Desenvolvimento de Sistemas

Linguagem de *scripts*: sintaxe, operadores, palavras reservadas, identificadores, delimitadores, comentários, função, passagem de parâmetros e bibliotecas; chamadas assíncronas

Introdução

JavaScript, também conhecida como JS, é uma linguagem de programação criada em 1995 por Brendan Eich para o Netscape, um navegador popular nos anos 1990. A intenção com a linguagem era criar animações e interação. Inicialmente, foi chamada de Mocha, depois de LiveScript e, por fim, de JavaScript. Logo foi adotada pela Microsoft e por seu Internet Explorer, tornando-se, rapidamente, uma linguagem de *script* padrão para desenvolvimento *web*, sendo suportada por todos os navegadores atuais.

Originalmente criada para ser executada no lado do cliente (também conhecido como *front-end*), a tecnologia JavaScript foi expandida e tornou-se executável também no lado do servidor, após o desenvolvimento do Node.js – um ambiente de servidor multiplataforma que permite executar códigos JavaScript fora do navegador *web*. Com isso, muitas bibliotecas começaram a surgir (como Angular, React, React Native, Vue.js e muitas outras), e o JavaScript deixou de ser uma linguagem de programação apenas para a *web*, sendo possível usá-la para desenvolver também aplicações para ambientes *desktop* e dispositivos móveis.

Neste conteúdo, você será introduzido ao mundo do JavaScript e o foco será o desenvolvimento de aplicações *web* com a linguagem pura, isto é, sem uso de bibliotecas.

Muitas pessoas confundem Java e JavaScript porque ambos são linguagens de programação e têm o termo “Java” no nome. Além disso, ambas as linguagens podem ser usadas para criar aplicativos para *web* e têm pontos em comum quanto à sintaxe, o que contribui para a confusão. No entanto, Java e JavaScript têm propósitos diferentes e são usados para diferentes tarefas. Java é mais relevante para criação de aplicativos para *desktops* e servidores (*back-end*), enquanto JavaScript é mais voltada para o *front-end*, na criação de aplicativos da *web* ricos em recursos e interativos.

A principal diferença entre Java e JavaScript é que o Java é uma linguagem de programação de propósito geral, enquanto o JavaScript é uma linguagem de *script*. Embora eles tenham algumas semelhanças, como a sintaxe, Java é uma linguagem de programação orientada a objetos, enquanto JavaScript é orientado a eventos.

Começando com JavaScript

Agora que você já foi introduzido à linguagem JavaScript, chegou o momento de começar a “pôr a mão na massa”. No decorrer deste conteúdo, serão realizadas diversas práticas, então é interessante que você crie uma pasta na sua área de trabalho. Ela será como a pasta de um projeto JavaScript.

Comece criando um arquivo **index.html**, que será a sua página *web*. Dentro desse arquivo, insira o seguinte trecho de código:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
  </body>
</html>
```

Agora, dentro das *tags* **<body>**, insira a *tag* **<script>**, que servirá para dizer ao navegador o que deve ser interpretado como HTML (*hyper text markup language*) e o que deve ser interpretado como JavaScript. Nesse caso, tudo que for escrito entre as *tags* **<script></script>** será interpretado pelo navegador como a linguagem JavaScript. O seu código ficará assim:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <script type="text/javascript">
        // Código JavaScript
    </script>
  </body>
</html>
```

Essa é uma das formas de trabalhar com o JavaScript, incorporando o código diretamente na página HTML.

Outra forma de trabalhar com o JavaScript é referenciando um arquivo de extensão **JS (.js)**. Nesse caso, use a propriedade **src** do HTML para apontar para o arquivo em questão e o seu código ficaria da seguinte maneira:

```
<script type="text/javascript" src="script.js"></script>
```

Nessa abordagem, ainda é possível informar um *script* que esteja fora da estrutura do projeto, passando a URL (*uniform resource locator*) completa. Por exemplo:

```
<script type="text/javascript" src="https://exemplo.com.br/js/script.js"></script>
```

A vantagem de um arquivo separado é que o navegador o baixará e armazenará em seu *cache*. Outras páginas que fazem referência ao mesmo *script* o retirarão do *cache* em vez de baixá-lo, então o arquivo é baixado apenas uma vez. Isso reduz o tráfego e torna as páginas mais rápidas.

Por questões didáticas, aqui você criará seus primeiros códigos JavaScript incorporando-os diretamente no HTML.

Após escrever o código, salve o arquivo e abra-o no seu navegador *web* – para isso, basta clicar no arquivo **index.html** criado e abri-lo com o seu navegador padrão.

Apesar de a *tag* <script> poder ser inserida em qualquer lugar da página HTML, incluindo a área de cabeçalho (*tag* <head>), é uma boa prática sempre inserir a *tag* no final do corpo do HTML, antes do fechamento da *tag* </body>. Lembre-se de que o navegador *web* sempre interpretará uma página *web* na ordem em que o código foi escrito. Se adicionar um código JavaScript que manipula um elemento HTML antes de esse elemento existir para o navegador, o seu código JavaScript provavelmente não funcionará. Portanto, procure sempre deixar o conteúdo da página (no caso, o conteúdo HTML) ser carregado primeiro para, só depois, carregar os *scripts* JavaScript.

Na linguagem Java, usa-se **System.out.print()**; para escrever saídas de dados. No JavaScript, usa-se **console.log()**;, da seguinte forma:

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <script type="text/javascript">
      console.log("Olá, mundo!");
    </script>
  </body>
</html>
```

Se você salvar o arquivo agora e tentar abri-lo depois, perceberá que não aparecerá nada na tela. Isso porque a mensagem foi impressa no console, que só se consegue acessar por meio das ferramentas de desenvolvedor do navegador.

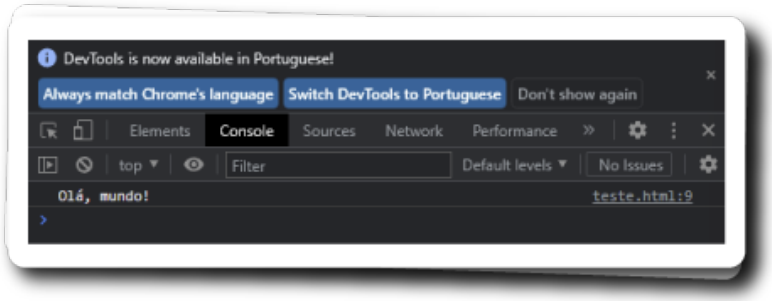

Figura 1 – DevTools do Google Chrome

Fonte: Senac EAD (2023)

Ferramentas de desenvolvedor

Todo código é propenso a erros. Porém, no navegador, os usuários não veem erros por padrão. Portanto, se algo der errado no *script*, você não verá o que está errado e não poderá corrigir. Para que os desenvolvedores possam ver os erros e obter outras informações úteis sobre seus *scripts*, as “ferramentas do desenvolvedor” foram incorporadas aos navegadores.

As ferramentas do desenvolvedor são potentes e oferecem muitos recursos. Para revelar essas ferramentas no navegador *web*, pressione a tecla **F12** ou **Ctrl + Shift + i**. As ferramentas do desenvolvedor serão abertas e você terá acesso a um painel com várias opções. A que nos interessa é a opção **Console**, que funcionará para o JavaScript da mesma forma que o console do NetBeans funciona para o Java.

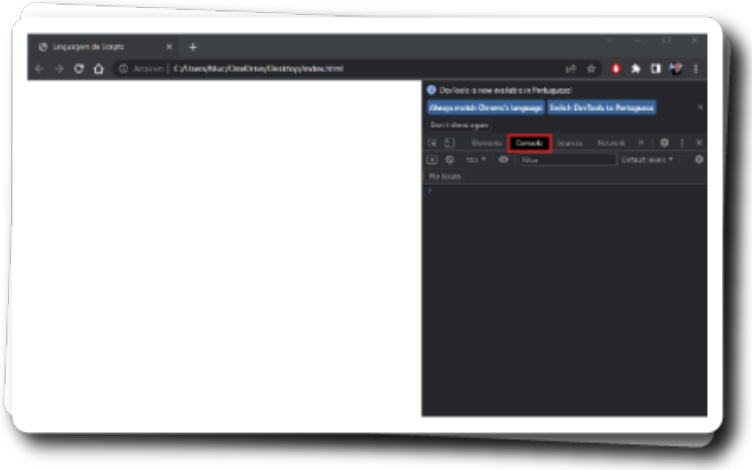

Figura 2 – DevTools do Google Chrome

Fonte: Senac EAD (2023)

Esse recurso é muito útil quando é preciso testar um *script* extenso. Usando o **console.log()** em vários trechos do código, é possível visualizar como o código está sendo lido e descobrir erros que talvez não estejam tão claros à primeira vista.

Para facilitar a visualização dos dados, e não precisar acessar sempre as ferramentas de desenvolvedor, use uma função nativa do JavaScript muito útil, que é a função **alert()**:

```
<html lang="pt-br">
  <head>
```

```
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <script type="text/javascript">
      alert("Olá, mundo!");
    </script>
  </body>
</html>
```

Agora, tente executar a página mais uma vez. Você verá que uma caixa de alerta aparecerá e exibirá a mensagem que foi passada para a função, com um botão **Ok** para dispensá-la.

Figura 3 – Alerta do JavaScript
Fonte: Senac EAD (2023)

A caixa de alerta tira o foco da página atual e força o usuário a ler a mensagem. Esse recurso será muito usado nos exemplos deste conteúdo, mas para fins didáticos. Em um projeto real, evite o uso excessivo do **alert()**, pois isso impede que o usuário acesse outras partes da página até que a caixa de alerta seja fechada, o que pode gerar insatisfações na experiência do usuário.

Por ora, serão tratados os aspectos técnicos da linguagem JavaScript, por isso não haverá mudanças no código HTML. Sendo assim, nos próximos códigos, haverá apenas *scripts* JavaScript. Para reproduzir esses exemplos, basta copiar e colar o código dentro das *tags* **<script>**. Conforme seus estudos forem avançando, serão apresentados exemplos mais complexos, com códigos HTML e JavaScript interagindo um com o outro.

Sintaxe

A linguagem JavaScript tem a sintaxe muito próxima de linguagens de alto nível, por exemplo, a linguagem Java.

Case-sensitive

Assim como a maioria das linguagens de programação, JavaScript é *case-sensitive* – o que significa que palavras com letras maiúsculas e minúsculas são tratadas de forma diferente e, por isso, é preciso prestar bastante atenção na hora de escrever certas palavras. Por exemplo, se uma variável “teste” for referenciada como “Teste” em alguma parte do código, provavelmente haverá um erro.

Palavras reservadas

Em JavaScript, existem as seguintes palavras reservadas, que se referem a elementos da linguagem e que não podem ser usadas pelo programador como nome de variável, classe ou função:

abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instaceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	var	void
volatile	while	with	yield

Tabela 1 – Palavras reservadas
Fonte: Senac EAD (2023)

Comentários

Um comentário é uma parte no código que não é lida ao executar o código. Geralmente é utilizado para o desenvolvedor escrever anotações ou impedir que alguns trechos de código sejam executados. A sintaxe de comentários em JavaScript é semelhante à sintaxe da linguagem Java e pode ser usada de duas formas.

A primeira forma é comentando uma linha inteira. Para isso, usa-se **//** e tudo que vier depois será comentado. Por exemplo:

```
// Esse é um exemplo de comentário em uma linha
```

A segunda forma é usando delimitadores **/*** e ***/** para definir um bloco de comentário. Por exemplo:

```
/* Esse é o inicio do comentário.
Podemos extende-lo o quanto quisermos.
Aqui, finalizamos nosso comentário. */
```

Ponto e vírgula

Em JavaScript, o caractere **ponto e vírgula (;)** pode ser omitido na maioria dos casos quando existe uma quebra de linha. Isso significa que o seu código também funcionará se for escrito da seguinte forma:

```
alert('Olá')
alert('Mundo')
```

Aqui, o JavaScript interpreta a quebra de linha como um ponto e vírgula “implícito”. Isso é chamado de “inserção automática de ponto e vírgula”.

Na maioria dos casos, uma nova linha significará um ponto e vírgula, mas “na maioria dos casos” não significa “sempre”! Sendo assim, há casos em que uma nova linha não significa um ponto e vírgula. Por exemplo:

```
alert(3 +
1
+ 2);
```

O código gerado será **6**, pois o JavaScript não insere ponto e vírgula aqui. É intuitivamente óbvio que, se a linha terminar com um sinal de mais (+), é uma “expressão incompleta”, portanto um ponto e vírgula estaria incorreto e, nesse caso, isso funcionaria como foi pretendido.

Contudo, há situações em que o JavaScript “falha” em assumir um ponto e vírgula onde é realmente necessário. Os erros que ocorrem nesses casos são bastante difíceis de encontrar e corrigir.

Variáveis

Em JavaScript, as variáveis não são tipadas. Isso significa que não é preciso especificar um tipo, e essa mesma variável pode ter valores de texto, numérico ou booleano atribuídos a qualquer momento. Para declarar uma variável, usa-se a palavra-chave **let**.

A instrução a seguir cria (ou seja, declara) uma variável com o nome **mensagem**:

```
let mensagem;
```

Agora que você tem uma variável declarada, pode armazenar dados nela usando o operador de atribuição **=** e ler esses dados usando a referência a essa variável.

```
let mensagem;
mensagem = 'Olá, mundo!'; //atribuindo um valor em “mensagem”
alert(mensagem); //lendo um valor de “mensagem”
```

Também possível declarar várias variáveis em uma única linha:

```
let mensagem = 'Olá, mundo!', usuario = "admin", idade = 45
```

Isso pode parecer mais curto e produtivo, porém não é recomendado. Para facilitar a leitura, procure usar uma única linha por variável. Essa abordagem é um pouco mais extensa, mas torna mais fácil a leitura do código:

```
let mensagem = 'Olá, mundo!';
let usuario = "admin";
let idade = 45;
```

Também é possível declarar duas variáveis e copiar os dados de uma para a outra:

```
let ola = 'Olá, mundo!';
let mensagem = ola;
```

Vale lembrar que uma variável deve ser declarada apenas uma vez. Uma declaração repetida da mesma variável resultará em erro:

```
let mensagem = "Mensagem 1";
let mensagem = "Mensagem 2";
```

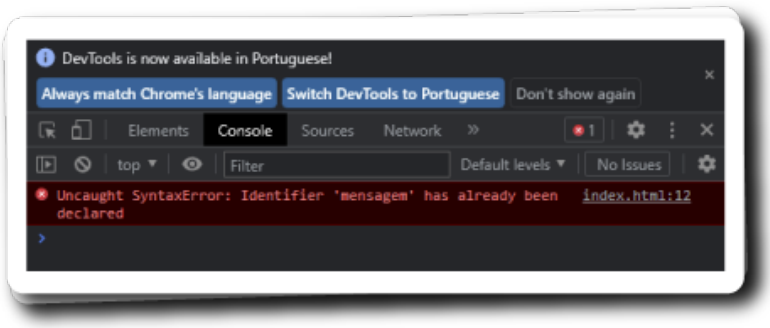

Figura 4 – Console do Google Chrome

Fonte: Senac EAD (2023)

Em *scripts* mais antigos, você também pode encontrar uma palavra-chave sendo usada para declarar variável: **var** em vez de **let**.

```
var mensagem = 'Olá, mundo!';
```

A palavra **var** faz algo parecido, mas não igual, em comparação à palavra **let**. Ela declara uma variável em escopo de função, enquanto a variável **let** atua em escopo de bloco. Veja uma comparação delas, lado a lado, começando pela declaração de **var**:

```
var x = 1; // declaramos uma var x
if (true) {
    var x = 2; // redeclaramos x
    alert(x); // vai exibir 2
}
alert(x); // vai exibir 2
// Conclusão: a variável x que declaramos no if irá subscrever a variável x que declaramos no começo pois VAR atua em contexto de função
```

Agora, veja a declaração com **let**:

```
let x = 1; // declaramos uma let x
if (true) {
    let x = 2; //redeclaramos uma let x
    alert(x); // vai exibir 2
}
alert(x); // vai exibir 1
// Conclusão: como as variáveis let atuam em contexto de bloco, a variável do bloco IF não subscreve a variável declarada no inicio
```

O comportamento de **var** é bem diferente de outras linguagens de programação e pode causar *bugs* de difícil resolução. A palavra-chave **let** permite um escopo de variável mais preciso e previsível e permite que os programadores reutilizem com segurança nomes para variáveis temporárias dentro da mesma função.

Tipos de variáveis

As variáveis em JavaScript, diferente da linguagem Java, não são tipadas, ou seja, não têm tipos. A mesma variável pode mudar (receber um novo valor) de não iniciada (**undefined**) para número (**number**), booleano (**boolean**) e texto (**string**).

```
let x;       // undefined
x = 0;       // number
x = true;    // boolean
x = "texto"; // string
```

Isso ocorre porque o JavaScript define os tipos de dados dinamicamente, enquanto outras linguagens (como Java e C#) são tipadas estaticamente, ou seja, informa-se um tipo (**int**, **string**, **boolean** etc) no momento da declaração e aquela variável só poderá carregar esse tipo de dado especificado. Já em JavaScript, as variáveis podem ser reatribuídas a valores de qualquer tipo, portanto nunca se especifica o tipo de variáveis ou o tipo de retorno das funções.

Por exemplo, em JavaScript, o seguinte código, em que se atribui um valor inteiro e depois um texto a uma mesma variável, é válido:

```
let x = 5;
x = "Olá";
```

Nomenclatura de variáveis

Existem duas limitações nos nomes de variáveis em JavaScript:

1. O nome deve conter apenas letras, dígitos ou os símbolos **$** e **_**.
2. O primeiro caractere não deve ser um dígito.

Veja alguns exemplos de nomes válidos:

```
let nomeUsuario;
let teste123;
```

Quando o nome contém várias palavras, é de boa prática utilizar **camelCase** – prática de juntar as palavras compostas e usar a caixa-alta na primeira letra de cada nova palavra, como feito em **nomeUsuario**.

Procure sempre declarar nomes, tanto de variáveis quanto de funções, de forma curta, legível e simples. Isso pode soar óbvio, mas é muito fácil se distrair e escrever nomes como **x1** e **x2** ou, em casos mais extremos, nomes como **calcularASomaDeDoisValoresInteiros**. Nada disso faz muito sentido. Bons nomes de variáveis e funções devem ser simples e fáceis de entender e dizer o que está acontecendo.

Manter o inglês também é uma boa ideia, já que linguagens de programação foram construídas nesse idioma e estão em inglês. Veja seu código como uma narrativa. Se você puder ler linha por linha e entender o que está acontecendo, perfeito!

Operadores

Você conhece muitos operadores da escola, como adição, multiplicação, subtração e assim por diante. Aqui, você começará com operadores simples e em seguida se concentrará nos aspectos específicos do JavaScript.

Operadores matemáticos

Operação	Sinal	Exemplo	Resultado
Adição	+	`let resultado = 10 + 10;` `alert(resultado);`	20
Subtração	-	`let resultado = 33 - 18;` `alert(resultado);`	15
Multiplicação	*	`let resultado = 9 * 8;` `alert(resultado);`	72
Divisão	/	`let resultado = 29 / 3;` `alert(resultado);`	9,66666666

Operação	Sinal	Exemplo	Resultado
Resto da divisão	%	`let resultado = 29 % 3;` `alert(resultado);`	2
Exponencial	**	`let resultado = 9 ** 3; //9³` `alert(resultado);`	729

Tabela 2 – Operadores matemáticos

Fonte: Senac EAD (2023)

Concatenação de texto com +

Normalmente, o operador **mais (+)** soma números. Porém, se esse operador for aplicado a **strings**, ele as mesclará (concatenará):

```
let texto = "olá, " + "mundo";
alert(texto); // olá, mundo
```

Se algum dos operandos for uma **string**, o outro também será convertido em uma **string**. Por exemplo:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Observe que não importa se o primeiro operando é uma **string** ou o segundo. Veja um exemplo mais complexo:

```
alert(2 + 2 + '1' ); // "41" e não "221"
```

Aqui, os operadores trabalham um após o outro. O primeiro **+** soma dois números, então ele retorna quatro. O próximo **+** adiciona a **string 1** e, por fim, tem-se **4 + '1' = '41'**.

Agora, veja um exemplo com os operadores inversos:

```
alert('1' + 2 + 2); // "122" e não "14"
```

Aqui, o primeiro operando é uma **string**, então o compilador tratará os outros dois operandos como **strings** também. Logo, você terá as operações **'1' + 2 = "12"** e, após isso, **"12" + 2 = "122"**.

O operador **+** é o único que lida com **strings** dessa forma. Outros operadores aritméticos trabalham apenas com números e sempre convertem seus operandos em números. Por exemplo:

```
alert( 6 - '2' ); // a string '2' é convertida para 2 e o resultado da operação é 4

alert( '6' / '2' ); // ambas string são convertidas e o resultado da operação é 3
```

Operadores lógicos

Assim como o Java, o JavaScript tem três principais operadores lógicos:

Operador	Sintaxe	Exemplo

Ou	||	`alert( true || true );  // true`
		`alert( false || true );  // true`
		`alert( true || false );  // true`
		`alert( false || false ); // false`

E	&&	`alert( true && true );  // true`
		`alert( false && true );  // false`
		`alert( true && false );  // false`
		`alert( false && false ); // false`

Operador	Sintaxe	Exemplo
Não	!	`alert( !true ); // false`

Tabela 3 – Operadores lógicos

Fonte: Senac EAD (2023)

Embora sejam chamados de “lógicos”, podem ser aplicados a valores de qualquer tipo, não apenas booleanos.

Operadores de comparação

Você conhece muitos operadores de comparação da matemática. Em JavaScript, eles são escritos assim:

Operador	Sintaxe	Exemplo

Maior/menor que	a > b, a < b	`alert( 2 > 1 );  // true`
		`alert( 2 < 1 );  // false`

Maior/menor que ou igual a	a >= b, a <= b	`alert( 2 >= 1 );  // true`
		`alert( 2 <= 1 );  // false`

Igual a	a == b	`alert( 2 == 1 ); // false`
Não é igual	a != b	`alert( 2 != 1 ); // true`

Tabela 4 – Operadores de comparação

Fonte: Senac EAD (2023)

Conversões de tipo

Na maioria das vezes, operadores e funções convertem automaticamente os valores dados a eles para o tipo certo. Por exemplo, a função **alert()** converte automaticamente qualquer valor em uma **string** para mostrá-lo. As operações matemáticas convertem valores em números. Mas há casos em que é preciso converter explicitamente um valor para o tipo esperado.

Identificando o tipo de dado

Para descobrir o tipo de dado que o JavaScript está lidando, usa-se a palavra-chave **typeof** seguindo o valor.

```
let string = "123";
alert(typeof string); // string

let numero = 123;
alert(typeof numero); // number

let booleano = true;
alert(typeof booleano); // boolean
```

Dessa forma, é possível identificar como o JavaScript está tratando determinado dado e aplicar a conversão certa para o tipo que se quer.

Conversão de *string*

A conversão de **string** acontece quando se precisa da forma de **string** de um valor. Para realizar essa conversão, usa-se a função **String();** e passa-se, dentro dos parênteses, o valor que se quer converter.

```
let booleano = true;
alert(typeof booleano); // boolean

let string = String(booleano); // conversão para string
alert(typeof string); //string
```

Conversão numérica

A conversão numérica em funções e expressões matemáticas ocorre automaticamente. Por exemplo, quando a divisão **/** é aplicada a não números:

```
alert( "6" / "2" ); // 3, textos são convertidos para números
```

Para realizar uma conversão explícita para números, usa-se a função **Number()**:

```
let string = "123";
alert(typeof string); //string

let numero = Number(string);
alert(typeof numero); //number
```

Além do método **Number()**, existem outros métodos JavaScript disponíveis para realizar conversões específicas para outros tipos numéricos – como **float** e **int**.

Para converter para **int**, usa-se **parseInt()**; para **float**, **parseFloat()**:

```
let string = "1.23";
alert(typeof string); //string

let numeroInt = parseInt(string);
alert("Valor: " + numeroInt + " - Tipo: " + typeof numero);
let numeroFloat = parseFloat(string);
alert("Valor: " + numeroFloat + " - Tipo: " + typeof numeroFloat);
```

Caso deseje limitar o número de casas decimais, use também a função **toFixed()** com a quantidade que você quer. Por exemplo:

```
let string = "1.23333333333";
alert(typeof string); //string

let numero = parseFloat(string); // valor completo
alert("Valor: " + numero + " - Tipo: " + typeof numero); // valor completo

alert("Valor: " + numero.toFixed(2) + " - Tipo: " + typeof numero); // valor limitado a 2 casas decimais
```

Em JavaScript, há apenas um tipo de número, **number**, ao contrário do Java, que tem os tipos **double**, **int** e **float**. Por isso, não é possível lidar nativamente com um número decimal de 21 dígitos, e essa é a razão de não haver um método **parseDouble()** em JavaScript.

Condicionais

Às vezes, é preciso realizar diferentes ações com base em diferentes condições. Para fazer isso, pode-se usar a instrução **if**.

No JavaScript, tanto a teoria quanto a prática são similares ao Java. A instrução **if(...)** avalia uma condição entre parênteses e, se o resultado for verdadeiro (*true*), executa um bloco de código que é delimitado por chaves, **{...}**. Por exemplo:

```
if( 1 < 2 ) {
  alert("A condição é verdadeira!");
}
```

A instrução pode conter um bloco **else** opcionalmente. Este é executado quando a condição é falsa (*false*).

```
if( 1 > 2 ) {
  alert("A condição é verdadeira!");
} else {
  alert("A condição é falsa!");
}
```

Se você quiser testar várias variantes de uma condição, também é possível. Nesse caso, usa-se o **else if** após a instrução **if**.

```
let nota = 5;
if (nota > 6) {
    alert("Nota média.");
} else if (nota == 10) {
    alert("Nota máxima!");
} else {
    alert("Nota baixa");
}
```

No código acima, o JavaScript primeiro verifica se a nota é maior que 6. Se for falso, vai para a próxima condição. Se também for falso, mostra o último alerta.

Lembre-se desta regra: você só irá para a próxima condição se a atual for falsa. Isso significa que, se caso o valor da nota for 10, não haverá o alerta “Nota máxima!” sendo apresentado, pois a primeira condição será verdadeira.

Repetição

Em desenvolvimento de sistemas, muitas vezes é preciso repetir ações, o que é feito por meio de laços de repetição. Aqui, em JavaScript, tem-se exatamente a mesma abordagem que na linguagem Java.

Loop		**Sintaxe**	**Exemplo**

While	Enquanto a condição for verdadeira, o bloco de código será executado.	`while (condição) {`
		`  // código`
		`}`
			`let i = 0;`
			`while (i < 3) {`
			`  alert( i );`
			`  i++;`
			`}`

Do... While	O *loop* primeiro executará o bloco de código e depois verificará a condição. Caso seja verdadeiro, execute-o novamente.	`do {`
		`  // código`
		`} while (condição);`
			`do {`
			`  alert( i );`
			`  i++;`
			`} while (i < 3);`

Loop		Sintaxe	Exemplo

For	Esse *loop* é mais complexo. Examine a declaração parte a parte:
	◆ **início**: É executado uma vez ao entrar no *loop*.
	◆ **condição**: É verificado antes de cada iteração do *loop*. Se for falso, o *loop* para.
	◆ **código**: É o que será executado repetidamente enquanto a condição for verdadeira.
	◆ **fim**: É executado após o fim de cada iteração.
		`for (inicio; condição; fim) {`
		`  // código`
		`}`
			`for (let i = 0; i < 3; i++) {`
			`  alert(i);`
			`}`

Tabela 5 – *Loops*

Fonte: Senac EAD (2023)

Função

As funções são os principais “blocos de construção” do programa. Eles permitem que o código seja chamado várias vezes sem repetição, o que é muito útil quando é preciso executar uma mesma ação em contextos diferentes de um *software*.

Declaração de função

Para criar uma função, use a seguinte sintaxe:

```
function nome(parametro1, parametro2) {
  // corpo
}
```

A palavra-chave **function** vai primeiro, depois o nome da função e, por fim, os parâmetros entre parênteses (separados por vírgula). Caso a função não tenha nenhum parâmetro, basta deixá-los em branco.

Funções são ações, portanto seu nome geralmente é um verbo. Procure sempre definir nomes de forma breve, sendo o mais preciso possível e descrevendo o que a função faz, para que alguém que leia o código tenha uma indicação do que a função realmente faz.

Perceba que aqui, no JavaScript, não é preciso se preocupar em informar o tipo de retorno da função, diferentemente do Java.

Para o primeiro exemplo, foi criada uma função para mostrar a mensagem “Olá, mundo” e chamá-la duas vezes:

```
function mostrarMensagem() {
  alert("Olá, mundo!");
}
mostrarMensagem();
mostrarMensagem();
```

Este exemplo demonstra claramente um dos principais propósitos das funções: evitar a duplicação de código. Se alguma vez você precisar mudar a mensagem ou a forma como ela é mostrada, basta modificar o código em um lugar: a função que a gera.

Variáveis locais

Uma variável local é uma variável declarada dentro de uma função, e esta só é visível dentro dessa função. Por exemplo:

```
function mostrarMensagem() {
  let mensagem = "Olá, mundo! Eu sou JavaScript!"; // variável local
  alert(mensagem);
}
mostrarMensagem(); // Olá, mundo! Eu sou JavaScript!
alert(mensagem); // <-- Erro! "mensagem is not defined"
```

Variáveis externas

Consideram-se variáveis externas qualquer variável que esteja declarada fora do contexto de uma função. Além das variáveis locais, uma função também pode acessar uma variável externa:

```
let linguagem = "JavaScript.";
function mostrarMensagem() {
    let mensagem = "Olá, mundo! Eu sou " + linguagem;
    alert(mensagem);
}
mostrarMensagem();
alert(mensagem);
```

A função tem acesso total à variável externa e também pode modificá-la.

```
let linguagem = "JavaScript.";
function mostrarMensagem() {
    linguagem = "JS."; // mudamos o nome da linguagem
    let mensagem = "Olá, mundo! Eu sou " + linguagem;
    alert( mensagem );
}
alert(linguagem); // Exibimos o nome JavaScript
mostrarMensagem(); // Chamamos o método que irá alterar o nome
alert(linguagem); // Exibimos o novo nome
```

A variável externa só é usada se não houver uma local. Se uma variável com o mesmo nome for declarada dentro da função, ela ignorará a externa. Por exemplo, no código a seguir, a função usa a variável local **linguagem** enquanto a externa é ignorada:

```
let linguagem = "JavaScript.";
function mostrarMensagem() {
    let linguagem = "JS.";
    let mensagem = "Olá, mundo! Eu sou " + linguagem;
    alert( mensagem );
}
alert(linguagem);
mostrarMensagem();
alert(linguagem);
```

As variáveis declaradas fora de qualquer função, como a variável externa **linguagem** no código acima, são chamadas de **variáveis globais**. Essas variáveis são visíveis para qualquer função a menos que sejam sombreadas por outra variável local.

Parâmetros

Ao falar sobre funções, os termos “parâmetros” e “argumentos” são frequentemente usados e podem passar a impressão de que são a mesma coisa, mas há uma diferença muito sutil:

Parâmetros são variáveis listadas como parte da definição da função.

Argumentos são valores passados para a função quando ela é invocada.

Assim como na linguagem Java, é possível passar dados arbitrários para funções usando parâmetros. No exemplo a seguir, foi escrita uma função que tem dois parâmetros: **numero1** e **numero2**.

```
function somar(numero1, numero2) {
  alert(numero1 + numero2);
}
somar(27, 30);
```

Quando a função é chamada na última linha, passam-se os argumentos, em que os valores fornecidos são copiados para as variáveis locais **numero1** e **numero2** para serem, em seguida, usadas pela função.

Para tornar o código limpo e fácil de entender, é recomendável usar principalmente variáveis e parâmetros locais na função, e não variáveis externas. É sempre mais fácil entender uma função que obtém parâmetros, trabalha com eles e retorna um resultado do que uma função que não obtém parâmetros, mas modifica variáveis externas como efeito colateral.

Valores padrão

Se uma função for chamada, mas um argumento não for fornecido, o valor correspondente será **undefined**. Por exemplo, a função acima poderia ser chamada com um único argumento:

```
function somar(numero1, numero2) {
  alert(numero1 + numero2);
}
somar(13);
```

Tal chamada produziria o seguinte efeito **13 + undefined**. Como o valor de **numero2** não é passado, ele se torna **undefined** e o cálculo não é realizado.

Para evitar esse tipo de situação, é possível especificar valores padrões para os parâmetros na declaração da função usando **=**:

```
function somar(numero1 = 0, numero2 = 0) {
  alert(numero1 + numero2);
}
somar(13);
```

Dessa forma, ambos os parâmetros **numero1** e **numero2** terão o valor padrão **0** caso algum desses parâmetros não for passado. Dessa forma, minimizam-se erros no sistema que possam quebrar alguma funcionalidade.

Retornando um valor

Uma função pode retornar um valor de volta ao código de chamada como resultado. Um exemplo simples seria retornar o valor da função **somar()**:

```
function somar(numero1 = 0, numero2 = 0) {
  return numero1 + numero2;
}
let resultado = somar(30, 103);
alert(resultado);
```

Vetores

Em JavaScript, é possível trabalhar com vetores usando *arrays*. Um *array* é uma estrutura de dados que permite armazenar vários valores em uma única variável.

Existem duas sintaxes para criar um vetor vazio:

```
let vetor = new Array();
```

ou

```
let vetor = [];
```

Em JavaScript, quase sempre a segunda forma é usada.

Para adicionar itens ao vetor, basta separar os valores com virgula. A seguir, veja alguns exemplos de vetores que armazenam diferentes tipos de dados.

```
let stringArray = ["um", "dois", "três"];
let numericoArray = [1, 2, 3, 4];
let decimalArray = [1.1, 1.2, 1.3];
let booleanArray = [true, false, false, true];
```

Em linguagens de programação como Java, tem-se como regra que todos os elementos do vetor devem ser do mesmo tipo. Já na linguagem JavaScript não é necessário armazenar o mesmo tipo de valores em um vetor, como é possível ver no exemplo a seguir.

```
let dados = [1, "Dois", "3", true, 4000, 5.5];
```

Para acessar o valor de um determinado item do vetor, é preciso informar o nome do vetor seguido do número da posição que se quer acessar entre colchete. Lembre-se de que cada item do vetor está associado a um índice numérico começando com 0.

```
let frutas = ["Maçã", "Laranja", "Abacaxi"];
alert( frutas[0] ); // Maçã
alert( frutas[1] ); // Laranja
alert( frutas[2] ); // Abacaxi
```

Se desejar percorrer um vetor inteiro e exibir todos os valores armazenados, use um laço de repetição **for**.

```
let frutas = ["Maçã", "Laranja", "Abacaxi"];
for (let i = 0; i < frutas.length; i++) {
    alert(frutas[i]);
}
```

Para adicionar um novo elemento, use o método **push**, que adicionará um elemento no final do vetor:

```
let frutas = ["Maçã", "Laranja", "Abacaxi"];
frutas.push("Uva");
for (let i = 0; i < frutas.length; i++) {
    alert(frutas[i]);
}
```

Além disso, o JavaScript fornece uma série de métodos que podem ser usados para manipular vetores: o **pop** (para remover o último elemento do vetor), o **shift** (para remover o primeiro elemento do vetor), o **unshift** (para adicionar um elemento no início do vetor), entre outros.

Eventos

O JavaScript depende de um *loop* de eventos que escuta mensagens (como cliques do *mouse*, pressionamentos de tecla ou mensagens criadas por outro objeto) e enfileira essas mensagens para processamento. Podem ser criadas funções que escutam determinadas mensagens e são acionadas quando esses eventos acontecem.

Essa arquitetura orientada a eventos faz todo o sentido para o propósito original (e ainda primário) do JavaScript – a interação do usuário do *site*. Ele permite que o aplicativo ouça entradas criadas pelo usuário. Ter um sistema integrado de detecção de eventos torna (relativamente) fácil criar aplicativos em tempo real em domínios como bate-papo, jogos, colaboração e envio.

Um evento é como um sinal ou mensagem de que algo aconteceu. Por exemplo, toda vez que seu *mouse* clica em um elemento de uma página, o evento **onclick** desse elemento é disparado. Em circunstâncias normais, isso não faz com que nada mais aconteça. Mas você pode usar esses eventos para acionar a execução de outro código. Você pode escrever um código que diga (em essência): toda vez que isso acontecer, faça outra coisa.

Lista de eventos JavaScript

A maioria dos nomes descreve exatamente quando eles disparam. Alguns deles são aplicáveis apenas a alguns tipos de elementos.

Veja, a seguir, os ventos do *mouse*.

onclick

O evento acontece quando um elemento da página é clicado.

oncontextmenu

O evento acontece quando o botão direito do *mouse* é clicado sobre um elemento.

onmouseover/onmouseout

O evento acontece quando o cursor do *mouse* passa por cima ou sai de um elemento.

onmousedown/onmouseup

O evento acontece quando o botão esquerdo do *mouse* é pressionado ou solto sobre um elemento.

onmousemove

O evento acontece quando o *mouse* é movido.

Veja, a seguir, os eventos do teclado.

onkeydown

O evento acontece quando uma tecla do teclado é pressionada.

onkeyup

O evento acontece quando uma tecla do teclado é liberada.

Veja, agora, eventos de elementos de formulários.

submit

O evento acontece quando o usuário envia um formulário **<form>**.

onfocus

O evento acontece quando um elemento de um formulário recebe foco, ou seja, quando está sendo usado.

onblur

O evento acontece quando um elemento de um formulário perde foco, ou seja, quando se sai dele.

Manipulação de eventos

Para reagir a eventos, atribui-se um **handler** (manipulador, em português) – uma função que é executada no caso de o evento acontecer.

A forma mais simples de fazer isso é por meio de um atributo no HTML.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
      <title>Linguagem de Scripts</title>
      <meta charset="utf-8">
  </head>
  <body>
    <form id="formulario">
      <h1 id="titulo">Login</h1>
      <p>
        <label>Usuário:</label>
        <input type="text" id="usuario">
      </p>
      <p>
        <label>Senha:</label>
        <input type="password" id="senha">
      </p>
      <input type="submit" value="Acessar" onclick="alert('Login efetuado!');">
    </form>
  </body>
</html>
```

Com um clique no *mouse* sobre o botão **Acessar**, o código interno do **onclick** é executado. Observe que, dentro de **onclick**, usam-se aspas simples, pois o próprio atributo está em aspas duplas. Se você se esquecer de que o código está dentro do atributo e colocar aspas dentro – assim: onclick="alert(**''**..");" –, então seu evento não funcionará, pois a segunda aspa dupla, destacada em negrito, será reconhecida como o fechamento da primeira.

Além disso, um atributo HTML não é um lugar conveniente para escrever muito código. Então, é melhor criar uma função JavaScript e chamá-la.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <form id="formulario">
      <h1 id="titulo">Login</h1>
      <p>
        <label>Usuário:</label>
        <input type="text" id="usuario">
      </p>
      <p>
        <label>Senha:</label>
```

```
        <input type="password" id="senha">
      </p>
      <input type="submit" value="Acessar" onclick="fazerLogin()">
    </form>
    <script type="text/javascript">
      function fazerLogin() {
          alert("Login efetuado!");
      }
    </script>
  </body>
</html>
```

DOM

Imagine o seguinte: você está com a televisão ligada. Você não gosta do programa que está sendo transmitido e deseja alterá-lo. Você também deseja aumentar o volume. Para fazer isso, deve haver uma maneira de interagir com a televisão. E o que você usa para fazer isso? Um controle remoto!

O controle remoto serve de ponte para que você interaja com a televisão. Você torna a televisão ativa e dinâmica por meio dele. E da mesma forma o JavaScript torna a página HTML ativa e dinâmica por meio do DOM (*document object model*). Assim como a televisão não pode fazer muito por si mesma, o JavaScript não faz muito mais do que permitir que você execute alguns cálculos ou trabalhe com **strings** básicas. Portanto, para tornar um documento HTML mais interativo e dinâmico, o *script* precisa ser capaz de acessar o conteúdo do documento e também precisa saber quando o usuário está interagindo com ele. Ele faz isso se comunicando com o navegador, usando as propriedades, os métodos e os eventos na interface chamada DOM.

O DOM é uma interface de programação para documentos da *web*. Ele representa a página para que os programas possam alterar a estrutura, o estilo e o conteúdo do documento.

A representação do DOM é de um tipo de estrutura de dados chamada de árvore. Nessa estrutura, ele representa a página da *web* usando uma série de objetos. O objeto principal é o **document**, que, por sua vez, abriga outros objetos que também abrigam os próprios objetos e assim por diante.

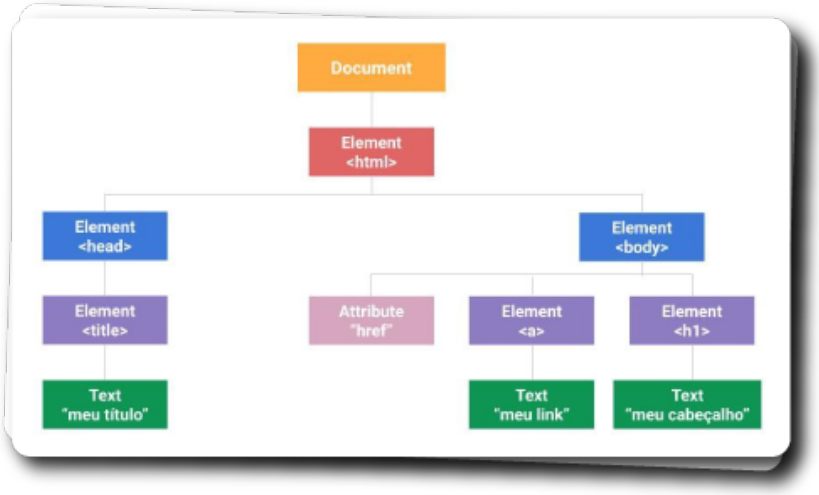

Figura 5 – Árvore de elementos do DOM

Fonte: Barro (2022)

Objeto *document*

Quando um documento HTML é carregado na janela de um navegador *web*, ele se torna um objeto **document**. Esse é o objeto mais alto na estrutura do DOM e tem propriedades e métodos que podem ser acessados e usados como desejar. Para fazer isso no código, após digitar **document**, basta colocar um ponto seguido de uma propriedade ou método que deseja utilizar.

Como exemplo, a seguir, a propriedade URL foi acessada, seu valor foi passado para uma variável e, por fim, o valor da variável foi exibido em um alerta:

```
<html lang="pt-br">
  <head>
      <title>Linguagem de Scripts</title>
      <meta charset="utf-8">
  </head>
  <body>
      <script type="text/javascript">
          let url = document.URL;
          alert(url);
      </script>
  </body>
</html>
```

Todas as operações no DOM começarão com o objeto **document**. A partir desse ponto, é possível acessar qualquer nó da estrutura, que será visto como um objeto com métodos e propriedades. Esses nós são conhecidos como **elementos**.

Pesquisando elementos

Antes de modificar qualquer elemento, é preciso saber identificar e localizar esses elementos no documento. Para isso, usam-se métodos de pesquisa do DOM.

getElementById()

Se um elemento HTML tiver o atributo **id**, é possível localizar o elemento usando o método **document.getElementById(id)**:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
      <title>Linguagem de Scripts</title>
      <meta charset="utf-8">
  </head>
  <body>
      <form id="formulario">
          <h1 id="titulo">Login</h1>
```

```
            <p>
                <label>Usuário:</label>
                <input type="text" id="usuario">
            </p>
            <p>
                <label>Senha:</label>
                <input type="password" id="senha">
            </p>
            <input type="submit" value="Acessar">
        </form>
        <script type="text/javascript">
            let elemento = document.getElementById('titulo');
            alert(elemento);
        </script>
    </body>
</html>
```

Se o elemento for encontrado, o método retornará o elemento como um objeto. Caso o elemento não seja encontrado, será retornado **null**.

Se você estiver lidando com um campo de entrada de dados (**input**), é possível acessar a propriedade **value** e descobrir qual informação consta nesse formulário. Por exemplo:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
      <title>Linguagem de Scripts</title>
      <meta charset="utf-8">
  </head>
  <body>
      <form id="formulario">
          <h1 id="titulo">Login</h1>
          <p>
              <label>Usuário:</label>
              <input type="text" id="usuario">
          </p>
          <p>
              <label>Senha:</label>
              <input type="password" id="senha" value="123">
          </p>
          <input type="submit" value="Acessar">
      </form>
      <script type="text/javascript">
          let elemento = document.getElementById('senha');
          alert(elemento.value);
      </script>
  </body>
</html>
```

Isso é muito útil em situações em que é preciso validar se alguma informação foi preenchida corretamente ou se um campo de formulário foi de fato preenchido.

getElementsBy*

Visto que o **id** de um elemento deve ser um identificador único, ou seja, não deve se repetir, usa-se **getElementById()** para localizar esse único elemento. Agora, se for preciso selecionar mais de um elemento, usam-se os métodos **getElementsBy** seguido da propriedade que desejar usar como identificação.

getElementsByTagName()

Nesse exemplo, pegue todos os elementos com a *tag* <input> do documento. Esses elementos serão do tipo **radio**. Isso significa que haverá várias opções que podem ser selecionadas (propriedade **checked**). Aproveite esse exemplo para aprender também como descobrir qual opção está selecionada ou não.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <table id="tabela">
      <tr>
        <td>O que você está aprendendo?:</td>
        <td>
          <label>
            <input type="radio" name="tecnologia" value="HTML"> HTML
          </label>
          <label>
            <input type="radio" name="tecnologia" value="CSS"> CSS
          </label>
          <label>
            <input type="radio" name="tecnologia" value="JavaScript" checked> JS
          </label>
        </td>
      </tr>
    </table>
    <script type="text/javascript">
      // Buscamos todos os elementos com a tag input
      let inputs = document.getElementsByTagName('input');
      // Percorremos item a item dessa lista e o tratamos individualmente como o objeto input
      for (let input of inputs) {
          // Exibimos o valor da input e verificamos se ela está marcada (true/false)
          alert(input.value + ': ' + input.checked);
      }
    </script>
  </body>
</html>
```

Perceba que os métodos **getElementsBy*** retornam uma coleção de elementos, e não um elemento. Portanto, não se consegue extrair diretamente o valor dos elementos como se fez com o exemplo anterior. A única forma de fazer isso é acessando cada elemento da lista e sua respectiva propriedade – para isso, usa-se o laço de repetição **for**.

getElementsByName()

Com esse método, buscam-se todos os elementos com a propriedade **name**. Aproveite esse exemplo para usar a propriedade **length**, que dirá quantos elementos foram encontrados no documento.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <table id="tabela">
      <tr>
        <td>O que você está aprendendo?:</td>
        <td>
          <label>
            <input type="radio" name="tecnologia" value="HTML"> HTML
          </label>
          <label>
            <input type="radio" name="tecnologia" value="CSS"> CSS
          </label>
          <label>
            <input type="radio" name="tecnologia" value="JavaScript" checked> JS
          </label>
        </td>
      </tr>
    </table>
    <script type="text/javascript">
      // Buscamos todos os elementos com a tag input
      let tecnologias = document.getElementsByName('tecnologia');
      // Retornamos a quantidade de elementos encontrados
      alert("Quantidade de tecnologias: " + tecnologias.length)
    </script>
  </body>
</html>
```

getElementsByClassName()

Nesse método, buscam-se todos os elementos que tenham determinado nome de classe CSS.

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <style>
      .vermelho {
          color: red;
      }
      .azul {
          color: blue;
      }
    </style>
    <table id="tabela">
      <tr>
        <td>O que você está aprendendo?:</td>
        <td>
          <label class="vermelho">
            <input type="radio" name="tecnologia" value="HTML"> HTML
          </label>
          <label class="vermelho">
            <input type="radio" name="tecnologia" value="CSS"> CSS
```

```
          </label>
          <label class="azul">
            <input type="radio" name="tecnologia" value="JavaScript" checked> JS
          </label>
        </td>
      </tr>
    </table>
    <script type="text/javascript">
      const tecnologias = document.getElementsByClassName("vermelho");
      alert("Classes encontradas: " + tecnologias.length);
    </script>
  </body>
</html>
```

Modificando elementos

A modificação do DOM é a chave para criar páginas vivas. Aqui, você verá como criar novos elementos e modificar o conteúdo da página existente.

style

A propriedade **style** permite que você modifique as propriedades CSS de um elemento. O objeto **style** contém propriedades CSS como **color**, **font-size**, **background-color** etc., e seus valores podem ser atribuídos diretamente ao elemento HTML. Por exemplo:

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="meuElemento" style="color: blue">
      <p>Conteúdo original</p>
    </div>
    <button onclick="atualizarConteudo()">Atualizar</button>

    <script type="text/javascript">
      function atualizarConteudo() {
        document.getElementById("meuElemento").style.color = "red";
        document.getElementById("meuElemento").style.fontSize = "20px";
      }
    </script>
  </body>
</html>
```

Nesse exemplo, a cor do texto do elemento **"meuElemento"** é definida como vermelho e o tamanho da fonte é definido como 20 *pixels*. Perceba que, para aplicar a propriedade **font-size**, é preciso aplicar as práticas de **camelCase** para que que o DOM possa interpretar corretamente o nome da propriedade. Essa prática deve ser aplicada para todas as propriedades com nomes compostos. Por exemplo, em vez de usar **"background-color"**, use **"backgroundColor"**.

Também é possível definir múltiplas propriedades CSS de uma só vez usando o método **cssText**. Por exemplo:

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="meuElemento" style="color: blue">
      <p>Conteúdo original</p>
    </div>
    <button onclick="atualizarConteudo()">Atualizar</button>

    <script type="text/javascript">
      function atualizarConteudo() {
          document.getElementById("meuElemento").style.cssText = "color: red; font-size: 20px";
      }
    </script>
  </body>
</html>
```

innerHTML

A propriedade **innerHTML** permite que você modifique o conteúdo HTML de um elemento. Por exemplo:

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="meuElemento">
      <p>Conteúdo original</p>
    </div>
    <button onclick="atualizarConteudo()">Atualizar</button>

    <script type="text/javascript">
      function atualizarConteudo() {
          document.getElementById("meuElemento").innerHTML = "<p>Novo conteúdo</p>";
      }
    </script>
  </body>
</html>
```

Isso substituirá o conteúdo anterior pelo novo conteúdo, alterando a exibição do elemento na página.

Caso deseje adicionar um novo conteúdo a um determinado elemento sem substituir o conteúdo anterior, concatene o conteúdo antigo com o novo da seguinte maneira:

```
<html lang="pt-br">
  <head>
```

```
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="meuElemento">
      <p>Conteúdo original</p>
    </div>
    <button onclick="atualizarConteudo()">Atualizar</button>

    <script type="text/javascript">
      function atualizarConteudo() {
        let conteudoAntigo = document.getElementById("meuElemento").innerHTML;
        let conteudoNovo = "<p>Novo conteúdo</p>";
        document.getElementById("meuElemento").innerHTML = conteudoAntigo + conteudoNovo;
      }
    </script>
  </body>
</html>
```

Lista de compras

Agora que já se sabe como manipular um elemento, você desenvolverá um exemplo mais complexo. Para isso, criará uma página na qual possa cadastrar itens a uma lista de compras.

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Compras do mercado</h1>
    <p>
      Nome do item:
      <input type="text" id="item">
    </p>

    <button onclick="adicionar()">Adicionar</button>

    <h3>Lista:</h3>
    <div id="listaDeCompras">
      <!-- Aqui será apresentado os itens cadastrados -->
    </div>

    <script type="text/javascript">
      function adicionar() {
        let conteudoAntigo = document.getElementById("listaDeCompras").innerHTML;
        let conteudoNovo = "<p>" + document.getElementById("item").value + "</p>";
        document.getElementById("listaDeCompras").innerHTML = conteudoAntigo + conteudoNovo;
      }
    </script>
  </body>
</html>
```

Perceba que, nesse exemplo, os itens foram adicionados diretamente ao elemento **"listaDeCompras"**. Caso deseje usar um vetor para armazenar esses dados e, depois, organizá-los na página, terá um código um pouco mais complexo. Implemente essa solução e adicione um trecho de código que limpará o seu formulário

após o item ser cadastrado:

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>

  <body>
    <h1>Compras do mercado</h1>
    <p>
      Nome do item:
      <input type="text" id="item">
    </p>

    <button onclick="adicionar()">Adicionar</button>

    <h3>Lista:</h3>
    <div id="listaDeCompras">
      <!-- Aqui será apresentado os itens cadastrados -->
    </div>

    <script type="text/javascript">
      let listaDeItens = []; // Vetor para armazenar os itens adicionados

      function adicionar() {
        let novoItem = document.getElementById("item").value; // Obter o valor do novo item
        listaDeItens.push(novoItem); // Adicionar o novo item ao vetor
        document.getElementById("item").value = ""; // Limpar o campo de entrada de texto

        // Exibir os itens armazenados no vetor na página
        let conteudoLista = "";
        for (let i = 0; i < listaDeItens.length; i++) {
            conteudoLista += "<p>" + listaDeItens[i] + "</p>";
        }
        document.getElementById("listaDeCompras").innerHTML = conteudoLista;
      }
    </script>
  </body>
</html>
```

Perfeito! Que tal evoluir um pouco mais e aplicar o que você aprendeu sobre manipulação de estilo com DOM também? Agora, sempre que um item da lista for clicado, você verificará se há alguma estilização aplicada. Se não houver, você deve “riscar” o item da lista. Caso o item já esteja riscado (ou seja, caso tenha estilização), remova o risco e deixe o texto legível novamente.

```
<html lang="pt-br">
  <head>
      <title>Linguagem de Scripts</title>
      <meta charset="utf-8">
  </head>

  <body>
    <h1>Compras do mercado</h1>
    <p>
      Nome do item:
      <input type="text" id="item">
    </p>
```

```
    <button onclick="adicionar()">Adicionar</button>

    <h3>Lista:</h3>
    <div id="listaDeCompras">

    </div>

    <script type="text/javascript">
      let listaDeItens = []; // Vetor para armazenar os itens adicionados

      function adicionar() {
        let novoItem = document.getElementById("item").value; // Obter o valor do novo item
        listaDeItens.push(novoItem); // Adicionar o novo item ao vetor
        document.getElementById("item").value = ""; // Limpar o campo de entrada de texto

        // Exibir os itens armazenados no vetor na página (opcional)
        let conteudoLista = "";
        for (let i = 0; i < listaDeItens.length; i++) {
          // Adicionamos o item da lista com o evento onclick preparado para acionar a função riscar
          // Aqui, usaremos a posição do vetor para diferenciar um elemento do outro
          conteudoLista += "<p id='"+ i +"' onclick='riscar("+i+")'>" + listaDeItens[i] + "</p>";
        }
        document.getElementById("listaDeCompras").innerHTML = conteudoLista;
      }

      function riscar(id) {
        // Verificamos se o elemento não possui estilização
        if(document.getElementById(id).style.cssText == "") {
          // Se não tiver, aplicamos o seguinte estilo
          document.getElementById(id).style.cssText="text-decoration: line-through;"
        } else {
          // Se tiver, removemos o estilo
          document.getElementById(id).style.cssText=""
        }
      }
    </script>
  </body>
</html>
```

Validação de formulários

Agora, você continuará aplicando seus conhecimentos em um novo exemplo, implementando a validação de um formulário.

Para isso, algumas regras devem ser estabelecidas:

1. O formulário deve conter dois campos: *e-mail* e senha.
2. Os campos devem ser validados após cada elemento ser desfocado pelo usuário.
3. Cada campo deve ter uma função de validação onde:
    a. A validação do *e-mail* deve verificar se o caractere @ foi digitado – dica: utilize o método **includes()** no valor do elemento.

    b. A validação da senha deve verificar se o tamanho do valor digitado é maior que 6.
4. Quando o usuário clicar em **Acessar**, as validações devem ser realizadas novamente para verificar os campos caso o usuário não tenha focados eles em nenhum momento.

5. Se ambos os campos estiverem válidos, uma mensagem de sucesso deve aparecer. Do contrário, uma mensagem de erro é exibida.

Tente realizar esse desafio aplicando o que aprendeu. Após isso, confira a resposta a seguir e compare com a solução que você implementou.

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>
  <body>
    <form id="formulario">
      <h1 id="titulo">Login</h1>
      <p>
        <label>E-mail:</label>
        <input type="text" id="email" onblur="validarEmail()">
      </p>
      <p>
        <label>Senha:</label>
        <input type="password" id="senha" onblur="validarSenha()">
      </p>
      <input type="submit" value="Acessar" onclick="fazerLogin()">
    </form>
    <script type="text/javascript">
      // Verificamos se o valor digitado possui o caracter @
      function validarEmail() {
        let email = document.getElementById("email").value;
        if(email.includes('@')) {
          return true;
        } else {
          alert("E-mail inválido!");
          return false;
        }
      }
      // Verificamos se a senha é maior que 6 digitos
      function validarSenha() {
        let senha = document.getElementById("senha").value;
        if(senha.length < 6) {
          alert("A senha precisa ter ao menos 6 caracteres");
          return false;
        } else {
          return true;
        }
      }

      function fazerLogin() {
        // Validamos se o email e a senha foram preenchidos corretamente
        if(validarEmail() && validarSenha()) {
          // Se a validação retornar TRUE
          alert("Login efetuado!");
        } else {
          // Se a validação retornar FALSE
          alert("Não foi possível realizar o login.");
        }
      }
    </script>
  </body>
</html>
```

Chamadas assíncronas

AJAX é uma abreviação de *asynchronous JavaScript and XML* (*extensible markup language*). É uma técnica usada para criar aplicações *web* interativas, permitindo que o código JavaScript faça requisições a servidores remotos para recuperar dados sem precisar atualizar a página.

Um exemplo de uso de AJAX seria um *site* de notícias *on-line*. Quando o usuário abrir a página, o código JavaScript fará uma requisição ao servidor remoto para recuperar os dados das últimas notícias. Em vez de atualizar a página inteira, os dados são enviados de volta ao código JavaScript, que os exibe na página do usuário. Isso permite que o usuário receba as últimas notícias sem precisar recarregar a página.

Para usar AJAX, é preciso, primeiro, criar um *script* JavaScript que contenha o código necessário para realizar a requisição. O código precisa usar a API (*application programming interface*) **XMLHttpRequest** para realizar a requisição HTTP (*hyper text transfer protocol*) para o servidor remoto.

Em seguida, adiciona-se uma chamada JavaScript para esse arquivo em sua página HTML. Essa chamada deve ocorrer após o carregamento da página HTML, pois o código JavaScript precisa ser carregado antes de poder ser executado.

Uma vez que o código JavaScript estiver carregado, você pode usá-lo para realizar a requisição AJAX. O código precisa usar a API **XMLHttpRequest** para definir o método e a URL da requisição, além de configurar um manipulador de eventos para lidar com a resposta do servidor remoto.

Depois que a requisição é realizada e a resposta é recebida, o manipulador de eventos é chamado para processar a resposta. O código JavaScript, então, pode usar essa resposta para atualizar a página do usuário e exibir os dados recuperados.

A seguir, veja algo bem simples para exemplificar como se faz essa chamada assincrônica com AJAX, colocando em prática todo esse passo a passo.

```
<html lang="pt-br">
  <head>
    <title>Linguagem de Scripts</title>
    <meta charset="utf-8">
  </head>

  <body>
    <button type="button" onclick="carregarAjax()">Carregar AJAX</button>
    <div id="resultado">
      <!-- Os dados carregados serão exibidos aqui -->
    </div>

    <script type="text/javascript">
      function carregarAjax() {
        //Criar um objeto XMLHttpRequest
        let xhr = new XMLHttpRequest();

        // Apontar em qual URL será feita a requisição
        let url = 'http://example.com/api/data'

        //Definir a função de manipulação de eventos para processar a resposta
        xhr.onreadystatechange = function () {
```

```
            //Verificar se a resposta foi recebida com sucesso
            if (xhr.readyState == 4 && xhr.status == 200) {
              //Processar a resposta
              let resposta = xhr.responseText;
              //Atualizar a página do usuário com os dados da resposta
              document.getElementById('resultado').innerHTML = resposta;
            }
          };

          //Realizar a requisição
          xhr.open('GET', url, true);
          xhr.send();
        }
      </script>
    </body>
  </html>
```

A biblioteca jQuery fornece uma interface simplificada para realizar requisições AJAX e manipular os dados retornados. Além disso, o jQuery também oferece outros recursos, como animações, seletores de elementos e manipulação de eventos. Por isso, você voltará a estudar chamadas assíncronas mais adiante.

Encerramento

Parabéns por chegar até aqui e adquirir todos esses conhecimentos sobre linguagens de *script*. Como você pode ver, a linguagem JavaScript oferece uma vasta gama de recursos e ferramentas para desenvolver e organizar a interface e os elementos visuais das suas aplicações *web*. Não é à toa que essa é uma das linguagens de programação mais populares para o desenvolvimento de aplicações *web*. Seu uso permite criar interfaces dinâmicas e interativas para os usuários, tornando a experiência mais agradável e intuitiva.

Com esses conhecimentos, você já tem um bom começo para desenvolver aplicações ainda mais ricas em conteúdo. No entanto, não pare por aqui! Existem muitos recursos ainda não explorados, que irão ajudar você a desenvolver páginas para web ainda melhores. Continue seus estudos em Javascript, coloque em prática tudo que aprendeu e continue evoluindo para criar projetos ainda mais incríveis.