



DEFINIÇÃO

Apresentação dos conceitos básicos e das principais funcionalidades do JavaScript, uma das linguagens de programação mais importantes e utilizadas na internet.

PROPÓSITO

Compreender a linguagem JavaScript, sua sintaxe e utilização em páginas HTML.

PREPARAÇÃO

Para aplicação dos exemplos, será necessário um editor de texto com suporte à marcação HTML. No sistema operacional Windows, é indicado o Notepad++. No Linux, o Nano Editor.

Uma alternativa aos editores instalados no computador são os interpretadores online, como CodePen e JSFiddle.

OBJETIVOS

MÓDULO 1

Identificar os conceitos básicos, a sintaxe e as formas de utilização do JavaScript

MÓDULO 2

Aplicar as estruturas de decisão e de repetição

MÓDULO 3

MÓDULO 4

Reconhecer os recursos assíncronos Ajax e JSON

INTRODUÇÃO

Segundo Flanagan (2011), JavaScript é a linguagem de programação da Web mais utilizada pelos sites. Além disso, todos os navegadores – estejam eles em desktops, jogos, consoles, tablets ou smartphones – incluem interpretadores JavaScript, fazendo desta a linguagem de programação mais onipresente da história.

Ao longo deste tema, veremos os **conceitos básicos e as principais funcionalidades do JavaScript**. Além disso, **aprenderemos a integrá-lo às páginas HTML e a utilizá-lo** – desde tarefas básicas, como manipular a interface DOM, a tarefas mais complexas, como transmitir dados entre o cliente e o servidor de forma assíncrona.

Vamos começar nossa jornada acessando os códigos-fontes originais propostos para o aprendizado de Javascript. Baixe o arquivo aqui, descompactando-o em seu dispositivo. Assim, você poderá utilizar os códigos como material de apoio ao longo do tema!

MÓDULO 1

-
- 🕒 Identificar os conceitos básicos, a sintaxe e as formas de utilização do JavaScript

JAVASCRIPT



Fonte: pixcon/Shutterstock

Esta linguagem faz parte da **tríade de tecnologias que compõe o ambiente Web**, juntando-se à **HTML** – que cuida da estrutura e do conteúdo das páginas – e ao **CSS** – responsável pela apresentação. **Sua função, nesse ambiente, é cuidar do comportamento e da interatividade das páginas Web. Trata-se de uma linguagem de programação interpretada e multiparadigma, uma vez que possui suporte à programação estruturada, orientada a objetos e funcional.**

O JavaScript, comumente chamado de JS, foi criado inicialmente para o ambiente Web, no lado cliente. Entretanto, evoluiu ao ponto de atualmente ser utilizado também no lado servidor, além de ser uma das principais linguagens para o desenvolvimento de aplicativos mobile.

ATENÇÃO

É importante destacar que, embora possuam nomes parecidos, as linguagens de programação JavaScript e Java não têm nenhuma relação.

A INTERFACE DOM

Iniciaremos nosso estudo de JavaScript entendendo o que é e como **manipular a interface**, ou árvore, DOM. Isso auxiliará o entendimento de como essa linguagem se integra e interage com a HTML.

A sigla **DOM** significa Modelo de Objeto de Documento. Trata-se de uma interface que permite a manipulação, via programação, de documentos HTML (e outros, como XML, por exemplo). **Além de interface, é, também, comumente chamado de árvore, por fornecer uma representação estruturada do documento nesse formato.**

A árvore DOM é composta por **nós** e **objetos**, ambos possuindo **propriedades** e **métodos**, além de eventos. Através dessa estrutura, é possível acessar, por exemplo, o conteúdo textual de um elemento – como a tag <p> –, recuperar e até mesmo alterar o seu valor.

Embora frequentemente associados, **o DOM não faz parte da linguagem JavaScript**, podendo também ser manipulado por outras linguagens. Neste tema, porém ficaremos restritos à manipulação do DOM através do JavaScript.

A Figura 1 ilustra a árvore DOM e seus elementos:

ÁRVORE DOM

● NÓ DOCUMENTO
● NÓ ELEMENTO
● NÓ ATRIBUTO
● NÓ TEXTO

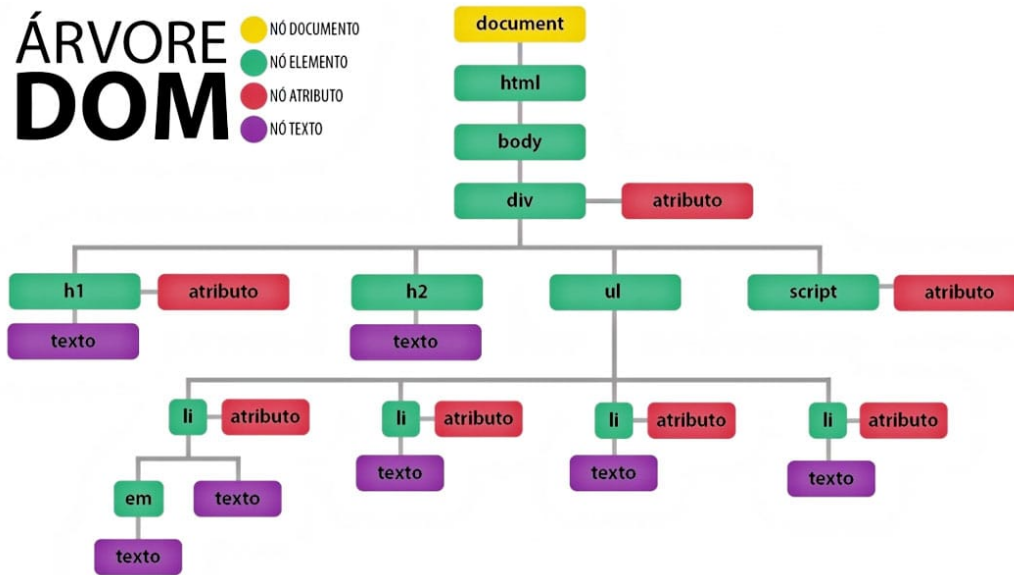


Figura 1: Árvore DOM (BARBOSA, 2017).

MANIPULANDO O DOM COM JAVASCRIPT

Por meio do JavaScript, é possível inserir dinamicamente, em **tempo de execução**, novos elementos à árvore DOM, mesmo que estes não façam parte da estrutura inicial da página. Da mesma forma, é possível excluir e alterar elementos preexistentes ou dinamicamente criados.

Esses elementos, porém, existirão somente enquanto durar a sessão atual do usuário. Ou seja, **trata-se de uma manipulação dinâmica e dependente de estado, de ações e interações por parte do usuário, mas que se perderão quando ele sair da página.**

TEMPO DE EXECUÇÃO

É a sessão ou duração da visita de um usuário a uma página.

DICA

Com JavaScript, é possível até mesmo manipular os estilos de um documento, de forma similar ao que é feito via CSS.

INCORPORANDO O JAVASCRIPT À HTML

A incorporação do JavaScript a páginas HTML pode ser feita de duas formas:

CÓDIGOS NO CORPO DA PÁGINA

Incluindo os códigos diretamente no corpo da página – dentro da seção `<head>` e da tag `<script>`

ARQUIVOS EXTERNOS

Através de arquivos externos, com extensão `js`, linkados ao documento também dentro da seção `<head>`

Para a otimização do desempenho do carregamento da página, deve-se mover todo o código JavaScript para o seu final, após o fechamento da tag `</body>`.

ATENÇÃO

Deve-se tomar cuidado para não mover códigos ou scripts incluídos que sejam necessários à correta visualização ou aos comportamentos da página, já que os códigos movidos para o final só serão lidos e interpretados após todo o restante da página.

Veja a seguir as duas formas citadas de inserção:

```
1  <!doctype html>
2  <html lang="pt-BR">
3  <head>
4  <meta charset="utf-8">
5  <title>Incorporando Javascript em Páginas HTML</title>
6  <!-- Incorporando um arquivo .js externo -->
7  <script src="script.js"></script>
8  </head>
9  <body>
10  <div id="exibe_resultado"> Resultado da Multiplicação: </div>
11  </body>
12  <!-- Incorporando códigos Javascript diretamente na página -->
13  <script type="text/javascript">
14  //Com duas barras criamos um comentário de linha em Javascript
15  //Comentários de mais de uma linha podem ser feitos dentro de /* */
16
17  var variavel; //Declarando uma variável cujo nome é 'variavel'
18
19  variavel = 3 + 3; // Atribuindo valores e aplicando uma operação matemática em uma variável
20
```



```
21  /* Utilizando a função nativa 'alert' para exibir uma caixa de
22  diálogo na tela cujo conteúdo será o valor da variável 'variavel' */
23  alert(variavel);
24
25  var resultadoMultiplicacao = multiplique(10, 50); // chamando a função 'multiplique' passando dois valores - 10 e 50
26
27  //Manipulando a Árvore DOM a fim de exibir, dentro da DIV declarada no HTML, o resultado da multiplicação juntamente com o seu texto inicial
28  var divLocal = document.getElementById('exibe_resultado');
29  /*
30  Neste ponto a variavel divLocal é um objeto que representa a div declarada no HTML.
31  Sendo um objeto é possível acessar seus atributos, como innerHTML, precedido do nome do objeto seguido de um '.'
32  */
33  divLocal.innerHTML += resultadoMultiplicacao;
34
35  //Definição da função 'multiplique' que recebe dois parâmetros - numero1 e numero2
36  function multiplique(numero1, numero2){
37
38  /*
39  Declarando uma nova variável que guardará o resultado da operação de multiplicação;
40  Iniciando a variável com o valor de 0.
41  */
42  var resultado = 0;
43
44  //Atribuindo à variável 'resultado' o valor resutante da multiplicação dos 2 parâmetros recebidos
45  resultado = numero1 * numero2;
46
47  //Retornando (devolvendo) o valor da variável resultado
```

```
48     return resultado;
49
50 }
51 </script>
52 </html>
```

 Figura 2: Formas de incorporação do JavaScript numa página HTML.

ATENÇÃO

Baixe **aqui** o código usado na Figura 2

SINTAXE JAVASCRIPT

A Figura 2 mostra tanto as formas de inserção do JavaScript em uma página HTML (**linhas 7 e 13**) quanto alguns aspectos da sua **sintaxe**. Tais aspectos serão vistos em detalhes a seguir, mas, antes de começarmos, é recomendado que você copie o código anterior e o execute – pode ser em seu computador ou utilizando uma ferramenta de codificação online, como o CodePen ou JSFiddle. Você pode copiar direto do box, ou utilizar o arquivo Figura 2.

DICA

Serão utilizados os números das linhas do editor, vistos na imagem, para facilitar a localização dos fragmentos e elementos abordados.

💬 COMENTÁRIO

Existem duas formas de inserir comentários no código JavaScript: para os comentários de apenas uma linha, utilizam-se as duas barras “//”. Para os de múltiplas linhas, utiliza-se “/*” e “*/”. Veja, por exemplo, as linhas **14, 15, 21 e 22**.

VARIÁVEIS

São declaradas utilizando-se a palavra reservada “**var**”, **sucedida pelo seu nome. Não devem ser utilizados caracteres especiais como nomes de variáveis.**

Além disso, embora existam diferentes convenções, **procure utilizar um padrão e segui-lo ao longo de todo o seu código para a nomeação das variáveis, sobretudo as compostas.**

★ EXEMPLO

Na linha 25, a palavra composta “resultado multiplicação” foi transformada em uma variável através do padrão *camelcase*, ou seja, a segunda palavra (e demais palavras, quando for o caso) é iniciada com a primeira letra em maiúsculo.

Outra característica importante de uma variável em JS é que, por se tratar de uma **linguagem fracamente tipada**, não é necessário declarar o tipo de dados. Com isso, uma variável que armazenará um número (inteiro, decimal etc.) e outra que armazenará uma palavra (*string*, char etc.) são declaradas da mesma forma.

Após declaradas, as variáveis podem ser utilizadas sem a palavra reservada “var”, como visto nas **linhas 19 e 33**.



LINGUAGEM FRACAMENTE TIPADA

Uma linguagem é dita **fracamente tipada** quando não é necessário informar o tipo de dado no momento de criação de uma variável.

ATRIBUIÇÃO

As variáveis precisam ser declaradas antes de sua utilização. Entretanto, há uma forma simplificada, vista na **linha 25**, na qual é feita uma atribuição de valores no momento de declaração da variável “resultadoMultiplicacao”.

A respeito da atribuição de valores, é importante frisar que, **embora declarados da mesma maneira, os tipos de dados são atribuídos de formas distintas.** Um número, por exemplo, pode ser atribuído diretamente a uma variável, enquanto uma *string* precisará estar envolta em aspas – simples ou duplas. A **linha 13** mostra dois números sendo atribuídos à “variável”.

Veja o exemplo a seguir, no qual atribuímos uma *string* a uma variável:

```
VAR NOMEDISCIPLINA = “JAVASCRIPT”
```

```
=  
=
```

```
VAR NOMEDISCIPLINA = ‘JAVASCRIPT’
```

PONTO E VÍRGULA

Repare o final de cada linha de código – todas foram terminadas com a utilização de um **ponto e vírgula**. Diferentemente de outras linguagens, em JavaScript não é obrigatória a utilização de caracteres para indicar o final de uma linha de código, mas, seguindo uma linha de boas práticas, adote uma convenção e a utilize em todo o seu código.

OUTROS ELEMENTOS

Ao longo do código apresentado na Figura 2, foram utilizados outros elementos. A seguir, na tabela 1, cada um deles será descrito:

ELEMENTO	PARA QUE SERVE	LINHA DO CÓDIGO
alert	Exibir uma caixa de diálogo no navegador. Existem outras funções nativas de diálogo, inclusive para receber input de valores.	23

document.getElementById	Referenciar um elemento da árvore DOM através do valor do seu atributo “id”. Pesquise também sobre document.getElementsByClassName.	28
innerHTML	Propriedade DOM relativa ao conteúdo de um elemento. Permite tanto a inclusão quanto a exclusão e a modificação do conteúdo do elemento.	33
+=	Operador de atribuição composta. A linha em questão poderia ser escrita de forma simplificada, com esse operador, ou em sua forma completa, como: “divLocal.innerHTML = divLocal.innerHTML + resultadoMultiplicacao”.	33
function	Palavra reservada, utilizada para indicar que será declarada uma função. É precedida pelo nome da função e por parênteses. Caso receba parâmetros, eles devem ser declarados dentro dos parênteses.	36
return	Palavra reservada, utilizada para indicar o conteúdo a ser retornado pela função. Nem todas as funções retornam valores. Logo, essa instrução só deve ser utilizada por funções que retornem algum resultado.	48

📷 Tabela 1: Lista de elementos utilizados no código de exemplo.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

CONSIDERAÇÕES ADICIONAIS SOBRE O CÓDIGO UTILIZADO COMO EXEMPLO

Volte ao seu código e faça a seguinte modificação:

Mova o código que está entre as **linhas 12 e 51**, inclusive, para dentro da seção `<head>`.



Salve a alteração e carregue novamente sua página.



Repare que o alerta é exibido, mas a div “exibe_resultado” não recebeu o valor de resultado da multiplicação.

Isso acontece porque, quando o código está no início da página, ele é lido pelo navegador antes que o restante seja renderizado.



Portanto, a tag `<div>`, por exemplo, ainda não foi carregada e não está presente na árvore DOM.

Caso queira manter o seu código no início e ter o mesmo resultado de quando ele fica ao final, é necessário utilizar um evento dentro do mesmo, o “**onload**”, e modificar seu código para utilizá-lo.

ONLOAD

O evento “**onload**” pode ser usado quando queremos que algo seja carregado junto com o carregamento da página.

TEORIA NA PRÁTICA

Ao longo deste tema, os exercícios práticos serão fundamentais para a fixação do conteúdo visto. Partindo do código utilizado na Figura 2, modifique-o da seguinte forma:

Peça ao usuário para inserir dois números inteiros positivos;

Armazene os números inseridos pelo usuário em duas variáveis;

Crie uma função para dividir números inteiros;

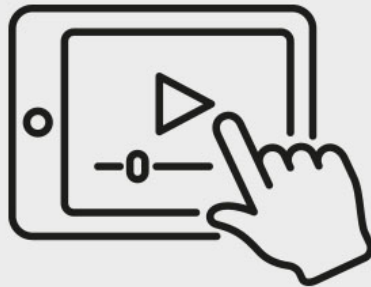
Exiba na tela uma caixa de diálogo com o resultado da divisão precedido pela frase “O resultado da divisão é igual a:”.

Caso queira ir além, crie uma calculadora para realizar as quatro operações matemáticas. Nesse caso, você precisará pedir ao usuário que escolha a operação a ser realizada, além dos números de entrada.

RESOLUÇÃO

No vídeo a seguir, o professor Alexandre Paixão nos apresenta um exercício resolvido. Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.




```
1  body {
2  <!doctype html>
3  <html lang="pt-BR">
4  <head>
5  <meta charset="utf-8">
6  <title>Praticando Javascript - Exercício 1</title>
7  </head>
8  <body>
9
10 </body>
11
12 <script type="text/javascript">
13
14   var numero1 = prompt("Insira o primeiro número: ");
15   var numero2 = prompt("Insira o segundo número: ");
16
17   var resultadoDivisao = divida(numero1, numero2);
18
19   alert('O resultado da divisão é igual a: ' + resultadoDivisao);
20
21   function divida(numero1, numero2){
22
23     var resultado = 0;
24
25     resultado = numero1 / numero2;
26
27     return resultado;
```

```
28
29  }
30  </script>
31  </html>
```

ATENÇÃO

Baixe **aqui** o código usado no exercício.

VERIFICANDO O APRENDIZADO

1. A LINGUAGEM JAVASCRIPT É UMA LINGUAGEM TIPICAMENTE DO LADO CLIENTE, EMBORA TAMBÉM USADA, MAIS RECENTEMENTE, NO LADO SERVIDOR. SOBRE SUA UTILIZAÇÃO NO LADO CLIENTE, E MAIS PRECISAMENTE SOBRE SUA RELAÇÃO COM O DOM, ASSINALE A AFIRMATIVA CORRETA:

A) JavaScript permite que a estrutura inicial de uma página HTML seja modificada. Além disso, como também é uma linguagem com suporte do lado servidor, ela permite que esses códigos HTML modificados sejam salvos na página HTML original.

B) Um script JS pode ser incluído tanto no corpo do documento HTML como através de um arquivo externo. A diferença principal entre essas duas formas está no fato de que o código inserido diretamente na HTML faz parte da árvore DOM – sendo, portanto, a única forma de manipular os elementos dessa interface.

C) Com a utilização da linguagem JavaScript, é possível ter acesso à árvore DOM. Com isso, tarefas como a modificação de elementos existentes e a inclusão de novos elementos, assim como conteúdos, se torna possível.

D) Os códigos JavaScript incorporados ao final da página não permitem a manipulação da árvore DOM, já que são interpretados apenas após o carregamento de todos os elementos.

2. A RESPEITO DOS TIPOS E DA UTILIZAÇÃO DE VARIÁVEIS EM JAVASCRIPT, ASSINALE A AFIRMATIVA INCORRETA:

A) Os valores podem ser atribuídos no momento em que a variável é declarada.

B) Valores de qualquer tipo podem ser atribuídos da mesma forma.

C) JavaScript é uma linguagem fracamente tipada. Logo, não é necessário informar o tipo de dado no momento de criação da variável.

D) As variáveis precisam ser declaradas antes de serem utilizadas.

GABARITO

1. A linguagem JavaScript é uma linguagem tipicamente do lado cliente, embora também usada, mais recentemente, no lado servidor. Sobre sua utilização no lado cliente, e mais precisamente sobre sua relação com o DOM, assinale a afirmativa correta:

A alternativa **"C "** está correta.

Através de JavaScript, é possível manipular a árvore DOM, independentemente do modo de incorporação ao documento HTML. A única ressalva diz respeito a eventos de manipulação que tentem acessar os nós e os elementos DOM antes que toda a página seja renderizada, como visto em um dos exemplos demonstrados.

2. A respeito dos tipos e da utilização de variáveis em JavaScript, assinale a afirmativa incorreta:

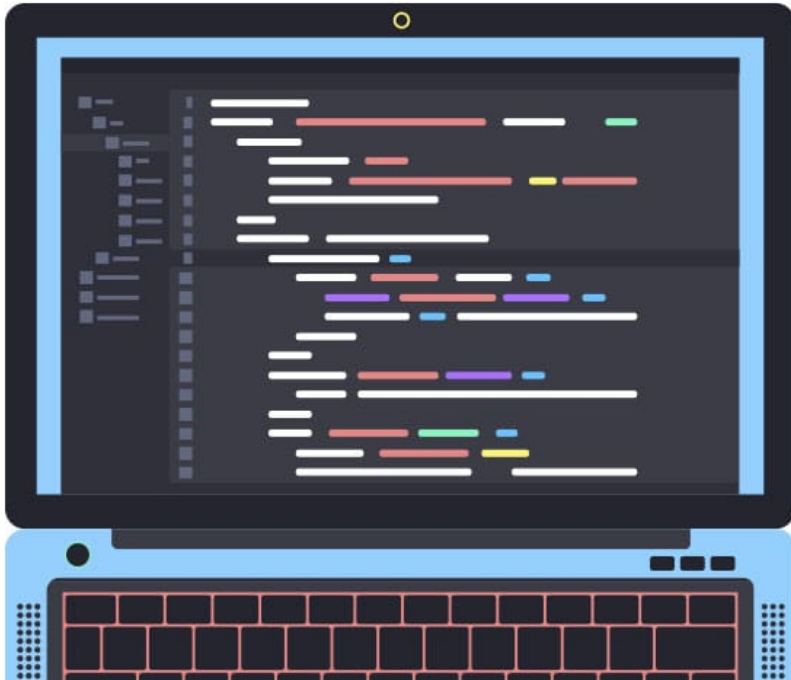
A alternativa **"B "** está correta.

JavaScript é uma linguagem bastante flexível em relação à declaração e à utilização de variáveis. Entretanto, alguns cuidados são necessários, entre eles a atribuição de valores do tipo *string*, que precisam ser englobados por aspas – duplas ou simples.

MÓDULO 2

⦿ Aplicar as estruturas de decisão e de repetição

ESTRUTURAS DE DECISÃO



Fonte: Adnrey/Shutterstock

Segundo Flanagan (2011), as **estruturas de decisão**, também conhecidas como “**condicionais**”, **são instruções que executam ou pulam outras instruções dependendo do valor de uma expressão especificada**. São os pontos de decisão do código, também conhecidos como ramos, uma vez que podem alterar o fluxo do código, criando um ou mais caminhos.

Para melhor assimilação do conceito, vamos usar um exemplo a partir do código construído no módulo anterior:

As orientações do programa diziam que deveria ser realizada a divisão de dois números inteiros positivos.

Entretanto, o que acontece se o usuário inserir um número inteiro que não seja positivo?

Ou como forçá-lo a inserir um número positivo?

Para essa função, podemos utilizar uma condição. Ou seja:

“... Caso o usuário insira um número inteiro não positivo, avise a ele que o número não é válido e peça que insira um número válido ...”

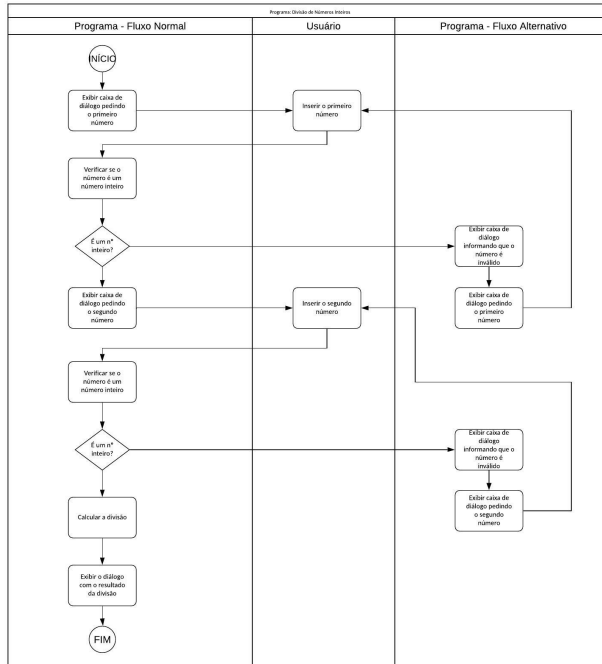
Nesse caso, o fluxo normal do programa é receber dois números positivos, calcular a divisão e exibir o resultado. Perceba que a condição cria um novo fluxo, um novo ramo, onde outro diálogo é exibido e o usuário, levado a inserir novamente o número.

O fluxo normal e o fluxo resultado da condicional podem ser vistos na Figura 3, a seguir. Nela, são apresentados os passos correspondentes ao nosso exercício, separando as ações do programa e também as do usuário.

Repare que a verificação “é um nº inteiro positivo” permite apenas duas respostas: “sim” e “não”. Tal condição, mediante a resposta fornecida, é responsável por seguir o fluxo normal do código ou o alternativo.

💡 DICA

Observação: o fluxograma de exemplo foi simplificado para fornecer mais detalhes. Logo, a respectiva notação padrão não foi utilizada em sua confecção.



📷 Figura 3: Fluxo normal e fluxo alternativo.

Nas linguagens de programação, utilizamos as **instruções condicionais** para implementar o tipo de decisão apresentado no exemplo. Em JavaScript, estão disponíveis as instruções "**if/else**" e "**switch**". Tais instruções serão apresentadas a seguir.

IF

A sintaxe da instrução "if/else" em JavaScript possui algumas formas. A primeira e mais simples é apresentada do seguinte modo:

if (condição) instrução

Nessa forma, é verificada uma única condição. Caso seja verdadeira, a instrução será executada. Do contrário, não.

Antes de continuarmos, é importante destacar os elementos da instrução:

Ela é iniciada com a palavra reservada "if".



Dentro de parênteses, é inserida a condição (ou condições, como veremos a seguir).



Por fim, é inserida a instrução a ser executada caso a condição seja verdadeira.

Outro detalhe importante: caso exista mais de uma instrução para ser executada, é necessário envolvê-las em chaves. Veja o exemplo:

```
1  if (condição1 && condição2){  
2  instrução1;
```

```
3   instrução2;  
4   }
```

Nesse segundo caso, além de **mais de uma instrução**, também temos **mais de uma condição**. Quando é necessário verificar mais de uma condição, em que cada uma delas precisará ser verdadeira, utilizamos os caracteres “&&”.

Na prática, as instruções 1 e 2 só serão executadas caso as condições 1 e 2 sejam verdadeiras. Vamos a outro exemplo:

```
1   if (condição1 || condição2){  
2       instrução1;  
3       instrução2;  
4   }
```

Repare que, nesse código, os caracteres “&&” foram substituídos por “||”. Esses últimos são utilizados **quando uma OU outra condição precisa ser verdadeira para que as instruções condicionais sejam executadas.**

E o que acontece se quisermos verificar mais condições?

Nesse caso, podemos fazer isso tanto para a forma onde todas precisam ser verdadeiras, e separadas por “&&”, quanto para a forma onde apenas uma deve ser verdadeira, separadas por “||”. Além disso, é possível, ainda, combinar, numa mesma verificação, os dois casos. Veja o exemplo:

```
1   if ( (condição1 && condição2) || condição3){  
2       instrução1;  
3       instrução2;  
4   }
```

Nesse fragmento, temos as duas primeiras condições agrupadas por parênteses. A lógica aqui é:

Execute as instruções 1 e 2 SE ambas forem verdadeiras OU se a condição 3 for verdadeira.

Por fim, há outra forma: **a de negação.**

Ou seja, como verificar se uma condição é falsa (ou não verdadeira)?

Veja a seguir:

```
1  if (!condição1){  
2      instrução1;  
3      instrução2;  
4  }
```

O sinal “!” é utilizado para negar a condição. As instruções 1 e 2 serão executadas caso a condição 1 não seja verdadeira.

ELSE

A instrução **"else"** acompanha a instrução **"if"**. Logo, embora não seja obrigatória, como vimos nos exemplos, sempre que a primeira for utilizada, deve vir acompanhada da segunda. **O "else" indica se alguma instrução deve ser executada caso a verificação feita com o "if" não seja atendida.** Vejamos:

```
If(número fornecido é inteiro e positivo){  
  
    Guarde o número em uma variável;  
  
}else{  
  
    Avise ao usuário que o número não é válido;  
  
    Solicite ao usuário que insira novamente um número;  
  
}
```

Perceba que o "else" (senão) acompanha o "if" (se). Logo, **SE as condições forem verdadeiras, faça isto. SENÃO, faça aquilo.**

DICA

Uma observação importante: no último fragmento foi utilizado, de forma proposital, **português-estruturado** nas condições e instruções. Isso porque, mais adiante, você mesmo codificará esse "if/else" em JavaScript.

PORTUGUÊS-ESTRUTURADO

Linguagem de programação ou pseudocódigo que utiliza comandos expressos em português.

ELSE IF

Veja o exemplo a seguir:

```
1  if (numero1 < 0){  
2    instrução1;  
3  }else if(numero == 0){  
4    instrução2;  
5  }else{  
6    instrução3;  
7  }
```

Repare que uma nova instrução foi utilizada no fragmento. Trata-se da **"else if"**, utilizada quando queremos fazer verificações adicionais, sem agrupá-las todas dentro de um único "if". Além disso, repare que, ao utilizarmos essa forma, caso nenhuma das condições constantes no "if" e no(s) "if else" seja atendida, ao final a instrução "else" será executada obrigatoriamente.

SWITCH

A instrução "**switch**" é bastante útil quando uma série de condições precisa ser verificada. É bastante similar à "else if". Vejamos:

```
1  Switch(numero1){  
2    Case < 0:  
3    instrução1;  
4    Break;  
5    Case == 0:  
6    instrução2;  
7    Break;  
8    Default:  
9    instrução3;  
10   Break;  
11  }
```

Em linhas gerais, o código anterior é igual e tem a mesma função do visto no exemplo de "else if". Veja a seguir:

Nele, após o "Swicth" dentro de parênteses, temos a condição a ser verificada.



A seguir, temos os “Case”, em quantidade equivalente às condições que queremos verificar.



Dentro de cada “Case” temos a(s) instrução(ões) e o comando “**Break**”.



Por fim, temos a instrução “Default”, que será executada caso nenhuma das condições representadas pelos “Case” sejam atendidas.

BREAK

O comando “**Break**” indica que, satisfeita a condição do “**Case**” e tendo sido executada(s) a(s) sua(s) instrução(ões), o fluxo do programa deverá sair do “**Switch**”, ou seja, nenhuma outra condição dele deverá/precisará ser verificada.

ESTRUTURAS DE REPETIÇÃO

Essas estruturas – também chamadas de **laços** – **permitem que uma ação seja executada de forma repetitiva**. Em nosso exercício, por exemplo, temos uma ação recorrente, que é a de solicitar ao usuário que insira um número. Se fosse executada dentro de um laço, o nosso código diminuiria, facilitando o trabalho.

A sintaxe de uma estrutura de repetição pode ser vista no fragmento de código apresentado na Figura 4, a seguir. Após ler o código e os comentários explicativos, execute-o e veja o resultado. Você pode copiar direto do box a seguir, ou utilizar o arquivo Figura 4.

FOR

Sobre a sintaxe apresentada na Figura 4, temos o laço “**for**”, um dos presentes em JavaScript. Em sua forma mais comum, temos uma variável, que chamamos normalmente de **contador**, que recebe um valor inicial (no exemplo, 0) e é incrementada (pelo “++”) até atingir uma condição (ser menor que 10).

```
1  var contador;  
2  for (contador = 0; contador < 10; contador++){  
3  
4      //As instruções incluídas aqui serão executadas 10 vezes  
5  
6      /*  
7      Serão exibidas 10 caixas de diálogo exibindo o valor da variável contador.  
8      O primeiro número será 0 e o último será 9 (repare que começamos com a nossa  
9      variável contador recebendo o número 0 e sendo incrementada até ser menor do que 10,  
10     ou seja, até o número 9.  
11     */  
12     alert(contador);  
13 }
```

 Figura 4: Estrutura de repetição em JavaScript.

ATENÇÃO

Baixe **aqui** o código usado na Figura 4

EXEMPLO

Algumas variações possíveis nesse código seriam iniciar o contador em 1, por exemplo (o mais usual, em programação, é iniciarmos nossos índices em zero), ou irmos até o número 10. Ou seja, no lugar do sinal de menor, utilizaríamos o “menor igual”, dessa forma: “<= 10”. Teste também, em seu código, essas variações e veja as diferenças.

A seguir, veremos as outras estruturas de repetição, além do “for”, presentes na linguagem JavaScript.

WHILE

Veja o fragmento a seguir para entender o comportamento do laço “**while**” (enquanto):

```
1  var contador = 0;
2  while(contador < 10){
3    alert(contador);
4    contador++;
5  }
```

Esse código tem o mesmo resultado que o visto no exemplo utilizando o laço “for”. Sua sintaxe é simples: “**Enquanto uma condição fornecida for verdadeira, faça isso.**”

DO/WHILE

Embora semelhante ao laço "while", temos uma diferença fundamental entre eles:

“do/while”

A condição é testada no final.



“while”

A condição é testada no início.

Com isso, pelo menos uma instrução será, obrigatoriamente, executada. Vejamos o exemplo:

```
1  var contador = 0;  
2  do{  
3    contador += 1;  
4    alert(contador);  
5  }while (contador < 10);
```

Execute o código e veja a diferença em relação ao utilizado no “while”.

FOR/IN

Esse laço, assim como os demais em uma linguagem de programação, é bastante utilizado com *arrays* (vetor ou matriz contendo dados não ordenados. Veremos sobre eles no próximo módulo). Normalmente, precisamos percorrer o conteúdo de um *array* e manipular ou apenas exibir o seu valor. Para isso, podemos fazer uso de laços. Na Figura 5, a seguir, serão apresentados dois fragmentos de código para uma mesma função – um utilizando “**for**” e o outro, “**for/in**”.

```
1  var frutas = ['Laranja', 'Uva', 'Pera'];  
2  
3  
4  /*Imprimindo na caixa de diálogo o conteúdo do array 'frutas' utilizando o laço 'for'*/
```

```
5  for(var i = 0; i < frutas.length; i++){
6      alert('Nome da Fruta contida no Array: ' + frutas[i]);
7  }
8
9  /*Imprimindo na caixa de diálogo o conteúdo do array 'frutas' utilizando o laço 'for/in'*/
10  for(var fruta in frutas){
11      alert('Nome da Fruta contida no Array: ' + frutas[fruta]);
12  }
```

📷 Figura 5: Comparação entre os laços “for” e “for/in”.

📢 ATENÇÃO

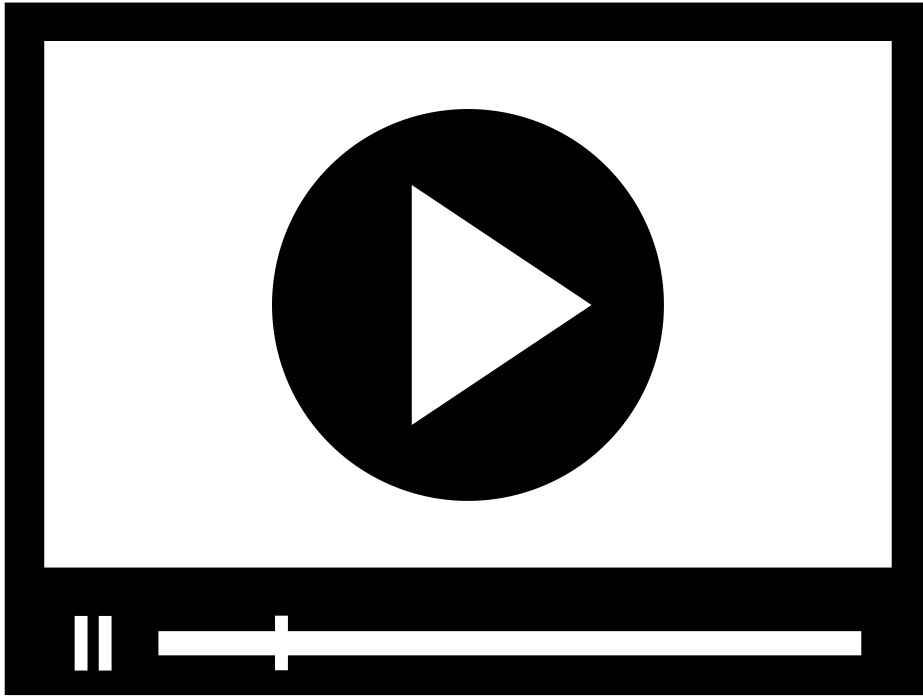
Baixe **aqui** o código usado na Figura 5.

Analisando o código, é possível notar as diferenças de sintaxe entre os laços “**for**” e “**for/in**”.

No primeiro, em “**for**”, definimos uma variável contador (“**i**”) e deveremos percorrer o *array* “frutas” começando em seu índice 0 até o seu tamanho (“**length**”).



Já no “**for/in**”, declaramos uma variável contador (“fruta”) e dizemos ao código que percorra o *array* imprimindo o seu conteúdo a partir do índice fornecido – nesse caso, a variável “fruta”.



EXERCÍCIO COM ESTRUTURAS DE DECISÃO E DE REPETIÇÃO

No vídeo a seguir, o professor Alexandre Paixão nos apresenta um exercício resolvido demonstrando as estruturas de controle de decisão e de repetição.
Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. NO QUE CONCERNE ÀS ESTRUTURAS DE DECISÃO, MAIS PRECISAMENTE À INSTRUÇÃO “SWITCH”, ASSINALE A AFIRMAÇÃO INCORRETA:

- A) Essa instrução serve para alterar o fluxo de execução de um programa.
- B) Com essa instrução, conseguimos realizar verificações que não são possíveis apenas utilizando "if" e "else".
- C) Essa instrução é uma forma de reduzir a complexidade proveniente da utilização de vários "if" e "else".
- D) Essa instrução é utilizada para testar várias opções de condicionais.

2. OBSERVE O FRAGMENTO DE CÓDIGO A SEGUIR. APÓS A SUA EXECUÇÃO, QUAL O VALOR DA VARIÁVEL CONT – EXIBIDA NA INSTRUÇÃO "ALERT(CONT)"?

```
VAR CONT = 1;  
DO{  
  CONT += 1;  
}WHILE (CONT < 10);  
ALERT(CONT);
```

- A) 10
- B) 1
- C) 9
- D) 11

GABARITO

1. No que concerne às estruturas de decisão, mais precisamente à instrução “switch”, assinale a afirmação incorreta:

A alternativa **"B "** está correta.

A “switch”, assim como as instruções “if/else”, permite que o fluxo de um programa seja alterado a partir de verificações de condicionais. Logo, tais instruções não se diferem, sendo a "switch" mais utilizada quando há muitas condições a serem verificadas, diminuindo assim a complexidade do código caso fosse utilizado “if/else”.

2. Observe o fragmento de código a seguir. Após a sua execução, qual o valor da variável cont – exibida na instrução "alert(cont)"?

```
var cont = 1;  
do{  
  cont += 1;
```

```
}while (cont < 10);
```

```
alert(cont);
```

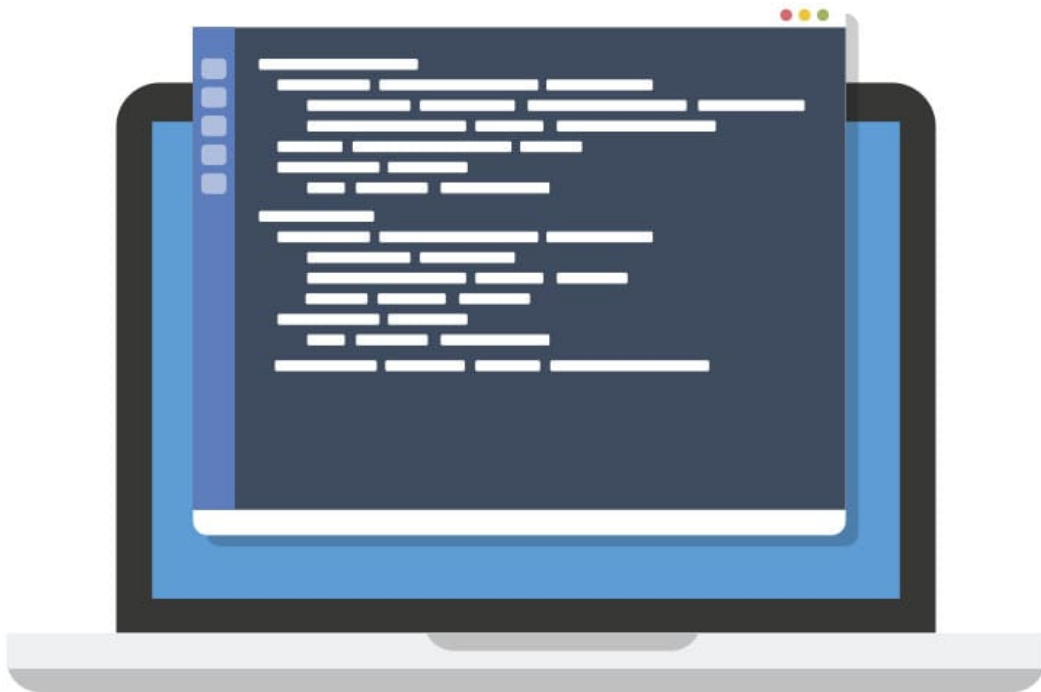
A alternativa **"A "** está correta.

O laço "do/while" executa a primeira instrução antes de testar a condição fornecida. Nesse caso, a instrução consiste em incrementar, de 1 em 1, o valor da variável "cont". Como se inicia em 1 e vai até 9, ao final o seu valor será 10.

MÓDULO 3

- ⦿ Identificar o conceito de vetor e sua utilização em JavaScript

VETORES EM JAVASCRIPT



Fonte: pixcon/Shutterstock

Um vetor é uma estrutura de dados simples utilizada para armazenamento de objetos do mesmo tipo. É chamado, na literatura relacionada a linguagens de programação, de *array*. Nesse contexto, é normalmente tratado em conjunto com outra estrutura: **a matriz**. Em linhas gerais:

Um vetor é um *array* unidimensional.



A matriz é um *array* multidimensional (um vetor de vetores).

A função prática de um vetor é simplificar a utilização de variáveis.

No exemplo do módulo anterior, vimos que eram necessárias duas variáveis para guardar os números solicitados ao usuário. Com a utilização de um vetor, porém, precisaríamos de apenas uma variável, de duas posições.

ATENÇÃO

Embora possa parecer sem importância o uso de *arrays* nesse caso, imagine, por exemplo, que seja necessário armazenar as notas de 50 alunos para, ao final, calcular as respectivas médias. Seriam necessárias 50 variáveis (ou apenas 1 *array*).

SAIBA MAIS

Cabe destacar, ainda, que os vetores são vistos também na Matemática: “Tabela organizada em linhas e colunas no formato **m x n**, sendo **m** o número de linhas e **n**, o de colunas.”

COMPOSIÇÃO DOS VETORES

Um vetor é composto por uma coleção de valores, onde cada um deles é chamado de **elemento**. Além disso, cada elemento possui uma **posição numérica** dentro do vetor, conhecida como **índice**. Veja, no exemplo a seguir, usando notação da linguagem JavaScript, um *array* contendo nomes de frutas:

```
1 var frutas = ['Laranja', 'Uva', 'Limão'];
```

Nesse exemplo, “Laranja”, “Uva” e “Limão” são os elementos do vetor “frutas”. Considerando que o índice de um *array* inicia-se em 0, temos: o conteúdo do vetor “frutas” na posição/índice 0 é “Laranja”; na posição/índice 1 é “Uva”; e na posição/índice 2 é “Limão”.

A seguir, veremos como declarar e utilizar vetores na linguagem JavaScript.

CRIAÇÃO DE VETORES EM JAVASCRIPT

Em JavaScript, os vetores não possuem tipo, a exemplo do que vimos quando tratamos das variáveis. Logo, é possível criar um *array* composto por números, *strings*, objetos e até mesmo outros *arrays*.

Em JS, um vetor pode ter, no máximo, 4.294.967.295 ($2^{32} - 2$) elementos. Outra característica importante é que, em JavaScript, os *arrays* possuem tamanho dinâmico, ou seja, não é necessário informar o tamanho do vetor ao declará-lo.

Vejamos mais alguns exemplos de criação de vetores em JS:

```
1  var alunos = []; //array vazio
2  var alunos = ['Alex', 'Anna', 'João']; // array de strings
3  var notas = [10.0, 9.5, 9.5]; // array de números decimais
4  var mistura = ['Um', 2, 3, 'Quatro']; //array de diversos tipos de dados
```

Outra forma de criação de vetores em JavaScript é usando o construtor (conceito relacionado à programação orientada a objetos) Array. Vejamos o exemplo:

```
1  var alunos = new Array();
2  var alunos = new Array('Alex', 'Anna', 'João');
```

ACESSO E EXIBIÇÃO DE ELEMENTOS DO VETOR

Em termos de acesso aos elementos de um *array*, a forma mais simples é utilizando o seu índice. Vamos ao exemplo:

```
1  var alunos = ['Alex', 'Anna', 'João']; // array de strings
2  alert(alunos[0]); // exibirá “Alex” na caixa de diálogo
```

A função “alert”, imprimirá o conteúdo da posição zero do *array* “alunos”, ou seja, “Alex”;

Seguindo a mesma lógica, se quiséssemos imprimir “João”, utilizaríamos o índice 2;

Outra forma de acessar e exibir os elementos de um vetor é usando um laço de repetição. Veja novamente o exemplo contido na Figura 5.

O JavaScript possui métodos nativos para tratamento de *arrays*. Em termos de acesso e manipulação, veremos agora como utilizar o ***push***. Nos próximos tópicos, outros métodos serão apresentados.

PUSH

Para compreender em que situações o método ***push*** pode ser útil, vamos voltar ao nosso vetor “alunos”. Imagine que, após ter sido declarado inicialmente com 3 valores, seja necessário incluir novos valores a esse *array*, em tempo de execução. O método *push*, cuja sintaxe pode ser vista logo a seguir, nos auxilia nessa tarefa.

```
nome_do_array.push(valor)
```

Usando nosso *array* de exemplo, poderíamos adicionar um novo elemento desta forma:

```
1 alunos.push('Helena');
```

É possível, ainda, inserir múltiplos valores utilizando *push*:

```
1 alunos.push('Helena', 'Maria');
```

OUTRAS FORMAS DE ADICIONAR ELEMENTOS A UM VETOR

Como mencionado, há outras maneiras de adicionar elementos a um *array* de forma dinâmica. A primeira delas pode ser vista a seguir:

```
1  frutas[frutas.length] = 'Maria';
```

Nesse caso, devemos utilizar o tamanho do *array* para informar que desejamos adicionar um novo elemento. Isso pode ser feito informando o número, caso o saibamos, ou de forma dinâmica, usando a propriedade ***length*** – que retorna justamente o tamanho do *array*. Essa importante propriedade será apresentada logo adiante.

SPLICE

O ***splice*** é um método multiuso em JavaScript. Ele serve tanto para **excluir elementos de um *array***, como veremos a seguir, como para *substituir e inserir*. Sua sintaxe é:

```
Array.splice(posição,0,novo_elemento,novo_elemento,...)
```

Onde:

‘**posição**’ é o índice onde o novo elemento será incluído;

‘**0**’ indica ao método que nenhum elemento do *array* será excluído;

‘**novo_elemento**’ é o novo elemento que se deseja adicionar ao *array*.

Vejamos um exemplo prático:

```
1  var alunos = ['Alex', 'Anna', 'João'];
2  alunos.splice(3,0,'Helena');
3  alert(alunos); //imprimirá 'Alex', 'Anna', 'João', 'Helena'
```

Além disso, com esse método também é possível **substituir** um dos elementos do *array*. Veja o exemplo a seguir:

```
1   var alunos = ['Alex', 'Anna', 'João'];
2   alunos.splice(1,1,'Helena');
3   alert(alunos); //imprimirá 'Alex, 'Helena', 'João'
```

Aqui, ao passarmos o número 1 como segundo parâmetro, informamos ao método que um elemento, o de índice 1, deveria ser excluído. Entretanto, como inserimos ao final o nome 'Helena', o método realizou a substituição do elemento excluído pelo novo elemento inserido.

A propriedade *length*

Uma das necessidades mais comuns quando se trabalha com *arrays* é **saber o seu tamanho**. Como vimos em alguns de nossos exemplos, em JavaScript está disponível a propriedade *length*, que retorna o tamanho, ou número de elementos, de um *array*.

Sua sintaxe é:

```
nome_do_array.length
```

REMOÇÃO DE ELEMENTOS DO VETOR

A remoção de elementos de um *array*, em JavaScript, pode ser feita com a utilização do método nativo ***delete***. Vejamos como esse método funciona utilizando nosso *array* de exemplo:

```
1   delete frutas[0];
```

Como visto, sua sintaxe é composta pelo nome do método, ***delete***, pelo nome do ***array*** e pelo **índice do elemento** que queremos remover.

Esse método possui uma particularidade: embora o valor seja excluído do *array*, este não é 'reorganizado', permanecendo com o mesmo tamanho.

Faça o teste:

Utilize o método ***delete*** para remover um elemento de um vetor.



Em seguida imprima (utilizando ***alert***, por exemplo) o tamanho do *array* (usando a propriedade *length*).



Veja que o tamanho do *array* permanece igual ao inicial, antes da utilização do *delete*.

Isso acontece porque **esse método não remove o valor, apenas o torna indefinido (*undefined*)**.

OUTROS MÉTODOS PARA REMOVER ELEMENTOS DO VETOR

A linguagem JavaScript possui, além de "***delete***", outros 4 métodos para remoção de elementos, conforme veremos a seguir:

POP

Este método, que **não recebe parâmetros**, remove um elemento do final do *array*, atualizando seu tamanho. Sua sintaxe é:

```
frutas.pop();
```

SHIFT

Embora similar ao **pop**, este método **remove um elemento do início do array**. Após a remoção, este é **reindexado** (ou seja, o elemento de índice 1 passa a ser o de índice 0 e assim sucessivamente). Além disso, **o tamanho do array também é atualizado**. Sua sintaxe pode ser vista a seguir:

```
frutas.shift();
```

SPLICE

Este método, introduzido anteriormente, pode ser usado para exclusão de elementos. Para tanto, **ele recebe como parâmetros a quantidade de elementos que se deseja eliminar e o índice a partir do qual estes serão excluídos**. A sintaxe a seguir demonstra a remoção de 2 elementos, a partir do índice 2, do *array* fornecido:

```
var primos = [2,3,5,7,11,13,17];  
  
primos.splice(2,2);  
  
alert(primos); //imprimirá 2,3,11,13,17
```

Nesse método, para fins de remoção, **o primeiro parâmetro indica o índice e o segundo, a quantidade de elementos a serem excluídos**.

OUTRAS FORMAS DE REMOVER ELEMENTOS DO VETOR

Existem outras maneiras para excluir elementos de um *array*. Uma forma simples é **determinar o tamanho, utilizando a propriedade *length*, do array**. Isso fará com que este seja **reduzido ao novo tamanho informado**. Vejamos o exemplo prático:

```
var primos = [2,3,5,7,11,13,17];  
  
alert(primos.length); //imprimirá 7  
  
primos.length = 4;
```

```
alert(primos.length); //imprimirá 4
```

Nesse exemplo, ao definirmos o tamanho do *array* como 4, este será reduzido, sendo mantidos os elementos do índice 0 ao 3 e excluídos os demais.

⊕ SAIBA MAIS

Existe, ainda, outro método para a remoção de elementos de um *array*: ***filter***. Entretanto, **ele não modifica o vetor original, mas cria um novo a partir deste**. Esse método utiliza uma sintaxe mais complexa, assim como conceitos e funções de *call-back* que fogem ao escopo deste tema.

TEORIA NA PRÁTICA

Conhecendo as estruturas de decisão e de repetição e os conceitos e exemplos de utilização de vetores, chegou a hora de praticar.

Teremos, a seguir, dois exercícios:

O primeiro será mais simples;

O segundo será um desafio, pois exigirá a pesquisa e o estudo de conteúdos adicionais, uma vez que não vimos todos os conceitos necessários para resolvê-lo.

Vamos a eles:

1. Junte em um único arquivo todos os códigos vistos ao longo deste último módulo e execute-os novamente. Isso ajudará a fixar o conteúdo.

RESOLUÇÃO

```
1  var frutas = ['Laranja', 'Uva', 'Limão'];
```

```
2
```

3

4 var alunos = []; //array vazio

5 var alunos = ['Alex', 'Anna', 'João']; // array de strings

6 var notas = [10.0, 9.5, 9.5]; //array de números decimais

7 var mistura = ['Um', 2, 3, 'Quatro']; //array de diversos tipos de dados

8

9

10 var alunos = new Array();

11 var alunos = new Array('Alex', 'Anna', 'João');

12

13 var alunos = ['Alex', 'Anna', 'João']; // array de strings

14 alert(alunos[0]); //exibirá “Alex” na caixa de diálogo

15

16

17 alunos.push('Helena');

18

19

20 alunos.push('Helena', 'Maria');

21

22

23 frutas[frutas.length] = 'Maria';

24

25

26 var alunos = ['Alex', 'Anna', 'João'];

27 alunos.splice(3,0,'Helena');

28 alert(alunos); //imprimirá 'Alex', 'Anna', 'João', 'Helena'

29

```
30
31  var alunos = ['Alex', 'Anna', 'João'];
32  alunos.splice(1,1,'Helena');
33  alert(alunos); //imprimirá 'Alex', 'Helena', 'João'
34
35  delete frutas[0];
36
37  frutas.pop();
38
39  frutas.shift();
40
41
42  var primos = [2,3,5,7,11,13,17];
43  primos.splice(2,2);
44  alert(primos); //imprimirá 2,3,11,13,17
45
46
47  var primos = [2,3,5,7,11,13,17];
48  alert(primos.length); //imprimirá 7
49  primos.length = 4;
50  alert(primos.length); //imprimirá 4
```

Baixe aqui o código usado na exercício.

2. Faça um programa em JavaScript que:

Solicite ao usuário que insira dois números inteiros positivos;

Utilize um vetor para armazenar esses dois números;

Verifique se os números inseridos são inteiros positivos. Caso contrário, solicite ao usuário para inseri-los novamente;

Divida os dois números inteiros positivos;

Imprima na tela o resultado da divisão.

Observações:

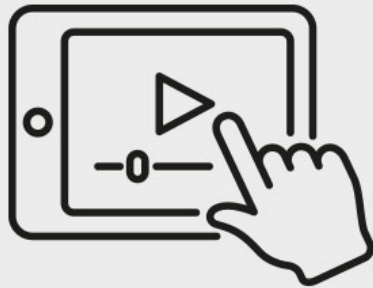
Há mais de uma forma de desenvolver o programa. Logo, não há código certo ou errado.

Você pode utilizar funções JavaScript para melhor organizar o seu código. Inclusive, pode usar um pouco de recursividade.

O código relacionado a uma das formas de resolver esse desafio está disponível a seguir:

RESOLUÇÃO

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



```
1
2 //Declaração do array 'numeros' sem tamanho definido e sem elementos atribuídos
3 var numeros = [];
4
```



```
5 //O primeiro elemento (o de índice 0) recebe o retorno da função que solicita o primeiro número
6 numeros[0] = soliticaPrimeiroNumero();
7
8 //O segundo elemento (o de índice 1) recebe o retorno da função que solicita o segundo número
9 numeros[1] = soliticaSegundoNumero();
10
11 //Declaração de atribuição de valor à variável que armazenará o resultado da divisão
12 //O resultado da divisão virá da função 'divida' (esse função recebe como parâmetro o array 'numeros')
13 var resultadoDivisao = divida(numeros);
14
15 //Exibindo o resultado da divisão na tela
16 alert('O resultado da divisão é igual a: ' + resultadoDivisao);
17
18 /*
19 Função Javascript
20 Esta função não recebe parâmetros
21 */
22 function soliticaPrimeiroNumero(){
23     //Declaração e atribuição de variável. Ela receberá o número inserido pelo usuário
24     var numero1 = prompt("Insira o primeiro número: ");
25
26     //Condição para verificar se o número é positivo.
27     //Caso não, o retorno da função será chamar a própria função novamente.
28     // Esta operação será repetida até que um número válido seja inserido.
29     //Caso sim, retorna o valor inserido pelo usuário
30     if(numero1 < 0){
31         alert("O número precisa ser inteiro e positivo");
```

```
32
33     //Este tipo de retorno, onde a própria função é chamada novamente, é conhecido como recursividade
34     return soliticaPrimeiroNumero();
35 }else{
36     return numero1;
37 }
38 }
39
40 function soliticaSegundoNumero(){
41     var numero2 = prompt("Insira o segundo número: ");
42
43     if(numero2 < 0){
44         alert("O número precisa ser inteiro e positivo");
45         return soliticaSegundoNumero();
46     }else{
47         return numero2;
48     }
49 }
50
51 /*
52 Esta função recebe como parâmetro um array - que contém os 2 números que desejamos dividir
53 */
54 function divida(numeros){
55     var resultado = 0;
56
57     //Os números a serem divididos são acessados através dos índices do array
58     resultado = numeros[0] / numeros[1];
```

```
59     return resultado;  
60 }
```

Baixe aqui o código usado no exercício.

VERIFICANDO O APRENDIZADO

1. EM RELAÇÃO AOS CONCEITOS E AO USO DE VETORES EM JAVASCRIPT, ASSINALE A AFIRMATIVA INCORRETA:

- A) Um vetor, ou *array*, é um grupo de variáveis que contém valores, do mesmo tipo ou de tipos diferentes.
- B) Um *array*, em JavaScript, só permite dados do mesmo tipo.
- C) Em JavaScript, o primeiro elemento de um *array* tem o índice igual a 0.
- D) Em JavaScript, podemos acessar o último elemento de um *array* da seguinte forma: `vetor[vetor.length-1]`.

2. DESEJA-SE EXCLUIR O ÚLTIMO ELEMENTO DO ARRAY ABAIXO. ASSINALE A ALTERNATIVA CUJO MÉTODO NÃO PODE SER APLICADO PARA REALIZAR ESSA AÇÃO:

VAR PARES = [2,4,6,8,10,12];

- A) `pares.pop()`
- B) `pares.splice(5,1)`

C) pares.length = 5

D) pares.splice(6,0,0)

GABARITO

1. Em relação aos conceitos e ao uso de vetores em JavaScript, assinale a afirmativa incorreta:

A alternativa "B " está correta.

JavaScript permite que um *array* seja composto por dados de diferentes tipos.

2. Deseja-se excluir o último elemento do array abaixo. Assinale a alternativa cujo método não pode ser aplicado para realizar essa ação:

var pares = [2,4,6,8,10,12];

A alternativa "D " está correta.

Como visto, o método *splice* pode ser utilizado tanto para remover quanto para adicionar ou substituir elementos de um *array*. Quando usado para remover, sua sintaxe corresponde ao código visto na alternativa 'b', na qual indicamos o índice e a quantidade de elementos, a partir dele, a ser removida. Já a alternativa 'd' faz com que seja adicionado um novo elemento, com valor 0, após o índice 6.

MÓDULO 4

🕒 Reconhecer os recursos assíncronos Ajax e JSON

REQUISIÇÕES ASSÍNCRONAS E SÍNCRONAS



Fonte: Eny Setiyowati/Shutterstock

O conceito de **requisição**, no ambiente Web, **diz respeito às informações solicitadas ou submetidas no lado cliente** – através do navegador, por exemplo – **e tratadas pelo lado servidor, que após processar a requisição devolverá uma resposta ao solicitante**. Nesse sentido, são possíveis dois tipos de requisições:

REQUISIÇÕES SÍNCRONAS

Quando realizadas, estas bloqueiam o remetente. Ou seja, o cliente faz a requisição e fica impedido de realizar qualquer nova solicitação até que a anterior seja respondida pelo servidor. Com isso, só é possível realizar uma requisição de cada vez.

REQUISIÇÕES ASSÍNCRONAS

Nestas não existe sincronismo. Logo, várias requisições podem ser realizadas em simultâneo, independentemente de ter havido resposta do servidor às solicitações anteriores.

 **ATENÇÃO**

Em comparação com as requisições síncronas, **deve-se dar preferência à utilização das assíncronas**. Isso porque estas últimas não apresentam os problemas de desempenho e congelamento do fluxo da aplicação, naturais das síncronas.

COMO AS REQUISIÇÕES ASSÍNCRONAS FUNCIONAM NA PRÁTICA

Para melhor entendimento de como as requisições assíncronas funcionam na prática, tomemos como exemplo **o feed das redes sociais**. Em tais páginas, novos conteúdos são carregados sem que seja necessário recarregar o navegador e, conseqüentemente, todo o conteúdo visto.

Eventos como a rolagem de tela, por exemplo, fazem com que os conteúdos sejam carregados do servidor e exibidos na tela do dispositivo.

AJAX - REQUISIÇÕES ASSÍNCRONAS EM JAVASCRIPT

Em JavaScript, quando falamos de requisições assíncronas, naturalmente falamos de AJAX. Esse termo foi empregado pela primeira vez em 2005 e engloba o uso não de uma, mas de várias tecnologias: **HTML (ou XHTML), CSS, JavaScript, DOM, XML (e XSLT), além do elemento mais importante, o objeto XMLHttpRequest**.

A utilização de AJAX permite que as páginas e aplicações Web façam requisições a scripts do lado servidor e carreguem, de forma rápida e muitas vezes incremental, novos conteúdos sem que seja necessário recarregar a página inteira.

Embora o “X” no acrônimo se refira a XML, esse não é o único formato disponível. Além dele, temos: **o HTML, arquivos de texto e o JSON**, sendo esse último o mais utilizado atualmente. Veremos sobre ele mais adiante.

Em relação aos recursos para realização de requisições, há dois disponíveis em JS: **o objeto XMLHttpRequest e a interface Fetch API**.

XMLHTTPREQUEST

Inicialmente, foi implementado no navegador Internet Explorer através de um objeto do tipo **ActiveX**. Posteriormente, outros fabricantes fizeram suas implementações, dando origem ao **XMLHttpRequest**, que se tornou o padrão atual.

💡 DICA

Versões antigas do Internet Explorer só possuem suporte ao ActiveX.

O XMLHttpRequest possui alguns métodos e propriedades. Alguns deles serão descritos após vermos um exemplo simples de sua utilização:

```
1  <!doctype html>
2  <html lang="pt-BR">
3  <head>
4    <meta charset="utf-8">
5    <title>Requisição XMLHttpRequest</title>
6  </head>
7  <body>
8    <h1>Imagens Aleatórios de Cachorros</h1>
9    <p>
```

A partir do click no botão abaixo uma nova imagem aleatória de cachorros será carregada utilizando requisições assíncronas com XMLHttpRequest</p>

```
10
11    <img id="img_dog" src="" alt="Aguardando a imagem ser carregada" />
12    <br/>
13    <button onclick="carregarImagens()">Carregar Imagens</button>
14  </body>
```

```
15     <script type="text/javascript">
16
17     function carregarImagens(){
18
19         var xmlhttpRequest = new XMLHttpRequest();
20         var url = "https://dog.ceo/api/breeds/image/random"
21
22         xmlhttpRequest.open("GET", url, true);
23
24         xmlhttpRequest.onreadystatechange = function() {
25             if (xmlhttpRequest.readyState == 3) {
26                 console.log('Carregando o conteúdo');
27             }
28             if (xmlhttpRequest.readyState == 4) {
29
30                 var jsonResponse = JSON.parse(xmlhttpRequest.responseText);
31
32                 console.log('Requisição Finalizada');
33                 console.log('Resultado da Requisição: ' + jsonResponse);
34                 console.log('Resultado da Requisição: ' + jsonResponse.message);
35
36                 var imgDog = document.getElementById("img_dog");
37                 imgDog.src = jsonResponse.message;
38             }
39         };
40
41         xmlhttpRequest.send(null);
```



```
42
43     }
44     </script>
45 </html>
46
47
48
49
```

📷 Figura 6: Exemplo de requisição utilizando XMLHttpRequest.

📢 ATENÇÃO

Baixe **aqui** o código usado na Figura 6.

O código na Figura 6 contém tanto funcionalidades **JavaScript** vistas neste tema quanto algumas novas, além do **XMLHttpRequest**, que veremos mais adiante. Ao utilizar esse código, você terá um exemplo real de dados sendo requisitados a um servidor – nesse caso, uma **API** que retorna imagens aleatórias de cachorros – e exibidos na página, sem que ela seja recarregada a cada nova requisição/click no botão.

Por ora, vamos nos concentrar no **XMLHttpRequest**. Utilizando as linhas contidas no código presente na figura anterior, veja:

Na **linha 19** uma instância do objeto é criada. Esse é o primeiro passo para sua utilização. A partir desse ponto, toda referência deverá ser feita pelo nome da variável utilizada (em nosso exemplo, `xmlHttpRequest`);

A **linha 22** mostra a utilização do método *open*, que recebe 3 parâmetros:

O método de requisição dos dados.

A url remota/do servidor que queremos acessar.

O tipo de requisição – onde “*true*” define que será feita uma requisição assíncrona e “*false*”, uma síncrona. Esse argumento é opcional. Logo, pode não ser definido, assumindo o valor padrão “*true*”.

Continuando o código, na **linha 24** temos a propriedade “**onreadystatechange**”, que monitora o status da requisição XMLHttpRequest – propriedade “**readyState**” – e especifica uma função a ser executada a cada mudança;

Repare, agora, na **linha 25**: o status 3 significa que a requisição ainda está sendo processada. Logo, poderíamos, por exemplo, exibir em nossa tela uma mensagem (ou imagem, como é muito comum) avisando que a informação requisitada está sendo carregada. Perceba que, dependendo do tempo de resposta do servidor remoto, nem sempre será possível ver essa informação;

Já na **linha 28** temos o tratamento do status quando ele for igual a 4, ou seja, quando a requisição estiver concluída. Além da propriedade “**readyState**”, poderíamos também monitorar a propriedade “**status**”, que armazena o código de resposta do servidor Http utilizado pela XMLHttpRequest;

Seguindo o código de exemplo, na **linha 30** vemos a propriedade “**responseText**”. Além dela, está disponível também a “**responseXML**”. Ambas dizem respeito a como trataremos o retorno da requisição: se como texto, no caso da primeira, ou como XML, no caso da segunda;

Ainda na **linha 30**, repare que também foi utilizado outro método, o **parse**. Esse método não pertence ao objeto XMLHttpRequest, mas este é necessário quando o recurso requisitado devolve o conteúdo em formato JSON;

Por fim, na **linha 41** é utilizado o método *send*, que envia a requisição.

OUTROS MÉTODOS E PROPRIEDADES

A Figura 6 mostra um exemplo simples do que é possível fazer utilizando AJAX. Além disso, apenas algumas propriedades e métodos foram vistos.

⊕ SAIBA MAIS

Na seção Explore +, estão disponíveis sugestões de conteúdo que permitirão um aprofundamento nesse tema. É recomendável a leitura desse material, assim como a implementação e até mesmo modificação do código anterior para uma melhor assimilação do conteúdo.

API FETCH

Essa API é, em termos conceituais, similar à XMLHttpRequest – ou seja, **permite a realização de requisições assíncronas a scripts do lado servidor**. Entretanto, por ser uma implementação mais recente, essa interface JavaScript apresenta algumas vantagens, como:

- O uso de ***promise***;

- O fato de poder ser utilizado em outras tecnologias, como ***service workers***, por exemplo.

A Figura 7 apresenta o mesmo exemplo utilizado na Figura 6, mas substituindo o XMLHttpRequest pela API Fetch. A seguir, alguns métodos e propriedades serão apresentados:

PROMISE


O *promise* (promessa) é um objeto utilizado para processamento assíncrono, representando um valor que pode estar disponível agora, no futuro ou nunca.

SERVICE WORKERS

Scripts executados em segundo plano no navegador, separados da página web.

```
1  <!doctype html>
2  <html lang="pt-BR">
3  <head>
4    <meta charset="utf-8">
5    <title>Requisição XMLHttpRequest</title>
6  </head>
7  <body>
8    <h1>Imagens Aleatórios de Cachorros</h1>
9    <p>A partir do click no botão abaixo uma nova imagem aleatória de cachorros será carregada utilizando requisições assíncronas com XMLHttpRequest</p>
10
11    <img id="img_dog" src="" alt="Aguardando a imagem ser carregada" />
12    <br/>
13    <button onclick="carregarImagens()">Carregar Imagens</button>
14  </body>
15  <script type="text/javascript">
16
17    function carregarImagens(){
18
19      var url = "https://dog.ceo/api/breeds/image/random"
20      fetch(url, {
```

```
21         method: 'get'
22     })
23     .then(function(response) {
24         response.json().then(function(data){
25             console.log('Resultado da Requisição: ' + data.message);
26
27             var imgDog = document.getElementById("img_dog");
28             imgDog.src = data.message;
29         });
30     })
31     .catch(function(err) {
32         console.error('O seguinte erro ocorreu durante a requisição: ' + err);
33     });
34
35 }
36 </script>
37 </html>
38
39
40
41
```

 Figura 7: Exemplo de requisição utilizando API Fetch.

ATENÇÃO

Baixe [aqui](#) o código usado na Figura 7.

Em relação à sua sintaxe, podemos notar algumas semelhanças com a XMLHttpRequest:

URL do servidor remoto, definida na **linha 19** e utilizada na **linha 21**;

Fetch options – em nosso exemplo utilizamos apenas um parâmetro, **o método**. Tal parâmetro, inclusive, é opcional. Veja a **linha 21**, onde declaramos o **GET**, que é o método padrão. Além desse parâmetro, há outros disponíveis;

Tipo de dado retornado pela requisição. Veja a **linha 24**, onde foi utilizado o objeto correspondente ao tipo de dado retornado pela requisição – nesse caso, **JSON**. Há outros tipos de objetos, como texto e até mesmo bytes, sendo possível, por exemplo, carregar imagens, arquivos pdf, entre outros.

ATENÇÃO

Ainda em relação à sintaxe, merece destaque a forma, própria, como a API Fetch trata a requisição (*request*) e o seu retorno (*response*): **quando o método *fetch* é executado, ele retorna uma promessa (*promise*) que resolve a resposta (*response*) à requisição, sendo esta bem-sucedida ou não.**

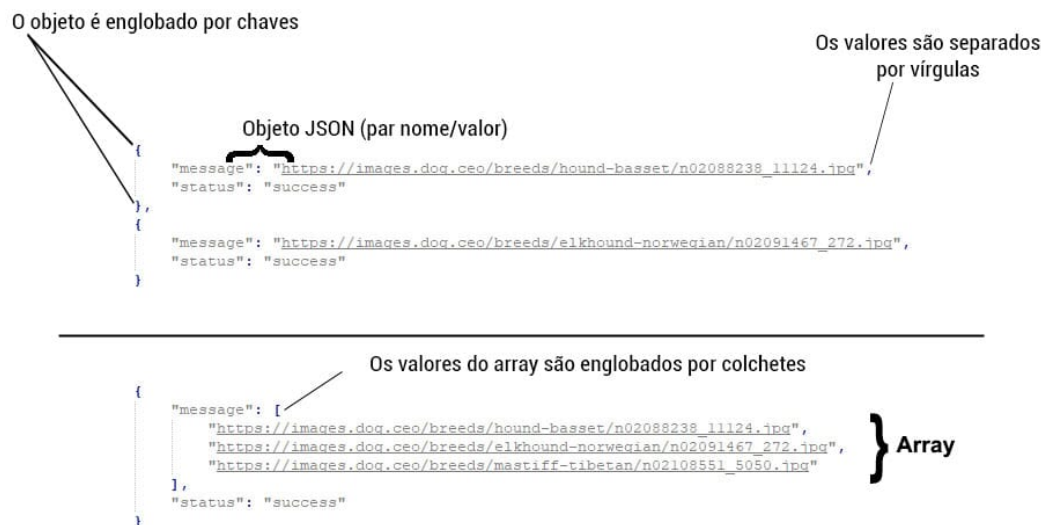
JSON

JSON pode ser traduzido para Notação de Objetos JavaScript. Trata-se de **um tipo, ou formatação, leve para troca de dados**. Essa, inclusive, é sua principal vantagem em relação aos outros tipos. Além disso, destaca-se também a sua **simplicidade**, tanto para ser interpretado por pessoas quanto por “máquinas”.

COMENTÁRIO

O JSON, embora normalmente associado ao JavaScript – tendo sido definido, inclusive, na especificação ECMA-262, de dezembro de 1999 –, é um formato de texto, independente de linguagem de programação. Essa facilidade de uso em qualquer linguagem contribuiu para que se tornasse um dos formatos mais utilizados para a troca de dados.

A Figura 8 apresenta alguns exemplos da estrutura de um objeto JSON:



Fonte: autor

📷 Figura 8: Estrutura de objetos JSON.

Como visto na imagem, um objeto JSON tem as seguintes características:

É composto por um conjunto de pares nome/valor. Na imagem, “*status*” é o nome de um objeto e “*success*”, o seu valor. Esses pares são separados por dois pontos “:”;

Para separar pares, valores ou objetos é utilizada a **vírgula**;

O objeto e seus pares são englobados por chaves “{}”;

É possível definirmos *arrays*. Esses são englobados por colchetes “[]”.

O JSON fornece suporte a uma gama de tipos de dados. Além disso, possui alguns métodos, como o *JSON.parse()*, visto em um de nossos exemplos.

SAIBA MAIS

A seção Explore + sugere materiais adicionais sobre o tema. A leitura desse material é fortemente recomendada.

TEORIA NA PRÁTICA

Como mencionado, os códigos utilizados para a apresentação do XMLHttpRequest e da API Fetch são funcionais. Copie esses códigos e os execute para ver, na prática, o comportamento das requisições assíncronas. Você pode copiar direto do box, ou utilizar o arquivo Figura 8.

Caso queira avançar um pouco mais, inclua outros elementos de interação na página que se comuniquem com servidores remotos;

Outra boa ideia é pesquisar e utilizar alguma outra API que retorne outro tipo de conteúdo, diferente do que foi mostrado.

RESOLUÇÃO

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Baixe aqui o código usado na Figura 8.

FRAMEWORKS JAVASCRIPT

A exemplo do que vimos nas demais tecnologias do lado cliente, no ambiente Web, existem inúmeros frameworks disponíveis também para JavaScript.

+ SAIBA MAIS

Embora não faça parte do escopo deste tema, cabe destacar que a utilização desses recursos é uma grande aliada no desenvolvimento Web. Logo, sempre que possível, é recomendada.

VERIFICANDO O APRENDIZADO

1. SOBRE AS REQUISIÇÕES ASSÍNCRONAS EM JAVASCRIPT – AJAX, É INCORRETO AFIRMAR QUE:

- A) Essas requisições tornam a interação na página mais lenta, já que dependem do retorno de dados que são requisitados ao servidor.
- B) Várias requisições podem ser realizadas a um mesmo servidor em paralelo.
- C) O objeto utilizado para realização da requisição fica aguardando o retorno do servidor e é capaz de processar esse retorno, sendo esse bem-sucedido ou não.
- D) As requisições assíncronas não bloqueiam o cliente – por exemplo, o navegador Web –, permitindo que outras operações sejam realizadas enquanto se aguarda o retorno da requisição.

2. A RESPEITO DO JSON, É CORRETO AFIRMAR QUE:

- A) O JSON é um formato leve de troca de informações e dados entre sistemas.
- B) Esse formato, cujo nome vem de JavaScript Object Notation, é exclusivo para a transmissão de dados na linguagem JavaScript.
- C) Quando utilizamos JavaScript, JSON é o único formato de transmissão de dados disponível, uma vez que é nativo desta linguagem.
- D) Não é possível transferir estruturas de dados mais complexas, como arrays, através de JSON.

GABARITO

1. Sobre as requisições assíncronas em JavaScript – AJAX, é incorreto afirmar que:

A alternativa "A " está correta.

Como discutido, as requisições assíncronas tornam a interação mais rápida no cliente, uma vez que a página não fica bloqueada, aguardando o retorno do servidor. Isso torna possível que outras ações, incluindo novas requisições, sejam realizadas.

2. A respeito do JSON, é correto afirmar que:

A alternativa "**A**" está correta.

JSON é uma notação simples para troca de dados. Embora proveniente de uma especificação JavaScript, não é exclusivo desta linguagem.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste tema, abordamos a linguagem de programação lado cliente do JavaScript. Ao longo dos módulos, vimos conceitos e sintaxes – propriamente ditos ou a ela relacionados, como a interface DOM e as requisições assíncronas, desde as formas de inclusão em arquivos HTML à declaração e utilização de variáveis.

Estudamos, ainda, outras estruturas de dados, como os *arrays*, e os conceitos de programação relacionados à função das estruturas de decisão e de repetição, bem como suas aplicações em JavaScript. Ao final, pudemos conhecer alguns recursos avançados, como o AJAX e o JSON. Para maior compreensão do tema, cada módulo foi amparado com a apresentação de exemplos práticos e funcionais. Assim, temos certeza de que os conhecimentos sobre o JavaScript aqui consolidados o ajudarão em sua vida acadêmica e profissional.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



AVALIAÇÃO DO TEMA

REFERÊNCIAS

BARBOSA, A. **DOM**. Publicado em: 4 set. 2017.

FLANAGAN, D. **JavaScript**: The Definitive Guide. Califórnia: O'Reilly Media, 2011.

RAUSCHMAYER, A. **Speaking JavaScript**. Califórnia: O'Reilly Media, 2014.

EXPLORE+

Para saber mais sobre DOM, leia os textos:

Modelo de Objeto de Documento (DOM) e **Examples of Web and XML development using the DOM**, da comunidade Mozilla.

What is the Document Object Model, **THE HTML DOM Document Object** e **JavaScript HTML DOM**, do site W3schools.

Acesse a jQuery, uma biblioteca de funções JavaScript.

Leia **AJAX – The XMLHttpRequest Object**, do site W3 schools.

Para aprender mais sobre JSON, leia:

JSON – Introduction, do site W3schools.

JSON e **Trabalhando com JSON**, da comunidade Mozilla.

Acesse os sites das comunidades CodePen e JSFiddle para testar códigos HTML, CSS e JavaScript.

Para aprofundar seus conhecimentos, leia: **Promise**; **JavaScript – Método Filter**; **Fetch API** e **Usando Fetch**, da comunidade Mozilla.

Leia o texto **Introdução aos service workers**, de Matt Gaunt.

CONTEUDISTA

Alexandre de Oliveira Paixão

 **CURRÍCULO LATTES**