam de matemática, alguns de nós tem a odiado desde que tivemos que aprender a tabuada de multiplicação e divisão na escola, e alguns de nós estão em algum lugar no meio dos dois cenários. Mas nenhum de nós pode negar que a matemática é uma parte fundamental da vida sem a qual não poderíamos ir muito longe. Isso é especialmente verdadeiro quando estamos aprendendo a programar em JavaScript (ou em qualquer outra linguagem, diga-se de passagem) — muito do que fazemos se baseia no processamento de dados numéricos, cálculo de novos valores, etc. Assim você não ficará surpreso em aprender que o JavaScript tem uma configuração completa de funções matemáticas disponíveis.

Este artigo discute apenas as partes básicas que você precisa saber agora.

• Tipos de números

Em programação, até mesmo o humilde sistema de números decimais que todos nós conhecemos tão bem é mais complicado do que você possa pensar. Usamos diferentes termos para descrever diferentes tipos de números decimais, **por exemplo:**

Integers (inteiros) são números inteiros, ex. 10, 400 ou -5.

Números de ponto flutuante (**floats**) tem pontos e casas decimais, por exemplo 12.5 e 56.7786543.

Doubles são tipos de **floats** que tem uma precisão maior do que os números de ponto flutuante padrões (significando que eles são mais precisos, possuindo uma grande quantidade de casas decimais).

Temos até mesmo diferentes tipos de sistemas numéricos! O decimal tem por base 10 (o que significa que ele usa um número entre 0-9 em cada casa), mas temos também algo como:

• Binário

A linguagem de menor nível dos computadores; 0s e 1s.

• Octal

Base 8, usa um caractere entre 0-7 em cada coluna.

• Hexadecimal

Base 16, usa um caractere entre 0-9 e depois um entre a-f em cada coluna. Você pode já ter encontrado esses números anteriormente, trabalhando com cores no CSS.

Antes de se preocupar com seu cérebro está derretendo, pare agora mesmo! Para um começo, vamos nos ater aos números decimais no decorrer desse curso; você raramente irá se deparar com a necessidade de começar a pensar sobre os outros tipos, se é que vai precisar algum dia.

A segunda boa notícia é que, diferentemente de outras linguagens de programação, o JavaScript tem apenas um tipo de dados para números, você adivinhou, Number. Isso significa que qualquer que seja o tipo de números com os quais você está lidando em JavaScript, você os manipula do mesmo jeito.

- **Tudo é número para mim**

Vamos brincar rapidamente com alguns números para nos familiarizarmos com a sintaxe básica que precisamos. Insira os comandos listados abaixo em seu console JavaScript, ou use o console simples incorporado abaixo, se preferir.

- **Abra em uma nova janela**

Primeiramente, vamos declarar duas variáveis e as inicializar com um integer e um float, respectivamente, então digitamos os nomes das variáveis para verificar se está tudo em ordem:

```
var meuInt = 5;  
var meuFloat = 6.667;  
meuInt;  
meuFloat;
```

Valores numéricos são inseridos sem aspas — tente declarar e inicializar mais duas variáveis contendo números antes de seguir em frente.

Agora vamos checar se nossas duas variáveis originais são do mesmo tipo de dados. Há um operador chamado `typeof` no JavaScript que faz isso. Insira as duas linhas conforme mostradas abaixo:

```
typeof meuInt;  
typeof meuFloat;
```

Você deve obter "number" de volta nos dois casos — isso torna as coisas muito mais fáceis para nós do que se diferentes tipos de números tivessem diferentes tipos de dados, e tivéssemos que lidar com eles de diferentes maneiras. Ufa!

• Operadores aritméticos

São os operadores que utilizamos para fazer as operações aritméticas básicas:

Operador	Nome	Propósito	Exemplo
+	Adição	Adiciona um número a outro.	6 + 9
-	Subtração	Subtrai o número da direita do número da esquerda.	20 - 15
*	Multiplicação	Multiplica um número pelo outro.	3 * 7
/	Divisão	Divide o número da esquerda pelo número da direita.	10 / 5
%	Restante (Remainder - as vezes chamado de modulo)	Retorna o resto da divisão em números inteiros do número da esquerda pelo número da direita.	8 % 3 (retorna 2; como três cabe duas vezes em 8, deixando 2 como resto.)

Nós provavelmente não precisamos ensinar a você como fazer matemática básica, mas gostaríamos de testar seu entendimento da sintaxe envolvida. Tente inserir os exemplos abaixo no seu console JavaScript, ou use o console incorporado visto anteriormente, se preferir, para familiarizar-se com a sintaxe.

Primeiro tente inserir alguns exemplos simples por sua conta, como:

10 + 7

9 * 8

60 % 3

Você pode tentar declarar e inicializar alguns números dentro de variáveis, e tentar usá-los nas operações — as variáveis irão se comportar exatamente como os valores que elas armazenam para a finalidade das operações. **Por exemplo:**

```
var num1 = 10;
var num2 = 50;
9 * num1;
num2 / num1;
Copy to Clipboard
```

Por último, nesta seção, tente inserir algumas expressões mais complexas, como:

**5 + 10 * 3;
num2 % 9 * num1;
num2 + num1 / 8 + 2;**

Alguns dos exemplos do último bloco podem não ter retornado os valores que você estava esperando; a seção abaixo pode lhe explicar o porquê.

• Precedência de operador

Vamos olhar para o último exemplo, assumindo que num2 guarda o valor 50 e num1 possui o valor 10 (como iniciado acima):

**num2 + num1 / 8 + 2;
Copy to Clipboard**

Como um ser humano, talvez você leia isso como "50 mais 10 é igual a 60", depois "8 mais 2 é igual a 10", e então "60 dividido por 10 é igual a 6".

No entanto seu navegador vai ler "10 dividido por 8 é igual a 1.25", então "50 mais 1.25 mais 2 é igual a 53.25".

Isto acontece por causa da precedência de operadores — algumas operações serão aplicadas antes de outras quando calcularmos o resultado de uma soma (referida em programação como uma expressão). A precedência de operadores em JavaScript é semelhante ao ensinado nas aulas de matemática na escola — Multiplicação e divisão são realizados primeiro, depois a adição e subtração (a soma é sempre realizada da esquerda para a direita).

Se você quiser substituir a precedência do operador, poderá colocar parênteses em volta das partes que desejar serem explicitamente tratadas primeiro. Então, para obter um resultado de 6, poderíamos fazer isso:

(num2 + num1) / (8 + 2);

Tente fazer e veja como fica.

Nota: Uma lista completa de todos os operadores JavaScript e suas precedências pode ser vista em Expressões e operadores.

• Operadores de incremento e decremento

Às vezes você desejará adicionar ou subtrair, repetidamente, um valor de uma variável numérica. Convenientemente isto pode ser feito usando os operadores incremento `++` e decremento `--`. Usamos `++` em nosso jogo "Adivinhe o número" no primeiro artigo Um primeiro mergulho no JavaScript, quando adicionamos 1 ao nosso `contagemPalpites` para saber quantas adivinhações o usuário deixou após cada turno.

contagemPalpites++;

Vamos tentar brincar com eles no seu console. Para começar, note que você não pode aplicá-las diretamente a um número, o que pode parecer estranho, mas estamos atribuindo à variável um novo valor atualizado, não operando no próprio valor. O seguinte retornará um erro:

`3++;`

Então, você só pode incrementar uma variável existente. Tente isto:

```
var num1 = 4;  
num1++;
```

Ok, segunda coisa estranha! Quando você fizer isso, verá um valor 4 retornado - isso ocorre porque o navegador retorna o valor atual e, em seguida, incrementa a variável. Você pode ver que ele foi incrementado se você retornar o valor da variável novamente:

```
num1;
```

Acontece a mesma coisa com -- : tente o seguinte

```
var num2 = 6;  
num2--;  
num2;
```

• Operadores de atribuição

Operadores de atribuição são os que atribuem um valor à uma variável. Nós já usamos o básico, =, muitas vezes, simplesmente atribuindo à variável do lado esquerdo o valor indicado do lado direito do operador:

```
var x = 3; // x contém o valor 3  
var y = 4; // y contém o valor 4  
x = y; // x agora contém o mesmo valor de y, 4  
Copy to Clipboard
```

Mas existem alguns tipos mais complexos, que fornecem atalhos úteis para manter seu código mais puro e mais eficiente. Os mais comuns estão listados abaixo:

Operator	Name	Purpose	Example	Shortcut for
<code>+=</code>	Atribuição de adição	Adiciona o valor à direita ao valor da variável à esquerda e, em seguida, retorna o novo valor da variável	<code>x = 3; x += 4;</code>	<code>x = 3; x = x + 4;</code>
<code>-=</code>	Atribuição de subtração	Subtrai o valor à direita do valor da variável à esquerda e retorna o novo valor da variável	<code>x = 6; x -= 3;</code>	<code>x = 6; x = x - 3;</code>
<code>*=</code>	Atribuição de multiplicação	Multiplica o valor da variável à esquerda pelo valor à direita e retorna o novo valor da variável	<code>x = 2; x *= 3;</code>	<code>x = 2; x = x * 3;</code>
<code>/=</code>	Atribuição de divisão	Divide o valor da variável à esquerda pelo valor à direita e retorna o novo valor da variável	<code>x = 10; x /= 5;</code>	<code>x = 10; x = x / 5;</code>

Tente digitar alguns dos exemplos acima em seu console para ter uma ideia de como eles funcionam. Em cada caso, veja se você pode adivinhar qual é o valor antes de digitar a segunda linha.

Note que você pode muito bem usar outros valores no lado direito de cada expressão, por exemplo:

```
var x = 3; // x contém o valor 3
var y = 4; // y contém o valor 4
x *= y; // x agora contém o valor 12
```

- **Operadores de comparação**

Às vezes, queremos executar testes verdadeiro / falso e, em seguida, agir de acordo, dependendo do resultado desse teste, para fazer isso, usamos operadores de comparação.

Operator	Name	Purpose	Example
<code>===</code>	Igualdade estrita	Verifica se o valor da esquerda e o da direita são idênticos entre si.	<code>5 === 2 + 4</code>
<code>!==</code>	Não-igualdade-estrita	Verifica se o valor da esquerda e o da direita não são idênticos entre si.	<code>5 !== 2 + 3</code>
<code><</code>	Menor que	Verifica se o valor da esquerda é menor que o valor da direita.	<code>10 < 6</code>
<code>></code>	Maior que	Verifica se o valor da esquerda é maior que o valor da direita.	<code>10 > 20</code>
<code><=</code>	Menor ou igual que	Verifica se o valor da esquerda é menor que ou igual ao valor da direita.	<code>3 <= 2</code>
<code>>=</code>	Maior ou igual que	Verifica se o valor da esquerda é maior que ou igual ao valor da direita.	<code>5 >= 4</code>

Tente digitar alguns dos exemplos acima em seu console para ter uma ideia de como eles funcionam. Em cada caso, veja se você pode adivinhar qual é o valor antes de digitar a segunda linha.

Note que você pode muito bem usar outros valores no lado direito de cada expressão, por exemplo:

```
var x = 3; // x contém o valor 3
var y = 4; // y contém o valor 4
x *= y; // x agora contém o valor 12
```

1.8 Concatenação (String)

Alguns exemplos são os booleanos (verdadeiro ou falso), números (inteiros, ponto flutuante) e alguns tipos mais complexos, como estruturas e objetos. Um dos tipos mais utilizados em programação são as strings, sequências (ou cadeias) de caracteres que usamos para, entre outras coisas, manipular textos.

No exemplo a seguir, temos uma cadeia de caracteres representada como um array:

```
const fruta = "banana"  
// ["b", "a", "n", "a", "n", "a"]
```

Imagine que estamos desenvolvendo o código de uma aplicação em JavaScript. A validação dos campos de login e senha foi solicitada e é preciso verificar se o tamanho da senha de cada usuário atende à regra de ter um tamanho mínimo de 8 caracteres e impedir que haja espaços no início ou no fim do login cadastrado.

Desse modo, precisaremos fazer o tratamento de strings.

Como podemos fazer isso? Quais opções o JavaScript pode oferecer para esses casos? No texto a seguir, vamos ver essas possibilidades de trabalhar com strings na linguagem.

• O que é uma String?

Por definição, strings são sequências de caracteres alfanuméricos (letras, números e/ou símbolos) amplamente usadas em programação. Em Javascript, uma string sempre estará entre aspas.

`const frase = "Mergulhando em tecnologia
com Digital College";`

ou

`const frase = 'Mergulhando em tecnologia
com Digital College';`

ou ainda

`console.log('Mergulhando em tecnologia
com Digital College')`

Espere! Mas eu declaro minhas strings com aspas duplas ou simples?

Podemos colocar nossas **strings** entre aspas duplas ou simples. Para o JavaScript, não há diferença, já que ele considera as duas formas de declaração válidas. Mas, atenção, essa regra pode não se aplicar a outras linguagens. No Java ou C#, por exemplo, aspas simples são usadas para definir um caractere.

Em alguns momentos, a string poderá ser um texto que contém aspas. Nesses casos, é preciso combinar a utilização das aspas simples com aspas duplas e vice-versa, porque um texto como: "Ela disse: "Adeus""", não funciona corretamente.

• Vamos ao exemplo:

`console.log('Ela disse: "Adeus!"')`

ou

`console.log("Ela disse: 'Adeus!' ")`

É importante ressaltar que, depois que a sequência de caracteres for definida, a string é imutável, ou seja, não poderá ter seu valor alterado. Então, como manipular a string?

Sempre que manipulamos uma string, é criada uma nova instância dela por baixo dos panos, o que significa que será gerado um novo espaço na memória com uma cópia do valor da string. Por isso, temos que utilizar uma variável para armazená-la.

• Objeto String

A linguagem JavaScript traz ainda como recurso um objeto global String que nos permite criar ou converter um tipo em uma string, **veja o exemplo abaixo:**

```
const numero = 256
const convertidoEmString = new
String(numero)
```

A saída após exibirmos a variável convertidoEmString usando o método console.log() é [String: '256']. Na construção do objeto usando new String(parâmetro), o parâmetro pode ser qualquer elemento do nosso código que queiramos transformar em string.

Também é possível converter outros tipos primitivos (por exemplo, números e booleanos) em strings com o método toString():

```
const num = 500
console.log(num.toString()) // '500'
```

• Usando Strings

É possível interpolar, concatenar, checar posições de caracteres ou ainda substituir partes de strings. Vamos ver algumas dessas utilizações com o JavaScript?

• Concatenando strings

Quando falamos em concatenar strings, quer dizer que vamos juntar duas ou mais strings e formar uma nova. Observe o exemplo abaixo:

```
let nome = "André"
let sobreNome = "Silva"
let nomeCompleto = "Meu nome completo
é : " + nome + sobreNome
```

Para concatenar as strings nome e sobreNome com a string de texto que é o valor de nomeCompleto, usamos o operador de adição (+). Podemos usar também +=, como no exemplo abaixo:

```
let nome = "André"
let saudacoes = "Seja bem-vindo "
saudacoes += nome
```

Dessa forma, temos a saída **Seja bem-vindo André**.

- **Interpolando strings (template strings)**

A interpolação de strings é um recurso bem interessante, presente em diversas linguagens. No JavaScript, é uma alternativa mais prática para manipular string sem a necessidade de fazer concatenação, porque para textos maiores, concatenar pode ser um pouco trabalhoso.

Usando as chamadas template strings ou templates literais, a pessoa desenvolvedora consegue ter uma flexibilidade maior no trabalho com strings, além de facilitar a escrita e leitura do código.

Retomando o exemplo da mensagem de boas vindas, veja abaixo a utilização de template strings:

```
let nome = "André"
let saudacoes = `Seja bem-vindo ${nome}`
```

Veja como exemplo o poema “E agora, José?” de Carlos Drummond de Andrade:

```
let nome = "André"
let poema = `
E agora, ${nome}?
A festa acabou,
a luz apagou,
o povo sumiu,
a noite esfriou,
e agora, ${nome}?
e agora, você?
você que é sem nome,
que zomba dos outros,
você que faz versos,
que ama, protesta?
e agora, ${nome}?`
```

Observe que, para a utilização da template string, ela deve estar entre acentos graves (`) e, para fazer a interpolação, o valor ou variável deve estar dentro da estrutura \${valor}. Vale ressaltar que usando *template strings* temos a opção de utilizar a quebra de linha normalmente, sem caracteres de escape para isso, como \n`.

- **Métodos para strings**

Antes de começarmos, é importante ressaltar que o JavaScript diferencia strings como tipos primitivos (com aspas duplas ou simples) de objetos Strings (quando usamos a palavra reservada new). Mas, por baixo dos panos toda string, mesmo as que criamos com a chamada “forma literal”, por exemplo const texto = “Digital College”, acaba convertida para um objeto do tipo String. Por isso, temos acesso a uma série de métodos e propriedades deste objeto.

Agora que entendemos isso, vamos ver algumas propriedades e métodos úteis e bem práticos para trabalhar com strings em nossas aplicações.

- **.length**

A propriedade length serve para nos informar o tamanho de uma string. E por que isso é útil?

Caso sua aplicação tenha como uma das regras para criação de senhas (que em geral são alfanuméricas) o tamanho de 8 caracteres, usar length será uma boa opção, pois ajudará a contar a quantidade de caracteres da string.

Para testar a propriedade length, vamos usar a string Digital College, que retornará o tamanho 5.

```
const palavra="Digital College";
console.log(palavra.length) //5
```

Veja que length é exatamente a mesma propriedade que acessamos quando queremos descobrir o comprimento (ou seja, a quantidade de elementos) em um array.

• **charAt()**

Com o método charAt() conseguimos acessar um caractere de uma string. Lembre-se que, por baixo dos panos, strings são arrays de caracteres, e em cada posição temos o caractere que compõe a string.

Veja o exemplo abaixo:

```
const palavra="Digital College".charAt(3) //r
```

Após a execução do método charAt(), ela retornará o caractere r, que é o valor que consta na posição 3 da string - lembrando que arrays em JavaScript começam na posição 0 (zero).

Outra alternativa será usar a notação de colchetes para encontrar um caractere da string, da seguinte forma:

```
const palavra="Digital College"  
console.log(palavra[0]) //A
```

Será exibido o caractere A, ou seja, o que está na primeira posição da string. O resultado da execução do charAt() é uma string.

Mas e se quisermos saber qual a posição de um caractere dentro da string?

• **indexOf()**

Respondendo a pergunta anterior, existe a função indexOf(), que retorna a posição de um caractere dentro da string.

Por exemplo:

```
const palavra="Digital College"  
console.log(palavra.indexOf("a")) //4
```

O resultado é a posição 4. Porém, na utilização do indexOf(), fique atento caso o caractere que se busca na string seja encontrado em mais de uma posição, pois será retornada somente a primeira ocorrência. veja o código abaixo:

```
const palavra="Divertidamente"  
console.log(palavra.indexOf("e")) //3
```

O resultado da execução do indexOf() é um valor numérico.

toUpperCase() e toLowerCase()

São duas funções bastante utilizadas quando estamos trabalhando com string e precisamos deixar o texto todo em letras minúsculas (lower case) ou todo em maiúsculas (upper case). Vamos ver o código abaixo:

```
const palavra="Digital College";  
  
console.log(palavra.toUpperCase())  
//DIGITAL COLLEGE  
  
console.log(palavra.toLowerCase()) //digital  
college
```

Após a execução do código, o console irá exibir Digital College e digital college respectivamente. O resultado da execução dos métodos toUpperCase() e toLowerCase() é uma nova string.

- **substr()**

Outra função muito interessante é a substr() (substring), que permite que façamos a extração de parte de uma string, conforme o código abaixo:

```
let frase= "Mergulhando em tecnologia."
console.log(frase.substr(0,11)) //
Mergulhando
```

A função recebe como parâmetro o início e o fim da nova string a ser retirada da string principal. Na execução do código acima, temos como resultado a palavra Mergulhando. Bem útil, né?

O resultado da execução do método substr() é uma nova string.

- **slice()**

Podemos utilizar também o método slice(), que usamos com arrays. Ele é similar ao substring() e retornará parte de uma string, desde que passemos nos parâmetros o índice de início e de fim. Veja abaixo:

```
let frase= "Mergulhando em tecnologia."
console.log(frase.slice(0,11)) //
Mergulhando
```

O resultado da execução do método slice() é uma nova string.

- **replace()**

Com a função replace() temos a possibilidade de substituir parte de uma string por outra. Essa função recebe como parâmetros duas informações: a string que você quer substituir e a string que será colocada no lugar.

Olhe o exemplo abaixo, em que precisamos substituir a string nomeusuario no texto padrão de comunicacao.

```
let nome = "André";
```

```
let comunicacao = " Olá, sr. nomeusuario,
informamos que a partir da presente data
o senhor tem 50% de desconto em nossa
loja.;"
```

```
console.log(comunicacao.replace("nomeus
uario",nome));
```

Na execução deste exemplo, a string nomeusuario será substituída pelo conteúdo da variável nome. Como resultado da execução do método replace() teremos uma nova string.

- **concat()**

O método concat() é uma opção para concatenar strings sem a utilização do operador de adição (+). Ele concatena duas strings, adicionando a nova string ao fim da anterior.

Observe uma utilização do concat():

```
let novaString = "Programe nas principais
linguagens e plataformas. Explore
linguagens como ";
```

```
console.log(novaString.concat("JavaScript,"
).concat(" Python,").concat(" e C#."))
```

O resultado obtido será: Programe nas principais linguagens e plataformas. Explore linguagens como JavaScript, Python, e C#.

Para a execução do método replace() teremos como resultado uma nova string.

- **split()**

O método split() é bem interessante, pois com ele conseguimos quebrar uma string com base em caracteres separadores que vamos informar para o método como parâmetro.

Vamos ver um exemplo:

```
let linguagens =  
"JavaScript;Java;C#;PHP;Python;Go;Vb;SQL;  
C;C++";  
  
let arrayLinguagens = linguagens.split(";" );  
  
console.log(arrayLinguagens)
```

Quando trabalhamos com o split(), devemos nos atentar, pois a execução gerará como resultado um array de strings com os elementos que foram separados com base no separador desejado. Portanto a execução do código resulta em um array como mostrado a seguir:

```
[ 'JavaScript', 'Java', 'C#', 'PHP', 'Python',  
'Go', 'Vb', 'SQL', 'C', 'C++' ]
```

Lembre-se que o resultado da execução do método split() é um array de strings.

- **trim()**

O trim() remove os espaços em branco no início ou fim de uma string. Se em alguma situação precisarmos fazer uma verificação de que o usuário não digitou o login com espaços, faremos;

```
let login = " andre@emailteste.com ";  
  
let loginSemEspaco = login.trim();  
  
console.log(loginSemEspaco);  
//andre@emailteste.com
```

A variável loginSemEspaco conterá uma nova string, sem os espaços em branco no início ou fim que podem ter sido digitados. Então, quando executado o método trim(), o resultado é uma nova string.

No JavaScript ainda temos algumas variações desta função como: trimEnd(), trimStart(), trimLeft() e trimRight(), teste também estas variantes e veja o resultado obtido, ok?

1.9 Interagindo (Prompt)

O próximo tipo de caixa de diálogo é o método prompt(). Seu objetivo é obter alguma informação da pessoa usuária da página. Confira a sua sintaxe:

```
window.prompt("argumento1", ["argumento2"]);
```

No qual:

argumento1: a mensagem destinada à pessoa usuária, que corresponde à solicitação feita.

argumento2: texto opcional que serve como um exemplo do que deve ser preenchido no prompt.

Além dos argumentos, o método prompt() também exibe dois botões: OK e Cancelar. O OK indica que a pessoa preencheu o campo, enquanto o cancelar significa que ela deseja cancelar a operação. Vamos continuar com o exemplo anterior, entretanto, agora vamos adicionar um novo elemento à lista. Veja o código na página a seguir.

```
<!DOCTYPE html>
<html>
<head>
<title>
    Teste do método window.prompt()
</title>
<meta charset="utf-8">
</head>
<body>
    <p>Vamos adicionar novos itens:</p>
    <select name="itens" id="itens" size="6" style="width: 150px">
        <option>Camiseta</option>
        <option>Calça</option>
        <option>Sapato</option>
        <option>Bolsa</option>
    </select>
    <br/>
    <input type="button" value="Adicionar item"
    onclick=adicionarItem()

<script>
    function adicionarItem(){
        var item = prompt("Qual objeto você deseja incluir na lista?",
        "Adicione um novo objeto");
        if (item == null || item == "") {
            alert("O uso do prompt foi cancelado!");
        } else {
            var itens = document.getElementById("itens");
            var option = document.createElement("option");
            option.text = item;
            itens.add(option, itens[0]);
        }
    }
</script>
</body>
</html>
```

Perceba que na função adicionarItem() usamos o método prompt() para solicitar à pessoa o nome do objeto que ela deseja adicionar. O próximo passo é verificar se ela pressionou a opção cancelar e exibir uma mensagem de alerta. Já se for informado um valor, ele será adicionado à lista de objetos.

Observe que usamos o método prompt() com o segundo argumento preenchido. Por isso, aparece a frase “Adicione um novo objeto” na linha do prompt.

2.0 Hora da prática

utilizando os conhecimentos adquiridos construa uma calculadora.

3.1 Declaração de variáveis

JavaScript pega emprestado a maior parte de sua sintaxe do Java, mas também é influenciado por Awk, Perl e Python.

JavaScript é case-sensitive e usa o conjunto de caracteres Unicode. Por exemplo, a palavra Früh (que significa "cedo" em Alemão) pode ser usada como nome de variável.

```
var Früh = "foobar";
```

JavaScript pega emprestado a maior parte de sua sintaxe do Java, mas também é influenciado por Awk, Perl e Python.

JavaScript é case-sensitive e usa o conjunto de caracteres Unicode. Por exemplo, a palavra Früh (que significa "cedo" em Alemão) pode ser usada como nome de variável.

```
var Früh = "foobar";
```

• Declarações

Existem três tipos de declarações em JavaScript

• var

Declara uma variável, opcionalmente, inicializando-a com um valor.

• let

Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.

• const

Declara uma constante de escopo de bloco, apenas de leitura.

• Variáveis

Você usa variáveis como nomes simbólicos para os valores em sua aplicação. O nome das variáveis, chamados de identificadores, obedecem a determinadas regras.

Um identificador JavaScript deve começar com uma letra, underline (_), ou cifrão (\$); os caracteres subsequentes podem também ser números (0-9). Devido JavaScript ser case-sensitive, letras incluem caracteres de "A" a "Z" (maiúsculos) e caracteres de "a" a "z" (minúsculos).

Você pode usar a ISO 8859-1 ou caracteres Unicode tal como os identificadores à e ü. Você pode também usar as sequências de escape Unicode como caracteres e identificadores.

Alguns exemplos de nomes legais são **Numeros_visitas, temp99, e _nome**.

• Declarando variáveis

Você pode declarar uma variável de três formas:

Com a palavra chave var. Por exemplo, var x = 42. Esta sintaxe pode ser usada para declarar tanto variáveis locais como variáveis globais.

Por simples adição de valor. Por exemplo, x = 42. Isso declara uma variável global. Essa declaração gera um aviso de advertência no JavaScript. Você não deve usar essa variante.

Com a palavra chave let. Por exemplo, let y = 13. Essa sintaxe pode ser usada para declarar uma variável local de escopo de bloco. Veja o escopo de variável abaixo.

- **Classificando variáveis**

Uma variável declarada usando a declaração var ou let sem especificar o valor inicial tem o valor **undefined**.

Uma tentativa de acessar uma variável não declarada resultará no lançamento de uma exceção **ReferenceError**:

```
var a;
```

```
console.log("O valor de a é " + a); // saída "O  
valor de a é undefined"
```

```
console.log("O valor de b é " + b); // executa  
uma exception de erro de referência  
(ReferenceError)
```

Você pode usar undefined para determinar se uma variável tem um valor. No código a seguir, não é atribuído um valor de entrada na variável e a declaração if será avaliada como verdadeira (true).

```
var input;  
if(input === undefined){  
  facaIsto();  
} else {  
  facaAquilo();  
}
```

O valor undefined se comporta como falso (false), quando usado em um contexto booleano. Por exemplo, o código a seguir executa a função myFunction devido o elemento myArray ser undefined:

```
var myArray = [];  
if (!myArray[0]) myFunction();
```

O valor undefined converte-se para NaN quando usado no contexto numérico.

```
var a;
```

```
a + 2; // Avaliado como NaN
```

Quando você avalia uma variável nula, o valor nulo se comporta como 0 em contextos numéricos e como falso em contextos booleanos. **Por exemplo**:

```
var n = null;
```

```
console.log(n * 32); // a saída para o console  
será 0.
```

- **Escopo de variável**

Quando você declara uma variável fora de qualquer função, ela é chamada de variável global, porque está disponível para qualquer outro código no documento atual. Quando você declara uma variável dentro de uma função, é chamada de variável local, pois ela está disponível somente dentro dessa função.

JavaScript antes do ECMAScript 6 não possuía escopo de declaração de bloco; pelo contrário, uma variável declarada dentro de um bloco de uma função é uma variável local (ou contexto global) do bloco que está inserido a função. Por exemplo, o código a seguir exibirá 5, porque o escopo de x está na função (ou contexto global) no qual x é declarado, não o bloco, que neste caso é a declaração if.

```
if (true) {
    var x = 5;
}
console.log(x); // 5
```

Esse comportamento é alterado, quando usado a declaração let introduzida pelo ECMAScript 6.

```
if (true) {
    let y = 5;
}
```

console.log(y); // ReferenceError: y não está definido

• Variável de elevação

Outra coisa incomum sobre variáveis em JavaScript é que você pode utilizar a variável e declará-la depois, sem obter uma exceção. Este conceito é conhecido como hoisting; variáveis em JavaScript são num sentido "hoisted" ou lançada para o topo da função ou declaração. No entanto, as variáveis que são "hoisted" retornarão um valor undefined. Então, mesmo se você usar ou referir a variável e depois declará-la e inicializá-la, ela ainda retornará undefined.

• Exemplo 01

```
console.log(x === undefined); // exibe "true"
var x = 3;
```

• Exemplo 02

```
// retornará um valor undefined
var myvar = "my value";
```

```
(function() {
    console.log(myvar); //
undefined
    var myvar = "local value";
})();
```

Os exemplos acima serão interpretados como:

• Exemplo 01

```
var x;
console.log(x === undefined); // exibe "true"
x = 3;
```

• Exemplo 02

```
var myvar = "um valor";

(function() {
    var myvar;
    console.log(myvar); //
undefined
    myvar = "valor local";
})();
```

Devido o hoisting, todas as declarações var em uma função devem ser colocadas no início da função. Essa recomendação de prática deixa o código mais legível.

• Variáveis Globais

Variáveis globais são propriedades do objeto global. Em páginas web o objeto global é a window, assim você pode configurar e acessar variáveis globais utilizando a sintaxe window.variavel.

Consequentemente, você pode acessar variáveis globais declaradas em uma janela ou frame ou frame de outra janela.

Por exemplo, se uma variável chamada **phoneNumber** é declarada em um documento, você pode consultar esta variável de um frame como **parent.phoneNumber**.

• Constantes

Você pode criar uma constante apenas de leitura por meio da palavra-chave `const`. A sintaxe de um identificador de uma constante é semelhante ao identificador de uma variável: deve começar com uma letra, sublinhado ou cifrão e pode conter caractere alfabético, numérico ou sublinhado.

```
const PI = 3.14;
```

Uma constante não pode alterar seu valor por meio de uma atribuição ou ser declarada novamente enquanto o script está em execução. Deve ser inicializada com um valor.

As regras de escopo para as constantes são as mesmas para as variáveis `let` de escopo de bloco. Se a palavra-chave `const` for omitida, presume-se que o identificador represente uma variável.

Você não pode declarar uma constante com o mesmo nome de uma função ou variável que estão no mesmo escopo. Por exemplo:

```
// Isto irá causar um erro
function f() {};
const f = 5;
// Isto também irá causar um
erro.
function f() {
  const g = 5;
  var g;
```

• Declarações

Estrutura de dados e tipos de dados

O mais recente padrão ECMAScript define sete tipos de dados:

Seis tipos de dados são os chamados primitivos:

Boolean.

`true` e `false`.

null.

Uma palavra-chave que indica valor nulo. Devido JavaScript ser case-sensitive, `null` não é o mesmo que `Null`, `NULL`, ou ainda outra variação.

undefined.

Uma propriedade superior cujo valor é indefinido.

Number.

`42` ou `3.14159`.

String.

`"Howdy"`

Symbol (novo em ECMAScript 6).

Um tipo de dado cuja as instâncias são únicas e imutáveis.

e **Object**

Embora esses tipos de dados sejam uma quantidade relativamente pequena, eles permitem realizar funções úteis em suas aplicações. Objetos e funções são outros elementos fundamentais na linguagem. Você pode pensar em objetos como recipientes para os valores, e funções como métodos que suas aplicações podem executar.

- **Conversão de tipos de dados**

JavaScript é uma linguagem dinamicamente tipada. Isso significa que você não precisa especificar o tipo de dado de uma variável quando declará-la, e quais tipos de dados são convertidos automaticamente conforme a necessidade durante a execução do script. Então, por exemplo, você pode definir uma variável da seguinte forma:

```
var answer = 42;
```

E depois, você pode atribuir uma string para a mesma variável, por exemplo:

```
answer = "Obrigado pelos peixes...";
```

Devido JavaScript ser dinamicamente tipado, essa declaração não gera uma mensagem de erro.

Em expressões envolvendo valores numéricos e string com o operador +, JavaScript converte valores numéricos para strings. Por exemplo, considere a seguinte declaração:

```
x = "A resposta é " + 42 // "A resposta é 42"  
y = 42 + " é a resposta" // "42 é a resposta"
```

Nas declarações envolvendo outros operadores, JavaScript não converte valores numéricos para strings. **Por exemplo:**

```
"37" - 7 // 30  
"37" + 7 // "377"
```

3.2 Convenções

Codificação e convenções são diretrizes de estilo para a programação. Eles geralmente cobrem:

- Regras de nomeação e de declaração para variáveis e funções.
- Regras para o uso de espaço em branco, recuo, e comentários.
- Programação práticas e princípios
- Convenções de codificação de qualidade seguro:
- Melhora a legibilidade do código
- Faça a manutenção do código mais fácil

Usar variáveis em JavaScript parece ser uma tarefa muito simples, mas você tem que seguir algumas regras ao nomear suas variáveis, por exemplo, você terá que usar camelCase para nomes de identificadores. Todos os nomes devem começar com uma letra . O exemplo abaixo segue estas regras:

```
firstName = "John";
lastName = "Doe";

price = 19.90;
tax = 0.20;

fullPrice = price + (price * tax);
```

Para facilitar a leitura, evite linhas com mais de 80 caracteres, mas se sua instrução JavaScript não couber em uma linha, o melhor lugar onde você pode quebrá-la é depois de uma vírgula ou um operador. Dê uma olhada no exemplo abaixo:

```
document.getElementById("demo").innerHTML =
"Hello Dolly.;"
```

Sempre coloque espaços entre os operadores (= + - * /) em seu código JavaScript porque isso torna a aparência agradável e fácil de ler. Se você fizer isso como fizemos no exemplo abaixo, tudo bem.

```
var x = y + z;  
var values = ["Volvo", "Saab", "Fiat"];
```

A maioria dos servidores web (apache...) diferencia maiúsculas de minúsculas em nomes de arquivos. Por outro lado, alguns servidores web (Microsoft) não diferenciam maiúsculas de minúsculas em nomes de arquivos. Então, digamos que você mudou do servidor Apache para o servidor Microsoft, onde estava usando letras maiúsculas para nomes de arquivo. O Apache é sensível a maiúsculas e minúsculas, por isso pode corromper seu site. Para evitar essa confusão, sempre use nomes de arquivo em minúsculas (se possível).

Existem algumas convenções de codificação que você precisa considerar ao escrever suas instruções JavaScript:

- Sempre termine uma instrução simples com um ponto e vírgula.
- Coloque o colchete de abertura no final da primeira linha.
- Use um espaço antes do colchete de abertura.
- Coloque o colchete de fechamento em uma nova linha, sem espaços à esquerda.

```
var values = ["Volvo", "Saab", "Fiat"];  
  
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};
```

```
if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

Como você pode ver, seguir as convenções de codificação em JavaScript torna nosso código sustentável e fácil de ler por outros desenvolvedores. Eu só queria listar essas regras simples, porque todo desenvolvedor de JavaScript deve considerá-las. Então é isso neste artigo, se você está usando todas essas convenções, ótimo, você está se tornando um bom desenvolvedor, ou talvez já seja um bom desenvolvedor de JavaScript.

3.3 var

O variable statement declara uma variável, opcionalmente é possível atribuir à ela um valor em sua inicialização.

- **Sintaxe**

```
var varname1 [= value1 [, varname2 [, varname3 ... [, varnameN]]]];
```

varnameN

Nome da variável. Pode ser utilizado qualquer identificador legal.

valueN

Valor inicial da variável. Pode ser qualquer expressão legal.

- **Descrição**

Declarações de variáveis, onde quer que elas ocorram, são processadas antes que qualquer outro código seja executado. O escopo de uma variável declarada com var é seu contexto atual em execução, o qual é a função a qual pertence ou, para variáveis declaradas fora de qualquer função, o escopo é o global.

Atribuir um valor a uma variável não declarada anteriormente implica em criar uma variável global (ela se torna uma propriedade do objeto global) quando a atribuição é executada. As diferenças entre uma variável declarada e uma não declarada são:

1. Variáveis declaradas estão relacionadas com o contexto de execução quando elas são criadas (por exemplo, uma função, objeto). Por outro lado, as variáveis não declaradas sempre são globais.

- 1.** Variáveis declaradas estão relacionadas com o contexto de execução quando elas são criadas (por exemplo, uma função, objeto). Por outro lado, as variáveis não declaradas sempre são globais.

```
function x() {
    y = 1; // Lança a exceção ReferenceError em modo restrito (strict mode)
    var z = 2;
}
x();
console.log(y); // logs "1"
console.log(z); // Lança a exceção ReferenceError: z não foi definida fora
da função x()
```

- 2.** Variáveis declaradas são criadas antes de qualquer código ser executado. As variáveis não declaradas não existem até quando o código atribui um valor a ela.

```
console.log(a); // Lança a exceção ReferenceError.
console.log('still going...'); // Nunca será executado.
var a;
console.log(a); // mostra "undefined" ou "dependendo do
navegador".
console.log('still going...'); // mostra "still going...".
```

- 3.** Variáveis declaradas são propriedades não configuráveis no contexto de execução (função ou global). Variáveis não declaradas são configuráveis (por exemplo, podem ser excluídas).

```
var a = 1;
b = 2;
```

`delete this.a;` // Lança a exceção TypeError em modo restrito(strict mode). Caso contrário, falha silenciosamente.

```
delete this.b;
console.log(a, b); // Throws a ReferenceError.
```

// A propriedade 'b' foi deletada e não existe mais.

Por conta dessas três diferenças, falha para declarar variáveis, muito provavelmente, levar a resultados inesperados. Então, é recomendado sempre declarar as variáveis, independentemente se as variáveis estão em escopo de função ou escopo global. E o modo restrito (strict mode) do ECMAScript 5 sempre lançará uma exceção quando o código atribuir um valor a uma variável não declarada.

var hoisting

Como as declarações de variáveis (e declarações em geral) são processadas antes de qualquer código seja executado, declarar uma variável em qualquer lugar no código é equivalente a declarar no início. Isso também significa que uma variável pode aparecer para ser usada antes dela ser declarada. Esse comportamento é chamado de "hoisting", a variável é movida para o início da função ou do código global.

```
bla = 2
var bla;
// ...
// é implicitamente entendido como:
var bla;
bla = 2;
```

Por essa razão, recomenda-se sempre declarar variáveis na parte superior do seu escopo de aplicação (o topo do código global e a parte superior do código da função). Por isso, é claro que as variáveis são função de escopo (local) e que são resolvidos na cadeia de escopo.

Exemplos:

Declarando e inicializando duas variáveis

```
var a = 0, b = 0;
```

• Atribuindo duas variáveis com uma única expressão

```
var a = "A";
var b = a;
// Equivalente a:
var a, b = a = "A";
É sempre importante lembrar da ordem da declaração das variáveis:
var x = y, y = 'A';
console.log(x + y); // undefinedA
```

Então, x e y são declarados antes de qualquer código seja executado, a atribuição ocorre posteriormente. Quando "x = y" for executado, y existe e nenhuma exceção ReferenceError é lançada, e o valor de y será considerado como 'undefined'. Por este motivo, este valor é atribuído a x. Depois disso, o valor 'A' é atribuído a variável y. Consequentemente, depois da primeira linha, x === undefined & y === 'A', então o resultado.

- Iniciando diversas variáveis

```
var x = 0;
function f(){
    var x = y = 1; // x é declarado localmente, y não é!
}
f();
console.log(x, y); // 0, 1
// x é uma variável global como esperado
// y vazou para fora da função!
```

- Variável global implícita e fora do escopo da função

Variáveis que aparecem como variáveis globais implícitas podem ser referenciadas como variáveis fora do escopo da função:

```
var x = 0; // x é declarada como global e é igual a 0

console.log(typeof z); // undefined, desde que z não tenha sido criada
anteriormente

function a() { // quando a for chamada,
    var y = 2; // y é declarada como local desta função, e o valor 2 é
atribuido

    console.log(x, y); // 0 2

    function b() { // quando b for chamado,
        x = 3; // atribui o valor 3 a variável global existente, ele não cria
uma nova variável global
        y = 4; // atribui o valor 4 a uma variável fora, ele não cria uma nova
variável
        z = 5; // cria uma nova variável global e atribui o valor 5.
    } // (Lança a exceção ReferenceError em modo restrito.)

    b(); // chamando b, o código irá criar z como variável global
    console.log(x, y, z); // 3 4 5
}

a(); // chamando a, também irá chamar b
console.log(x, z); // 3 5
console.log(typeof y); // indefinido já que y é u
```

3.4 Case sensitive

JavaScript é uma linguagem que diferencia maiúsculas de minúsculas. Isso significa que as palavras-chave do idioma, variáveis, nomes de funções e quaisquer outros identificadores devem sempre ser digitados com letras maiúsculas consistentes.

Assim, os identificadores Time e TIME terão significados diferentes em JavaScript.

 // NOTA:

Deve-se tomar cuidado ao escrever nomes de variáveis e funções em JavaScript.

Exemplo:

O exemplo a seguir mostra que JavaScript é uma linguagem que diferencia maiúsculas de minúsculas:

```
<!DOCTYPE html>
<html>
  <body>
    <h3>My favorite subject</h3>
    <p id="demo"></p>
    <script>
      var subject, Subject;
      subject = "Java";
      Subject = "Maths";
      document.getElementById("demo").innerHTML = subject;
    </script>
  </body>
</html>
```

3.5 Tipagem

JavaScript é uma linguagem de tipagem dinâmica. Isso significa que você não necessita declarar o tipo de uma variável antes de sua atribuição. O tipo será automaticamente determinado quando o programa for processado. Isso também significa que você pode atribuir uma mesma variável com um tipo diferente:

```
var foo = 42; // foo é um Number agora
foo = "bar"; // foo é um String agora
foo = true; // foo é um Boolean agora
```

- **Tipos de Dados**

A última versão ECMAScript define sete tipos de dados:

Sete tipos de dados são: primitives:

```
Boolean
Null
Undefined
Number
BigInt
String
Symbol
e Object
```

- **Tipo "Null"**

O tipo Null tem exatamente um valor: null(nulo). Veja null e Null para mais detalhes.

- **Tipo "Undefined"**

Uma variável que não foi atribuída a um valor específico, assume o valor undefined (indefinido). Veja undefined e Undefined para mais detalhes.

- **Tipo "Number"**

De acordo com os padrões ECMAScript, existe somente um tipo numérico. O double-precision 64-bit binary format IEEE 754 value (número entre -(2⁵³ -1) e 2⁵³ -1). Não existe um tipo específico para inteiros. Além de poderem representar números de ponto-flutuante, o tipo number possui três valores simbólicos: +Infinity, -Infinity, e NaN (não numérico).

Para verificar o maior ou o menor valor disponível dentro de +/-Infinity, você pode usar as constantes Number.MAX_VALUE ou Number.MIN_VALUE, e a partir do ECMAScript 6, você também consegue verificar se um número está dentro da região de um ponto flutuante do tipo double-precision, usando Number.isSafeInteger(), como também Number.MAX_SAFE_INTEGER, e Number.MIN_SAFE_INTEGER. Fora dessa região, números inteiros em JavaScript não são mais precisos e serão uma aproximação de um número de ponto flutuante do tipo double-precision.

O tipo number possui apenas um inteiro que tem duas representações: 0 é representado como -0 ou +0. ("0" é um pseudônimo para +0). Na prática, isso não gera grandes impactos. Por exemplo +0 === -0 resulta em true.

Entretanto, você pode notar a diferença quando realiza uma divisão por 0:

```
> 42 / +0  
Infinity  
> 42 / -0  
-Infinity
```

Apesar de um número frequentemente representar somente seu valor, JavaScript fornece alguns operadores binários. Estes podem ser usados para representar muitos valores booleanos dentro de um único número usando bit masking. Entretanto, isto é usualmente considerado uma má prática, desde que JavaScript oferece outros meios para representar uma configuração de booleanos (como uma array de booleanos ou um objeto com valores booleanos assinalados em propriedades). Bit masking também tende a fazer o código mais difícil de ler, entender e de realizar manutenção. Isto pode ser necessário em um ambiente bastante limitado, como quando tentamos lidar com limitação de armazenamento ou armazenamento local ou em casos extremos quando cada bit na rede conta. Esta técnica somente deveria ser considerada quando é a última medida possível para otimizar o tamanho.

• Tipo "String"

O tipo String em JavaScript é usado para representar dados textuais. Isto é um conjunto de "elementos" de valores de 16-bits unsigned integer. Cada elemento na string ocupa uma posição na string. O primeiro elemento está no índice 0, o próximo no índice 1, e assim por diante. O comprimento de uma string é o número de elementos nela

Diferente de linguagens como C, strings em JavaScript são imutáveis. Isto significa que uma vez criada a string, não é possível modificá-la. Entretanto, ainda é possível criar outra string baseada em um operador na string original. Por exemplo:

Uma substring da original a partir de letras individuais ou usando String.substr().

Uma concatenação de duas strings usando o operador (+) ou String.concat().

Cuidado com "stringly-typing" (digitação no seu código!)

Pode ser tentador utilizar strings para representar dados complexos. Fazer isso traz os seguintes benefícios de curto prazo:

É fácil construir strings complexas com concatenação.

Strings são fáceis para debug (o que você vê escrito é o que está na string).

Strings são o denominador comum entre várias APIs (input fields, local storage values, XMLHttpRequest responses ao usar responseText, etc.) e pode ser tentador trabalhar apenas com strings.

Com convenções, é possível representar qualquer estrutura de dados com uma string. Ainda assim, isso não faz a prática ser uma boa ideia. Por exemplo, pode-se emular uma lista utilizando separadores (um JavaScript array seria mais adequado). Infelizmente, se o separador faz parte de um dos elementos da "lista", então, a lista é quebrada. Um caractere de escape pode ser utilizado, etc. Tudo isso requer convenções e cria uma carga de manutenção desnecessária.

É aconselhável usar strings para dados textuais. Quando representar dados complexos, analise as strings e utilize abstrações apropriadas.

- **Symbol type**

Symbols são novos no JavaScript ECMAScript edição 6. Um Symbol é um valor primitivo único e imutável e pode ser usado como chave de uma propriedade de Object (ver abaixo). em algumas linguagens de programação, Symbols são chamados de atoms (átomos). Você também pode compará-los à enumerações nomeadas (enum) em C. Para mais detalhes veja Symbol e o Symbol object wrapper em JavaScript.

- **Objetos**

Na ciência da computação, um objeto é um valor na memória que pode ser possivelmente referenciado por um identifier.

- **Propriedades**

No JavaScript, objetos podem ser vistos como uma coleção de propriedades. Com o object literal syntax, um conjunto limitado de propriedades podem ser inicializados; a partir daí propriedades podem ser adicionadas e removidas. Estas propriedades podem assumir valores de qualquer tipo, incluindo outros objetos, o que permite construir estruturas de dados mais complexas. Propriedades são identificadas usando valores chave. Um valor chave pode ser uma String ou um valor Symbol.

Existem dois tipos de propriedades de objetos que contém certos atributos: a propriedade de dados e a propriedade de acesso.

3.6 Interagindo (prompt)

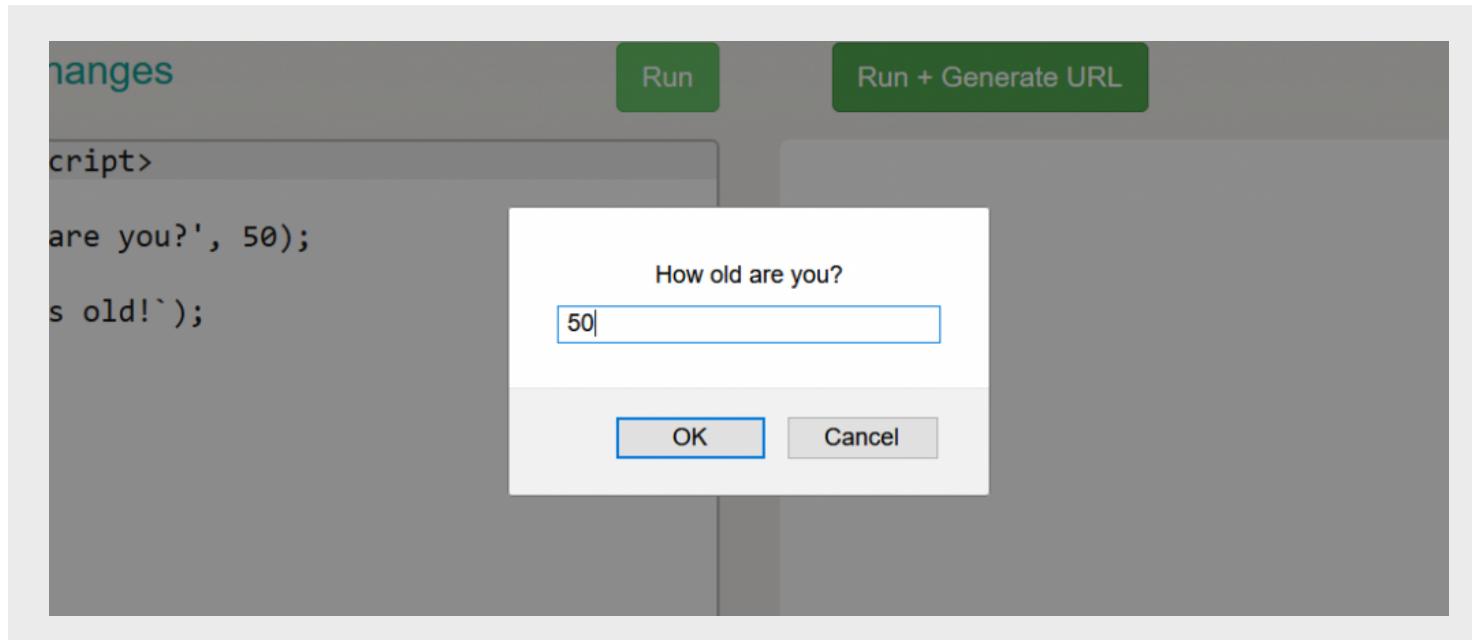
Prompt é outra função da interface do usuário que normalmente contém dois argumentos.

```
prompt ('texto', valor padrão);
```

O texto é basicamente o que você deseja mostrar ao usuário e o argumento do valor padrão é opcional, embora atue como um espaço reservado dentro de um campo de texto. É a interface mais usada, pois com ela você pode pedir ao usuário para inserir algo e então usar essa entrada para construir algo.

Exemplo: (com parâmetro padrão)

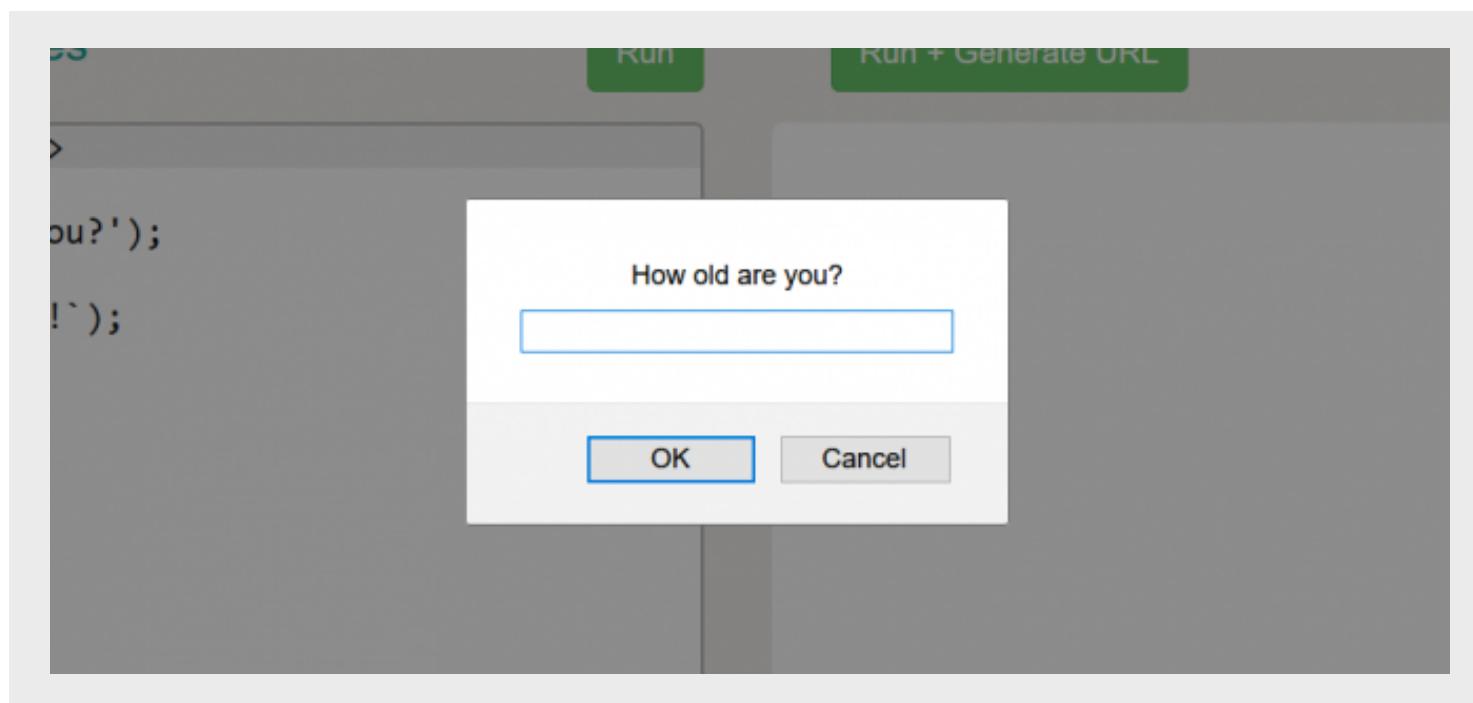
```
<script>
// prompt example
let age = prompt('How old are you?', 50);
alert(`You are ${age} years old!`);
</script>
```



Você pode inserir qualquer coisa e será impresso, não necessariamente tem que ser um número. Sem o valor padrão, você deve inserir algo no campo de texto, caso contrário, ele imprimirá um espaço em branco de forma simples.

Exemplo:

```
<script>
// prompt example
let age = prompt('How old are you?');
alert(`You are ${age} years old!`);
</script>
```

**Confirmar**

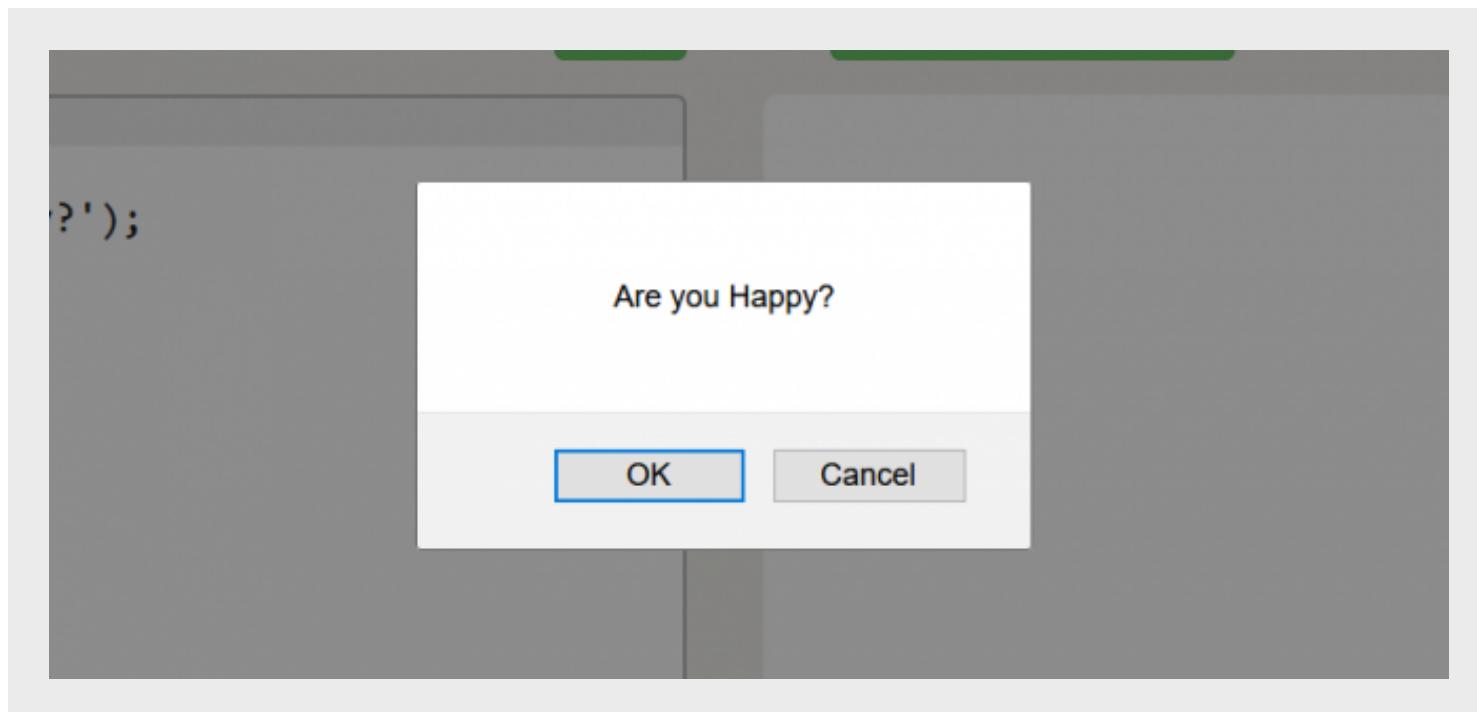
A função de confirmação basicamente emite uma janela modal com uma pergunta e dois botões 'OK' e 'CANCELAR'.

```
confirmar ('pergunta');
```

Exemplo:

```
<script>
// confirm example
let isHappy = confirm('Are you Happy?');
alert(`You are ${isHappy}`);
</script>
```

Saída:



Será impresso verdadeiro ou falso com base na sua escolha de clicar no botão 'OK' ou no botão 'CANCELAR', respectivamente.

4.0 Hora da prática

- Utilize os conhecimentos adquiridos e construa o cálculo do IMC.
- Utilize os aprendizados deste módulo e ajuste a calculadora produzida no módulo anterior.

5.1 Function

Function cria um novo objeto Function. Chamar o construtor diretamente pode criar funções dinamicamente, mas sofre com problemas de segurança e desempenho semelhante (mas muito menos significativo) a eval. No entanto, diferentemente de eval, a Função construtora cria funções que executam somente no escopo global.

- **Sintaxe**

```
new Function ([arg1[, arg2[, ...argN]],] functionBody)
Parâmetros
arg1, arg2, ... argN
```

Nomes para serem usados pela função como nomes formais de argumentos. Cada um deve ser uma string que corresponde para uma válida identidade JavaScript ou uma lista de certas strings separadas com uma vírgula; por exemplo "x", "theValue". ou "a,b".

- **FunctionBody**

Uma string que contém as instruções JavaScript que compõem a definição da função.

- **Descrição**

Objetos Function criados com o construtor Function são parseados quando a função é criada. Isto é menos eficiente que criar com uma expressão de função ou um declaração de função e chamando-a dentro do seu código, porque tais funções são parseadas com o resto do código.

Todos os argumentos passados para a função são tratados como os nomes dos identificadores dos parâmetros na função a ser criada, na mesma ordem na qual eles foram passados.



// NOTA:

Funções criadas com o construtor Function não criam closures para o seu contexto de criação; elas sempre são criadas no escopo global. Quando executadas, elas terão acesso apenas às suas variáveis locais ou globais, não terão acesso às variáveis do escopo na qual o construtor Function foi chamado. Isto é diferente de usar eval com o código de uma expressão de função.

Invocar o construtor Function como uma função (sem usar o operador new) tem o mesmo efeito de chamá-la como um construtor.

- **Propriedades e Métodos da Function**

O objeto global Function não tem métodos ou propriedades próprias, no entanto, como ela é uma função, ela herda alguns métodos e propriedades através do prototype chain do Function.prototype (en-US).

- **Function instances**

Function instances inherit methods and properties from Function.prototype (en-US). As with all constructors, you can change the constructor's prototype object to make changes to all Function instances.

Exemplos:

Exemplos: Especificando argumentos com o construtor Function

```
O código a seguir cria um objeto Function que recebe dois argumentos.
// O exemplo pode ser executado direto no seu console JavaScript
// Cria uma função que recebe 2 argumentos e retorna a soma entre os dois:
var adder = new Function('a', 'b', 'return a + b');
// Chamada da função
adder(2, 6);
// > 8
```

Os argumentos "a" e "b" são os argumentos que serão usados no corpo da função, "return a + b".

Exemplo:

Um atalho recursivo para modificar o DOM em massa

Creating functions with the Function constructor is one of the ways to dynamically create an indeterminate number of new objects with some executable code into the global scope from a function. The following example (a recursive shortcut to massively modify the DOM) is impossible without the invocation of the Function constructor for each new query if you want to avoid closures.

```
<!doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>MDN Example - a recursive shortcut to massively modify the DOM</title>
<script type="text/javascript">
var domQuery = (function() {
  var aDOMFunc = [
    Element.prototype.removeAttribute,
    Element.prototype.setAttribute,
    CSSStyleDeclaration.prototype.removeProperty,
    CSSStyleDeclaration.prototype.setProperty
  ];

  function setSomething(bStyle, sProp, sVal) {
    var bSet = Boolean(sVal), fAction = aDOMFunc[bSet | bStyle << 1],
      aArgs = Array.prototype.slice.call(arguments, 1, bSet ? 3 : 2),
      aNodeList = bStyle ? this.cssNodes : this.nodes;
    if (bSet && bStyle) { aArgs.push(''); }
    if (bSet) { aArgs[0] = aNodeList; }
    if (fAction) { fAction.apply(aNodeList, aArgs); }
  }
  return setSomething;
})()
```

```

for (
    var nItem = 0, nLen = this.nodes.length;
    nItem < nLen;
    fAction.apply(aNodeList[nItem++], aArgs)
);
this.follow = setSomething.caller;
return this;
}
function setStyles(sProp, sVal) { return setSomething.call(this, true,
sProp, sVal); }
function setAttribs(sProp, sVal) { return setSomething.call(this, false,
sProp, sVal); }
function getSelectors() { return this.selectors; }
function getNode() { return this.nodes; }
return (function(sSelectors) {
var oQuery = new Function('return
arguments.callee.follow.apply(arguments.callee, arguments);');
oQuery.selectors = sSelectors;
oQuery.nodes = document.querySelectorAll(sSelectors);
oQuery.cssNodes = Array.prototype.map.call(oQuery.nodes,
function(oInlineCSS) { return oInlineCSS.style; });
oQuery.attributes = setAttribs;
oQuery.inlineStyle = setStyles;
oQuery.follow = getNode;
oQuery.toString = getSelectors;
oQuery.valueOf = getNode;
return oQuery;
});
})();
</script>
</head>
<body>
<div class="testClass">Lorem ipsum</div>
<p>Some text</p>
<div class="testClass">dolor sit amet</div>
<script type="text/javascript">
domQuery('.testClass')
    .attributes('lang', 'en')('title', 'Risus abundat in ore stultorum')
    .inlineStyle('background-color', 'black')('color', 'white')('width',
'100px')('height', '50px');
</script>
</body>
</html>

```

5.2 Chamada de função

A definição de uma função não a executa. Definir a função é simplesmente nomear a função e especificar o que fazer quando a função é chamada. Chamar a função executa realmente as ações especificadas com os parâmetros indicados. Por exemplo, se você definir a função square, você pode chamá-la do seguinte modo:

```
square(5);
```

A declaração anterior chama a função com o argumento 5. A função executa as instruções e retorna o valor 25.

Funções devem estar no escopo quando são chamadas, mas a declaração de uma função pode ser puxada para o topo (aparecem abaixo da chamada no código), como neste exemplo:

```
console.log(square(5));
/* ... */
function square(n){return n*n}
```

O escopo de uma função é a função na qual ela é declarada, ou todo o programa se ela é declarada no nível superior.



// NOTA:

*Nota: Isso funciona apenas quando a definição da função usa a sintaxe acima (ex., **function funcNome(){ }**). O código a seguir não vai funcionar.*

```
console.log(square(5));
var square = function (n) {
  return n * n;
}
```

Os argumentos de uma função não estão limitados a strings e números. Você pode passar objetos para uma função. A função **show_props** (definido em Trabalhando com Objetos) é um exemplo de uma função que recebe um objeto como um argumento.

Uma função pode chamar a si mesma. Por exemplo, a função que calcula os fatoriais recursivamente:

```
function factorial(n){
  if ((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
}
```

Você poderia, então, calcular os fatoriais de um a cinco:

```
var a, b, c, d, e;
a = factorial(1); // a recebe o
valor 1
b = factorial(2); // b recebe o
valor 2
c = factorial(3); // c recebe o
valor 6
d = factorial(4); // d recebe o
valor 24
e = factorial(5); // e recebe o
valor 120
```

Há outras maneiras de chamar funções. Muitas vezes há casos em que uma função precisa ser chamada dinamicamente, ou o número de argumentos de uma função varia, ou em que o contexto da chamada de função precisa ser definido para um objeto específico determinado em tempo de execução.

5.3 Convenção

Ao ter muitos membros trabalhando em um projeto, é necessário ter um certo padrão a ser seguido por uma questão de escalabilidade.

Hoje, vamos falar sobre o padrão mais básico, as convenções de nomenclatura: o que devemos e o que não devemos fazer.

Vamos entrar nisso.

- **Use nomes específicos**

```
// Don't
function fetchData() {}
// Do
function fetchUsers() {}
```

- **Evite nomes de uma letra**

Exceto pelos nomes comuns, como para index instruções de loop, você deve evitar nomes de uma letra porque isso o confundirá mais tarde.

```
// Don't
let n = 0;
// Do
let number = 0;
```

Muitas vezes, buscamos nomes curtos. Em alguns casos, no entanto, para ser mais claro e descriptivo, um nome longo não deve ser um problema.

// Don't

```
function findBooks() {}
```

// Do

```
function findBooksByAuthor() {}
```

Escrever um comentário apenas para uma definição de variável é mais provavelmente redundante. O nome de uma variável deve ser descriptivo para que, quando você olhe para ele, saiba exatamente para que é usado.

```
// Don't
let d = '01/01/2021';
let number = 28;
// Do
let date = '01/01/2021';
let age = 28;
```

```
// Don't
let book_title = 'JavaScript';
let BOOKTITLE = 'JavaScript';
let booktitle = 'JavaScript';
// Do
let bookTitle = 'JavaScript';
```

Constantes em JavaScript não podem ser alteradas. Para separar de outras variáveis, todas as letras em uma constante devem ser **MAIÚSCULAS**.

```
// Don't
const pi = 3.14;
// Do
const PI = 3.14;
Um booleano é como uma pergunta
sim / não:
Está lido?
Você fez?
São iguais?
// Don't
let decoration = true;
let red = true;
let visible = false;
// Do
let hasDecoration = true;
let isRed = true;
let isVisible = false;
```

Você chama uma função quando deseja realizar uma ação. Portanto, um nome de função deve ser iniciado por um verbo.

```
// Don't
function data() {}
// Do
function fetchData() {}
```

```
// Don't
function getUsers() {}
function fetchBooks() {}
function retriveAuthors() {}
// Do
function fetchUsers() {}
function fetchBooks() {}
function fetchAuthors() {}
```

Como outras linguagens de programação, nomear um nome de classe deve seguir PascalCase e um nome de classe deve ser um substantivo.

```
// Don't
class mobileDevice {}
// Do
class MobileDevice {}
```

Para arquivos JavaScript, temos dois estilos comuns de nomenclatura: **kebab-case** e **PascalCase**.

kebab-case:

```
-components/
--product/
---product-details.js
---product-list.js
-components/
--product/
---ProductDetails.js
---ProductList.js
```

5.4 Indentação

A indentação é um recurso fundamental para a legibilidade de um código, pois adiciona informação importante, do tipo: qual bloco de código pertence a uma função ou método? Ela também ajuda a entender a hierarquia de elementos em código HTML e XML. Acredito que todo programador já espera encontrar algum tipo de indentação quando vai ler qualquer código.

Basicamente indentar é um termo utilizado para escrever o código do programa de forma hierárquica, facilitando assim a visualização e o entendimento do programa. O exemplo abaixo mostra um código indentado.

```
if (a) {  
    if (b) {  
        while (c) {  
            d();  
        }  
    } else if (e) {  
        f();  
    } else if (g) {  
        h();  
    }  
} else if (i) {  
    while (j) {  
        k();  
    }  
}
```

agora o mesmo código sem indentação:

```
if (a) {  
  if (b) {  
    while (c) {  
      d();  
    }  
    } else if (e) {  
      f();  
    } else if (g) {  
      h();  
    }  
    } else if (i) {  
      while (j) {  
        k();  
      }  
    }
```

• Qual é sua importância?

A indentação é bastante importante. Para pessoas que seguem profissões de programação há que ter em conta que não estamos a trabalhar sozinhos e que outras pessoas poderão ter de via a pegar no nosso trabalho. Quem diz a indentação diz o uso de comentários.

Algumas empresas grandes como a Google/Microsoft têm até regras de organização de código para diferentes projectos. Estas regras incluem indentação/comentários/organização do código em geral etc.

5.5 Funções parametrizadas

As funções em JavaScript são parametrizadas: uma definição de função pode incluir uma lista de identificadores, conhecidos como parâmetros, que funcionam como variáveis locais para o corpo da função. Frequentemente, as funções utilizam seus valores de argumentos para calcular um valor de retorno, que se torna o valor da expressão da chamada função. Além dos argumentos, cada chamada tem outro valor - o contexto da chamada -, que é o valor da palavra-chave **"this"**. Além dos parâmetros declarados, toda função recebe dois adicionais: **this** e **arguments**. O parâmetro **this** é muito importante na programação orientada a objeto, seu valor é determinado pelo padrão de invocação.

Há quatro padrões de invocação em JavaScript:

- método**
- função**
- construtor**
- aplicação**

O operador de invocação é um para de **parênteses ()** que é posto em seguida de qualquer expressão que produza um valor que seja uma função. Os parênteses podem conter zero ou mais expressões, separadas por vírgula. Cada uma delas produz um valor de argumento. Cada um desses valores será atribuído aos nomes de parâmetro da função. Não há erro no tempo de execução quando o número de argumentos e de parâmetros não coincidem. Se os valores informados excederem os argumentos, os valores excedentes serão ignorados. Se for o inverso, **'undefined'** será atribuído aos valores faltantes. N

Não há checagem de tipo em valores de argumento: qualquer tipo de valor pode ser passado a qualquer parâmetro.

Veja um exemplo:

// Função que soma um monte de coisas.

```
var sum = function () {  
    var i, sum = 0;  
    for (i = 0; i <  
        arguments.length; i += 1) {  
        sum += arguments[i];  
    }  
    return sum;  
};  
console.log(sum(4, 8, 15, 16, 23,  
42)); // 108
```

5.6 Comentários

Um comentário é uma parte de código que não é interpretada pelo navegador. Pode ser utilizada, também, para colocar textos que ajudam a identificar o que está sendo feito naquele trecho de código. Pensando num trabalho em equipe, esta é uma excelente prática.

Existem dois tipos de comentários:

- **Barra dupla:** serve para comentar uma linha de código.

```
<script>
  // window.alert("Este alerta não será mostrado");
</script>
```

- **O outro comentário pode ser utilizado para comentar várias linhas.**

deve-se iniciar com /* e terminar com */.

```
<script>
  /* Exemplo de comentário para
  várias linhas.
  Basta escolher o início e fim. */
</script>
```

5.7 Retorno de funções (return)

Algumas funções não retornam um valor significativo após a conclusão, mas outras o fazem, e é importante entender quais são seus valores, como utilizá-los em seu código e como fazer com que suas próprias funções personalizadas retornem valores úteis. Nós vamos cobrir tudo isso abaixo.

• Quais são os valores de retorno?

Valores de retorno são exatamente como soam — valores retornados pela função quando são concluídos. Você já conheceu os valores de retorno várias vezes, embora possa não ter pensado neles explicitamente. Vamos voltar para algum código familiar:

```
var myText = 'I am a string';
var newString =
myText.replace('string',
'sausage');
console.log(newString);
// the replace() string function
takes a string,
// replaces one substring with
another, and returns
// a new string with the
replacement made
```

Estamos invocando a função `replace()` na string `myText` e passando a ela dois parâmetros — a `substring` para localizar e a `substring` para substituí-la. Quando essa função é concluída (termina a execução), ela retorna um valor, que é uma nova string com a substituição feita. No código acima, estamos salvando esse valor de retorno como o valor da variável `newString`.

Se você observar a página de referência **MDN** da função de substituição, verá uma seção chamada Valor de retorno. É muito útil conhecer e entender quais valores são retornados por funções, portanto, tentamos incluir essas informações sempre que possível.

Algumas funções não retornam um valor de retorno como tal (em nossas páginas de referência, o valor de retorno é listado como `void` ou `undefined` em tais casos). Por exemplo, na função `displayMessage()` que criamos no artigo anterior, nenhum valor específico é retornado como resultado da função que está sendo chamada. Apenas faz uma caixa aparecer em algum lugar na tela — é isso!

Geralmente, um valor de retorno é usado onde a função é uma etapa intermediária em um cálculo de algum tipo. Você quer chegar a um resultado final, que envolve alguns valores. Esses valores precisam ser calculados por uma função, que retorna os resultados para que possam ser usados no próximo estágio do cálculo.

• Usando valores de retorno em suas próprias funções

Para retornar um valor de uma função personalizada, você precisa usar ... aguarde por isso ... a palavra-chave `return`. Vimos isso em ação recentemente em nosso exemplo `random-canvas-circles.html`. Nossa função `draw()` desenha 100 círculos aleatórios em algum lugar em um HTML `<canvas>`:

```
function draw() {
    ctx.clearRect(0,0,WIDTH,HEIGHT);
    for (var i = 0; i < 100; i++) {
        ctx.beginPath();
        ctx.fillStyle = 'rgba(255,0,0,0.5)';
        ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 *
Math.PI);
        ctx.fill();
    }
}
```

Dentro de cada iteração de loop, três chamadas são feitas para a função **random()**, para gerar um valor aleatório para a coordenada x do círculo atual, coordenada y e raio, respectivamente. A função **random()** recebe um parâmetro — um número inteiro — e retorna um número aleatório inteiro entre 0 e esse número. Se parece com isso:

```
function randomNumber(number) {
    return Math.floor(Math.random()*number);
}
```

Isso pode ser escrito da seguinte maneira:

```
function randomNumber(number) {
    var result = Math.floor(Math.random()*number);
    return result;
}
```

Mas a primeira versão é mais rápida de escrever e mais compacta.

Estamos retornando o resultado do cálculo **Math.floor(Math.random()*number)** cada vez que a função é chamada. Esse valor de retorno aparece no ponto em que a função foi chamada e o código continua. Então, por exemplo, se nós rodássemos a seguinte linha:

```
ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
```

e as três chamadas **random()** retornaram os valores 500, 200 e 35, respectivamente, a linha seria executada como se fosse isso:

```
ctx.arc(500, 200, 35, 0, 2 * Math.PI);
```

As chamadas de função na linha são executadas primeiro e seus valores de retorno substituem as chamadas de função, antes que a própria linha seja executada.

- **Aprendizagem ativa:** nossa própria função de valor de retorno

Vamos escrever nossas próprias funções com valores de retorno.

Primeiro de tudo, faça uma cópia local do arquivo [function-library.html](#) do GitHub. Esta é uma página HTML simples contendo um campo de texto `<input>` e um parágrafo. Há também um elemento `<script>` no qual armazenamos uma referência a ambos os elementos HTML em duas variáveis. Esta pequena página permitirá que você insira um número na caixa de texto e exiba diferentes números relacionados a ele no parágrafo abaixo.

Vamos adicionar algumas funções úteis para este elemento `<script>`. Abaixo das duas linhas existentes de JavaScript, adicione as seguintes definições de função:

```
function squared(num) {  
    return num * num;  
}  
  
function cubed(num) {  
    return num * num * num;  
}  
  
function factorial(num) {  
    var x = num;  
    while (x > 1) {  
        num *= x-1;  
        x--;  
    }  
    return num;  
}
```

As funções `squared()` e `cubed()` são bastante óbvias — elas retornam o quadrado ou cubo do número dado como um parâmetro. A função `factorial()` retorna o fatorial do número fornecido.

Em seguida, vamos incluir uma maneira de imprimir informações sobre o número digitado na entrada de texto. Digite o seguinte manipulador de eventos abaixo das funções existentes:

```
input.onchange = function() {
    var num = input.value;
    if (isNaN(num)) {
        para.textContent = 'You need to enter a number!';
    } else {
        para.textContent = num + ' squared is ' + squared(num) + '. ' +
                           num + ' cubed is ' + cubed(num) + '. ' +
                           num + ' factorial is ' + factorial(num) + '.';
    }
}
```

Aqui estamos criando um manipulador de eventos onchange que é executado sempre que o evento de mudança é acionado na entrada de texto — ou seja, quando um novo valor é inserido na entrada de texto e enviado (insira um valor e pressione tab por exemplo). Quando essa função anônima é executada, o valor existente inserido na entrada é armazenado na variável **num**.

Em seguida, fazemos um teste condicional — se o valor inserido não for um número, imprimimos uma mensagem de erro no parágrafo. O teste analisa se a expressão **isNaN(num)** retorna true. Usamos a função **isNaN()** para testar se o valor num não é um número — se for, retorna true, e, se não, false. Se o teste retorna false, o valor num é um número, então imprimimos uma frase dentro do elemento de parágrafo informando o que é o quadrado, o cubo e o fatorial do número. A sentença chama as funções **squared()**, **cubed()**, e **factorial()** para obter os valores necessários.

- **Salve seu código, carregue-o em um navegador e experimente.**

Analisamos outro exemplo de como escrever erros em nossas funções. Geralmente, é uma boa ideia verificar se os parâmetros necessários foram fornecidos e, no tipo de dados correto, e se eles são opcionais, que algum tipo de valor padrão é fornecido para permitir isso. Desta forma, o seu programa terá menos probabilidade de lançar erros.

Pense na ideia de criar uma biblioteca de funções. À medida que você avança na sua carreira de programação, você começará a fazer o mesmo tipo de coisa uma e outra vez. É uma boa idéia começar a manter sua própria biblioteca de funções utilitárias que você usa com muita frequência — você pode então copiá-las para o seu novo código, ou até mesmo aplicá-las a qualquer página HTML onde você precisar.

5.8 Interagindo (prompt)

Permite abrir uma caixa de diálogo para entrada de dados

Pertence ao objeto window

Sintaxe:

```
prompt("arg01", "[arg02]");
```

Onde:

arg01 é uma mensagem de instrução direcionada ao usuário

arg02 é um valor padrão, geralmente usado para fornecer uma dica ao usuário. É opcional.

Os argumentos devem estar sempre entre aspas, simples ou duplas. Se o usuário clicar no botão OK, retorna os dados digitados na caixa; se clicar em Cancelar, retornará o valor null

Exemplo do método prompt():

```
<script type="text/javascript">
    var a = prompt("Digite seu nome:", "Nome");
    var b = prompt("Digite seu sobrenome:");
    document.write("Bom dia, " + a + " " + b + " <br />");
</script>
```

5.9 parseInt

Em JavaScript temos uma função chamada **parseInt()**, que vai, em outras palavras, analisar a string e retornar um número inteiro:

```
const salario = pegaSalarioDoFuncionario();
const salarioFormatado = parseInt(salario);
const aumento = 20;
const novoSalario = ( salario * aumento)/100) + salarioFormatado;
console.log( "Seu novo salário é: " + novoSalario );
```

Como cálculos envolvendo salário geralmente utilizam representação de pontos flutuantes, isto é, contém casas decimais, podemos utilizar a função **parseFloat()** que vai justamente analisar a string e retornar um número com representação decimal, ou seja, com ponto flutuante.

```
const salario = pegaSalarioDoFuncionario();
const aumento = 0.9;
const salarioFormatado = parseFloat(salario);
const novoSalario = ( salario * aumento)/100) + salarioFormatado;
console.log( "Seu novo salário é: " +novoSalario );
```

Podemos perceber que dependendo do tipo de entrada que a função recebe ela pode mudar de comportamento e afetar a nossa lógica dentro do sistema.

Tendo em vista que em algumas vezes iremos capturar dados dos usuários para realizar alguma operação de adição, temos que lembrar que em muitas dessas vezes esses inputs virão em formato de string.

Portanto, para não correr o risco de acabar concatenando esses dados ao invés de realizar uma soma, é sempre importante lembrar de utilizar as funções **parseInt()** ou **parseFloat()** dependendo do tipo de número que será utilizado para realizar as operações.

6.0 Hora da prática

Com os conhecimentos adquiridos melhore seu sistema de calculadora e o sistema de IMC.

7.1 if e else

A condicional if é uma estrutura condicional que executa a afirmação, dentro do bloco, se determinada condição for verdadeira. Se for falsa, executa as afirmações dentro de else.

- **Sintaxe**

```
if (condição) afirmação1 [else
afirmação2]
```

- **Condição**

Uma expressão (premissa) que pode ser avaliada como verdadeira (true) ou falsa (false), veja lógica de programação para entender melhor.

Condição1

Condicional que será executada caso a condição em if seja verdadeira (true). Pode ser qualquer instrução, incluindo mais condicionais if aninhadas à instrução. Para executar multiplas instruções, faça um agrupamento com uma instrução em bloco `{ ... }`. Para não executar nenhuma instrução, utilize uma instrução vazia (empty).

Descrição

Múltiplas condicionais if ... else podem ser aninhados quando necessário. Observe que não existe elseif (em uma palavra). O correto é a instrução com espaços (else if), conforme a seguir:

```
if (condição1)
    instrução1
else if (condição2)
    instrução2
else if (condição3)
    instrução3
...
else
    instruçãoN
```

Para ver seu funcionamento, abaixo está a instrução como deveria ser caso identada corretamente.

```
if (condição1)
    instrução1
else
    if (condição2)
        instrução2
    else
        if (condição3)
            ...
        else
            instruçãoN
```

Para executar múltiplas instruções dentro de uma condição, utilize um bloco `{ ... }`. Em geral, é sempre uma boa prática utilizar instruções dentro de blocos, especialmente em códigos que envolvam condicionais if aninhadas:

```
if (condição) {
    instrução1
} else {
    instrução2
}
```

Não confunda os valores boolean primitivos true e false com os valores true e false do objeto Boolean. Qualquer valor que não for undefined, null, 0, NaN, ou uma string vazia (""), e qualquer objeto, incluindo um objeto Boolean cujo valor é false, é avaliado como true quando passado por uma condicional. **Por exemplo:**

```
var b = new Boolean(false);
if (b) // essa condição é avaliada
como true
Exemplos
Usando if...else
if (cipher_char === from_char) {
  result = result + to_char;
  x++;
} else {
  result = result + clear_char;
}
```

• Usando else if

Perceba que não existe sintaxe de elseif em JavaScript. Entretanto, você pode escrevê-la com um espaço entre o if e o else.

```
if (x > 5) {
} else if (x > 50) {
} else {
```

• Atribuições junto de expressões condicionais

É recomendado não utilizar atribuições simples junto de uma expressão condicional, visto que atribuições podem ser confundidas com igualdade ao olhar o código. Por exemplo, não use o código abaixo:

```
if (x = y) {
  /* faça a coisa certa */
}
```

Caso você precise utilizar uma atribuição em uma expressão condicional, uma prática comum é inserir parênteses adicionais em volta da atribuição. **Por exemplo:**

```
if ((x = y)) {
  /* faça a coisa certa */
}
```

7.2 Operações lógicas

Operadores em Javascript são símbolos especiais que envolvem um ou mais operandos com a finalidade de produzir um determinado resultado.

Os operadores podem ser aritméticos como soma, subtração e divisão; de comparação como a igualdade; operadores sobre cadeias de caracteres como o de concatenação de strings e, por fim, operadores lógicos JavaScript como o ‘and’ e o ‘or’.

• Operador de atribuição

O operador de atribuição básico é o sinal de igual (`=`). Ele atribui um valor a uma variável:

```
x = 5;
```

Essa operação atribui a `x` o valor 5.

Existem também os operadores de atribuição composta que serão explicados com exemplos.

• Atribuição composta de soma (`+=`)

```
x += 10;
```

O código acima irá verificar o valor de `x`, somá-lo a 10 e atribuí-lo novamente a `x`. Por exemplo, se o valor de `x` for 5, será feita a soma de 5 com 10 e o valor final, 15, será atribuído novamente a variável `x`.

• Atribuição composta de subtração (`-=`)

```
x -= 3;
```

Como na adição composta, a subtração irá considerar o valor da variável `x`, subtrair dela o segundo operando, 3, e armazená-lo de volta na variável `x`.

• Atribuição composta de multiplicação (`*=`)

```
x *= 8;
```

Você já deve ter entendido como é o funcionamento da composição do operador de atribuição com operadores aritméticos. No código acima, portanto, o valor de `x` será multiplicado por 8 e atribuído novamente a `x`.

• Atribuição composta de divisão (`/=`) e resto (`%=`)

```
x /= 2;  
x %= 3;
```

Por fim, a atribuição composta de divisão e resto armazenará na variável `x` o resultado da divisão de `x` por 2, e o resto da divisão de `x` por 3, respectivamente.

Esses operadores compostos são uma contração de uma operação aritmética e outra de atribuição. Por exemplo, a instrução:

```
p += 7;
```

é equivalente a:

```
p = p + 7;
```

• Operadores de comparação

Os operadores de comparação envolvem dois operandos e retornam um resultado lógico. O operador de igualdade (`==`), por exemplo, retorna verdadeiro (`true`), caso os operandos tenham o valor igual, e retornará falso (`false`), caso sejam diferentes.

Vamos ver em detalhes quais são os operadores de comparação que podemos usar quando programamos em Javascript.

- **Igual (==)**

O operador de igualdade compara dois operandos e retorna verdadeiro, se eles forem iguais, e falso, se forem diferentes. Veja os exemplos abaixo:

```
var x = 5;
x == 5; // retornará verdadeiro
x == '5' // também retornará
verdadeiro
x == 4 // retornará falso
x == 'c' // também retornará falso
```

Observe que para o Javascript não importa se os operandos recebidos pelo operador de igualdade são do mesmo tipo. No exemplo acima, o Javascript faz a conversão (casting) de um dos operandos e depois faz a comparação.

- **Diferente (!=)**

Esse operador compara os dois operandos e retorna verdadeiro, se eles forem diferentes, e falso, se forem iguais. A lógica é a inversa do operador de igualdade.

```
'string1' != 'string2' // retorna
verdadeiro
```

O exemplo acima também serve para mostrar que além de caracteres, os operadores de comparação também podem ser aplicados a strings.

- **Igualdade estrita (===)**

A igualdade estrita, além de considerar o valor dos operandos, leva em conta também seu tipo:

```
var op = 100;
op === 100; // retorna true
op === '100' // retorna false
op === 3 // também retorna false
Diferença estrita (!==)
```

Tem funcionamento análogo à diferença (!=), no entanto, considera se o tipo dos operandos é o mesmo, como o operador de igualdade estrita.

- **Maior (>)**

É usado para indicar se o operador da esquerda é maior que o operador da direita:

```
5 > 4 // retornará verdadeiro
Menor (<)
```

Retorna verdadeiro se o operador da esquerda for menor que o operador da direita:

```
6 < 10 // retornará verdadeiro
```

- **Maior ou igual (>=)**

Seu funcionamento é semelhante ao operador ‘maior’, mas retorna verdadeiro caso o operador da esquerda seja maior ou igual ao operador da direita. Veja:

```
var x = 6;
var y = 4;
x >= y // retorna verdadeiro
x >= 6 // também retorna
verdadeiro
y >= x // retorna falso
Menor ou igual (<=)
```

Partindo da declaração das variáveis acima, vamos analisar o seguinte trecho de código:

```
x <= y
```

Essa instrução retornará falso, já que 6 é maior que 4. Por outro lado, se invertermos os operandos, a instrução abaixo retornará true.

```
y <= x
```

• Operadores aritméticos

Esses operadores podem ser velhos conhecidos se você já frequentou aulas de matemática elementar. São usados para fazer operações de soma, subtração, multiplicação, divisão, módulo e exponenciação.

Vamos explorar o uso desses operadores por meio de exemplos. Primeiramente, definiremos dois operandos:

```
var x = 2;
var y = 4;
```

Vamos somar os dois operandos:

```
var res = x + y; // o resultado
será 6
Agora, a subtração:
var res = y - x; // o resultado
será 2
```

A multiplicação:

```
var res = x * y; // o valor da
operação será 8
```

E a divisão:

```
var res = x / y // como 2 dividido
por 4 é 0.5, o res receberá esse
valor.
Para a exponenciação, a sintaxe é:
y ** x // retornará 16 que é 4 ao
quadrado.
```

E, finalmente, para o operador de módulo (%), vamos usar variáveis diferentes, pois a divisão entre as variáveis que definimos acima não tem resto:

```
var dividendo = 12;
var divisor = 5;
var resto = dividendo % divisor;
// o valor de resto será 2 que é o
resto da divisão de 12 por 5.
```

• Operadores bit a bit

Os operadores bit a bit são utilizados para manipular valores para comparações e cálculos, comparando cada bit do primeiro operando com o correspondente do segundo.

Eles não são considerados operadores lógicos, já que os operandos são tratados como uma sequência de 32 bits, representados por zeros e uns, em vez de números decimais, hexadecimais ou octais.

• Operadores lógicos

São usados para realizar operações lógicas. Elas podem ser do tipo AND, OR e NOT. Os operandos devem ser lógicos, verdadeiro ou falso. Também podem operar sobre expressões lógicas, ou seja, que retornem valores verdadeiro ou falso.

• AND

O operador AND (&&) recebe dois operandos e retorna verdadeiro se, e somente se ambos os operandos sejam verdadeiros. Retorna falso, caso contrário.

Por exemplo, a expressão:

```
true && true;
retorna true. Já a expressão:
true && false;
retorna false.
```

- **OR**

Para o operador OR (||) retornar verdadeiro, basta que um dos operandos seja verdadeiro. Ele também retorna verdadeiro caso os dois operandos sejam verdadeiros. Retorna falso, se os dois forem falsos.

A expressão:

```
true || true;
retorna true bem como a expressão:
true || false;
Já a expressão:
false || false;
retorna false.
```

NOT

O operador de negação (!) é um operador unário, isto é, opera sobre apenas um operando. Ele nega, inverte o valor lógico do operando. **Veja os exemplos:**

```
var x = 2 > 1; // o valor de x é
true
!x; // o valor retornado por essa
expressão é false
Operadores de string
```

- **Concatenação (+)**

O operador de concatenação é usado para unir strings em JavaScript. O resultado que ele produz pode ser visto como uma soma.

Este operador irá avaliar os valores dos operandos, que podem ser mais de dois neste caso, e retornar uma string que é a junção desses valores. Vale notar que os operandos não precisam necessariamente ser todos do tipo string.

Retornando aos operadores de comparação do início do artigo, relembremos que, da mesma forma, ao comparar valores de tipos diferentes, o Javascript tenta realizar a transformação (casting) dos valores antes de fazer a compa-

ração, na concatenação, o Javascript faz o casting dos operandos não-string para depois juntar todos na string resultante. **Vejamos um exemplo:**

```
var numItens = 10;
var mensagem = 'O número de itens
no pedido é ' + numItens + '.';
O valor da variável 'mensagem'
será: O número de itens no pedido
é 10.
```

Observe que nesse exemplo, há três operandos. A string final é o resultado da concatenação destes três operandos.

- **Operador condicional (ternário)**

Operador ternário JavaScript é uma contracção de um operador de atribuição e duas expressões lógicas. A sintaxe é um pouco diferente das que vimos até aqui, mas tem o benefício de evitar escrever um bloco IF para atribuir um valor a uma variável. **Vejamos um exemplo:**

É preciso calcular o valor final de uma compra. Se o valor da compra for superior a 100, é aplicado um desconto de 10%.

É possível escrever essa lógica da seguinte forma:

```
var precoCompra = 200;
var desconto = 0.1;
var precoFinal = 0;
if (precoCompra > 100 ) {
    precoFinal = precoCompra;
} else {
    precoFinal = precoCompra -
    (precoCompra*desconto);
}
```

Podemos simplificar este código com o operador ternário:

```
precoFinal = precoCompra > 100 ?  
    precoCompra -  
    (precoCompra*desconto) :  
    precoCompra
```

No operador ternário, antes da interrogação (?) deve vir uma expressão lógica. A expressão antes dos dois pontos (:) será o resultado do operador, caso a expressão lógica seja verdadeira. A expressão depois dos dois pontos será o resultado caso ela seja falsa.

• Operador vírgula

A vírgula em JavaScript é usada para denotar uma lista de valores. Ela pode ser usada para declarar várias variáveis de uma vez:

```
var x, y, z;
```

Para denotar elementos de uma lista:

```
var lista = [0, 1, 2, 3];
```

• Operadores unários

Operadores unários são aqueles que realizam uma operação sobre apenas um operador. Um exemplo de operador unário é o operador de negação (!) que irá inverter o valor lógico do único operador que precede.

• Operadores relacionais

Operador in: é usado para definir se um objeto tem uma determinada propriedade. Ele retorna os valores true ou false. Um uso bastante comum do operador in é para verificar se um elemento pertence a um array:

```
1 in [1,2,3,4] // retorna true.
```

Na expressão acima, o JavaScript retornará verdadeiro, pois 1 pertence ao array. Já a expressão abaixo retorna falso, pois o valor 1 não está no array (que é vazio):

```
1 in [];
```

instanceof: outro operador relacional, retorna verdadeiro se um objeto é uma instância de outro objeto. Caso contrário, retorna falso.

• Precedência de Operadores

As operações que vimos até agora são avaliadas pelo compilador do JavaScript para que elas possam retornar um resultado.

Precedência é a ordem que essa avaliação é feita. Dizer que um operador tem precedência sobre o outro significa dizer que uma operação será avaliada antes da outra.

A ordem precedência dos operadores deve sempre ser considerada para que expressões sejam avaliadas da forma que pretendemos.

Expressões entre parênteses tem precedência mais alta. Em ordem de precedência, multiplicação e divisão são avaliadas antes de somas e subtrações, por exemplo.

Em operações lógicas envolvendo expressões, estas são avaliadas primeiro para depois o operador lógico ser avaliado. **Vejamos alguns exemplos:**

```
2 * 3 + 2; // O resultado é 8 pois  
a multiplicação tem precedência  
sobre a soma.
```

```
2*(3+2); // O resultado é 10 pois  
a expressão entre parênteses será  
avaliada antes da multiplicação.
```

```
4>6 && 5<2;
```

Na expressão acima, que retorna false, primeiro serão avaliadas as expressões que envolvem o AND. O resultado será false && false. Logo depois, a expressão resultante (false && false) será avaliada, resultando em false.

- **Onde aprender mais sobre operadores JavaScript?**

Você pode aprender mais sobre a sintaxe desta linguagem aqui no blog da Kenzie e, quando possível, consulte o manual de referência para se aprofundar ainda mais em seus estudos.

7.3 Math.random

A função **Math.random()** retorna um número pseudo-aleatório no intervalo $[0, 1[$, ou seja, de 0 (inclusivo) até, mas não incluindo, 1 (exclusivo), que depois você pode dimensionar para um intervalo desejado. A implementação seleciona uma semente para o algoritmo de geração de números aleatórios; esta semente não pode ser escolhida ou atribuída.

- **Sintaxe**

```
Math.random()
```

- **Valor retornado**

Um número pseudo-aleatório entre 0 (inclusivo) e 1 (exclusivo).

Exemplos:

Note que os números em JavaScript são pontos flutuantes que seguem o padrão IEEE 754 com comportamento arredondar-para-o-par-mais-próximo, os intervalos que serão citados nos exemplos a seguir (exceto o exemplo do Math.random()), não são exatas. Se limites extremamente grandes forem escolhidos (2⁵³ ou maior), em raros casos é possível que o limite superior (que seria exclusivo) seja retornado.

Gerando um número aleatório entre 0 (inclusivo) e 1 (exclusivo)

```
function getRandom() {
    return Math.random();
}
```

- **Gerando um número aleatório entre dois valores**

Este exemplo retorna um número entre dois valores definidos. O valor retornado será maior ou igual a min, e menor que max.

```
function getRandomArbitrary(min, max) {
    return Math.random() * (max - min) + min;
}
```

- **Gerando um número inteiro aleatório entre dois valores**

Este exemplo retorna um número inteiro entre dois valores definidos. O valor não poderá ser menor que min (ou do próximo inteiro maior que min, caso min não seja inteiro), e será menor (mas não igual) a max.

```
function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}
```

Pode ser tentador usar `Math.round()` para arredondar min e max, mas dessa maneira a aleatoriedade dos números seguiria uma distribuição não-uniforme, que talvez não seja o que você precisa.

- **Gerando um número inteiro aleatório entre dois valores, inclusive**

A função `getRandomInt()` acima tem intervalo com o valor mínimo incluído e o máximo excluído. Mas se você precisar que a função inclua, tanto o mínimo quanto o máximo, em seus resultados? A função `getRandomIntInclusive()` abaixo faz isso.

```
function getRandomIntInclusive(min, max) {  
    min = Math.ceil(min);  
    max = Math.floor(max);  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

7.4 Math.round

A função **Math.round()** retorna o valor de um número arredondado para o inteiro mais próximo.

Sintaxe

```
Math.round(x)
```

Parâmetros

x

Um número.

- **Retorno**

O valor de um número dado aproximado para o inteiro mais próximo

- **Descrição**

Se a parte fracionária do número for maior ou igual a 0.5, o argumento x é arredondado para o próximo número inteiro acima, entretanto se a parte fracionária do número for menor que 0.5, então o valor de x é arredondado para o próximo número inteiro abaixo. Se a parte fracionária for exatamente igual a 0.5, o número é arredondado para o próximo inteiro na direção de $+\infty$.

Por round ser um método estático de Math, você sempre irá usá-lo como **Math.round()**, ao invés de usá-lo como um método da instância do objeto Math que você criou.

Exemplos:

Exemplo: Uso de Math.round

```
// Retorna o valor 20
x = Math.round(20.49);
```

```
// Retorna o valor 21
x = Math.round(20.5);
```

```
// Retorna o valor -20
x = Math.round(-20.5);
```

```
// Retorna o valor -21
x = Math.round(-20.51);
```

```
// Retorna 1 (!)
// Note o erro de arredondamento por causa da inacurácia de aritmética de
// ponto flutuante
// Compare o exemplo abaixo com Math.round(1.005, -2)
x = Math.round(1.005*100)/100;
```

Exemplo: Arredondamento decimal.

```
// Closure
(function(){

  /**
   * Ajuste decimal de um número.
   *
   * @param {String} type O tipo de arredondamento.
   * @param {Number} value O número a arredondar.
   * @param {Integer} exp O expoente (o logaritmo decimal da base pretendida).
   *
   * @returns {Number} O valor depois de ajustado.
   */
  function decimalAdjust(type, value, exp) {
    // Se exp é indefinido ou zero...
    if (typeof exp === 'undefined' || +exp === 0) {
      return Math[type](value);
    }
    value = +value;
    exp = +exp;
    // Se o valor não é um número ou o exp não é inteiro...
    if (isNaN(value) || !(typeof exp === 'number' && exp % 1 === 0)) {
      return NaN;
    }
    // Transformando para string
    value = value.toString().split('e');
    value = Math[type](+(value[0] + 'e' + (value[1] ? (+value[1] - exp) : -exp)));
    // Transformando de volta
    value = value.toString().split('e');
    return +(value[0] + 'e' + (value[1] ? (+value[1] + exp) : exp));
  }

  // Arredondamento decimal
  if (!Math.round) {
    Math.round = function(value, exp) {
      return decimalAdjust('round', value, exp);
    };
  }
})
```

```
// Decimal arredondado para baixo
if (!Math.floor) {
  Math.floor = function(value, exp) {
    return decimalAdjust('floor', value, exp);
  };
}

// Decimal arredondado para cima
if (!Math.ceil) {
  Math.ceil = function(value, exp) {
    return decimalAdjust('ceil', value, exp);
  };
}

))();

// Round (arredondamento)
Math.round(55.55, -1); // 55.6
Math.round(55.549, -1); // 55.5
Math.round(55, 1); // 60
Math.round(54.9, 1); // 50
Math.round(-55.55, -1); // -55.5
Math.round(-55.551, -1); // -55.6
Math.round(-55, 1); // -50
Math.round(-55.1, 1); // -60
Math.round(1.005, -2); // 1.01 -- compare este resultado com
Math.round(1.005*100)/100 no exemplo acima
// Floor (para baixo)
Math.floor(55.59, -1); // 55.5
Math.floor(59, 1); // 50
Math.floor(-55.51, -1); // -55.6
Math.floor(-51, 1); // -60
// Ceil (para cima)
Math.ceil(55.51, -1); // 55.6
Math.ceil(51, 1); // 60
Math.ceil(-55.59, -1); // -55.5
Math.ceil(-59, 1); // -50
```



8.0 Hora da prática

- Melhorar sistema Calcular IMC.
- Melhorar sua Calculadora.
- Construa um jogo de adivinhação.

9.1 While

A declaração **while** cria um laço que executa uma rotina específica enquanto a condição de teste for avaliada como verdadeira. A condição é avaliada antes da execução da rotina.

- **Syntax**

```
while (condição) {
    rotina
}
```

- **Condição**

Uma expressão avaliada antes de cada passagem através do laço. Se essa condição for avaliada como verdadeira, a rotina é executada. Quando a condição for avaliada como falsa, a execução continua na declaração depois do laço while.

- **Rotina**

Uma declaração que é executada enquanto a condição é avaliada como verdadeira. Para executar múltiplas declarações dentro de um laço, use uma declaração em bloco (`{ ... }`) para agrupar essas declarações.

Exemplos:

O seguinte laço while itera enquanto n é menor que três.

```
var n = 0;
var x = 0;

while (n < 3) {
    n++;
    x += n;
}
```

A cada iteração, o laço incrementa n e soma à x.

Portanto, x e n assumem os seguintes valores:

Depois da primeira passagem: n = 1 e x = 1

Depois da segunda passagem: n = 2 e x = 3

Depois da terceira passagem: n = 3 e x = 6

Depois de completar a terceira passagem, a condição `n < 3` não é mais verdadeira, então o laço termina.

9.2 For

A instrução **for** cria um loop que consiste em três expressões opcionais, dentro de parênteses e separadas por ponto e vírgula, seguidas por uma declaração ou uma sequência de declarações executadas em sequência.

- **Sintaxe**

```
for ([inicialização]; [condição];
      [expressão final])
      declaração
```

- **Inicialização**

Uma expressão (incluindo expressões de atribuição) ou declarações variáveis. Geralmente usada para iniciar o contador de variáveis. Esta expressão pode, opcionalmente, declarar novas variáveis com a palavra chave **var**. Essas variáveis não são locais no loop, isto é, elas estão no mesmo escopo que o loop **for** está. Variáveis declaradas com **let** são locais para a declaração.

O resultado desta expressão é descartado.

Condição

Uma expressão para ser avaliada antes de cada iteração do loop. Se esta expressão for avaliada para true, declaração será executado. Este teste da condição é opcional. Se omitido, a condição sempre será avaliada como verdadeira. Se a expressão for avaliada como falsa, a execução irá para a primeira expressão após a construção loop **for**.

Expressão final

Uma expressão que será validada no final de cada iteração de loop. Isso ocorre antes da próxima avaliação da condição. Geralmente usado para atualizar ou incrementar a variável

do contador.

Declaração

Uma declaração que é executada enquanto a condição **for** verdadeira. Para executar múltiplas condições dentro do loop, use uma instrução de bloco **{...}** para agrupar essas condições. Para não executar declarações dentro do loop, use uma instrução vazia **();**.

Exemplos de uso:

Usando **for**

A declaração **for** começa declarando a variável **i** e inicializando-a como 0. Ela verifica se **i** é menor que nove, executa as duas instruções subsequentes e incrementa 1 a variável **i** após cada passagem pelo loop.

```
for (var i = 0; i < 9; i++) {
  console.log(i);
  // more statements
}
```

- **Expressões for opcionais**

Todas as três expressões na condição do loop **for** são opcionais.

Por exemplo, no bloco de inicialização, não é necessário inicializar variáveis:

```
var i = 0;
for (; i < 9; i++) {
  console.log(i);
  // more statements
}
```

Assim como ocorre no bloco de inicialização, a condição também é opcional. Se você está omitindo essa expressão, você deve certificar-se de quebrar o loop no corpo para não criar um loop infinito.

```

for (var i = 0;; i++) {
    console.log(i);
    if (i > 3) break;
    // more statements
}
    
```

Você também pode omitir todos os três blocos. Novamente, certifique-se de usar uma instrução break no final do loop e também modificar (incrementar) uma variável, para que a condição do break seja verdadeira em algum momento.

```

var i = 0;

for (;;) {
    if (i > 3) break;
    console.log(i);
    i++;
}
    
```

• Usando for sem uma declaração

O ciclo **for** a seguir calcula a posição de deslocamento de um nó na seção [expressão final] e, portanto, não requer o uso de uma declaração ou de um bloco de declaração, e no seu lugar é usado um ponto-vírgula - ;.

```

function showOffsetPos (sId) {
    var nLeft = 0, nTop = 0;

    for (var oItNode = document.getElementById(sId); // inicialização
        oItNode; // condition
        nLeft += oItNode.offsetLeft, nTop += oItNode.offsetTop, oItNode =
oItNode.offsetParent) // expressão final
        /* empty statement */ ;

        console.log("Offset position of \'" + sId + "\' element:\n left: " +
nLeft + "px;\n top: " + nTop + "px;");
    }

// Exemplo de call:

showOffsetPos("content");

// Resultado:
// "Offset position of "content" element:
// left: 0px;
// top: 153px;"
```

9.3 ++

O operador de incremento (`++`) incrementa (adiciona um a) seu operando e retorna um valor.

• Sintaxe

Descrição

Se usado postfix, com operador após operando (por exemplo, `x++`), o operador de incremento incrementa e retorna o valor antes de incrementar.

Se usado prefixo, com operador antes do operando (por exemplo, `++x`), o operador de incremento incrementa e retorna o valor após o incremento.

Exemplos

Incremento pós-fixado

```
let x = 3;
y = x++;

// y = 3
// x = 4
```

Incremento de prefixo

```
let a = 2;
b = ++a;

// a = 3
// b = 3
```

• Sintaxe

```
break [label];
label
```

Opcional. Identificador associado ao label de um comando. Se a estrutura não for um loop ou switch, ele será um pré-requisito.

Descrição

O comando `break` inclui um label opcional que permite ao programa encerrar a execução da estrutura que possui o nome informado na label. O comando `break` deve estar dentro dessa estrutura informada no label. A estrutura que possui o nome informada na label pode ser qualquer comando block; não é necessário que seja precedida por um loop.

Exemplos

A função a seguir possui um comando `break` que encerra o loop `while` quando a variável `i` vale 3, e então retorna o valor `3 * x`.

```
function testaBreak(x) {
    var i = 0;

    while (i < 6) {
        if (i == 3) {
            break;
        }
        i += 1;
    }
    return i * x;
}
```

9.4 Break

O comando `break` encerra o loop atual, `switch`, ou o loop que foi informado no label e transfere o controle da execução do programa para o comando seguinte.

O código a seguir possui o comando `break` dentro de uma estrutura nomeada. O comando `break` deverá estar dentro da estrutura na qual o label se refere. Veja que `inner_block` está dentro de `outer_block`.

```

bloco_externo:{

    bloco_interno:{
        console.log ('1');
        break bloco_externo; // encerra bloco_interno e
    bloco_externok
        console.log (':-('); // não é executado
    }

    console.log ('2'); // não é executado
}

```

O código a seguir também utiliza o comando `break` com blocos nomeados mas gera um erro de sintaxe pois o comando `break` está dentro do bloco₁ mas faz uma referência ao bloco₂. Um comando `break` sempre deverá estar dentro da estrutura nomeada na qual fizer referência.

```

bloco_1:{
    console.log ('1');
    break bloco_2; // SyntaxError: label not found
}

bloco_2:{
    console.log ('2');
}

```

9.5 HTML + JavaScript

A tag HTML `<script>` é usada para definir um script do lado do cliente (JavaScript). O `<script>` elemento contém instruções de script ou aponta para um arquivo de script externo por meio do `src` atributo.

Os usos comuns para JavaScript são manipulação de imagem, validação de formulário e alterações dinâmicas de conteúdo. Para selecionar um elemento HTML, o JavaScript geralmente usa o `document.getElementById()` método.

Este exemplo de JavaScript escreve "Hello JavaScript!" em um elemento HTML com `id="demo"`:

Exemplo:

```

<script>
document.getElementById("demo").innerHTML = "Hello JavaScript!";
</script>

```

9.6 Input

A `<input>` tag especifica um campo de entrada onde o usuário pode inserir dados.

O `<input>` elemento é o elemento de forma mais importante.

O `<input>` elemento pode ser exibido de várias maneiras, dependendo do atributo

- **Type**

Os diferentes tipos de entrada são os seguintes:

```

<input type="button">
<input type="checkbox">
<input type="color">
<input type="date">
<input type="datetime-local">
<input type="email">
<input type="file">
<input type="hidden">
<input type="image">
<input type="month">
<input type="number">

```

```

<input type="password">
<input type="radio">
<input type="range">
<input type="reset">
<input type="search">
<input type="submit">
<input type="tel">
<input type="text">(valor padrão)
<input type="time">
<input type="url">
<input type="week">
    
```

Veja o atributo **type** para ver exemplos para cada tipo de entrada!



// NOTA:

Sempre use a tag <label> para definir rótulos para <input type="text">, <input type="checkbox">, <input type="radio">, <input type="file"> e <input type="password">.

9.7 Button.Onclick

O **onclickevent** executa uma determinada funcionalidade quando um botão é clicado. Isso pode acontecer quando um usuário envia um formulário, quando você altera determinado conteúdo na página da Web e outras coisas assim.

Você coloca a função JavaScript que deseja executar dentro da tag de abertura do botão.

onclickSintaxe básica

```

<element
onclick="functionToExecute()">Cli
ck</element>
Por exemplo
<button
onclick="functionToExecute()">Cli
ck</button>
    
```

Observe que o onclick atributo é puramente JavaScript. O valor que ele recebe, que é a função que você deseja executar, diz tudo, pois é invocado diretamente na tag de abertura.

Em JavaScript, você invoca uma função chamando seu nome e, em seguida, coloca um parêntese após o identificador da função (o nome).

onclick exemplo de evento

segue HTML básico com um pouco de estilo para que possamos colocar o onclickevent em prática no mundo real.

```

<div>
  <p
  class="name">freeCodeCamp</p>
  <button>Change to Blue</button>
</div>
    
```

10.0 Hora da prática

- Melhorar jogo de adivinhação.
- campo texto e botão.

11.1 Arrays

• A declaração

O objeto Array do JavaScript é um objeto global usado na construção de 'arrays': objetos de alto nível semelhantes a listas.

Criando um Array

```

var frutas = [ 'Maçã', 'Banana' ];
console.log(frutas.length); // 2
    
```

- Acessar um item (index) do Array

```
var primeiro = frutas[0]; // Maçã

var ultimo = frutas[frutas.length - 1]; // Banana
```

- Iterar um Array

```
frutas.forEach(function (item, indice, array) {
  console.log(item, indice);
});
// Maçã 0
// Banana 1
```

- Adicionar um item ao final do Array

```
var adicionar =
frutas.push('Laranja');
// ['Maçã', 'Banana', 'Laranja']
```

- Remover um item do final do Array

```
var ultimo = frutas.pop(); // remove Laranja (do final)
// ['Maçã', 'Banana'];
```

- Remover do início do Array

```
var primeiro = frutas.shift(); // remove Maçã do início
// ['Banana'];
```

- Adicionar ao início do Array

```
var adicionar =
frutas.unshift('Morango') // adiciona ao início
// ['Morango', 'Banana'];
```

- Procurar o índice de um item na Array

```
frutas.push('Manga');
// ['Morango', 'Banana', 'Manga']

var pos =
frutas.indexOf('Banana');
// 1
```

- Remover um item pela posição do índice

```
var removedItem =
frutas.splice(1); // é assim que se remove um item
// ['Morango', 'Manga']
```

- Remover itens de uma posição de índice

```
var vegetais = ['Repolho', 'Nabo', 'Rabanete', 'Cenoura'];
console.log(vegetais);
// ['Repolho', 'Nabo', 'Rabanete', 'Cenoura']
```

```
var pos = 1, n = 2;
var itensRemovidos =
vegetais.splice(pos, n);
// Isso é como se faz para remover itens, n define o número de itens a se remover,
// a partir da posição (pos) em direção ao fim da array.
```

```
console.log(vegetais);
// ['Repolho', 'Cenoura'] (o array original é alterado)
```

```
console.log(itensRemovidos);
// ['Nabo', 'Rabanete']
Copy to Clipboard
Copiar um Array
var copiar = frutas.slice(); // é assim que se copia
// ['Morango', 'Manga']
```

- **Sintaxe**

```
[element0, element1, ...,  
elementN]  
new Array(element0, element1,  
..., elementN)  
new Array(arrayLength)  
  
element0, element1, ..., elementN
```

Um array JavaScript é inicializado com os elementos contém, exceto no caso onde um único argumento é passado para o construtor do Array e esse argumento é um número (veja o parâmetro arrayLength abaixo).

Esse caso especial só se aplica para os arrays JavaScript criados com o construtor Array , e não para literais de array criados com a sintaxe de colchetes [].

- **arrayLength**

Se o único argumento passado para o construtor do Array for um número inteiro entre 0 e 232-1 (inclusive), um novo array com o tamanho desse número é retornado. Se o argumento for qualquer outro número, uma exceção RangeError é lançada.

Descrição

Arrays são objetos semelhantes a listas que vêm com uma série de métodos embutidos para realizar operações de travessia e mutação. Nem o tamanho de um array JavaScript nem os tipos de elementos são fixos.

Já que o tamanho de um array pode ser alterado a qualquer momento e os dados podem ser armazenados em posições não contíguas, arrays JavaScript não tem a garantia de serem densos; isso depende de como o programador escolhe usá-los.

De uma maneira geral, essas são características convenientes, mas, se esses recursos não são desejáveis para o seu caso em particular, você pode considerar usar arrays tipados.

Arrays não podem usar strings como índices (como em um array associativo), devem ser usados números inteiros. Definir ou acessar não-inteiros usando notação de colchetes (ou notação de ponto) não vai definir ou recuperar um elemento do array em si, mas sim definir ou acessar uma variável associada com a coleção de propriedades de objeto daquele array. As propriedades de objeto do array e a lista de elementos do array são separados, e as operações de travessia e mutação não podem ser aplicadas a essas propriedades nomeadas.

Acessando elementos de um array

Arrays JavaScript começam com índice zero: o primeiro elemento de um array está na posição 0 e o último elemento está na posição equivalente ao valor da propriedade length (tamanho) menos 1.

```
var arr = ['este é o primeiro elemento', 'este é o segundo elemento'];  
console.log(arr[0]); // exibe 'este é o primeiro elemento'  
console.log(arr[1]); // exibe 'este é o segundo elemento'  
console.log(arr[arr.length - 1]); // exibe 'este é o segundo elemento'
```

Elementos de um array são somente propriedades de objetos, da forma que `toString` é uma propriedade. Contudo, note que tentando acessar o primeiro elemento de um array da seguinte forma causará um erro de sintaxe, pois o nome da propriedade é inválido:

```
console.log(arr.0); // um erro de sintaxe
```

Não há nada de especial a respeito de arrays JavaScript e suas propriedades que causam isso. As propriedades JavaScript que começam com um dígito não podem ser referenciadas com notação de ponto. Elas necessitam usar notação de colchetes para poderem ser acessadas. Por exemplo, se você tivesse um objeto com a propriedade "3d", também teria que ser referenciá-la usando notação de colchetes. **Por exemplo:**

```
var anos = [1950, 1960, 1970, 1980, 1990, 2000, 2010];
console.log(anos.0); // um erro de sintaxe
console.log(anos[0]); // funciona corretamente
```

```
renderer.3d.setTexture(model, 'personagem.png'); // um erro de sintaxe
renderer['3d'].setTexture(model, 'personagem.png'); //funciona corretamente
```

Note que no exemplo 3d, '3d' teve de ser colocado entre aspas. É possível também colocar entre aspas os índices de arrays JavaScript (ou seja, `years['2']` ao invés de `years[2]`), contudo isto não é necessário. O valor 2 em `years[2]` eventualmente será convertido a uma string pela engine do JavaScript através de uma conversão explícita com o método `toString`. E é por esta razão que '2' e '02' irão referenciar dois slots diferentes no objeto `anos` e o seguinte exemplo pode ser true:

```
console.log(anos['2'] != anos['02']);
```

De forma similar, propriedades de objeto que sejam palavras reservadas(!) só podem ser acessadas como strings em notação de colchetes:

```
var promessa = {
  'var': 'texto',
  'array': [1, 2, 3, 4]
};

console.log(promessa['var']);
```

• Relação entre `length` e propriedades numéricas

As propriedades `length` e numéricas de um array Javascript são conectadas. Vários dos métodos javascript predefinidos (por exemplo, `join`, `slice`, `indexOf` etc.) levam em conta o valor da propriedade `length` de um array quando eles são chamados. Outros métodos (por exemplo, `push`, `splice` etc.) também resultam em uma atualização na propriedade `length` do array.

```
var frutas = [];
frutas.push('banana', 'maça', 'pêssego');
```

```
console.log(frutas.length); // 3
```

Quando configurar uma propriedade num array Javascript em que a propriedade é um índice válido do array e este índice está fora do atual limite do array, o array irá crescer para um tamanho grande o suficiente para acomodar o elemento neste índice, e a engine irá atualizar a propriedade `length` do array de acordo com isto:

```
frutas[5] = 'manga';
console.log(frutas[5]); // 'manga'
console.log(Object.keys(frutas)); // ['0', '1', '2', '5']
console.log(frutas.length); // 6
```

Configurar a propriedade length diretamente, também resulta em um comportamento especial:

```
frutas.length = 10;
console.log(Object.keys(frutas)); // ['0', '1', '2', '5']
console.log(frutas.length); // 10
```

Diminuir o valor de length, entretanto, apaga elementos:

```
frutas.length = 2;
console.log(Object.keys(frutas)); // ['0', '1']
console.log(frutas.length); // 2
```

Criando um array usando o resultado de uma comparação

O resultado de uma comparação entre uma expressão regular e uma string pode criar um array Javascript. Este array tem propriedades e elementos que disponibilizam informações sobre a comparação. Esse array é o valor de retorno dos métodos RegExp.exec, String.match, e String.replace. Para explicar melhor sobre estas propriedades e elementos, veja o seguinte exemplo e então consulte a tabela abaixo:

```
// Encontra um d seguido por um ou mais b's seguido por um d
// Salva os b's encontrados e o d seguinte
// Ignora caixa (maiúscula/minúscula)

var minhaRegex = /d(b+)(d)/i;
var meuArray = minhaRegex.exec('cdbBdbsbz');
```

Propriedades

Array.length - Propriedade comprimento do construtor Array, cujo valor é 1.
 get Array[@@species] (en-US)
 A função de construtor que é utilizada para criar objetos derivados.

Array.prototype - Permite a adição de propriedades para todos os objetos array.

Métodos

Array.from() - Cria uma nova instância de Array a partir de um objeto semelhante ou iterável.

Array.isArray() - Retorna true se a variável é um array e false caso contrário.

Array.of() - Cria uma nova instância de Array com um número variável de argumentos, independentemente do número ou tipo dos argumentos.

- **Instâncias de Array**

Todas as instâncias de Array herdam de **Array.prototype**. O protótipo do construtor Array pode ser modificado de forma a afetar todas as instâncias de Array.

Propriedades

```
{{ page('/pt-BR/docs/JavaScript/Reference/Global_Objects/Array/prototype',
'Properties') }}
```

Métodos

Métodos modificadores

```
{{ page('/pt-BR/docs/JavaScript/Reference/Global_Objects/Array/prototype',
'Mutator_methods') }}
```

Métodos de acesso

```
{{ page('/pt-BR/docs/JavaScript/Reference/Global_Objects/Array/prototype',
'Accessor_methods') }}
```

Métodos de iteração

```
{{ page('/pt-BR/docs/JavaScript/Reference/Global_Objects/Array/prototype',
'Iteration_methods') }}
```

- **Métodos genéricos de Array**

Métodos genéricos de arrays não seguem o padrão, são obsoletos e serão removidos em breve. Algumas vezes você poderá querer aplicar métodos de arrays para strings ou outros objetos parecidos com arrays (como em argumentos de funções). Ao fazer isto, você trata uma string como um array de caracteres (ou em outros casos onde trata-se não-arrays como um array). Por exemplo, para checar se cada caractere em uma variváel str é uma letra, você poderia escrever:

```
function isLetter(character) {
  return (character >= "a" && character <= "z");
}

if (Array.prototype.every.call(str, isLetter))
  alert("A string '" + str + "' contém somente letras!");
```

Esta notação é um pouco dispendiosa e o JavaScript 1.6 introduziu a seguinte abreviação genérica:

```
if (Array.every(isLetter, str))
  alert("A string '" + str + "' contém somente letras!");
```

Generics também estão disponíveis em String.

Estes não são atualmente parte dos padrões ECMAScript (através do ES2015 Array.from() pode se conseguir isto). A seguir segue uma adaptação para permitir o uso em todos os navegadores:

```

/*globals define*/
// Assumes Array extras already present (one may use shims for these as well)
(function () {
    'use strict';

    var i,
        // We could also build the array of methods with the following, but the
        // getOwnPropertyNames() method is non-shimable:
        // Object.getOwnPropertyNames(Array).filter(function (methodName)
    {return typeof Array[methodName] === 'function'});
    methods = [
        'join', 'reverse', 'sort', 'push', 'pop', 'shift', 'unshift',
        'splice', 'concat', 'slice', 'indexOf', 'lastIndexOf',
        'forEach', 'map', 'reduce', 'reduceRight', 'filter',
        'some', 'every', 'isArray'
    ],
    methodCount = methods.length,
    assignArrayGeneric = function (methodName) {
        var method = Array.prototype[methodName];
        Array[methodName] = function (arg1) {
            return method.apply(arg1,
        Array.prototype.slice.call(arguments, 1));
        };
    };

    for (i = 0; i < methodCount; i++) {
        assignArrayGeneric(methods[i]);
    }
}());

```

Exemplos

Exemplo: Criando um array

O exemplo a seguir cria um array, msgArray, com length 0, então atribui valores para msgArray[0] e msgArray[99], trocando o length do array para 100.

```

var msgArray = new Array();
msgArray[0] = "Hello";
msgArray[99] = "world";

if (msgArray.length == 100)
    print("O length é 100.");

```

Exemplo: Criando um array bi-dimensional

O exemplo a seguir cria um tabuleiro de xadrez usando dois arrays bi-dimensionais de string. A primeira jogada é feita copiando o 'p' em 6,4 para 4,4. A posição antiga de 6,4 é colocada em branco.

```
var board =
[ ['R','N','B','Q','K','B','N','R'],
  ['P','P','P','P','P','P','P'],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ 'p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r']];
print(board.join('\n') + '\n\n');

// Fazendo o King's Pawn avançar 2
board[4][4] = board[6][4];
board[6][4] = ' ';
print(board.join('\n'));
```

Copy to Clipboard
Saída:

R,N,B,Q,K,B,N,R

P,P,P,P,P,P,P

```
 ' ' ' ' '
 ' ' ' ' '
 ' ' ' ' '
 ' ' ' ' '
p,p,p,p,p,p,p
r,n,b,q,k,b,n,r
```

R,N,B,Q,K,B,N,R

P,P,P,P,P,P,P

```
 ' ' ' ' '
 ' ' ' ' '
 ' ',p, ' '
 ' ' ' ' '
p,p,p,p, ,p,p,p
r,n,b,q,k,b,n,r
```

- Utilizando um array para tabular um conjunto de valores

```
values = [];
for (var x = 0; x < 10; x++){
  values.push([
    2 ** x,
    2 * x ** 2
  ])
}
console.table(values)
Saída:
0 1 0
1 2 2
2 4 8
3 8 18
4 16 32
5 32 50
6 64 72
7 128 98
8 256 128
9 512 162
```

12.0 Hora da prática

- Melhorar jogo de adivinhação.



Digital College

ENSINO DE HABILIDADES DIGITAIS

digitalcollege.com.br