

VIEW

TORNANDO A INTERFACE

COM O USUÁRIO MAIS DINÂMICA

HENRIQUE R. POYATOS NETO



06

LISTA DE FIGURAS

Figura 1 – Exemplo de JavaScript.....	7
Figura 2 – E agora?.....	11
Figura 3 – Acessando o Inspetor.....	12
Figura 4 – Acessando o Console e o depurador JavaScript	13
Figura 5 – Depurar entregando o culpado.....	14
Figura 6 – Exemplo de variáveis em JavaScript.....	17
Figura 7 – Exemplo de conversões de variáveis em JavaScript	18
Figura 8 – Exemplo de comando de entrada usando <i>prompt()</i>	20
Figura 9 – Exemplo de comando de entrada usando caixa de texto	22
Figura 10 – Exemplo utilizando operadores aritméticos	24
Figura 11 – Exemplo utilizando funções matemáticas	26
Figura 12 – Exemplo de estrutura condicional usando <i>if</i>	28
Figura 13 – Exemplo de estrutura condicional usando <i>if</i> 's aninhados/encadeados ..	30
Figura 14 – Exemplo de estrutura condicional usando <i>switch</i>	32
Figura 15 – Exemplo de estrutura de repetição usando <i>while</i>	33
Figura 16 – Exemplo de estrutura de repetição usando <i>do...while</i>	34
Figura 17 – Exemplo de estrutura de repetição usando <i>for</i>	35
Figura 18 – Exemplo de estrutura de repetição usando <i>for...of</i>	36
Figura 19 – Exemplo de declaração e chamada de função	38
Figura 20 – Exemplo de declaração e chamada de função com parâmetros.....	39
Figura 21 – Exemplo de declaração e chamada de função com parâmetros e retorno	41
Figura 22 – Exemplo de declaração e chamada de função com parâmetros, retorno e caixas de texto	42
Figura 23 – Exemplo de eventos <i>onmouseover</i> e <i>onmouseout</i>	46
Figura 24 – Exemplo de validação de campo de formulário com ganho e perda de foco	48
Figura 25 – Exemplo de requisição Ajax	50

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de JavaScript obstrutivo	7
Código-fonte 2 – Exemplo de JavaScript obstrutivo melhorado	8
Código-fonte 3 – Exemplo de JavaScript não obstrutivo	9
Código-fonte 4 – Exemplo de JavaScript não obstrutivo melhorado	9
Código-fonte 5 – Exemplo de JavaScript não obstrutivo totalmente excelente	10
Código-fonte 6 – Exemplo de código-fonte JavaScript com erro de digitação	12
Código-fonte 7 – Exemplo de variáveis em JavaScript	16
Código-fonte 8 – Exemplo de conversões de variáveis em JavaScript	18
Código-fonte 9 – Exemplo de comando de entrada usando <i>prompt()</i>	19
Código-fonte 10 – Exemplo de comando de entrada usando caixa de texto	21
Código-fonte 11 – Exemplo utilizando operadores aritméticos	24
Código-fonte 12 – Exemplo utilizando funções matemáticas	25
Código-fonte 13 – Exemplo de estrutura condicional usando <i>if</i>	27
Código-fonte 14 – Exemplo de estrutura condicional usando <i>if</i> 's aninhados/encadeados	29
Código-fonte 15 – Exemplo de estrutura condicional usando <i>if</i> e <i>else if</i>	30
Código-fonte 16 – Exemplo de estrutura condicional usando <i>switch</i>	31
Código-fonte 17 – Exemplo de estrutura de repetição usando <i>while</i>	33
Código-fonte 18 – Exemplo de estrutura de repetição usando <i>do...while</i>	34
Código-fonte 19 – Exemplo de estrutura de repetição usando <i>for</i>	35
Código-fonte 20 – Exemplo de estrutura de repetição usando <i>for...of</i>	36
Código-fonte 21 – Exemplo de declaração e chamada de função	38
Código-fonte 22 – Exemplo de declaração e chamada de função com parâmetros	39
Código-fonte 23 – Exemplo de declaração e chamada de função com parâmetros e retorno	40
Código-fonte 24 – Exemplo de declaração e chamada de função com parâmetros, retorno e caixas de texto	42
Código-fonte 25 – Exemplo de eventos <i>onmouseover</i> e <i>onmouseout</i>	45
Código-fonte 26 – Exemplo de validação de campo de formulário com ganho e perda de foco	47
Código-fonte 27 – Exemplo de arquivo-texto representando um <i>script</i> dinâmico	49
Código-fonte 28 – Exemplo de requisição Ajax	50

SUMÁRIO

1 TORNANDO A INTERFACE COM O USUÁRIO MAIS DINÂMICA.....	5
1.1 Introdução	5
1.2 Sobre a linguagem JavaScript.....	5
1.3 Considerações a respeito da versão	6
1.4 Pré-requisitos do capítulo.....	6
1.5 Como aplicar o JavaScript em um documento html	6
1.6 “Ih... deu erro! E agora?”	10
1.7 Estruturas básicas.....	14
1.7.1 Variáveis.....	14
1.7.2 Comandos de entrada e saída	18
1.7.3 Operadores aritméticos e funções matemáticas	22
1.7.4 Estruturas condicionais	26
1.7.5 Estruturas de repetição	32
1.8 Funções: como criá-las?	37
1.9 Programação orientada a eventos	43
1.10 Ajax	48
CONCLUSÃO.....	52
REFERÊNCIAS.....	53

1 TORNANDO A INTERFACE COM O USUÁRIO MAIS DINÂMICA

1.1 Introdução

Um sistema como o Health Track jamais atingiria seu potencial sem a possibilidade de reagir às interações de um usuário, como um alerta quando um usuário preenche um campo de formulário de maneira incorreta, por exemplo. A linguagem JavaScript é a grande responsável por interagir com o usuário na camada mais externa do sistema: a de Visualização (View). Neste capítulo, aprenderemos essa importante linguagem de programação, indispensável em sistemas web modernos.

1.2 Sobre a linguagem JavaScript

O JavaScript (também conhecido como JScript ou pelo seu nome oficial, ECMAScript) é uma linguagem de programação do tipo *script* que roda do lado cliente. Seu objetivo é prover o que chamamos de programação orientada a eventos, sendo eventos disparados, em sua maioria, pelo usuário. A linguagem reage à interação do usuário, seja por teclado, mouse e, mais recentemente, toque por telas sensíveis.

O navegador web precisa oferecer suporte ao JavaScript, já que a linguagem é interpretada inteiramente do lado cliente e, portanto, pelo navegador. Por essa razão, o código-fonte completo é repassado a ele, podendo ser inspecionado e até mesmo modificado. Procedimentos como validação de informações em formulário devem ser feitos duplamente: pelo lado cliente, em JavaScript (pela acessibilidade e praticidade que a linguagem proporciona), e pelo lado servidor, fora do alcance do usuário.

Graças a uma interface conhecida como DOM (*Document Object Model*), todos os elementos em documentos HTML podem ser manipulados em JavaScript como se fossem objetos. Assim, podemos alterar propriedades de objetos que representem janelas, menus, caixas de diálogo, tabelas, imagens, enfim, qualquer elemento. Com a linguagem, é possível ainda acessar o histórico do navegador ou utilizar comandos de entrada e saída por meio de janelas de alerta. A linguagem deu um grande salto com o lançamento do HTML 5, podendo acessar equipamentos de

GPS, manipular *WebSockets*, desenhar figuras utilizando um *Canvas*, dentre outras novidades.

1.3 Considerações a respeito da versão

JavaScript é uma linguagem em constante evolução e, por isso, focaremos nosso aprendizado na especificação ECMAScript na versão 5.1, publicada em junho de 2011. Apesar de a versão 6 ter sido publicada em 2015 e a versão 7 já estar em discussão, o principal motivo para utilizar a versão 5.1 é que a maioria dos navegadores modernos atende a essa especificação, possibilitando a seu código ser executado por mais navegadores. Contudo, caso queira utilizar versões mais recentes, é importante ter em mente que nem todo navegador é compatível à especificação.

1.4 Pré-requisitos do capítulo

O material a seguir assume como premissa que o leitor possui noções de lógica de programação ou desenvolvimento de algoritmos. E, além disso, conhece tudo o que foi abordado em nossos materiais sobre HTML e CSS.

1.5 Como aplicar o JavaScript em um documento html

Assim como o CSS, o JavaScript tem várias maneiras de se relacionar com os documentos HTML, desde a maneira mais prática (e preguiçosa) até a menos prática e mais organizada (e, portanto, recomendada).

A forma mais fácil é misturar HTML com JavaScript, um procedimento conhecido como JavaScript obstrutivo. Os eventos permitidos nos elementos HTML podem ser definidos usando atributos na *tag* em que se deseja aplicar.

No exemplo a seguir, aplicaremos um evento *onClick* em um botão – há várias possibilidades de evento em um botão, mas o mais comum a ser programado é no ato do clique –, o navegador exibirá uma janela de notificação com os dizeres “Alô, Mundo”:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript obstrutivo.</title>
    <meta charset="utf-8">
  </head>
  <body>
    <button onclick="window.alert('Alô, Mundo!');">
Teste</button>
  </body>
</html>
```

Código-fonte 1 – Exemplo de JavaScript obstrutivo
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E a magia acontece ao se clicar no botão:

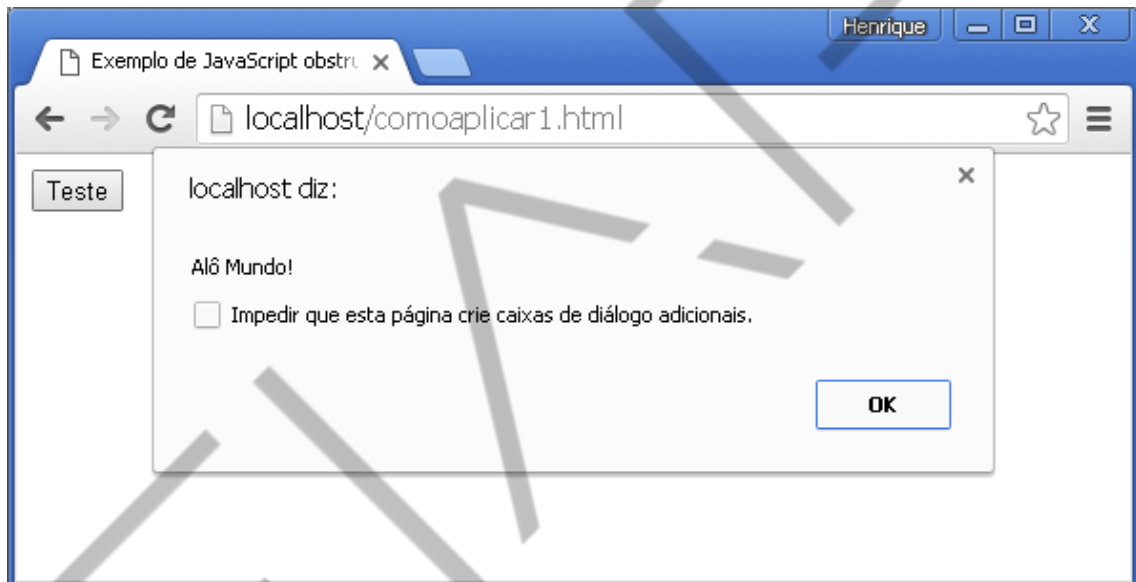


Figura 1 – Exemplo de JavaScript
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Simples, porém, não recomendado. A manutenção dos procedimentos JavaScript misturados com HTML é muito penosa. Além disso, não possibilita o reuso de procedimentos por meio de funções.

Uma evolução seria o atributo *onClick* chamar uma função que, por sua vez, chama a janela de notificações: o código fica dentro da *tag* `<script>`:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript obstrutivo.</title>
    <meta charset="utf-8">
  </head>
  <body>
    <button onclick="exibirFrase();">Teste</button>
```

```
</body>
</html>
<script>
  function exibirFrase()
  {
    window.alert("Alô, Mundo!");
  }
</script>
```

Código-fonte 2 – Exemplo de JavaScript obstrutivo melhorado
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Obs.: O atributo `type` na `tag` `<script>` se tornou obsoleto no HTML 5.

Embora o resultado prático seja rigorosamente o mesmo, essa abordagem permite o reúso de código: afinal, posso aplicar um evento em outro elemento que exiba a mesma frase. Além disso, dificilmente um evento tem apenas uma instrução (em nosso caso, uma função `window.alert()`), é mais comum apresentar várias instruções, condicionais, laços e nada disso caberia de forma adequada e organizada.

Vamos, mais uma vez, melhorar esse código-fonte (aliás, isso que estamos fazendo é chamado *refactoring*!). Dessa vez, trabalhando em uma abordagem conhecida como JavaScript não obstrutivo: a separação total entre procedimentos HTML e JavaScript.

Para tanto, devemos acessar o DOM e identificar o objeto que representa nosso botão. Ele deverá receber o identificador (o atributo `id` no HTML) e seu objeto correspondente será localizado utilizando uma função chamada `document.getElementById()`.

Essa função recebe como parâmetro o identificador do objeto e, se ela tiver sucesso, retorna o objeto correspondente. Para fins práticos, armazenaremos o objeto em uma variável (vamos falar mais sobre elas adiante!). Depois disso, basta chamar nosso evento `onClick`, dessa vez no JavaScript, de uma maneira bem peculiar. Veja:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript não-obstrutivo.</title>
    <meta charset="utf-8">
  </head>
  <body>
    <button id="botao">Teste</button>
  </body>
</html>
<script>
```



```
var objBotao = document.getElementById("botao");
objBotao.onclick = function()
{
    window.alert("Alô, Mundo!");
}
</script>
```

Código-fonte 3 – Exemplo de JavaScript não obstrutivo
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Quase perfeito! Mas por que quase? Porque nos resta separar os dois códigos em dois arquivos diferentes, como fizemos com o CSS. Basta mover os procedimentos JavaScript para um arquivo de extensão **.js**, e chamá-lo usando a mesma *tag* **<script>**, que ganha um atributo *src* (*source*):

COMOAPLICAR4 . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript não-obstrutivo.</title>
    <meta charset="utf-8">
  </head>
  <body>
    <button id="botao">Teste</button>
    <script src="comoaplicar4.js"></script>
  </body>
</html>
```

COMOAPLICAR4 . JS

```
var objBotao = document.getElementById("botao");

objBotao.onclick = function()
{
    window.alert("Alô, Mundo!");
}
```

Código-fonte 4 – Exemplo de JavaScript não obstrutivo melhorado
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Excelente! Esta é a boa prática que promove uma organização e manutenção de código melhores. Além disso, é assim que os motores de busca preferem, então pense em **SEO** (*Search Engine Optimization*).

Uma última mudança para ficar TOTALMENTE EXCELENTE: o tempo de processamento entre HTML e JavaScript é diferente. Se colocarmos a *tag* **<script>** dentro da *tag* **<head>** (que é convenção de mercado), os dois processam simultaneamente e pode acontecer de o `document.getElementById("botão")` tentar localizar o botão antes que este se torne disponível pelo HTML.

Para evitar que isso aconteça, vamos colocar todo o código que depende do DOM dentro de um evento chamado `window.onload`. Esse evento é disparado quando todo o documento HTML está renderizado e disponível, e o DOM pronto para ser acessado:

COMOAPLICAR5.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript não-obstrutivo.</title>
    <meta charset="utf-8">
    <script src="comoaplicar5.js"></script>
  </head>
  <body>
    <button id="botao">Teste</button>
  </body>
</html>
```

COMOAPLICAR5.JS

```
window.onload = function()
{
  var objBotao = document.getElementById("botao");

  objBotao.onclick = function()
  {
    window.alert("Alô, Mundo!");
  }
}
```

Código-fonte 5 – Exemplo de JavaScript não obstrutivo totalmente excelente
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Agora, sim, perfeito!

1.6 “Ih... deu erro! E agora?”

Primeira coisa: NÃO ENTRE EM PÂNICO! Feche os olhos, respire fundo, conte até dez e concentre-se em sua respiração. Pronto. Tenho certeza de que está mais calmo agora.

Isso é absolutamente normal quando estamos programando, especialmente em uma linguagem nova. Diferentemente do HTML e do CSS (nos quais o navegador tenta, a todo custo, renderizar alguma coisa), o JavaScript falha e dá erro.



Figura 2 – E agora?
Fonte: Shutterstock (2020)

Quando conhecemos bem a lógica de programação e passamos a conhecer também a linguagem em que estamos trabalhando, esses erros são frutos de desatenção, causada talvez por noites mal dormidas ou meras distrações. Pois bem, vamos aprender, então, a resolvê-los.

O código-fonte abaixo tem um erro proposital: trocamos o “m” por “n” em `document.getElementById`, tornou-se `document.getElenentById`, simulando um erro de digitação. Segue:

ERRO . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de JavaScript (ERRO).</title>
    <meta charset="utf-8">
    <script src="erro.js"></script>
  </head>
  <body>
    <button id="botao">Teste</button>
  </body>
</html>
```

ERRO . JS

```
window.onload = function()
{
```

```
var objBotao = document.getElenentById("botao");  
  
objBotao.onclick = function()  
{  
    window.alert("Alô, Mundo!");  
}
```

Código-fonte 6 – Exemplo de código-fonte JavaScript com erro de digitação
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Ao chamar a página em nosso navegador e pressionar o botão de “Teste”, não é exatamente uma surpresa descobrir que nada funciona. Para acessar o depurador JavaScript em um navegador como o Google Chrome, devemos clicar com o botão direito do mouse e escolher a opção “Inspecionar” (ou fazer isso por meio dos atalhos F12 ou Ctrl+Shift+I):

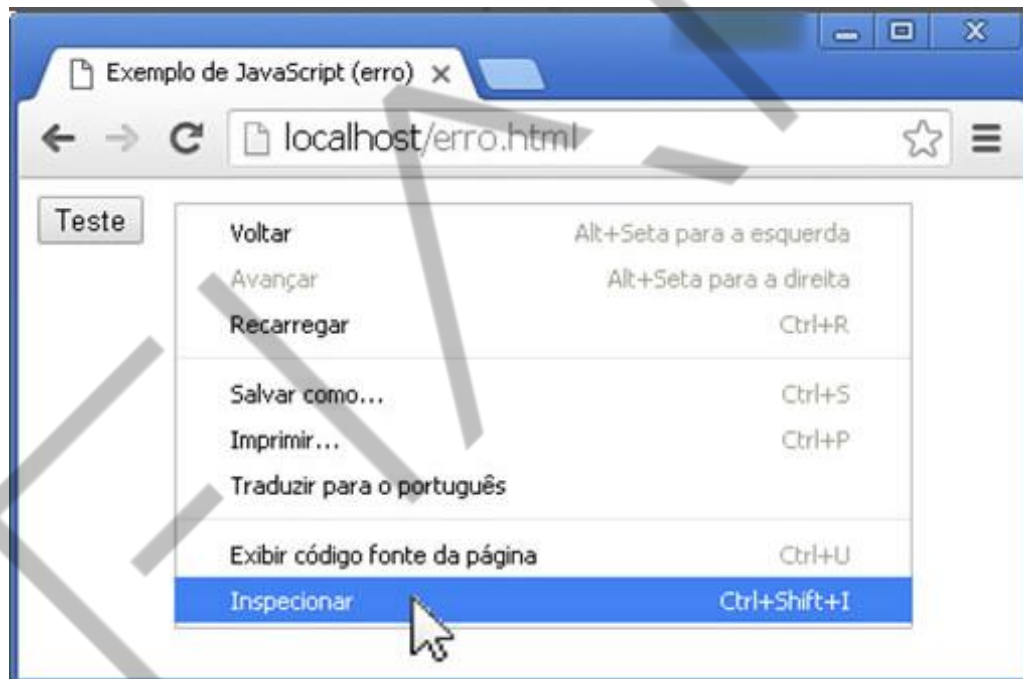


Figura 3 – Acessando o Inspetor
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

O Inspetor que surge é útil de várias maneiras, mas a que nos auxiliará neste momento é a opção “Console”:

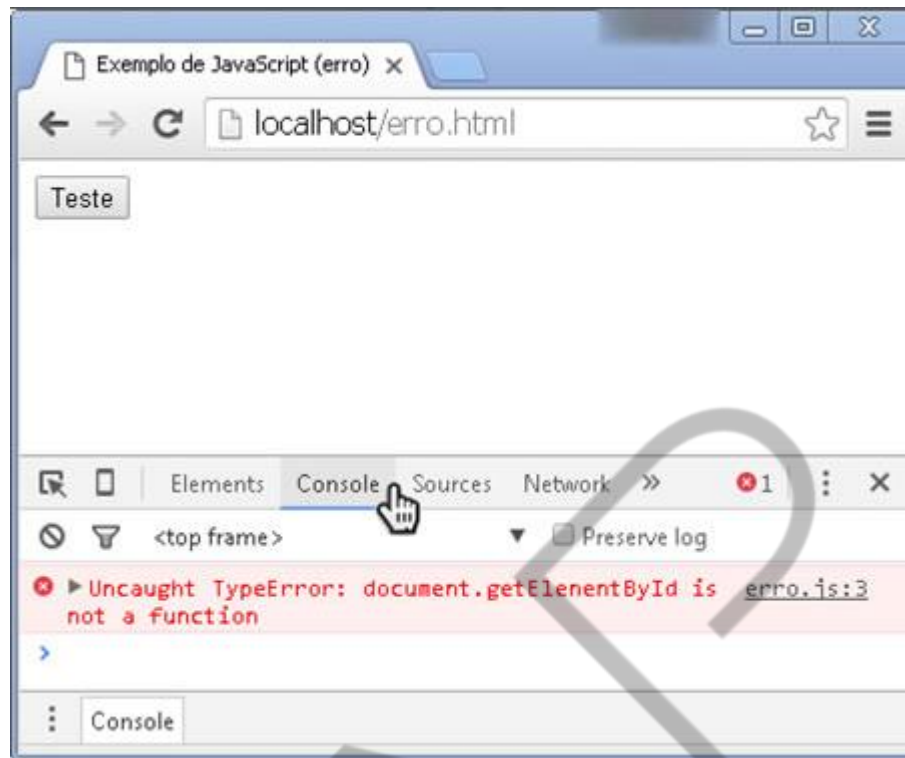


Figura 4 – Acessando o Console e o depurador JavaScript
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Eis que aparece o depurador JavaScript entregando o culpado: na minha linha 3 do arquivo erro.js, ele me diz que `document.getElenentById` não é uma função, e ele tem razão, não é mesmo? Escrevemos seu nome errado...

Corrija o “n” para “m” e tudo voltará a funcionar como antes. Ao acessar o “Console”, temos a janela limpa, livre de erros.

Fácil, não é mesmo? Em caso de erros, basta investigá-los e resolvê-los!



Figura 5 – Depurar entregando o culpado
Fonte: FIAP (2017)

1.7 Estruturas básicas

1.7.1 Variáveis

Assim como várias linguagens de *script* para internet, o JavaScript é uma linguagem “não tipada”, ou seja, ao se declarar uma variável, não é necessário (e nem é possível) identificar o tipo de informação que você guardará. Na verdade, nem sequer declarar a variável é necessário. As variáveis são do tipo *variant* e, por isso, assumem automaticamente seu tipo sempre que um valor é atribuído a elas.

“Ah, então, existem tipos”. Sim, eles existem. A diferença é que os tipos são assumidos no ato da atribuição de valor, e essa abordagem tem suas vantagens e desvantagens. A vantagem é a praticidade da abordagem, uma vez que o tipo é configurado automaticamente. Contudo, o risco de se perder a natureza da informação guardada (Será um alfanumérico? Será um inteiro?) ou de se declarar por engano uma variável nova quando, na verdade, você gostaria apenas de alterar o valor de uma já existente são algumas das desvantagens.

Os tipos mais comuns que um *variant* pode assumir são:

- **undefined**: indefinido é como o JavaScript expõe uma variável que foi declarada, mas não recebeu qualquer valor e, portanto, não assumiu nenhum tipo.
- **null**: isso acontece quando o valor null é literalmente atribuído à variável e ela está sem valor definido. Repare que é diferente de undefined; null é utilizado sempre que desejamos informar que a variável existe, mas ainda não possui um valor definido.
- **boolean**: tipo lógico, permite armazenar valores como true (verdadeiro) e false (falso).
- **string**: cadeia de alfanuméricos; cada símbolo é armazenado em 16 bits no padrão UTF-16.
- **number**: tipo numérico, sejam valores inteiros ou fracionados. São utilizados 64 bits por número, armazenados no padrão de dupla precisão estabelecido pelo IEEE.
- **object**: tipo objeto, usado para armazenar objetos instanciados a partir de uma classe ou retornados por funções específicas.
- **array**: utilizado quando for necessário armazenar mais de um valor simultaneamente, de qualquer tipo mencionado acima – possibilitando um array de números, de caracteres, de booleanos ou mesmo de objetos.

Uma última coisa antes de prosseguirmos com um exemplo: “Se, por acaso, perdermos a rastreabilidade da variável e quisermos saber qual tipo ela assume naquele momento?”. Utilize o comando **typeof**, seguido do nome da variável.

Vejamos um exemplo disso tudo:

VARIAVEIS.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de variáveis</title>
    <meta charset="utf-8">
    <script src="variaveis.js"></script>
  </head>
  <body></body>
</html>
```

VARIAVEIS.JS

```
/*Exemplo de variável indefinida*/
var vIndefinida;
console.log(typeof vIndefinida);

/*Exemplo de booleano*/
var vBooleano = true;
console.log(typeof vBooleano);

/*Exemplo de alfanumérico (string)*/
var vAlfaNumerica = "Exemplo curso de JavaScript";
console.log(typeof vAlfaNumerica);

/*Exemplos numéricos*/
var vNumeroInteiro = -159;
console.log(typeof vNumeroInteiro);
var vNumeroFracionado = 2435.45;
console.log(typeof vNumeroFracionado);

/*Exemplo de array*/
var vArrayNums = [10, 20, 30];
console.log(typeof vArrayNums);

/*Exemplo de objeto*/
var vAluno = {name:'Fulano de Tal', age:18};
console.log(typeof vAluno);
```

Código-fonte 7 – Exemplo de variáveis em JavaScript

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

O comando **console.log()** é uma alternativa de comando de saída, em particular quando estamos testando uma aplicação, como é o caso. Ele não dá a saída no documento HTML, em vez disso, registra na janela de *log* presente nos principais navegadores de mercado.

Veja o resultado:

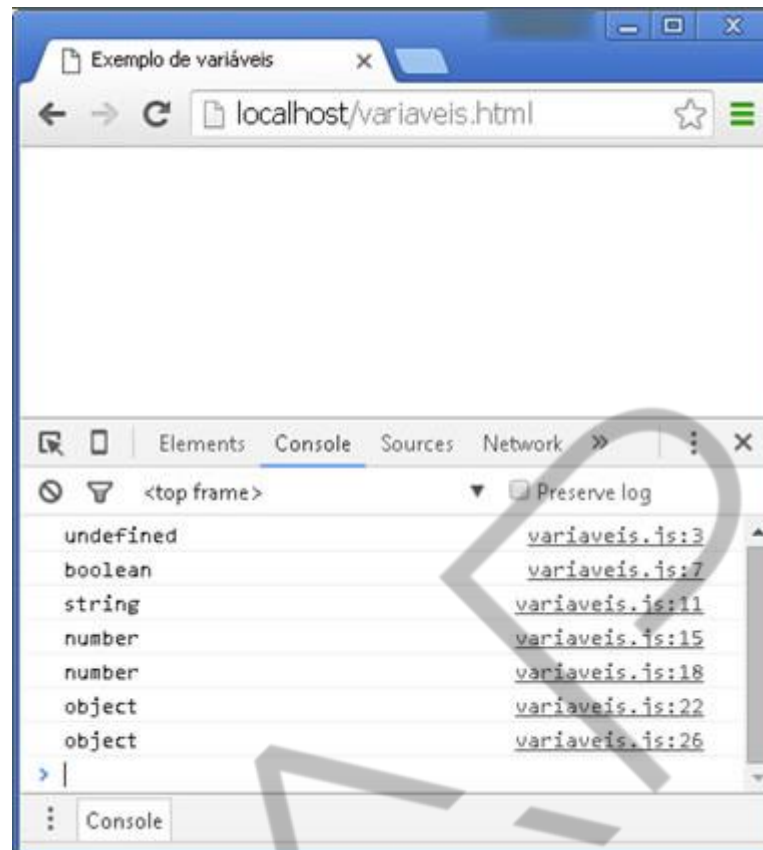


Figura 6 – Exemplo de variáveis em JavaScript

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Repare que o *array* é considerado um objeto, pois a sua estrutura é um objeto.

Em determinadas situações, um tipo de variável pode ser convertido para outro. Isso pode acontecer de forma explícita, por meio de funções de conversão, e implícita, quando a própria linguagem identifica a necessidade de conversão e o faz automaticamente.

Veja um exemplo das duas situações:

VARIAVEIS2.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de variáveis</title>
    <meta charset="utf-8">
    <script src="variaveis2.js"></script>
  </head>
  <body></body>
</html>
```

VARIAVEIS2.JS

```
/*Exemplo de conversão explícita*/
var vTeste = "10";
console.log(typeof vTeste); //retorna string
```

```
/*parseInt() converte um string, retornando um number*/  
vTeste = parseInt(vTeste);  
console.log(typeof vTeste); // retorna number  
/*Exemplo de conversão implícita*/  
var vTeste2 = "10";  
console.log(typeof vTeste2); //retorna string  
/*conversão implícita*/  
vTeste2 = vTeste2 * 2;  
console.log(typeof vTeste2); // retorna number
```

Código-fonte 8 – Exemplo de conversões de variáveis em JavaScript
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Conseguindo o seguinte resultado:

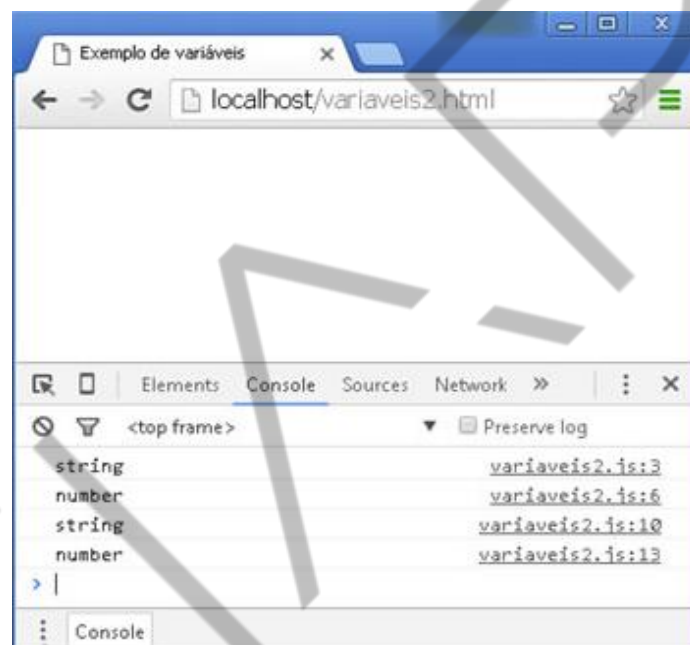


Figura 7 – Exemplo de conversões de variáveis em JavaScript
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Como em toda linguagem, existem algumas regras para nomear variáveis. É ideal que sejam nomes curtos e que possuam descrições claras de seu conteúdo, como idade, total, valor do produto, entre outros. Além disso, os nomes podem possuir letras, dígitos, *underline* (_) e cifrão e devem começar por uma letra, cifrão ou *underline*; maiúsculas e minúsculas são diferenciadas e palavras reservadas não podem ser usadas (não se pode batizar uma variável de “var” ou “if”).

1.7.2 Comandos de entrada e saída

Vamos agora aprender algumas maneiras de receber informações digitadas pelo usuário e como exibi-las na tela.

O primeiro comando de entrada é o **prompt()**, que possibilita a renderização de uma janela de alerta com uma frase e uma caixa de texto solicitando digitação do usuário. Ele funciona com dois parâmetros: o primeiro é o texto da janela e o segundo (opcional) é a informação que queremos preenchida previamente na caixa de texto.

Vamos usar **window.alert()** para mostrar a saída da informação:

ENTRADAESAIDA1.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de comandos de entrada e saída</title>
    <meta charset="utf-8">
    <script src="entradaesaida1.js"></script>
  </head>
  <body>
  </body>
</html>
```

ENTRADAESAIDA1.JS

```
/*Comando de entrada*/
var vNome = prompt("Digite seu nome", "Fulano de Tal");
/*Comando de saída*/
window.alert("Seu nome é " + vNome);
```

Código-fonte 9 – Exemplo de comando de entrada usando *prompt()*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

A última linha possui uma operação de concatenação: a frase “Seu nome é” é “grudada” ao valor presente na variável *Nome* que, ao receber o conteúdo de **prompt()**, se torna uma variável do tipo *string* (se fosse de qualquer outro tipo, aconteceria a conversão implícita). Vamos ao teste:

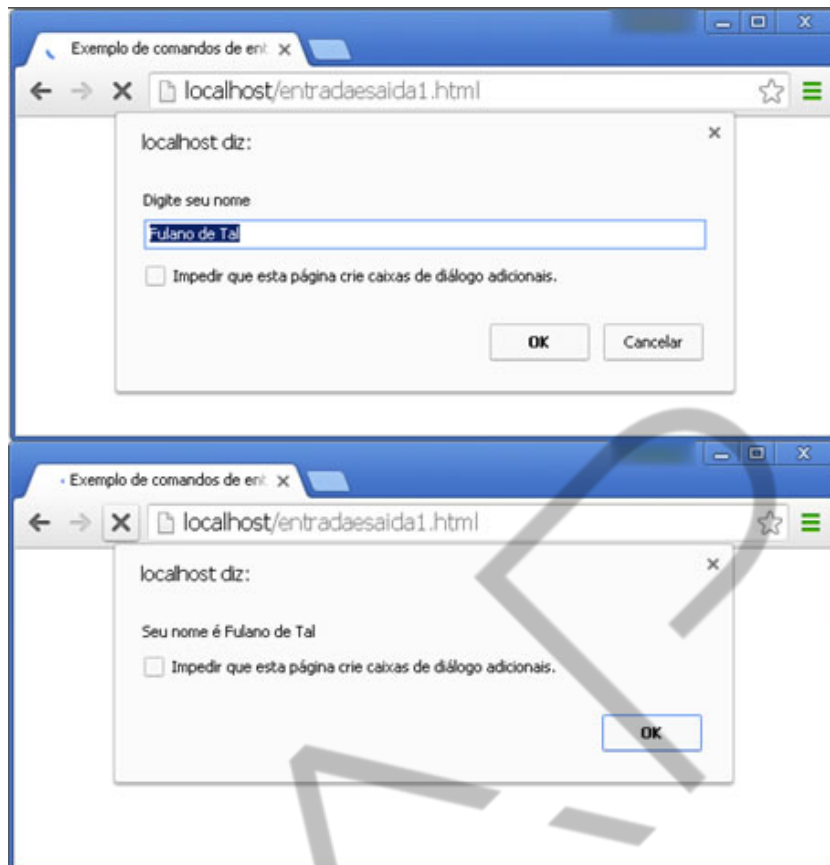


Figura 8 – Exemplo de comando de entrada usando *prompt()*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Entretanto, **prompt()** e **window.alert()** não são as melhores abordagens. Por seu comportamento de interrupção (impedindo o usuário de fazer qualquer coisa), elas são extremamente incômodas, a ponto de as versões mais recentes dos navegadores possibilitarem ao usuário impedir suas aberturas, como podemos ver na imagem. Essas caixas impedem todo o fluxo de processamento do JavaScript e a navegação do usuário.

Vamos aprender a fazer a mesma coisa, mas, dessa vez, utilizando uma caixa de texto fornecida pela tag `<input>`, típica de formulários, e vamos dar a saída em uma `<div>`. Para tanto, é necessário utilizar o DOM, acessando os objetos que representam esses elementos:

ENTRADAESAIIDA2 . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de comandos de entrada e saída</title>
    <meta charset="utf-8">
```

```
<script src="entradaesaida2.js"></script>
</head>
<body>
  <form>
    <label for="nome">Digite seu nome: </label>
    <input type="text" id="nome">
    <input type="button" id="botao" value="Enviar">
    <br><br>
    <div id="resultado"></div>
  </form>
</body>
</html>
```

ENTRADAESAIIDA2.JS

```
window.onload = function()
{
  var objCxNome = document.getElementById("nome");
  var objBotao = document.getElementById("botao");
  var objDiv = document.getElementById("resultado");
  objBotao.onclick = function()
  {
    objDiv.innerHTML = "Seu nome é " + objCxNome.value;
  }
}
```

Código-fonte 10 – Exemplo de comando de entrada usando caixa de texto

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

A <div> e vários outros elementos, como <td>, e outros, possuem um atributo conhecido como `innerHTML`, que permite às informações serem inseridas em formato HTML diretamente no elemento e renderizadas *on-the-fly*. Embora nosso exemplo não as utilize, quaisquer *tags* HTML seriam válidas aqui.

A caixa de texto, por sua vez, possui o atributo `value`, de onde é possível recuperar valores digitados na caixa ou mesmo alterá-los. Em nosso exemplo, a informação presente na caixa foi transferida para o <div> e será precedida por “Seu nome é”. Veja o código em funcionamento:



Figura 9 – Exemplo de comando de entrada usando caixa de texto
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Existem, é claro, outras possibilidades, como `document.write()`, que escreve o código HTML que foi passado como parâmetro no fim do documento, incrementando-o.

1.7.3 Operadores aritméticos e funções matemáticas

Em toda linguagem de programação, precisamos ter acesso aos operadores aritméticos, que nos possibilitam realizar os cálculos mais comuns, bem como funções matemáticas para cálculos mais complexos.

Veja, a seguir, os operadores matemáticos:

- **Símbolo “+”:** possibilita operações de adição/soma.
- **Símbolo “-”:** possibilita operações de subtração.
- **Símbolo “*” (asterisco):** possibilita operações de multiplicação.
- **Símbolo “/”:** possibilita operações de divisão.
- **Símbolo “%”:** possibilita realizar divisões, recuperando o resto da divisão.

Além dos operadores aritméticos tradicionais, temos alguns atalhos: o nome da variável seguido de dois sinais de mais (variável++) significa um pós-incremento

em um, ou seja, se `variável = 5`, e na sequência, temos um comando como `variável++`, o valor contido na variável se torna 6. Também é possível fazer um pós-decremento, ou seja, a `variável--` fará o inverso, subtraindo um do valor da variável e armazenando o novo valor.

As quatro operações básicas possibilitam outros atalhos para acúmulo de valores: o comando `variável = variável + 5` pode ser escrito como `variável += 5`; trata-se apenas de uma forma reduzida de escrita, sendo assim, as duas linhas terão um mesmo resultado. Isso pode ser feito igualmente com subtração, multiplicação e divisão.

Vejamos um exemplo abordando tudo isso:

MATEMATICA1.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de operadores matemáticos</title>
    <meta charset="utf-8">
    <script src="matematica1.js"></script>
  </head>
  <body>
  </body>
</html>
```

MATEMATICA1.JS

```
console.log("Adição 4+4:");
console.log(4 + 4);

console.log("Subtração 4-2:");
console.log(4 - 2);

console.log("Multiplicação 4x2:");
console.log(4 * 2);

console.log("Divisão 4/2:");
console.log(4 / 2);

console.log("Resto da divisão de 10 por 3:");
console.log(10 % 3);

var num = 5;
console.log("Valor de núm: " + num);

num++;
console.log("Valor de núm: " + num);
```

```
num--;  
console.log("Valor de núm: " + num);  
  
num *= 2;  
console.log("Valor de núm: " + num);
```

Código-fonte 11 – Exemplo utilizando operadores aritméticos

Fonte: Elaborado pelo autor (2016)

E ao chamar o HTML em um navegador:

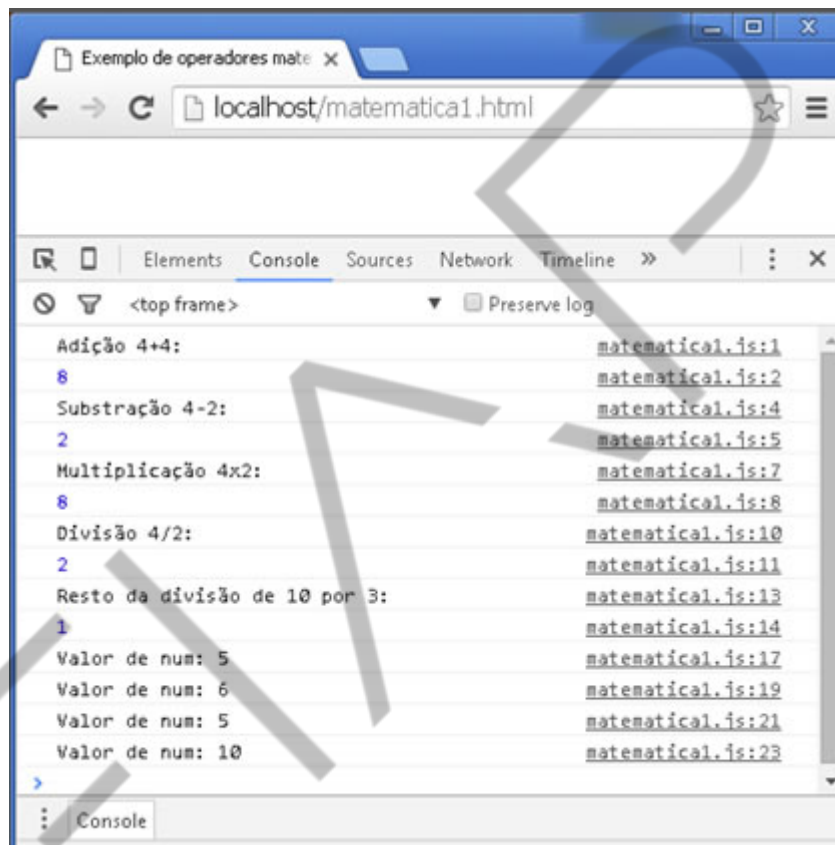


Figura 10 – Exemplo utilizando operadores aritméticos

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Conforme mencionado, o JavaScript conta com algumas funções matemáticas, para ser mais exato, alguns atributos e métodos estáticos da classe Math. Abaixo, segue uma relação com alguns deles:

- Math.PI: retorna o valor de PI (aproximadamente 3,14).
- Math.E: retorna o número de Euler (aproximadamente 2,718).
- Math.LN2: retorna o logaritmo natural de 2 (aproximadamente 0,693).
- Math.LN10: retorna o logaritmo natural de 10 (aproximadamente 2,302).
- Math.ceil(x): arredonda o valor de x para cima.

- `Math.cos(x)`: calcula o cosseno de x (x deve estar em radianos).
- `Math.floor(x)`: arredonda o valor de x para baixo.
- `Math.log(x)`: calcula o logaritmo de x (a base é o número de Euler).
- `Math.pow(x,y)`: retorna o valor de x elevado a y.
- `Math.random()`: retorna um número aleatório entre 0 e 1.
- `Math.round(x)`: arredonda o valor de x para cima ou para baixo, utilizando o que estiver mais próximo.
- `Math.sin(x)`: calcula o seno de x (x deve estar em radianos).
- `Math.sqrt(x)`: retorna a raiz quadrada de x.
- `Math.tan(x)`: calcula a tangente de x (x deve estar em radianos).

Veja a seguir um exemplo de como utilizar esses atributos e métodos estáticos:

```
MATEMATICA2.HTML  
<!DOCTYPE html>  
<html lang="pt-br">  
  <head>  
    <title>Exemplo utilizando funções  
matemáticas</title>  
    <meta charset="utf-8">  
    <script src="matematica2.js"></script>  
  </head>  
  <body>  
  </body>  
</html>
```

```
MATEMATICA2.JS  
var num = 5;  
console.log("Valor de num: " + num);  
  
num = Math.pow(5,2);  
console.log("Valor de num: " + num);  
  
num = num - Math.random();  
console.log("Valor de num: " + num);  
  
num = num * Math.PI;  
console.log("Valor de num: " + num);
```

Código-fonte 12 – Exemplo utilizando funções matemáticas
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Acompanhe a execução:

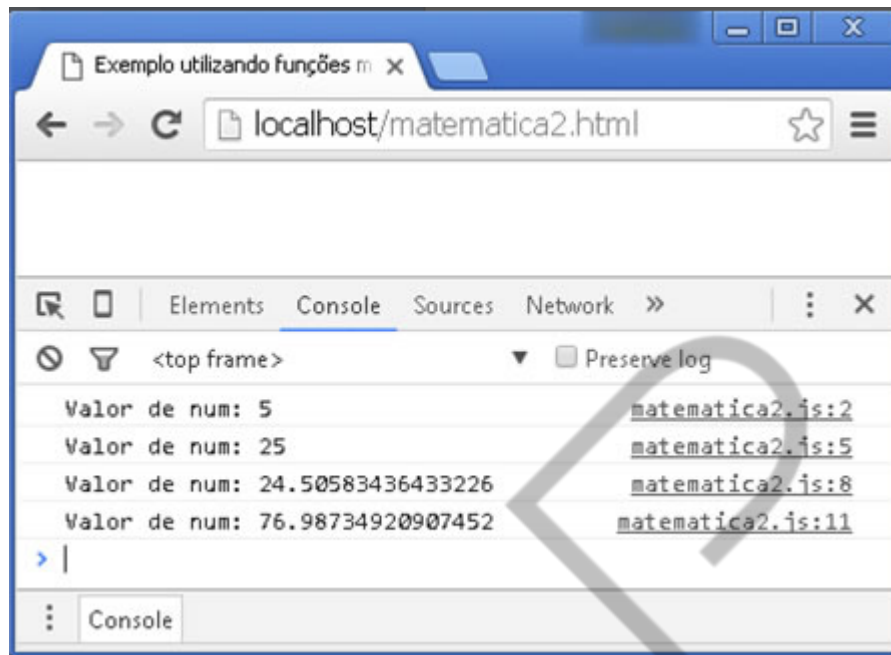


Figura 11 – Exemplo utilizando funções matemáticas
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

1.7.4 Estruturas condicionais

Toda linguagem de programação precisa de suas estruturas condicionais. Elas controlam o fluxo do código, ou seja, dizem se uma ou mais linhas de códigos devem ser executadas. A mais comum delas, o comando **if**, está presente em JavaScript.

A melhor forma de apresentar a estrutura é demonstrá-la. No exemplo a seguir, criamos uma condicional clássica do ensino acadêmico: “O aluno está aprovado ou reprovado?”. A condicional foi construída como se deve: um valor comparado a outro, usando um operador de comparação. As possibilidades são: igualdade (símbolo “==”), diferente (símbolo “!=”), maior (símbolo “>”), maior igual (símbolo “>=”), menor (símbolo “<”) ou menor igual (símbolo “<=”).

CONDICIONAL1.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de estrutura condicional</title>
    <meta charset="utf-8">
    <script src="condicional1.js"></script>
  </head>
  <body>
```

```
<label for="nota">Nota: </label>
<input type="number" id="nota" style="width: 50px">
<input type="button" id="botao" value="Calcular">
<div id="resultado"></div>
</body>
</html>
```

CONDICIONAL1.JS

```
window.onload = function() {
    var objTxtNota = document.getElementById("nota");
    var objDivResultado =
document.getElementById("resultado");
    var objBotao = document.getElementById("botao");

    objBotao.onclick = function() {
        var objFloNota = parseFloat(objTxtNota.value);
        if (objFloNota >= 6.0) {

            objDivResultado.innerHTML = "Aluno aprovado.";
            console.log("Aluno aprovado com nota " +
objFloNota);

        } else {

            objDivResultado.innerHTML = "Aluno reprovado.";
            console.log("Aluno reprovado com nota " +
objFloNota);

        }
    }
}
```

Código-fonte 13 – Exemplo de estrutura condicional usando *if*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

O exemplo utiliza uma condicional composta, isto é, possui um bloco *else* que trata a condição sendo falsa; o primeiro bloco é o tratamento tradicional, caso a condição seja satisfeita.

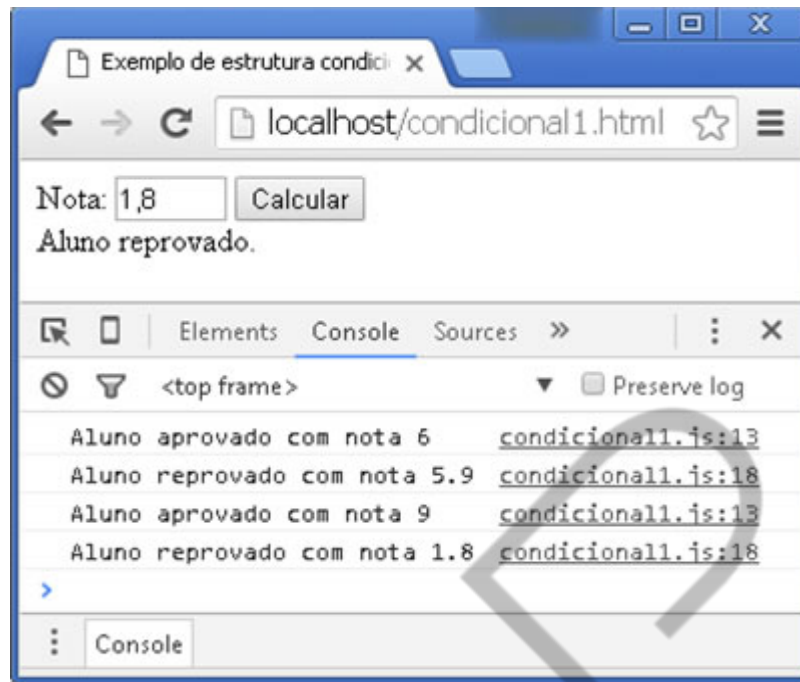


Figura 12 – Exemplo de estrutura condicional usando *if*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Vamos evoluir o exemplo utilizando o que chamamos de *if* aninhados ou encadeados: e se houvesse uma situação em que o aluno ficasse de exame, tirando uma nota 4 até 5,9999? Teríamos, então, três resultados... e duas perguntas a serem feitas:

CONDICIONAL2 . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de estrutura condicional</title>
    <meta charset="utf-8">
    <script src="condicional2.js"></script>
  </head>
  <body>
    <label for="nota">Nota: </label>
    <input type="number" id="nota" style="width: 50px">
    <input type="button" id="botao" value="Calcular">
    <div id="resultado"></div>
  </body>
</html>
```

CONDICIONAL2 . JS

```
window.onload = function() {
  var objTxtNota = document.getElementById("nota");
```

```
var objDivResultado =  
document.getElementById("resultado");  
var objBotao = document.getElementById("botao");  
objBotao.onclick = function() {  
    var objFloNota = parseFloat(objTxtNota.value);  
    if (objFloNota >= 6.0) {  
  
        objDivResultado.innerHTML = "Aluno aprovado.";  
        console.log("Aluno aprovado com nota " +  
objFloNota);  
  
    } else {  
  
        if (objFloNota >= 4.0) {  
            objDivResultado.innerHTML = "Aluno em exame.";  
            console.log("Aluno em exame com nota " +  
objFloNota);  
  
        } else {  
  
            objDivResultado.innerHTML = "Aluno reprovado.";  
            console.log("Aluno reprovado com nota " +  
objFloNota);  
  
        } // fim do if (objFloNota >= 4.0)  
  
    } // fim do if (objFloNota >= 6.0)  
    } // fim do onclick  
} // fim do window.onload
```

Código-fonte 14 – Exemplo de estrutura condicional usando *if*'s aninhados/encadeados
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E temos como resultado:

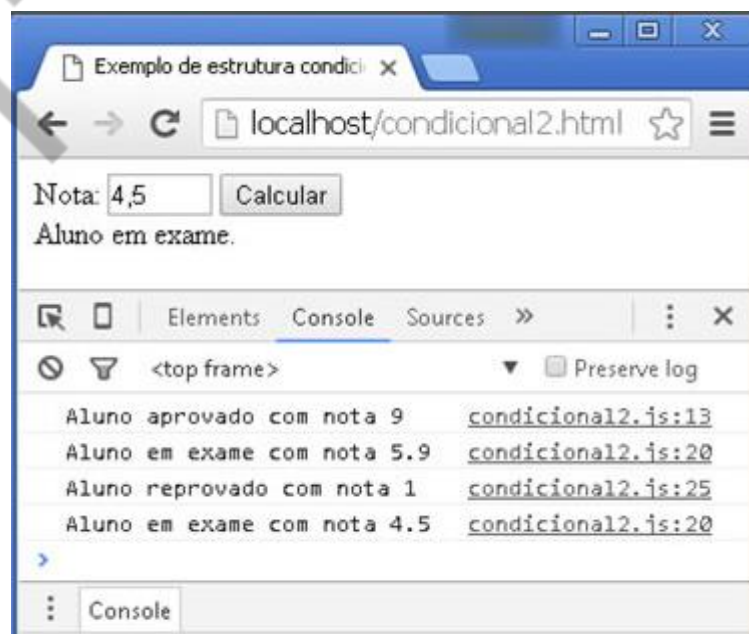


Figura 13 – Exemplo de estrutura condicional usando *if*'s aninhados/encadeados
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Existe uma alternativa para essa estrutura: trata-se do uso de *else if*, é apenas um atalho, uma força menos encadeada de estrutura e mais rápida de se digitar, poupando um abre-e-fecha chaves.

```
CONDICIONAL2.JS (ALTERNATIVO)
window.onload = function() {
    var objTxtNota = document.getElementById("nota");
    var objDivResultado =
document.getElementById("resultado");
    var objBotao = document.getElementById("botao");

    objBotao.onclick = function() {
        var objFloNota = parseFloat(objTxtNota.value);
        if (objFloNota >= 6.0) {

            objDivResultado.innerHTML = "Aluno aprovado.";
            console.log("Aluno aprovado com nota " +
objFloNota);

        } else if (objFloNota >= 4.0) {
            objDivResultado.innerHTML = "Aluno em exame.";
            console.log("Aluno em exame com nota " +
objFloNota);

        } else {

            objDivResultado.innerHTML = "Aluno reprovado.";
            console.log("Aluno reprovado com nota " +
objFloNota);

        } // fim dos ifs
    } // fim do on-click
} // fim do window.onload
```

Código-fonte 15 – Exemplo de estrutura condicional usando *if* e *else if*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

O JavaScript possui ainda o comando *switch*, que permite várias verificações de igualdade em uma única estrutura. O exemplo a seguir mostra seu funcionamento:

```
CONDICIONAL3.HTML
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <title>Exemplo de estrutura condicional</title>
        <meta charset="utf-8">
        <script src="condicional3.js"></script>
    </head>
```

```
<body>
  <label for="sigla">Digita a sigla do Estado: </label>
  <input type="text" id="sigla" style="width: 30px"
maxlength="2">
  <input type="button" id="botao" value="Descobrir">
  <div id="resultado"></div>
</body>
</html>
```

CONDICIONAL3.JS

```
window.onload = function()
{
  var objTxtSigla = document.getElementById("sigla");
  var objDivResultado =
document.getElementById("resultado");
  var objBotao = document.getElementById("botao");

  objBotao.onclick = function()
  {
    switch(objTxtSigla.value)
    {
      case "SP":
        objDivResultado.innerHTML = "São Paulo";
        break;
      case "RJ":
        objDivResultado.innerHTML = "Rio de Janeiro";
        break;
      case "MG":
        objDivResultado.innerHTML = "Minas Gerais";
        break;
      case "ES":
        objDivResultado.innerHTML = "Espírito Santo";
        break;
      default:
        objDivResultado.innerHTML = "Não é um Estado
do Sudeste";
    }
  }
}
```

Código-fonte 16 – Exemplo de estrutura condicional usando *switch*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E temos como resultado:

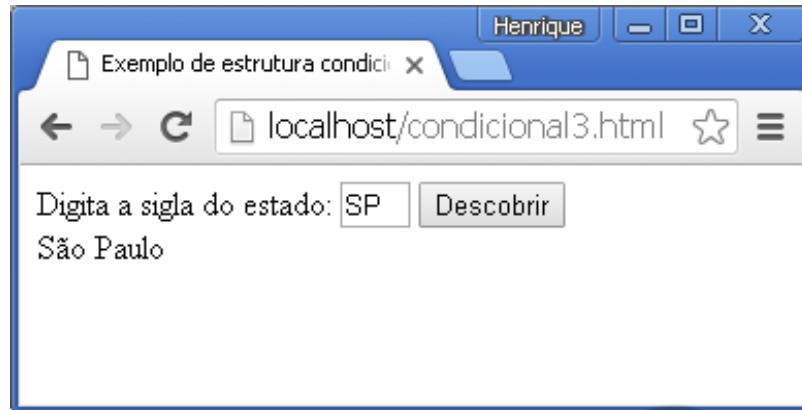


Figura 14 – Exemplo de estrutura condicional usando *switch*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

1.7.5 Estruturas de repetição

Igualmente essenciais nos sistemas, as estruturas de repetição *while*, *do..while* e *for* estão disponíveis em JavaScript.

O comando *while* nada mais é do que uma condicional simples que se repete enquanto o teste resulta em verdadeiro; sendo assim, possui todos os recursos de uma condicional *if*: podemos trabalhá-lo com várias condições amarradas com operadores lógicos E (símbolo “&&”) ou (símbolo “||”).

Um exemplo simples para verificar sua sintaxe e seu funcionamento: um laço *while* que exibe os números de 1 a 10 na tela:

REPETICA01.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de estrutura de repetição</title>
    <meta charset="utf-8">
    <script src="repeticao1.js"></script>
  </head>
  <body>
    <div id="saida"></div>
  </body>
</html>
```

REPETICA01.JS

```
window.onload = function() {
  var objDivSaida = document.getElementById("saida");

  var i = 1;
```



```
while(i <= 10) {  
    objDivSaida.innerHTML += i + "<br>";  
    i++;  
}  
}
```

Código-fonte 17 – Exemplo de estrutura de repetição usando *while*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E temos como resultado:



Figura 15 – Exemplo de estrutura de repetição usando *while*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Outra possibilidade é o laço `do...while`; sua única diferença é que a verificação da condição fica ao final do bloco e isso faz uma diferença crucial em relação ao `while`: temos a absoluta garantia de que o laço vai executar pelo menos uma única vez (diferente do `while`, que, se a condição não for satisfeita logo em seu início, não dá nem uma única volta sequer):

```
REPETICAO2.HTML  
<!DOCTYPE html>  
<html lang="pt-br">  
  <head>  
    <title>Exemplo de estrutura de repetição</title>  
    <meta charset="utf-8">  
    <script src="repeticao2.js"></script>  
  </head>  
  <body>  
    <div id="saida"></div>  
  </body>  
</html>
```

REPETICA02.JS

```
window.onload = function()
{
    var objDivSaida = document.getElementById("saida");

    var i = 1;
    do {
        objDivSaida.innerHTML += i + "<br>";
        i++;
    } while(i > 10); //A condição é falsa, logo de início..
}
```

Código-fonte 18 – Exemplo de estrutura de repetição usando *do...while*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

A estabelecida ($i > 10$) já começa falsa desde o início, afinal, o valor de i inicial é 1. Como podemos ver na demonstração a seguir, ao interpretar o código o processamento entra no bloco iniciado em “do”, quebrando o laço apenas ao fim da primeira execução, ao se deparar com a condição estabelecida em *while*:



Figura 16 – Exemplo de estrutura de repetição usando *do...while*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

A estrutura *for* é a preferida ao se construir laços como estes que usamos no exemplo, os chamados laços com número de voltas conhecido. Isso se deve ao fato de que os três elementos essenciais (inicialização da variável, condição do laço e incremento da variável) ficam agrupados em uma única linha, minimizando as falhas ao se esquecer de um deles. Veja o exemplo agora com *for*:

REPETICA03.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de estrutura de repetição</title>
    <meta charset="utf-8">
    <script src="repeticao3.js"></script>
  </head>
  <body>
    <div id="saida"></div>
  </body>
</html>
```

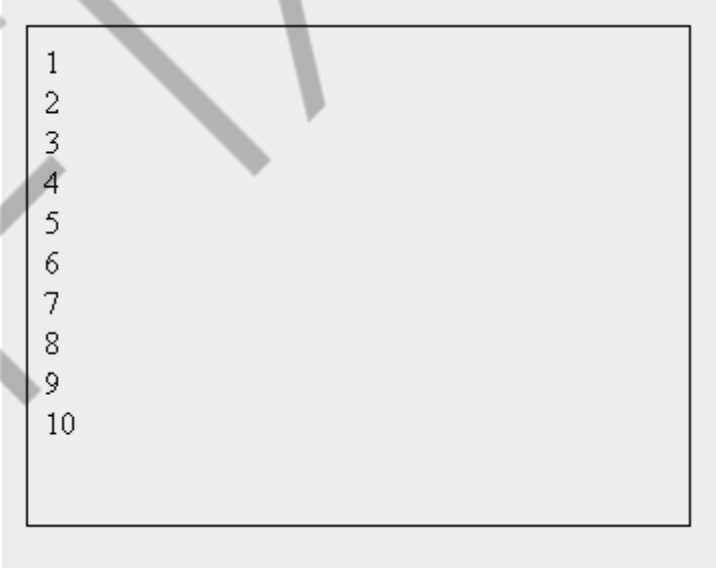
REPETICA03.JS

```
window.onload = function()
{
  var objDivSaida = document.getElementById("saida");

  for(var i = 1; i <= 10; i++) {
    objDivSaida.innerHTML += i + "<br>";
  }
}
```

Código-fonte 19 – Exemplo de estrutura de repetição usando *for*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Veja:



```
1
2
3
4
5
6
7
8
9
10
```

Figura 17 – Exemplo de estrutura de repetição usando *for*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Existe, ainda, uma estrutura de laço específica para ser usada com *arrays*. O comando *for...of* percorre automaticamente um array, do primeiro ao seu último elemento. Veja o exemplo a seguir:

REPETICA04 .HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de estrutura de repetição</title>
    <meta charset="utf-8">
    <script src="repeticao4.js"></script>
  </head>
  <body>
    <div id="saida"></div>
  </body>
</html>
```

REPETICA04 .JS

```
window.onload = function()
{
  //Declaração de array com 5 elementos.
  var aLinguagens = ["JavaScript", "Java", "PHP",
"Python"];

  var objDivSaida = document.getElementById("saida");
  //Estrutura for..of
  for(sLinguagem of aLinguagens) {
    objDivSaida.innerHTML += sLinguagem + "<br>";
  }
}
```

Código-fonte 20 – Exemplo de estrutura de repetição usando *for...of*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E ao executar:



```
JavaScript
Java
PHP
Python
```

Figura 18 – Exemplo de estrutura de repetição usando *for...of*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

DICA: Existem outras estruturas similares, como *for...in* e *for each*, não abordadas aqui. Procure a especificação oficial para mais informações e suporte de navegadores.

1.8 Funções: como criá-las?

Funções são indispensáveis nas linguagens de programação modernas por promoverem produtividade: existem dezenas, às vezes centenas, de procedimentos prontos para serem usados, porque implementá-las do zero levaria muito tempo.

Outra grande vantagem é quando nos deparamos com um conjunto de procedimentos que precisamos utilizar em várias partes do site. Em vez de fazer o tão automático “CTRL+C CTRL+V”, pergunte-se: existe uma maneira mais inteligente de fazer isso? Geralmente, a resposta para isso é “SIM!”. Podemos criar nossas próprias funções, promovendo o reúso de código, resultando um código mais organizado e mais fácil de manter.

No exemplo abaixo, aprenderemos a declarar e a chamar uma função em JavaScript:

```
FUNCOES1 . HTML
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de função</title>
    <meta charset="utf-8">
    <script src="funcoes1.js"></script>
  </head>
  <body>
    <input type="button" id="botao" value="Chamar
função">
    <div id="resultado"></div>
  </body>
</html>

FUNCOES1 . JS
//Declaração da função aloMundo()
function aloMundo()
{
  objResultado = document.getElementById("resultado");
  objResultado.innerHTML += "Alô, Mundo!<br>";
}

window.onload = function()
```

```
{  
  objBotao = document.getElementById("botao");  
  objBotao.onclick = function(){  
    aloMundo(); //Chamada da função aloMundo()  
  }  
}
```

Código-fonte 21 – Exemplo de declaração e chamada de função
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

A função é declarada com o comando `function`, e todo o procedimento que será realizado deverá estar no bloco contido por chaves. Para chamar a função, basta usar o termo que foi usado para batizá-la – em nosso caso, `aloMundo` – precedido por abre-e-fecha parênteses. Aliás, toda função, seja em sua declaração ou chamada, possui abre-e-fecha parênteses.

Veja o que acontece quando o botão “Chamar função” é pressionado três vezes:

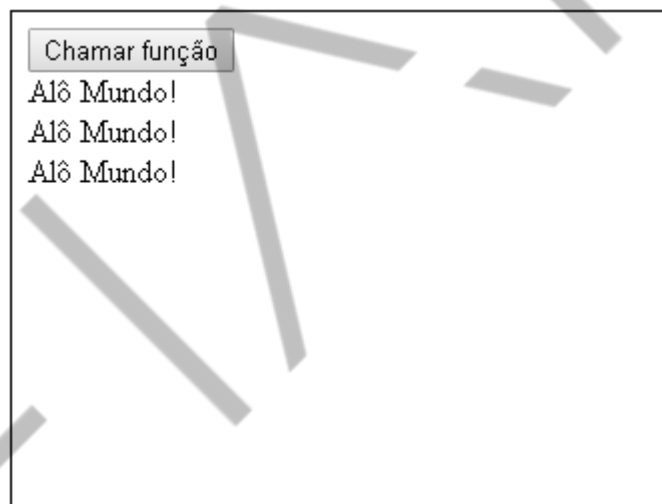


Figura 19 – Exemplo de declaração e chamada de função
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Claro que uma função como esta é “estática demais” e por isso não é útil. Informações poderiam ser passadas entre os parênteses de uma função, possibilitando parametrizar sua execução e dando a ela uma empregabilidade maior.

O exemplo a seguir é uma função de soma. Claro que ela é absolutamente desnecessária, pois o reaproveitamento de lógica é mínimo – ela é mais trabalhosa que simplesmente usar o símbolo de “+”, mas dá uma boa ideia de como os parâmetros funcionam e podem ser úteis:

```
FUNCOES2 . HTML
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de função</title>
    <meta charset="utf-8">
    <script src="funcoes2.js"></script>
  </head>
  <body>
    <input type="button" id="botao" value="Chamar
função">
    <div id="resultado"></div>
  </body>
</html>
```

FUNCOES2 . JS

```
//Declaração da função soma()
function soma(num1, num2)
{
  objResultado = document.getElementById("resultado");
  objResultado.innerHTML += (num1 + num2) + "<br>";
}

window.onload = function()
{
  objBotao = document.getElementById("botao");
  objBotao.onclick = function() {
    //Chamada da função soma()
    soma(3, 5);
  }
}
```

Código-fonte 22 – Exemplo de declaração e chamada de função com parâmetros
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

E testamos essa função clicando três vezes no botão:

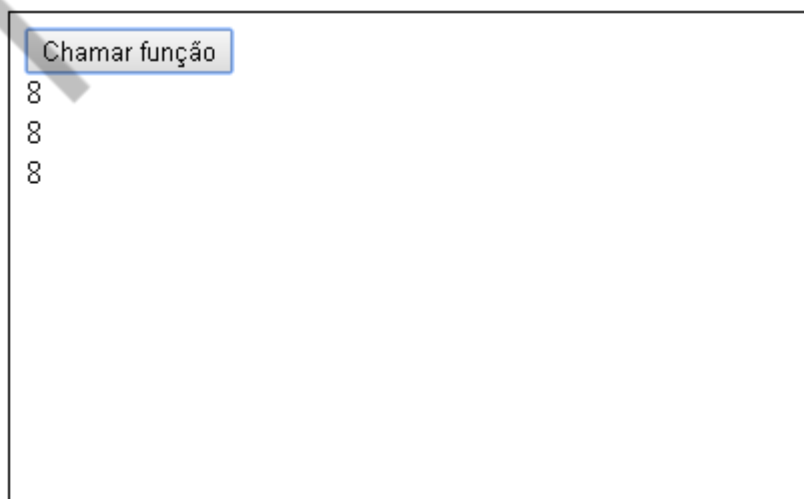


Figura 20 – Exemplo de declaração e chamada de função com parâmetros
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Ainda está longe de ser uma função dinâmica. Aplicar o resultado no <div> diretamente de dentro da função a torna “engessada” demais: não é possível recuperar o resultado da soma e utilizar em outra conta, o resultado não pode ser enviado em uma requisição, post ou aplicado em outro elemento, ou mesmo em outra <div>. Nada, exceto aplicar o resultado na <div> cujo id é “resultado”.

É possível retornar esse resultado para a chamada da função, utilizando o comando return. Somente depois disso decidimos o que fazer com o resultado. Dessa forma, aumentamos a reusabilidade da função.

Veja, a seguir, o exemplo melhorado:

```
FUNCOES3 . HTML
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de função</title>
    <meta charset="utf-8">
    <script src="funcoes3.js"></script>
  </head>
  <body>
    <input type="button" id="botao" value="Chamar
função">
    <div id="resultado"></div>
  </body>
</html>

FUNCOES3 . JS
//Declaração da função soma()
function soma(num1, num2)
{
  return num1 + num2;
}

window.onload = function()
{
  objBotao = document.getElementById("botao");
  objBotao.onclick = function(){
    //Chamadas (sim, no plural!) da função soma()
    objResultado =
document.getElementById("resultado");

    objResultado.innerHTML += soma(soma(3, 5), 8) +
"<br>";
  }
}
```

Código-fonte 23 – Exemplo de declaração e chamada de função com parâmetros e retorno
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Ao testar, clicando no botão:



Figura 21 – Exemplo de declaração e chamada de função com parâmetros e retorno
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Repare que, embora tenhamos clicado apenas uma vez no botão, a função foi chamada duas vezes: existe uma chamada de função dentro de outra. Assim como em equações matemáticas, o que está separado por parênteses tem preferência, assim sendo, a soma entre 3 e 5. Seu resultado, 8, é retornado e alimenta a segunda chamada da função, que agora é responsável por somar 8 e 8, e é esse resultado que vemos no exemplo. Esse procedimento era impossível antes de reformularmos pela última vez.

Que tal incrementarmos o exemplo, adicionando duas caixas de texto e dando a opção para o usuário digitar os dois valores?

FUNCOES4 . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de função</title>
    <meta charset="utf-8">
    <script src="funções4.js"></script>
  </head>
  <body>
    <input type="text" id="num1" style="width: 20px"> +
    <input type="text" id="num2" style="width: 20px"> =
    <input type="button" id="botao" value="Somar">
    <div id="resultado"></div>
  </body>
</html>
```

FUNCOES4 . JS

```
//Declaração da função soma()
function soma(num1, num2)
{
    return num1 + num2;
} // fim da função soma()

window.onload = function()
{
    objBotao = document.getElementById("botao");
    objBotao.onclick = function(){
        objTxtNum1 = document.getElementById("num1");
        objTxtNum2 = document.getElementById("num2");
        objResultado = document.getElementById("resultado");
        objResultado.innerHTML += soma(
            parseInt(objTxtNum1.value),
            parseInt(objTxtNum2.value) ) + "<br>";
    } // fim do onclick
} // fim do window.onload
```

Código-fonte 24 – Exemplo de declaração e chamada de função com parâmetros, retorno e caixas de texto

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2021)

Finalmente:



Figura 22 – Exemplo de declaração e chamada de função com parâmetros, retorno e caixas de texto

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Uma pequena calculadora. Esse exemplo é apenas uma pequena demonstração do potencial das funções na programação.

1.9 Programação orientada a eventos

Por ser uma linguagem muitas vezes utilizada do lado do cliente e, portanto, interpretada pelo navegador, o grande trunfo do JavaScript é a possibilidade de se programar por meio de eventos: podemos criar procedimentos que serão executados quando um determinado gatilho for acionado.

Os eventos mais comuns são aqueles disparados pelo próprio usuário, quando ele clica em algum elemento com o botão do mouse, quando ele digita alguma coisa ou quando um determinado elemento ganha foco. Entretanto, existem outros disparados por tempo ou por alguma funcionalidade HTML 5 que tenha sido previamente solicitada.

Aliás, muitos são os novos eventos criados com o surgimento do HTML 5: para arrastar-e-soltar, controle de mídia, de toque em tela sensível, entre vários outros. Vamos nos ater aqui aos eventos mais comuns e simples, mas o convidamos a dar uma olhada em nosso material de HTML 5.

Os eventos mais comuns são:

- **onload:** específico para o elemento que representa a janela de navegação, é disparado quando ela está plenamente carregada (e todos os elementos HTML estão ativos).
- **onclick:** disparado quando o usuário clica com o botão do *mouse* em cima do elemento.
- **onchange:** disparado quando o elemento muda, como o conteúdo de uma caixa de texto, por exemplo.
- **onmouseover:** disparado quando o ponteiro do *mouse* passa sobre o elemento HTML.
- **onmouseout:** disparado quando o ponteiro do *mouse* deixa/sai do elemento HTML.
- **onkeydown:** disparado quando a tecla que está sendo utilizada foi pressionada e está em descida. É o primeiro evento disparado em uma digitação.

- **onkeypress:** disparado quando a tecla que está sendo utilizada foi totalmente pressionada. É o segundo evento disparado em uma digitação.
- **onkeyup:** disparado quando a tecla que está sendo utilizada foi pressionada e está subindo, retornando ao estado original. É o terceiro evento disparado em uma digitação.
- **onsubmit:** disparado no momento em que o formulário HTML está sendo submetido ao servidor.
- **onfocus:** disparado quando o elemento em questão ganha foco. O ganho de foco pode acontecer de várias maneiras. A mais comum é clicando em cima do elemento. E a outra é por meio de navegação pulando de um elemento a outro, usando a tecla TAB. É possível até mesmo conceder foco automaticamente a um dos elementos do documento, ao final de sua carga.
- **onblur:** disparado quando o elemento em questão perde o foco, deixa de estar em referência.

Já vimos o evento **onclick** sendo aplicado aos botões em exemplos anteriores, portanto, vamos a exemplos mais elaborados: o primeiro deles, um retângulo “desenhado” com CSS. Ao utilizar o evento **onmouseover**, mudamos sua cor de fundo quando o ponteiro do *mouse* passa por cima do elemento. O exemplo também exige um **onmouseout**, de forma que restauramos o fundo original da Figura “Exemplo de eventos *onmouseover* e *onmouseout*”, quando o ponteiro do *mouse* deixar o elemento:

```
EVENTOS1 . HTML
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de eventos</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css"
href="eventos1.css">
    <script src="eventos1.js"></script>
  </head>
  <body>
    <div id="retangulo"></div>
  </body>
</html>
```

EVENTOS1.CSS

```
#retangulo
{
    width: 100px;
    height: 100px;
    border: 2px solid black;
    background-color: #DEB887;
}
```

EVENTOS1.JS

```
window.onload = function()
{
    var oRetangulo =
document.getElementById("retangulo");

    //Pinta o fundo de verde quando o ponteiro do mouse
passar por cima
    oRetangulo.onmouseover = function(){
        oRetangulo.style.backgroundColor = "#008000";
    }

    //Restaurar a cor original quando o ponteiro sair
    oRetangulo.onmouseout = function(){
        oRetangulo.style.backgroundColor = "";
    }
}
```

Código-fonte 25 – Exemplo de eventos *onmouseover* e *onmouseout*

Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Na sequência, um teste do exemplo:

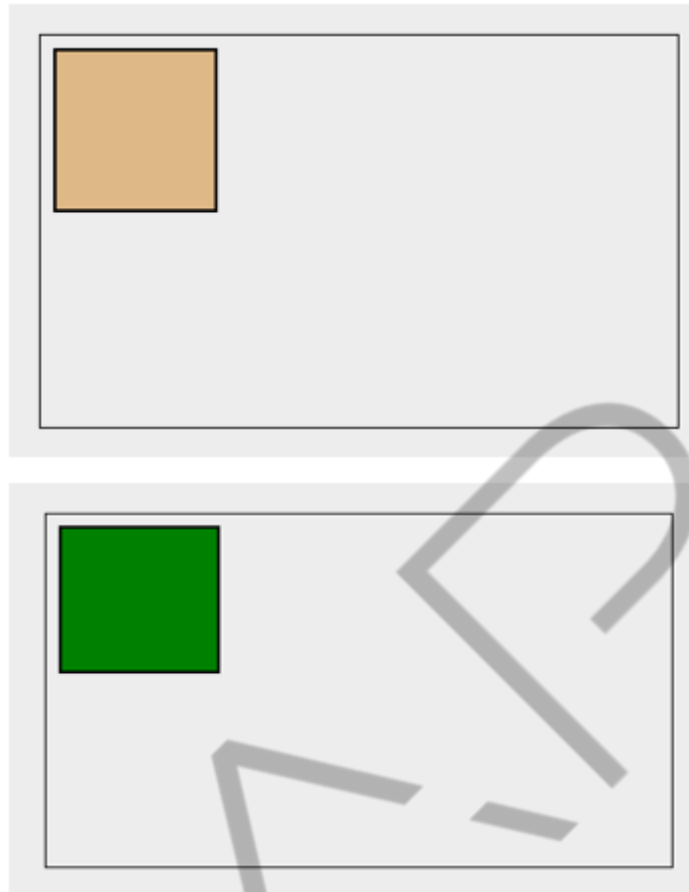


Figura 23 – Exemplo de eventos *onmouseover* e *onmouseout*
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

O exemplo seguinte é uma validação de um campo de formulário para CEP aplicada utilizando ***onblur***, ou seja, quando o usuário termina de digitar a informação e deixa o campo. Além da tradicional mensagem de erro, trocaremos a cor da borda do campo para vermelho.

O evento ***onfocus*** também deve ser usado – afinal, quando o usuário volta ao campo para resolver o problema, devemos apagar as referências ao erro cometido: a mensagem de erro some e a borda volta à sua cor original:

EVENTOS2 . HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de eventos</title>
    <meta charset="utf-8">
    <script src="eventos2.js"></script>
  </head>
  <body>
    <label for="cep">CEP: </label>
    <input type="text" id="cep" style="width: 70px">
    <span id="msgCep"></span>
```

```
</body>
</html>

EVENTOS2.JS
window.onload = function()
{
    var oTxtCep      = document.getElementById("cep");
    var oDivMsgCep   = document.getElementById("msgCep");

    //Disparado quando o campo de CEP perder o foco.
    oTxtCep.onblur = function(){
        var oRegExp = new RegExp("^([0-9]{5})\-[0-9]{3}$");
        if (oRegExp.test(oTxtCep.value) == true) {
            oTxtCep.style.borderColor = "#008000";
            oDivMsgCep.style.color = "#008000";
            oDivMsgCep.innerHTML = "Campo preenchido com
sucesso!";
        } else {
            oTxtCep.style.borderColor = "#FF0000";
            oDivMsgCep.style.color = "#FF0000";
            oDivMsgCep.innerHTML = "CEP inválido! Informe
novamente!";
        }
    }

    //Limpar"
    oTxtCep.onfocus = function(){
        oTxtCep.style.borderColor = "";
        oDivMsgCep.style.color = "";
        oDivMsgCep.innerHTML = "";
    }
}
```

Código-fonte 26 – Exemplo de validação de campo de formulário com ganho e perda de foco
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Outra boa novidade desse exemplo é o uso de expressões regulares. Trata-se da melhor forma de se definir padrões de caracteres e é aplicável em várias linguagens de programação. O padrão **`^[0-9]{5}\-[0-9]{3}$`** significa que os cinco primeiros caracteres são dígitos entre 0 e 9; o sexto caractere é, obrigatoriamente, uma barra ou hífen ("-"); e os três caracteres seguintes (e finais, graças ao cifrão - \$) são outros três dígitos válidos, ou seja, um padrão de CEP, com sua máscara. Procure saber mais a respeito de expressões regulares, temos certeza de que você vai se encantar.

Faça como nós e teste o código-fonte:

The figure consists of three vertically stacked screenshots of a web form, illustrating the dynamic behavior of a CEP validation field during focus changes.

- Top Screenshot:** The text "CEP: aaaa" is displayed. The input field "aaaa" has a red border, indicating it is the active element. To the right of the input, the text "CEP inválido! Informe novamente!" is shown in red, indicating a validation error.
- Middle Screenshot:** The text "CEP: aaaa" is displayed. The input field "aaaa" has a blue border, indicating it is no longer the active element.
- Bottom Screenshot:** The text "CEP: 03547-050" is displayed. The input field "03547-050" has a green border, indicating it is the active element. To the right of the input, the text "Campo preenchido com sucesso!" is shown in green, indicating a successful validation.

Figura 24 – Exemplo de validação de campo de formulário com ganho e perda de foco
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

1.10 Ajax

Ajax é o acrônimo de *Asynchronous Javascript and XML*, que quer dizer JavaScript e XML assíncronos. Trata-se de um método para acessar novas informações no servidor web sem a necessidade de se fazer uma carga completa em

um documento HTML. Embora a possibilidade já exista há décadas, apenas nos últimos anos ela se tornou mais difundida e, por que não dizer, necessária.

O grande trunfo é o fato de ser assíncrono, ou seja, o navegador não “trava” a navegação do usuário enquanto o conteúdo não for renderizado, como em um HTTP Post tradicional, que é síncrono. O navegador manda a requisição e fica monitorando a resposta, executando os procedimentos apenas quando a resposta efetivamente chega.

O exemplo a seguir está no padrão HTML 5, pois a requisição Ajax é realizada com a API do novo XMLHttpRequest 2.0. Faremos isso por duas boas razões: a requisição antiga é extremamente complicada (graças à falta de padrões do navegador Microsoft Internet Explorer) e a adesão é praticamente total. O exemplo funcionará em IE 10 em diante, a única exceção é o navegador portátil Opera Mini, atualmente sem suporte.

O arquivo-texto a seguir traz algumas cidades do estado de São Paulo e simula um *script* dinâmico acessando um banco de dados:

AJAX1_CIDADES.TXT

São Paulo
São Vicente
Santos

Código-fonte 27 – Exemplo de arquivo-texto representando um *script* dinâmico
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

No mesmo diretório, mas independente, a solução que, ao clicar no botão, chamará o arquivo-texto e exibirá as cidades em um <div>:

AJAX1.HTML

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Exemplo de eventos</title>
    <meta charset="utf-8">
    <script src="ajax1.js"></script>
  </head>
  <body>
    <span id="estado">Clique para ver algumas cidades do
Estado de São Paulo</span>
    <input type="button" id="botao" value="Buscar
cidades">
    <div id="cidades"></div>
  </body>
</html>
```

```
AJAX1.JS
window.onload = function()
{
    var oBotao = document.getElementById("botao");
    var oDiv    = document.getElementById("cidades");

    oBotao.onclick = function(){
        var xhttp = new XMLHttpRequest();
        //Disparado quando o servidor dá algum sinal de
vida..
        xhttp.onreadystatechange = function() {

            /*readyState em 4 significa que o retorno
aconteceu com sucesso. status em 200 significa requisição
HTTP código 200, significa OK, sucesso. */

            if (xhttp.readyState == 4 && xhttp.status == 200)
            {
                document.getElementById("cidades").innerHTML =
"<pre>" + xhttp.responseText + "</pre>";
            }
        };
        xhttp.open("GET", "ajax1_cidades.txt", true);
        xhttp.send();
    }
}
```

Código-fonte 28 – Exemplo de requisição Ajax
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Ao testar o exemplo:

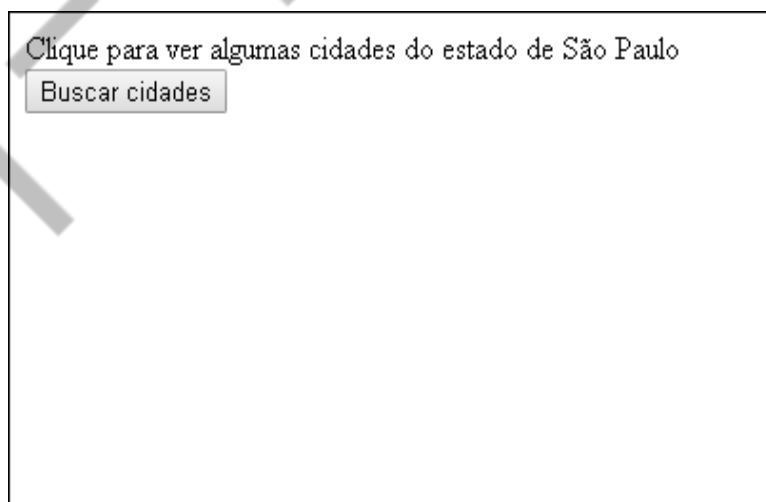


Figura 25 – Exemplo de requisição Ajax
Fonte: Elaborado pelo autor (2016), adaptado por FIAP (2017)

Vale repetir: o nome das três cidades estava em um arquivo à parte. O arquivo-texto foi solicitado e devolvido ao navegador de forma assíncrona e invisível. Não houve recarga da página ou travamento da navegação do usuário. Estas são as grandes razões para se usar Ajax e as possibilidades são infinitas.

EMANIP

CONCLUSÃO

A linguagem de programação JavaScript é indispensável em qualquer solução web nos dias de hoje. E ganhou ainda mais importância, pois pode ser utilizada em desenvolvimento *server-side* (com o chamado Node.js, assunto do ano que vem).

Este conteúdo cumpriu seu objetivo de apresentar as principais estruturas de linguagem e seus usos e funções mais relevantes; muitas outras existem e dezenas foram criadas com o advento do HTML 5. As possibilidades são infinitas!

REFERÊNCIAS

ECMA INTERNACIONAL. **ECMAScript® Language Specification 5.1 Edition**. Disponível em: <<http://www.ecma-international.org/ecma-262/5.1/index.html>>. Acesso em: 13 abr. 2021.

ECMA INTERNACIONAL. **ECMAScript® 2015 Language Specification 6th Edition**. Disponível em: <<http://www.ecma-international.org/ecma-262/6.0/index.html>>. Acesso em: 13 abr. 2021.

W3C. **Document Object Model (DOM)**. Disponível em: <<https://www.w3.org/DOM/>>. Acesso em: 13 abr. 2021.