



Apostila de JavaScript (ES6+)

Nível básico



Prof. Fausto G. Cintra
(professor@faustocintra.com.br)

19/08/2021

Sumário

APRESENTAÇÃO	3
1 COMEÇANDO COM JAVASCRIPT	5
1.1 Uma breve história do JavaScript (sim, é importante!)	5
1.2 Características da linguagem JavaScript	7
1.3 O JavaScript e as páginas Web	8
1.4 A estrutura de uma página HTML	8
1.5 Adicionando JavaScript a uma página HTML	10
1.5.1 Adicionando comentários	11
1.6 Usando o console JavaScript	12
2 VARIÁVEIS E TIPOS DE DADOS	14
2.1 Declaração de variáveis	14
2.1.1 Nomeando variáveis	15
2.1.2 Convenções de nomeação de variáveis	15
2.2 Atribuindo valores a variáveis	16
2.3 Tipos de dados	17
2.3.1 Descobrindo o tipo do valor de uma variável	18
3 OPERADORES	20
3.1 Operadores aritméticos	20
3.1.1 Quando os operandos não são números	21
3.2 Operadores compostos de atribuição	24
3.3 Incremento e decremento	25
3.4 Operadores lógicos	26
3.5 Operadores relacionais (ou de comparação)	26
3.6 Precedência	29
4 ENTRADA E SAÍDA	31
4.1 Exibindo informações	31
4.1.1 alert()	31
4.1.2 document.write()	33
4.1.3 console.log()	34
4.2 Coletando informações	34
4.2.1 prompt()	34
4.2.2 confirm()	36

5	ESTRUTURAS CONDICIONAIS	38
5.1	A estrutura if	38
5.1.1	if..else	39
	PROBLEMA REAL: conversão de tipos	40
5.1.2	if..else if..else	42
5.2	A estrutura switch..case	44
5.2.1	break	47
5.2.2	case s vazios	47
5.2.3	Limitação	49
5.3	O operador ternário	50
6	ESTRUTURAS DE REPETIÇÃO	52
6.1	while	52
6.2	do..while	54
6.3	for	55
7	FUNÇÕES (básico)	60
7.1	Declaração e chamada de função	60
7.1.1	Múltiplos parâmetros	62
7.1.2	Funções sem parâmetros	63
7.1.3	Funções sem valor de retorno	64
7.2	Expressão de função	65
7.3	Onde declarar ou colocar expressar funções no código	66
7.4	Variáveis e funções	67
8	MANIPULAÇÃO DE STRINGS	69
9	DOCUMENT OBJECT MODEL (DOM)	70
10	VETORES	71
11	OBJETOS	72
12	REVISITANDO O DOM	73
13	FUNÇÕES (avançado)	74

APRESENTAÇÃO

JavaScript é uma das linguagens de programação mais difundidas atualmente, sendo imprescindível para aplicações voltadas à Internet. Por isso, todos quantos queiram se tornar desenvolvedores Web precisam dominá-la.

Meu desejo, ao disponibilizar a você esta apostila, é ajudá-lo(a) a dar seus primeiros passos com a linguagem, com um conteúdo atualizado que abrange as principais modificações desde a versão ES6 (lançada em 2015). Ela é o resultado de minha experiência de oito anos como professor de Ensino Superior, quatro deles ensinando JavaScript. Procurei trazer para o texto respostas às dúvidas mais comuns que alunos iniciantes na linguagem têm em sala de aula.

Este material é de nível básico; portanto, não cobre todos os recursos do JavaScript nem pretende ser a última palavra sobre o assunto. Sempre que algum tópico for secundário para quem está começando, ou exigir algum aprofundamento que saia da proposta do texto, há *links* para fontes externas.

A quem se destina esta apostila

Para o melhor aproveitamento deste material, é desejável que o estudante já tenha noções de:

- algoritmos e lógica de programação;
- arquitetura e organização de computadores; e
- uso de editores de código.

E, claro, vontade de aprender. Programação é, sobretudo, prática. Não basta apenas **ler** o texto; é necessário colocar a mão na massa, codificar, testar e aprender com os erros.

A quem NÃO se destina esta apostila

Se você ainda não tem nenhum conhecimento de programação, peço minhas desculpas, mas esta apostila não é para você. Existem alguns passos que você deve percorrer antes de se aventurar com JavaScript. Recomendo os seguintes materiais de estudo:

- [Curso em Vídeo: Algoritmos](#)
- [IF Fluminense: Apostila de Lógica de Programação](#)
- [Livro: Algoritmos: Lógica Para Desenvolvimento de Programação de Computadores – Edição Revisada e Atualizada \(Manzano e Oliveira\)](#)

Meus melhores votos de bons estudos!

Um abraço,

Prof. Me. Fausto G. Cintra

- professor@faustocintra.com.br
- [Currículo Lattes](#)

Capítulo 1

COMEÇANDO COM JAVASCRIPT

1.1 Uma breve história do JavaScript (sim, é importante!)

Em 1989, **Tim Berners-Lee** inventou a linguagem HTML e, com ela, a possibilidade de criar páginas Web. Isso revolucionou a Internet da época, que, até então, era acessada principalmente por terminais de texto, com suporte muito limitado a imagens e outros tipos de mídia. Essa “nova” internet passou a ser conhecida pela sigla WWW (de *World Wide Web*, Teia de Alcance Mundial).

As páginas Web de Berners-Lee fizeram muito sucesso. Mas elas tinham uma limitação: apenas entregavam conteúdo multimídia, de forma estática, como a televisão. Não havia forma de o usuário interagir com esse conteúdo, nem de enviar informações de volta à origem. E isso restringia as possibilidades de uso das páginas Web.

Em 1994, o **Netscape Navigator** era o navegador Web líder de mercado. Para resolver o problema da falta de interatividade, a empresa Netscape decidiu por introduzir uma linguagem de programação que pudesse ser incorporada às páginas Web e executada pelo navegador ao carregá-las.

A linguagem em questão, depois de muitos nomes provisórios, ganhou o nome de JavaScript, tendo sido lançada em 1995. Foi desenvolvida originalmente por **Brendan Eich**, que, nos dias de hoje, é o nome por trás do navegador **Brave**, focado em privacidade e eliminação de propagandas indesejadas.

A escolha pelo nome JavaScript não foi por acaso, tratando-se, na verdade, de uma bela jogada de *marketing*. Na época do lançamento, uma outra linguagem, também recém-surgida, estava fazendo bastante sucesso: o **Java**, da empresa Sun Microsystems. A intenção da Netscape era embarcar na popularidade do Java, mas, no fim das contas, acabou por causar uma grande confusão, levando muitos a acreditar que o JavaScript era baseado em Java. No entanto, **JavaScript e Java são linguagens totalmente diferentes**, tanto em finalidade quanto na forma de se programar e, ainda hoje, os mais desavisados pensam que se trata da mesma

coisa por causa da semelhança do nome.

A Microsoft, uma das gigantes da tecnologia já naquela época, não podia ficar de fora da onda da WWW. Em 1996, a empresa lançou o **Internet Explorer 3.0** com sua própria “versão” da linguagem JavaScript, a qual denominou JScript. O JScript possuía várias extensões e diferenças em relação ao JavaScript original, fazendo com que *websites* feitos com ele só funcionassem corretamente no Internet Explorer.

Diante desse fato, em novembro de 1996, a Netscape anunciou que havia submetido o JavaScript à **ECMA** (*European Computer Manufacturers Association*, ou Associação Europeia de Fabricantes de Computadores) como candidata a padrão industrial. Com isso, a empresa abriu as especificações da linguagem para o público, como se quisesse dizer que “não tinha nada a esconder”, ao contrário da concorrente. O trabalho que se seguiu resultou na versão padronizada chamada **ECMAScript** (ECMA-262), versão 1.0 (junho de 1997). O padrão foi evoluindo ao longo dos anos, sendo a versão mais recente a ES2021 (ECMAScript 2021 ou, ainda, edição 12), lançada em junho de 2021. Atualmente, o ECMAScript é considerado o padrão de JavaScript a que todo navegador deve dar suporte.

Contudo, a chamada Primeira Guerra dos *Browsers* (~ 1995—2001) não demorou a começar. A Microsoft passou a fornecer seu navegador junto com o sistema operacional Windows, fazendo com que as pessoas não mais precisassem recorrer ao *download* do Netscape para ter um programa do tipo. Durante praticamente toda a primeira década do século, o Internet Explorer reinou absoluto na liderança do mercado de navegadores. Com a posição alcançada, a Microsoft ignorava o padrão ECMAScript, investindo em suas extensões proprietárias. Desenvolver para a Web tornou-se caótico, devido à falta de compatibilidade entre o JScript do Internet Explorer com o JavaScript padrão.

A empresa Netscape fechou as portas em 2002, exaurida por uma longa batalha judicial contra a Microsoft. Dos escombros da Netscape surgiram dois projetos, que resultaram nos navegadores **Opera** e **Firefox**, que existem até hoje. Embora fossem bons produtos e tivessem alcançado algum destaque, nenhum deles tinha força para competir com a toda poderosa Microsoft.

Foi necessário que outra gigante da tecnologia entrasse no páreo para ameaçar a liderança da Microsoft e do Internet Explorer entre os navegadores, iniciando a Segunda Guerra dos *Browsers*. Em 2008, a Google lançou o **Google Chrome**, com importantes diferenciais. Seu mecanismo de processamento de JavaScript, chamado V8, era extremamente rápido, fazendo com que as páginas Web carregassem bem mais depressa que no Internet Explorer. Além disso, o Chrome era baseado em um projeto de código aberto chamado **Chromium**, o que incentivou desenvolvedores a colaborar para o aperfeiçoamento do produto. Mais do que isso, a Google adotou como princípio a aderência aos chamados “padrões Web”, dentre os quais se encontra o ECMAScript.

Por volta de 2013, cinco anos após seu lançamento, o Google Chrome conquistou a lide-

rança do mercado de navegadores. A Microsoft se viu obrigada a substituir o Internet Explorer por um novo navegador, o Edge, mais conforme aos padrões Web.

Antes, em 2010, aproveitando-se do fato de o Chrome ter seu código-fonte aberto, Ryan Dahl isolou o código do interpretador de JavaScript V8 e criou o **Node.js**. Este *software* permite com que o JavaScript, uma linguagem de Internet projetada para ser executada dentro do navegador, possa ser executado também fora dele, tornando-a uma linguagem de propósito geral.

A aderência do Google aos padrões Web, a derrocada do Internet Explorer e seu JScript não-padrão e a popularidade do Node.js levaram a um impulsionamento do desenvolvimento da linguagem JavaScript, que vem ganhando uma nova versão por ano desde 2015, quando foi lançada a versão ES6.

A versão ES6 representou um marco de modernização na linguagem, e as versões subsequentes prosseguiram com a adição de funcionalidades. **Esta apostila tem como um de seus objetivos ensinar o JavaScript moderno**, abrangendo as principais características introduzidas desde o ES6, sem deixar de mencionar as formas originais, quando existirem.

1.2 Características da linguagem JavaScript

Dentre os principais atributos da linguagem, podemos destacar:

- ela é **interpretada**, ou seja cada linha de código é transformada em linguagem de máquina à medida que vai sendo executada pelo navegador. **Programas escritos em linguagens interpretadas são geralmente chamados de *scripts***. Existe um outro tipo de linguagem, as compiladas, em que todo o código de um programa é transformado em linguagem de máquina para só então poder ser executado.
- ela é **imperativa**: você deve fornecer instruções claras de como conseguir o que quer, assim como em Python, VBA e C, entre outras linguagens. Já nas linguagens do tipo declarativo, é necessário dizer apenas o que se quer, sem precisar indicar as instruções passo a passo.
- ela é **estruturada**, possuindo estruturas de construção de blocos lógicos de código, como a grande maioria das linguagens de programação utilizadas na atualidade. Sua sintaxe (forma de escrever as instruções) assemelha-se à das linguagens C e PHP.
- ela tem **tipagem dinâmica**: não é necessário indicar o tipo ao declarar uma variável, sendo ele determinado a partir do valor armazenado na variável. Como consequência, uma mesma variável pode inicialmente guardar um número e, mais tarde, ter seu valor trocado por uma *string*.

Caso você tenha tido contato com JavaScript antes, principalmente se há mais de cinco anos, pode se lembrar de que era costume colocar um **;** (ponto-e-vírgula) ao final de cada linha de código, como terminador de instruções. Saiba que esses **ponto-e-vírgulas finais são**

opcionais em JavaScript (exceto em raríssimos casos), e, assim sendo, você não os verá nesta apostila.

Quanto à sintaxe, isto é a forma de escrever as instruções, JavaScript faz parte de uma grande “família” iniciada pela linguagem C e que tem, entre seus membros, as linguagens C++, C#, PHP. Portanto, se você já entrou em contato com alguma delas, verá algo similar em JavaScript.

1.3 O JavaScript e as páginas Web

As páginas Web são formadas pela combinação de três tecnologias:

- **HTML** (*Hypertext Markup Language*, isto é, linguagem de marcação de hipertexto): é responsável por estruturar o conteúdo da página, ou seja, **o que** você vê nela.
- **CSS** (*Cascading Style Sheets*, folhas de estilo em cascata): cuida da aparência da página, como cores, fontes e alinhamento dos elementos. Controla **como** o conteúdo é exibido na página.
- **JavaScript**: é uma linguagem de **programação** que pode ser adicionada às páginas Web, conferindo-lhes interatividade para com o usuário. O JavaScript é capaz, por exemplo, de verificar em um formulário se o usuário digitou algo diferente do esperado ou de fazer coisas se movimentarem pela página.

IMPORTANTE

Como o foco desta apostila é a linguagem JavaScript, veremos apenas o necessário de HTML e CSS, sem aprofundamento. Caso você queira aprender mais sobre essas duas tecnologias, recomendo o [Curso em Vídeo de HTML e CSS](#).

A linguagem JavaScript será o objeto do nosso estudo. No entanto, como iremos utilizá-la **dentro** de uma página Web, é necessário aprender o básico de HTML.

1.4 A estrutura de uma página HTML

Uma página Web (também chamada de página HTML) é um arquivo de texto simples com extensão `.html` ou `.htm`. A Listagem 1.1 seguir apresenta a estrutura básica de um arquivo HTML.

IMPORTANTE

Os números que aparecem à esquerda do código **não** fazem parte dele. Servem apenas para que possamos nos referir a diferentes partes do código usando o número da linha.

Listagem 1.1 Estrutura básica de um arquivo HTML

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5   <title>Minha primeira página HTML</title>
6 </head>
7 <body>
8   <p>Olá, mundo!</p>
9 </body>
10</html>
```

Quando o arquivo HTML contendo este código for exibido em um navegador Web, veremos um resultado semelhante ao da Figura 1.1:

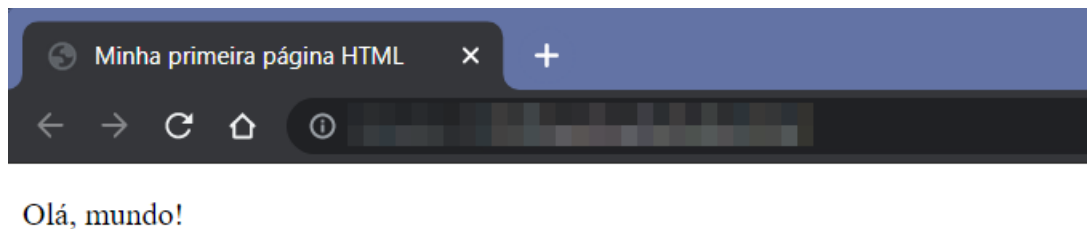


Figura 1.1: Exibição de uma página HTML

Como podemos ver, o HTML é formado por elementos delimitados pelos caracteres `<` e `>`, os quais são chamados **tags**.

Muitas *tags* vêm em pares, formando **seções**. Podemos observar, por exemplo, que a tag `<body>`, chamada *tag* de abertura, tem a correspondente *tag* de fechamento `</body>` (note a presença da `/` antes do nome da *tag*).

Agora, vamos analisar cada uma das partes do código.

- `<!DOCTYPE html>` (linha 1): essa *tag* serve para indicar ao navegador Web que irá exibir a página qual a versão da linguagem HTML está sendo usada. No caso, esse *doctype* indica que se trata da versão 5 do HTML, a mais recente.
- Seção **html** (linhas 2 a 10): a maior parte do código da página fica nessa grande seção. Ali dentro, temos as seções **head** e **body**.
- Seção **head** (linhas 3 a 6): aqui são colocadas *tags* de configuração da página, como a `<meta charset="UTF-8">` (linha 4), para garantir que os caracteres acentuados sejam exibidos corretamente. Os elementos dessa seção, normalmente, não têm um efeito visível para o usuário. Uma exceção é a tag `<title>` (linha 5), cujo conteúdo aparece na

aba do navegador onde a página estiver sendo exibida.

- Seção **body** (linhas 7 a 9): todo o conteúdo da página que será visível para o usuário é colocado nessa seção. No código de exemplo, temos um parágrafo (`<p>` , linha 8) contendo um texto a ser exibido.

1.5 Adicionando JavaScript a uma página HTML

Para utilizar JavaScript em uma página HTML, precisamos criar uma seção `<script></script>`, normalmente dentro da seção **head**, e adicionar o código JavaScript dentro dela.

A Listagem 1.2 apresenta uma nova versão da mensagem “Olá, mundo!” mas, desta vez, quem dará o recado é a linguagem JavaScript.

Listagem 1.2 Código JavaScript em uma página HTML

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5   <title>Minha primeira página HTML</title>
6   <script>
7     alert('Olá, mundo!')
8   </script>
9 </head>
10 <body>
11
12 </body>
13 </html>
```

IMPORTANTE

Para executar o código JavaScript dentro de uma página HTML, basta abri-la em um navegador Web. Recomendo usar o **Google Chrome** atualizado.

Observe com atenção as linhas 6 a 8. Aberto no navegador, um arquivo HTML com este código produz o seguinte resultado (Figura 1.2):

Portanto, agora você já sabe. Toda vez que formos usar JavaScript em uma página Web, devemos:

1. Criar um arquivo com extensão `.html` ou `.htm`.
2. Colocar dentro desse arquivo a estrutura básica de um arquivo HTML. Editores de código, como o **Visual Studio Code** ou o **Gitpod** possuem recursos que geram automaticamente

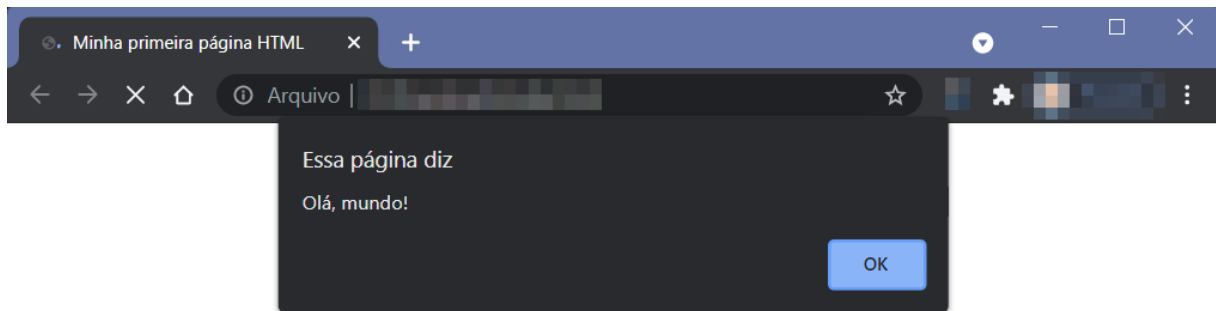


Figura 1.2: Mensagem exibida usando JavaScript

este código.

3. Adicionar uma seção `<script></script>` na seção **head** e colocar as instruções JavaScript dentro dela.

1.5.1 Adicionando comentários

Bons desenvolvedores não se preocupam apenas em escrever bem as instruções que serão executadas. Eles também cuidam da documentação, adicionando comentários que explicam os principais pontos, como forma de colaborar para que o código seja compreendido por outras pessoas (e até pelo autor, no futuro).

Todas as linguagens possuem maneiras de inserir comentários no código. Em JavaScript, eles assumem duas formas:

- **comentários de linha:** são iniciados com `//` (duas barras) e, como o próprio nome indica, terminam junto com o fim da linha onde foram colocados.
- **comentários de bloco:** começam com `/*` (barra asterisco) e terminam com `*/` (asterisco barra). Tudo o que estiver entre eles é considerado comentário, que pode ter várias linhas.

A Listagem 1.3 exemplifica esses tipos de comentários.

Listagem 1.3 Comentários em JavaScript

```
1 // Exibe uma mensagem em uma caixa de diálogo
2 alert('Olá, como vão seus estudos?') // Posso comentar aqui tb
3
4 /*
5     A linha abaixo exibe um texto na área <body>
6     do arquivo HTML
7 */
8 document.write('Estou adorando aprender JavaScript!')
```

IMPORTANTE

A partir desta listagem, mostraremos apenas o código JavaScript. Você já sabe que ele precisa estar dentro das tags `<script></script>` do arquivo HTML, não é mesmo? ;)

1.6 Usando o console JavaScript

Todos os navegadores mais utilizados atualmente tem uma parte “secreta”, desconhecida da maioria dos usuários. Essa parte é chamada de Ferramentas de Desenvolvedor e pode ser acessada ao pressionar a tecla **F12**. Será aberto um painel, no lado direito ou inferior da tela, conforme mostrado na Figura 1.3.

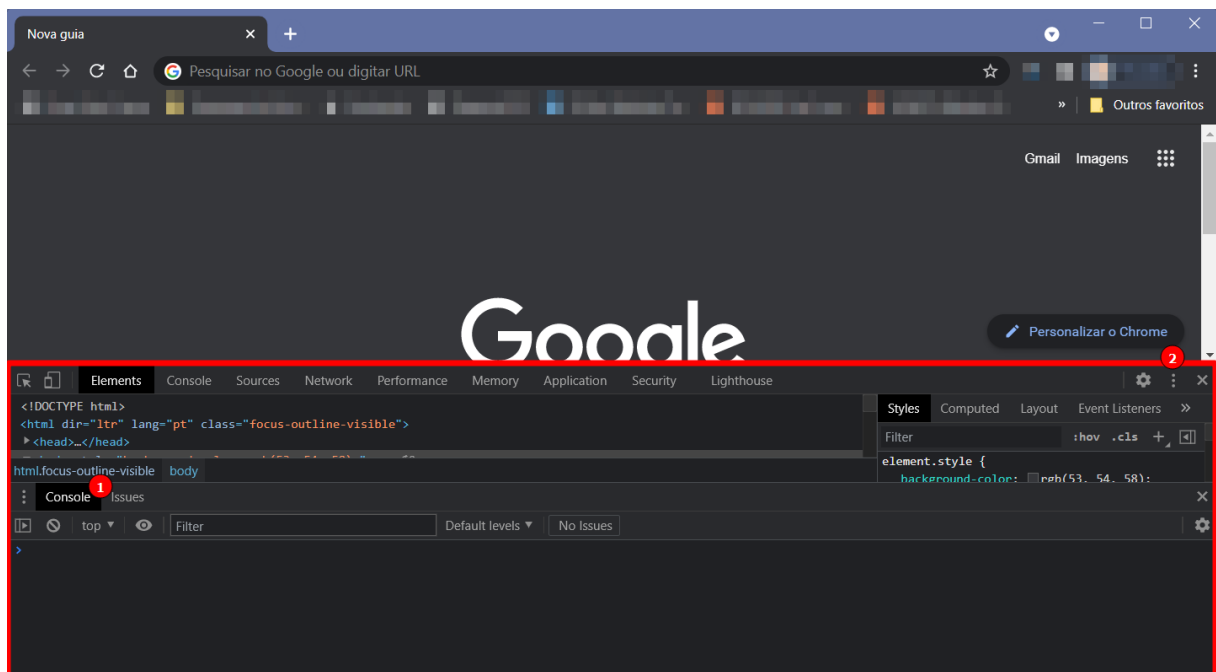


Figura 1.3: Navegador Web exibindo as Ferramentas de Desenvolvedor (no destaque). Note (1) a aba Console e (2) o menu de opções

DICA: usando o menu de opções (2), é possível mudar o posicionamento das Ferramentas de Desenvolvedor na tela.

Na aba Console, é possível digitar instruções JavaScript, incluindo operações aritméticas, e ver imediatamente seu resultado. Veja alguns exemplos na Figura 1.4.

No próximo capítulo, vamos aprender sobre variáveis e tipos de dados, e usaremos o console para fazer alguns testes.

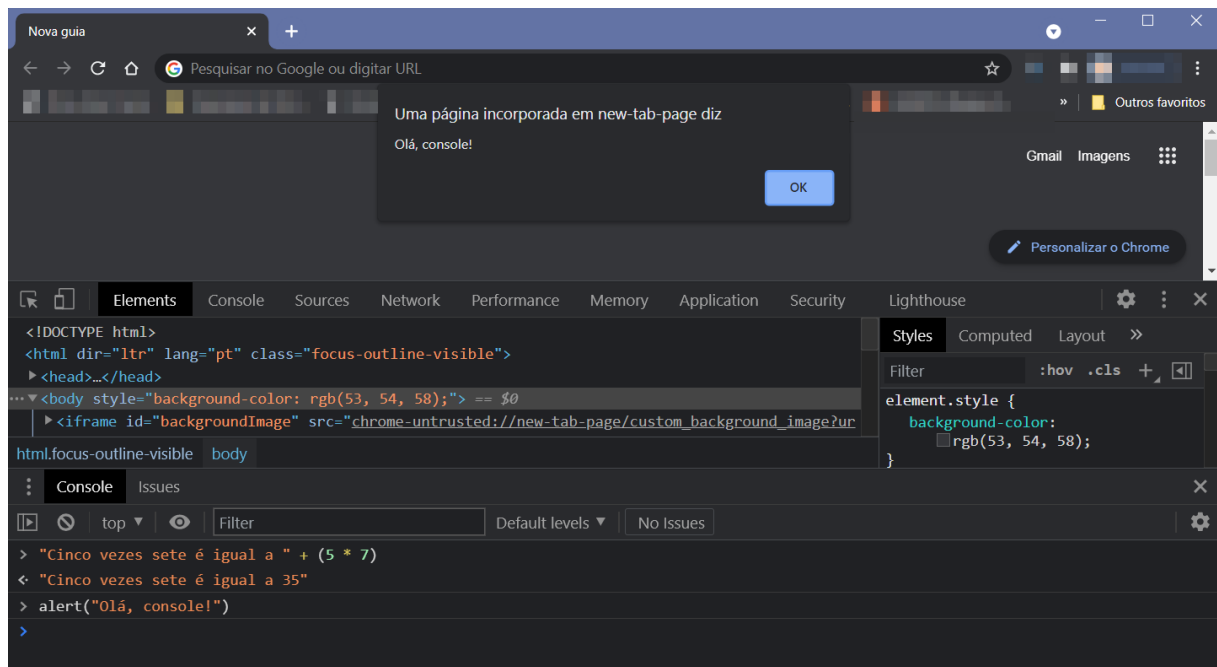


Figura 1.4: Resultado da execução de alguns comandos no console JavaScript do navegador Web

Capítulo 2

VARIÁVEIS E TIPOS DE DADOS

2.1 Declaração de variáveis

O Javascript é uma linguagem de tipagem dinâmica. Isso significa que os tipos de dados de suas variáveis não são determinados no momento em que são declaradas. Em vez disso, eles são deduzidos a partir dos **valores** atribuídos a elas.

Veja alguns exemplos na Listagem 2.1:

Listagem 2.1 Declarações de variáveis em JavaScript

```
1 let x
2 var preco
3 const meuNome = 'Fausto'
```

NOTE BEM: para declarar uma variável em JavaScript, basta uma das palavras-chave reservadas (`let` ou `var`) seguida do nome da variável, nada mais. `const` exige também que um valor seja atribuído à variável no momento da declaração.

Há três palavras-chave utilizadas para declarar variáveis em JavaScript:

- **let** : atualmente, é o método recomendado de criação de variáveis. Uma das vantagens de sua utilização é a impossibilidade de se declarar mais de uma variável com o mesmo nome, o que ajuda a evitar erros de lógica no código. Possui também outros benefícios e características que serão explicadas ao longo desta apostila. É uma adição relativamente recente à linguagem (foi introduzida na versão ES6, de 2015). Nesta apostila, seguindo as recomendações mais recentes, usaremos **let** para declarar as variáveis, a menos que o uso de outra palavra-chave seja justificado.
- **var** : é a palavra-chave originalmente disponível para a declaração de variáveis, desde a primeira versão do JavaScript. Seu uso apresenta alguns problemas, como a possibilidade

de redeclarar uma variável já existente, o que pode induzir a erros de lógica. Evite utilizá-la, até compreender completamente as consequências de seu emprego.

- **const** : em algumas situações, é necessário representar valores que não devem mais ser alterados posteriormente. São as chamadas **constantes**. Variáveis declaradas com **const** devem receber um valor quando declaradas e não aceitam que este valor seja modificado depois.

Várias variáveis podem ser declaradas simultaneamente (Listagem 2.2):

Listagem 2.2 Declarações múltiplas de variáveis

```
1 let quantidade, precoUnitario, precoTotal
2 let nome, email, telefone, celular
```

2.1.1 Nomeando variáveis

Em JavaScript, nomes de variáveis devem começar com uma letra; os caracteres `$` e `_` também são aceitos primeira posição. Dígitos (0 a 9) podem ser utilizados a partir da segunda posição.

Você também não pode usar para nomear variáveis nenhuma das **palavras reservadas** pelo JavaScript para o seu próprio uso.

É importante frisar que JavaScript é uma **linguagem sensível à diferença entre letras maiúsculas e minúsculas** (*case sensitive*, em inglês), isto é, ela trata letras maiúsculas e minúsculas como coisas diferentes. Assim, uma variável de nome `num` é diferente de outra variável de nome `NUM`, e ambas são diferentes de uma terceira variável nomeada como `Num`.

Apesar de as especificações da linguagem JavaScript assim permitirem, não é recomendável declarar variáveis que contenham caracteres acentuados.

Listagem 2.3 Exemplos de nomeação de variáveis

```
1 let x           // OK!
2 let primeiroNome // OK!
3 let 1nome      // INVÁLIDO: começa com um dígito
4 let $valor     // OK, mas pouco usual
5 let _num       // OK, mas pouco usual
6 let %resultado // Caractere inicial INVÁLIDO
7 let área      // OK, mas acentos não são recomendados
```

2.1.2 Convenções de nomeação de variáveis

Quando muitos desenvolvedores trabalham num mesmo projeto, é comum que surja, mais cedo ou mais tarde, alguma discórdia sobre a forma de nomear as variáveis.

Por isso, as comunidades de cada linguagem acabam adotando convenções, que não são regras definidas na própria linguagem, mas sim uma espécie de “combinado” entre seus membros.

A convenção mais comum entre os desenvolvedores JavaScript é a seguinte:

1. **Sempre iniciar** o nome das variáveis com uma **letra minúscula**; e
2. Se o nome da variável for composto por **mais de uma palavra**, é utilizada **inicial maiúscula a partir da segunda** palavra.

Observe os exemplos da Listagem 2.4:

Listagem 2.4 Uso da convenção ‘camel case’ na nomeação de variáveis

```
1 // Uma palavra, inicial minúscula
2 let area
3 // Duas palavras, a segunda com inicial maiúscula
4 let areaTerreno
5 // Três palavras, iniciais maiúsculas a partir da segunda
6 let areaTerrenoPadrao
```

CURIOSIDADE: esse tipo de convenção é chamado, em inglês, de *camel case* (*camel* significa “camelo”). O motivo é que as letras maiúsculas no meio do nome das variáveis acabam se parecendo com as corcovas do animal.

2.2 Atribuindo valores a variáveis

Para atribuir valor a uma variável, é usado o operador `=` em JavaScript. A Listagem 2.5 exemplifica os diferentes formas de atribuição.

Listagem 2.5 Exemplos de atribuição de valores a variáveis

```
1 let quantidade, valor
2
3 quantidade = 7
4 valor = 12.63
5
6 // Podemos atribuir um valor à variável quando a declaramos
7 let cargo = 'Gerente'
8
9 /* Podemos, inclusive, fazer várias declarações/atribuições
10  de uma só vez */
11 let marca = 'Volkswagen', modelo = 'Fusca', ano = 1969
```

A atribuição de valores pode ocorrer posteriormente à declaração da variável (linhas 3

e 4) ou acontecer ao mesmo tempo que esta (linhas 7 e 10). Neste último caso, dizemos que a variável, além de declarada, foi **inicializada**.

2.3 Tipos de dados

Vamos analisar em detalhes os **tipos de dados** disponíveis na linguagem JavaScript. Usaremos, como referência e exemplo, a Listagem 2.6.

Listagem 2.6 Exemplos de tipos de dados

```
1 let nome, sobrenome, naturalidade, idade, altura, peso, casado
2 let conjugue, ocupacao, filhos, nomeCompleto
3 nome = "Afrânio" // string
4 sobrenome = 'Azeredo' // string
5 naturalidade = `Morro Alto de Cima (MG)` // string
6 idade = 44 // number
7 altura = 1.77 // number
8 peso = undefined // undefined
9 casado = true // boolean
10 conjugue = { nome: 'Jeruza', sobrenome: 'Jordão' } // object
11 ocupacao = null // object
12 filhos = ['Zózimo', 'Zuleica'] // object
13 nomeCompleto = function(nome, sobrenome) { // function
14     return nome + " " + sobrenome
15 }
```

- **string** (linhas 3 a 5): representa uma sequência de caracteres, ou seja, um texto. Em JavaScript, *strings* podem ser delimitadas com aspas duplas ("), aspas simples (') ou acentos graves (`), muitas vezes chamados também de crases. Aspas simples e aspas duplas são totalmente equivalentes entre si, e a escolha por uma ou por outra acaba sendo decisão do programador. Já as *strings* delimitadas por acentos graves têm significado e funções especiais, que serão explicadas mais à frente.
- **number** (linhas 6 e 7): ao contrário de outras linguagens de programação, o JavaScript não faz distinção entre números inteiros e números com parte fracionária (também chamados de números de ponto flutuante), colocando-os todos sob um mesmo tipo. Para separar a parte inteira da parte fracionária, quando esta exista, é sempre usado o ponto (.), embora, na língua portuguesa, usemos a vírgula para esse fim. Existem várias outras formas de representar valores numéricos:
 - valores hexadecimais (base 16) podem ser representados usando-se o prefixo 0x . Por exemplo 0x1A representa o valor hexadecimal 1A (equivalente a 26 em decimal).
 - valores octais (base 8) são representados usando-se um 0 no início. **CUIDADO!** Para o JavaScript, 045 não é um 45 decimal com um inútil zero à esquerda, e sim o valor

- octal `45` (que, convertido em decimais, equivale a 37).
- Números muito grandes ou muito pequenos podem usar a chamada notação científica. Assim, `4e12` é o mesmo que `4` vezes `1` seguido de doze zeros, ou seja `4000000000000`.
 - Para números *realmente* grandes, foi introduzida na versão 2020 do EcmaScript mais um tipo de dados, o **bigint**. Você pode saber detalhes dessa novidade consultado a documentação da [Mozilla Developer Network](#).
 - **undefined** (linha 8): é um tipo especial, usado para indicar que uma variável não tem qualquer valor atribuído a ela. A propósito, toda variável declarada e que ainda não recebeu um valor é considerada **undefined**.
 - **boolean** (linha 9): como indicado pelo próprio nome, representa valores booleanos. Os únicos valores possíveis são **true** (verdadeiro) e **false** (falso), sempre escritos totalmente em minúsculas.
 - **object** (linhas 10 a 12): é o tipo de dados mais versátil da linguagem, usado, normalmente, para armazenar vários valores em uma única variável. Os vetores (linha 12) e objetos em sentido estrito (linha 10) fazem parte desta categoria. **null** é um valor especial utilizado para representar um objeto inexistente.
 - **function** (linhas 13 a 15): em JavaScript, funções podem ser atribuídas a variáveis. Essa característica é importante para usos de nível intermediário e avançado da linguagem.

2.3.1 Descobrindo o tipo do valor de uma variável

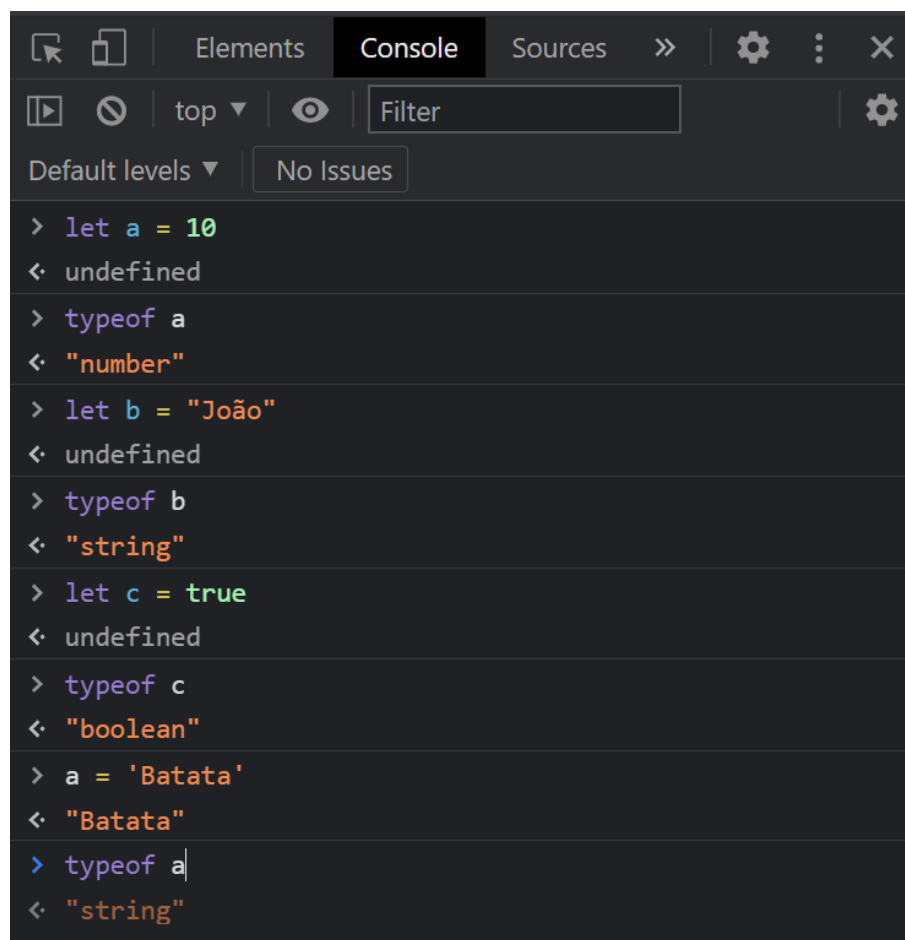
Em JavaScript, embora as variáveis não tenham um tipo determinado, os valores que elas abrigam têm. Para descobrir qual o tipo de dados do valor em um dado momento, usamos o operador **typeof**.

Observe os testes feitos no console JavaScript mostrados na Figura 2.1.

OBSERVAÇÃO: repare que as instruções que inicializam variáveis (começadas pela palavra **let**) retornaram **undefined**. Isso é normal, e significa que a instrução de inicialização da variável não retorna valor algum.

O operador **typeof** nos diz qual o tipo de dados que a variável possui **no momento do teste**. Veja o que aconteceu com a variável `a`. No primeiro teste, obtemos `"number"`, quando o valor que ela tinha era `10`. Mais à frente, depois que mudamos o valor de `a` para `"Batata"`, o teste retornou o tipo do novo valor, `"string"`. Com isso, não resta dúvida: **o que possui tipo é o valor da variável, não a variável em si mesma**.

Agora que sabemos como funcionam as variáveis e tipos de dados em JavaScript, precisamos saber o que fazer com eles. Bora aprender operadores?



The image shows a screenshot of a web browser's developer console, specifically the 'Console' tab. The console displays a series of JavaScript commands and their corresponding outputs. The commands are: `let a = 10`, `typeof a`, `let b = "João"`, `typeof b`, `let c = true`, `typeof c`, `a = 'Batata'`, and `typeof a`. The outputs are: `undefined`, `"number"`, `undefined`, `"string"`, `undefined`, `"boolean"`, `"Batata"`, and `"string"`. The console interface includes a toolbar with icons for running, pausing, and filtering, as well as a 'Filter' input field and a 'No Issues' button.

```
> let a = 10
< undefined

> typeof a
< "number"

> let b = "João"
< undefined

> typeof b
< "string"

> let c = true
< undefined

> typeof c
< "boolean"

> a = 'Batata'
< "Batata"

> typeof a
< "string"
```

Figura 2.1: Determinando o tipo do valor de algumas variáveis no console JavaScript

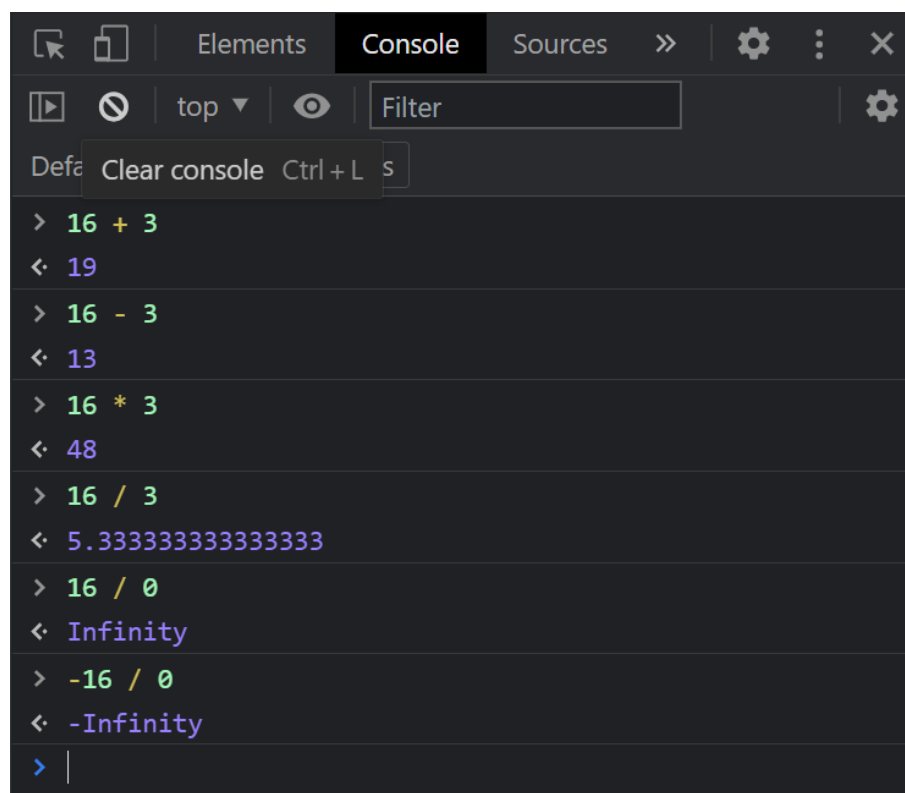
Capítulo 3

OPERADORES

3.1 Operadores aritméticos

Convivemos com os operadores aritméticos das quatro operações básicas desde o Ensino Fundamental. A maior parte das pessoas não terá problemas com os operadores das quatro operações aritméticas básicas. Quem usa computador há algum tempo sabe que o caractere `*` é usado para multiplicação e o caractere `/` é empregado na divisão.

É assim também no JavaScript, como ilustra a Figura 3.1.

A screenshot of a web browser's developer console, specifically the 'Console' tab. The console shows a series of JavaScript arithmetic operations being executed. Each operation is preceded by a green prompt character '>'. The results are displayed on the line following each operation, preceded by a purple prompt character '<'. The operations and their results are: 16 + 3 resulting in 19; 16 - 3 resulting in 13; 16 * 3 resulting in 48; 16 / 3 resulting in 5.333333333333333; 16 / 0 resulting in Infinity; and -16 / 0 resulting in -Infinity. The console interface includes a 'Filter' input field, a 'Clear console' button, and a keyboard shortcut 'Ctrl + L'. The background is dark, and the text is color-coded for readability.

```
> 16 + 3
< 19
> 16 - 3
< 13
> 16 * 3
< 48
> 16 / 3
< 5.333333333333333
> 16 / 0
< Infinity
> -16 / 0
< -Infinity
> |
```

Figura 3.1: Os quatro operadores aritméticos básicos em JavaScript

OBSERVAÇÃO: ao contrário da maioria das linguagens de programação (e até da sua calculadora), JavaScript não retorna erro quando há uma tentativa de divisão por zero. Em vez disso, ele retorna `Infinity` caso o dividendo seja positivo ou `-Infinity` se o dividendo for negativo. Existem razões matemáticas para tanto. Há uma bela discussão sobre isso [aqui](#).

Além deles, a linguagem conta com mais dois operadores aritméticos (Figura 3.2):

- `%`: é o operador de **resto da divisão**, também chamado de módulo da divisão. Calcula quanto sobra (o resto) da divisão de um número pelo outro.
- `**`: dois asteriscos consecutivos representam o operador de **potenciação**, que calcula o resultado do primeiro número elevado à potência do segundo.

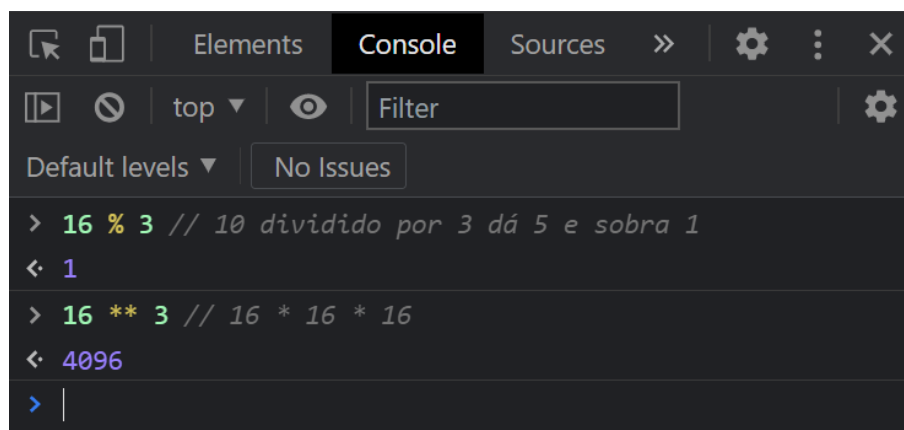


Figura 3.2: Os operadores de resto da divisão e de potenciação

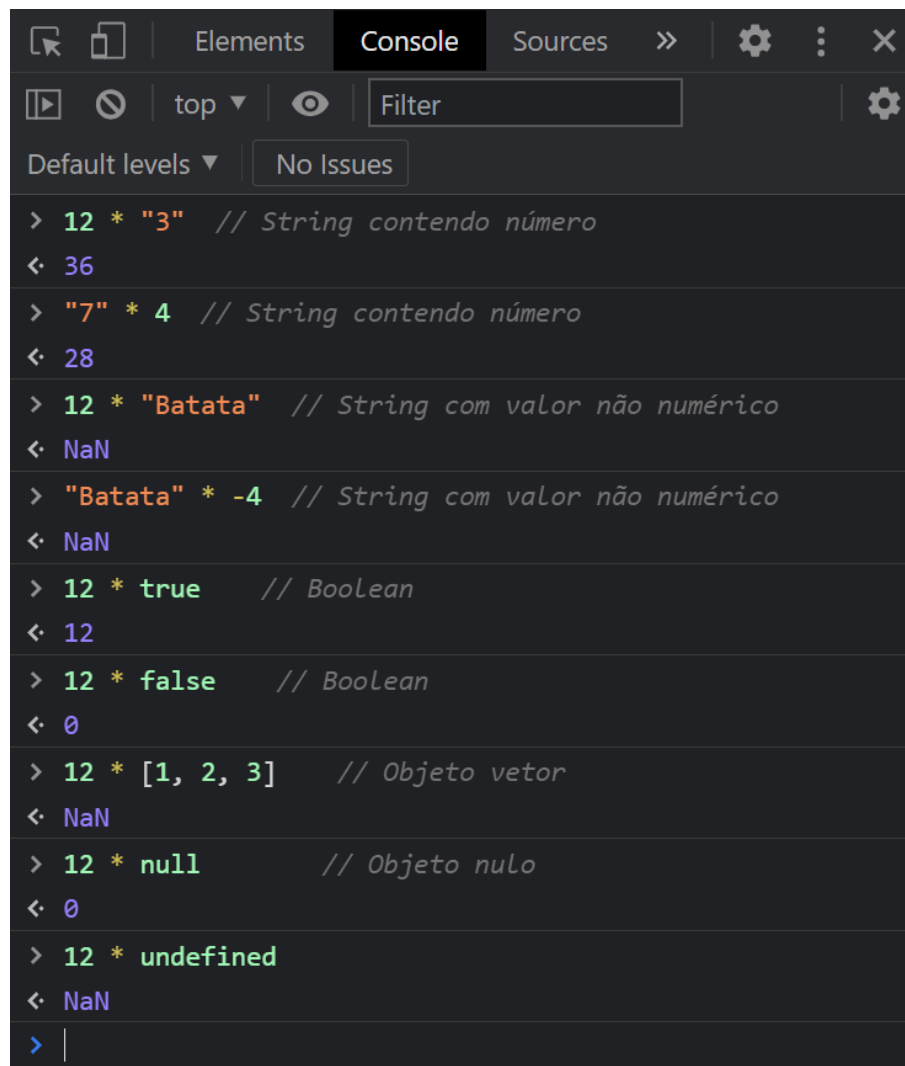
3.1.1 Quando os operandos não são números

Os operadores aritméticos funcionam como esperado quando seus operandos são números. Mas, e quando não são? Aí **depende**.

Vamos fazer alguns testes com o operador de multiplicação (Figura 3.3).

Sem dúvida, são resultados surpreendentes. Vamos analisar caso a caso:

1. Quando um dos operandos é *string*, temos duas possibilidades:
 - a) se o conteúdo da *string* equivaler a um valor numérico, o JavaScript efetua automaticamente a conversão de tipos e trata o valor da *string* como número, e temos o resultado da operação aritmética como se todos os operandos fossem numéricos.
 - b) se o conteúdo da *string* não contiver um valor que possa ser convertido para número, a operação aritmética é impossível e recebemos, para indicar esse fato, o valor especial `NaN`, que significa **Not a Number** (não é um número).
2. No caso de um dos operandos ser *boolean*, o valor `true` é tratado como se valesse `1` e o valor `false` é considerado como `0`, e a operação aritmética é feita normalmente.



The image shows a web browser's developer console with the 'Console' tab selected. The console displays the results of several JavaScript expressions involving multiplication with non-numeric operands. Each input line is followed by a comment in Portuguese and the resulting output.

```
> 12 * "3" // String contendo número
< 36

> "7" * 4 // String contendo número
< 28

> 12 * "Batata" // String com valor não numérico
< NaN

> "Batata" * -4 // String com valor não numérico
< NaN

> 12 * true // Boolean
< 12

> 12 * false // Boolean
< 0

> 12 * [1, 2, 3] // Objeto vetor
< NaN

> 12 * null // Objeto nulo
< 0

> 12 * undefined
< NaN

> |
```

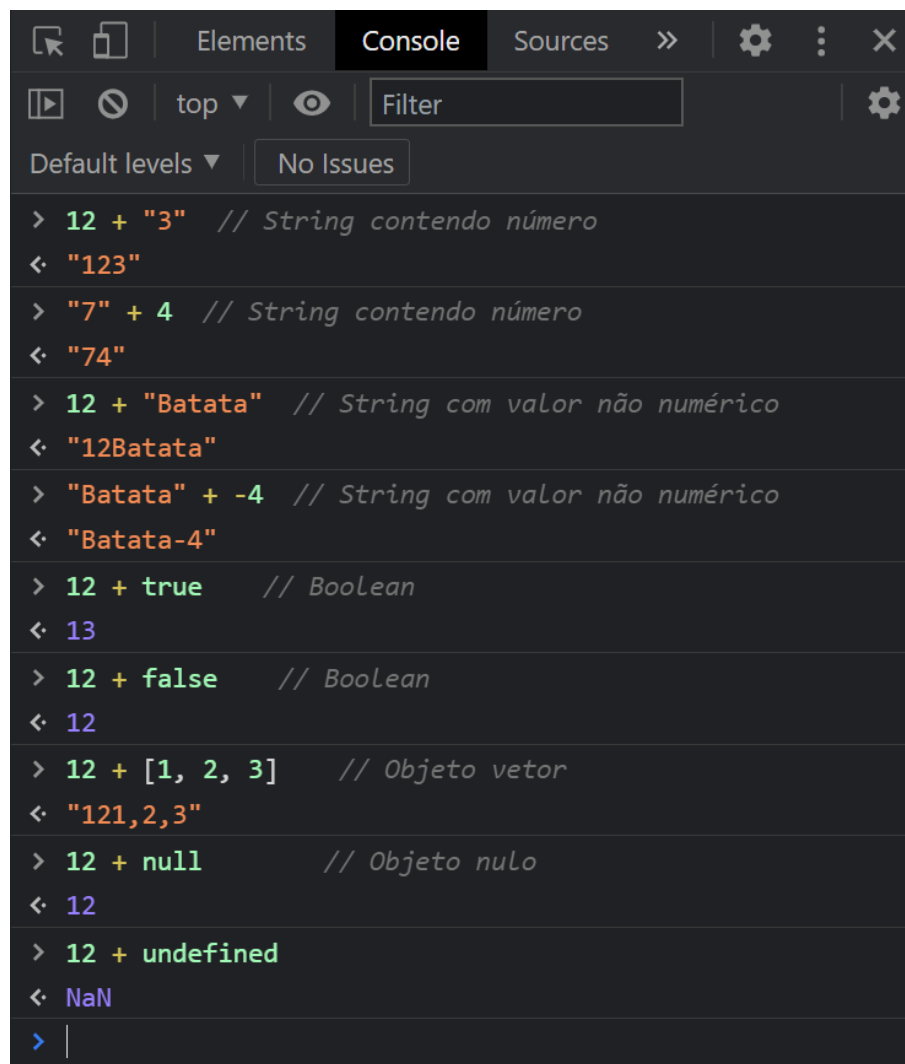
Figura 3.3: Operandos não numéricos

3. No caso de operandos do tipo *object*, não é possível efetuar a operação, portanto recebemos o resultado `NaN`. Uma exceção é o objeto nulo (`null`), que é tratado como `0` e possibilita a operação.

CONFIRA VOCÊ MESMO(A)

Resultados semelhantes são obtidos com os operadores de subtração (`-`), divisão (`/`), resto da divisão (`%`) e potenciação (`**`). Agora é a sua vez de abrir as Ferramentas do Desenvolvedor (tecle `F12` no navegador) e fazer seus próprios testes.

Se você for observador(a), vai notar que eu não mencionei o operador `+` na observação anterior. Isso foi proposital, pois ele tem um comportamento diferente dos demais (Figura 3.4).



```
> 12 + "3" // String contendo número
< "123"

> "7" + 4 // String contendo número
< "74"

> 12 + "Batata" // String com valor não numérico
< "12Batata"

> "Batata" + -4 // String com valor não numérico
< "Batata-4"

> 12 + true // Boolean
< 13

> 12 + false // Boolean
< 12

> 12 + [1, 2, 3] // Objeto vetor
< "121,2,3"

> 12 + null // Objeto nulo
< 12

> 12 + undefined
< NaN

> |
```

Figura 3.4: O operador '+' com operandos não numéricos

Esquisito? Que nada, vamos entender o porquê desses resultados.

Primeiramente, precisamos aprender que o **operador** `+` **tem duas funções** diferentes no JavaScript:

1. Se **todos os operandos forem numéricos**, ou puderem ser convertidos para números (ou tiverem valores equivalentes a números, como `true`, `false` e `null`), ele agirá como um **operador aritmético de adição**, como estamos acostumados.
2. No entanto, se **pelo menos um de seus operandos for uma *string***, ele terá a função de **operador de concatenação** de *strings*. Concatenar significa “emendar” uma *string* em outra. O JavaScript irá converter todos os demais operandos em *strings* e concatenará tudo em uma única *string* de resultado.

O operador `+` em conjunto com algum valor `undefined` sempre retornará `NaN`, e é um caso à parte.

IMPORTANTE

Comprender as duas funções do operador `+` (quando há e quando não há *strings* entre os operandos, ou seja, **concatenação** e **adição**, respectivamente) é **FUNDAMENTAL** para evitar frustrações futuras ao tratar com valores informados pelo usuário.

3.2 Operadores compostos de atribuição

Quem já estudou algoritmos sabe que uma das tarefas mais comuns em programação é a acumulação. Quando estamos somando uma lista de números por exemplo, iniciamos uma variável com o valor `0` e vamos **acumulando** o valor dos números, mais ou menos assim (Listagem 3.1):

Listagem 3.1 Exemplo de acumulação em variável

```
1 // Somando os valores 10, 20, 30 e 40
2 let soma = 0
3 soma = soma + 10
4 soma = soma + 20
5 soma = soma + 30
6 soma = soma + 40
7 alert(soma) // 100
```

Observe que, no processo de acumulação, a variável `soma` aparece **antes** e **depois** do sinal de atribuição (`=`). Para casos assim, o JavaScript dispõe de uma série de **operadores compostos de atribuição**, como o operador `+=`, que evitam a repetição do nome a variável. Usando-o, o código anterior fica assim (Listagem 3.2):

Veja alguns dos outros operadores compostos de atribuição na Tabela 3.3:

Listagem 3.2 Acumulação usando o operador '+='

```
1 // Somando os valores 10, 20, 30 e 40
2 let soma = 0
3 soma += 10
4 soma += 20
5 soma += 30
6 soma += 40
7 alert(soma) // 100
```

Tabela 3.3: Operadores de atribuição composta mais comuns

Nome	Operador	Exemplo de uso	Significado
Atribuição de subtração	<code>--</code>	<code>x -= 10</code>	<code>x = x - 10</code>
Atribuição de multiplicação	<code>*=</code>	<code>x *= 10</code>	<code>x = x * 10</code>
Atribuição de divisão	<code>/=</code>	<code>x /= 10</code>	<code>x = x / 10</code>
Atribuição de resto de divisão	<code>%=</code>	<code>x %= 10</code>	<code>x = x % 10</code>
Atribuição de potenciação	<code>**=</code>	<code>x **= 10</code>	<code>x = x ** 10</code>

3.3 Incremento e decremento

Sem dúvida, os operadores compostos de atribuição são muito práticos e deixam o código mais simples e limpo. Quer ainda mais praticidade? Outra tarefa comum em programação é aumentar ou diminuir o valor da variável numérica em uma unidade. Isso é chamado, respectivamente, de **incremento** e **decremento**.

Pois bem, o JavaScript possui operadores especializados para esses casos. Os operadores de incremento (`++`) e decremento (`--`) são **operadores unários**, isto é, agem sobre um único operando. Repare, na Listagem 3.3, que eles podem vir **antes** ou **depois** do seu operando?. Há uma diferença entre esses dois usos, que foge ao escopo desta apostila, mas, desde que esses operadores não estejam em uma expressão com outros operadores, o resultado prático é o mesmo.

Listagem 3.3 Uso dos operadores de incremento e decremento

```
1  let x = 10, y = 20
2
3  x++      // x = x + 1
4  alert(x) // 11
5
6  y--      // y = y - 1
7  alert(y) // 19
8
9  ++x      // x = x + 1
10 alert(x) // 12
11
12 --y      // y = y - 1
13 alert(y) // 18
```

3.4 Operadores lógicos

Quem já estudou algoritmos e lógica de programação sabe que existem os operadores **E**, **OU** e **NÃO**, que agem sobre valores booleanos, com resultado também booleano.

Em JavaScript:

- O operador lógico **E** é representado por **&&** (dois sinais de E comercial).
- O operador lógico **OU** é representado por **||** (duas barras verticais. Você encontra esse caractere na tecla à esquerda de **Z**, no seu teclado).
- O operador lógico **NÃO** é representado por **!** (um ponto de exclamação).

A Figura 3.5 exemplifica o uso dos operadores lógicos.

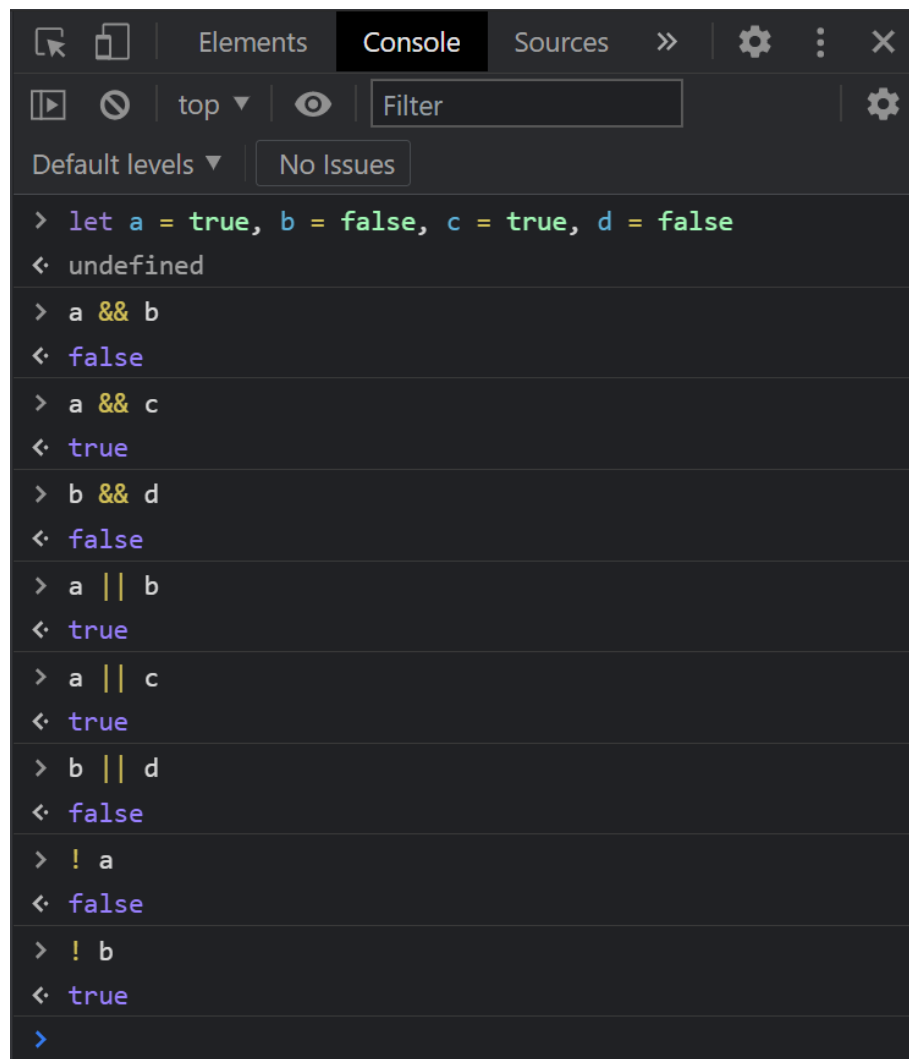
3.5 Operadores relacionais (ou de comparação)

Comparar significa colocar dois valores lado a lado e determinar se ambos são iguais ou se existe algum deles maior que o outro. O resultado de uma comparação é sempre um valor booleano (**true** ou **false**).

JavaScript possui uma série de operadores de comparação, alguns deles semelhantes aos usados em outras linguagens de programação:

- **>** (maior que)
- **<** (menor que)
- **>=** (maior que ou igual a)
- **<=** (menor que ou igual a)

No entanto, quando se trata de verificar igualdade, JavaScript possui **dois** operadores:



The image shows a screenshot of a web browser's developer console, specifically the 'Console' tab. The console displays a series of JavaScript commands and their corresponding outputs, demonstrating the behavior of logical operators. The commands and outputs are as follows:

```
> let a = true, b = false, c = true, d = false
< undefined

> a && b
< false

> a && c
< true

> b && d
< false

> a || b
< true

> a || c
< true

> b || d
< false

> !a
< false

> !b
< true

>
```

Figura 3.5: Operadores lógicos em JavaScript

- `==` operador de **igualdade**. Note que temos aqui **dois** sinais de igual.
- `===` operador de **igualdade estrita**. São **três** sinais de igual.

E por que isso?

Você deve se recordar, quando estudamos os operadores aritméticos, de que o JavaScript tenta converter *strings* que contêm valores numéricos em números, no contexto de uma operação aritmética. Por causa disso, a linguagem considera isso como verdadeiro:

```
3 == "3"    // true
```

Ou seja, não apenas em expressões aritméticas, mas também em expressões de comparação, a conversão ocorre. Por isso, JavaScript considera ambos os valores iguais.

No entanto, nem sempre é esse o comportamento que desejamos. Muitas vezes, não queremos que *strings* e números com o mesmo “conteúdo” sejam considerados iguais. Nesses casos, devemos usar o **operador de igualdade restrita** para efetuar a verificação.

```
3 === "3"   // false
```

O operador de igualdade estrita verifica não apenas os valores, mas também os **tipos** que estão sendo comparados, retornando `true` apenas quando ambos os lados são **absolutamente iguais**.

```
"3" === "3"    // true  
3 === 3        // true
```

CUIDADO!

Ao efetuar comparações de igualdade em JavaScript, você deve usar **PELO MENOS dois sinais de igual (=)** na sua expressão. Um único sinal de igual corresponde ao operador de atribuição, ou seja, você corre o risco de mudar o valor de alguma variável em vez de comparar.

Obviamente, se temos dois operadores de igualdade, temos também dois operadores de diferença:

- `!=` operador de *diferença*, composto por um ponto de exclamação e **um** sinal de igual.
- `!==` operador de *diferença estrita*, composto por um ponto de exclamação e **dois** sinais de igual.

O mesmo princípio da conversão (ou não) de **strings** numéricas em números, para efeitos de comparação, também se aplica aos operadores de diferença (Listagem 3.4):

Note que, para o operador de diferença comum (`!=`) dois valores que têm o mesmo conteúdo, mas tipos diferentes (número e *string*), não são diferentes entre si (linha 1).

Listagem 3.4 Exemplos de uso dos operadores de diferença

```
1 3 != "3"      // false
2 3 !== "3"     // true
3 3 != 4        // true
4 3 != "4"     // true
5 3 !== 4       // true
```

3.6 Precedência

Observe a seguinte expressão aritmética:

```
3 + 6 * 4
```

Qual o resultado? 36? 27?

Aprendemos no Ensino Fundamental que as operações de multiplicação e divisão devem ser resolvidas antes das operações de adição e subtração. Portanto, precisamos resolver primeiro `6 * 4` e somar o resultado obtido por `3`. Considerando isso, a resposta correta é 27.

Dizemos que os operadores de multiplicação e divisão têm **precedência** sobre os operadores de adição e subtração. Caso quiséssemos que a soma fosse resolvida primeiro, deveríamos usar parênteses:

```
(3 + 6) * 4
```

A questão da precedência de operadores surge também com os operadores lógicos. Qual seria, por exemplo, o resultado da expressão a seguir?

```
let a = true, b = true, c = true
a || b && !c    // true
```

Dentre os operadores lógicos, o `!` tem a maior precedência. Assim, ele será resolvido primeiro, invertendo o valor de `c` (de `true` para `false`).

Em seguida na ordem de precedência, vem o operador `&&`. Devemos, portanto, resolver `b && c`. Com um valor verdadeiro e outro false, o resultado é `false`.

Por fim, resolvemos o operador `||`. Analisando `a || false`, chegando a `true` como resultado final.

Teríamos outro resultado caso o operador `||` fosse executado antes do `&&`:

```
let a = true, b = true, c = true
(a || b) && !c    // false
```

Sim, eu sei que não é fácil decorar todas as ordens de precedência, e nem é necessário

fazer isso. Basta consultar a tabela de precedência (Tabela 3.5). Quanto mais acima o operador estiver, com maior antecedência ele será resolvido em uma expressão.

Tabela 3.5: Tabela de precedência de operadores

Operador	Significado	Observação
!	NÃO lógico	Precedência mais alta
* / %	Multiplicação e divisão	
+ -	Adição e subtração	
< <= > >=	Operadores relacionais	
== != === !==	Igualdade e diferença	
&&	E lógico	
	OU lógico	
= += -= *= /= %= **=	Atribuição	
		Precedência mais baixa

Até agora, aprendemos alguns dos conceitos mais básicos da linguagem JavaScript. Pode parecer que ainda não construímos nada, mas posso garantir que o aprendizado vale a pena.

No próximo capítulo, a ação começa de verdade. Vamos passar a interagir com o usuário, por meio de entradas e saídas.

Capítulo 4

ENTRADA E SAÍDA

4.1 Exibindo informações

A exibição de informações para o usuário é uma das atividades mais comuns em programação. Afinal, o usuário está usando o computador com um propósito e quer ter *feedback* de suas ações.

Vejamos os diferentes métodos que o JavaScript dispõe para essa finalidade.

4.1.1 `alert()`

A forma mais simples e comum de exibir algo para o usuário é pelo uso de `alert()`. Já o usamos nos capítulos anteriores, de modo que ele não constitui exatamente uma novidade.

Vamos rever seu funcionamento:

```
alert('Espero que você esteja gostando de estudar JavaScript!')
```

`alert()` é uma função, que leva como parâmetro a mensagem a ser exibida. Essa mensagem, normalmente, é uma *string*, ou uma variável contendo uma *string*. Caso o parâmetro não seja *string*, será convertido para esse tipo de dado.

A aparência da janela em que aparece a mensagem é definida pelo navegador, e, infelizmente, não temos controle sobre esse ponto. Essa aparência, inclusive, pode variar bastante de um navegador para outro (Figura 4.1).

A função `alert()` também pode aparecer escrita, às vezes, como `window.alert()`. Trata-se da mesma função, com as mesmas funcionalidades. É como se ela tivesse um nome longo, mais formal, e um nome curto, mais prático.

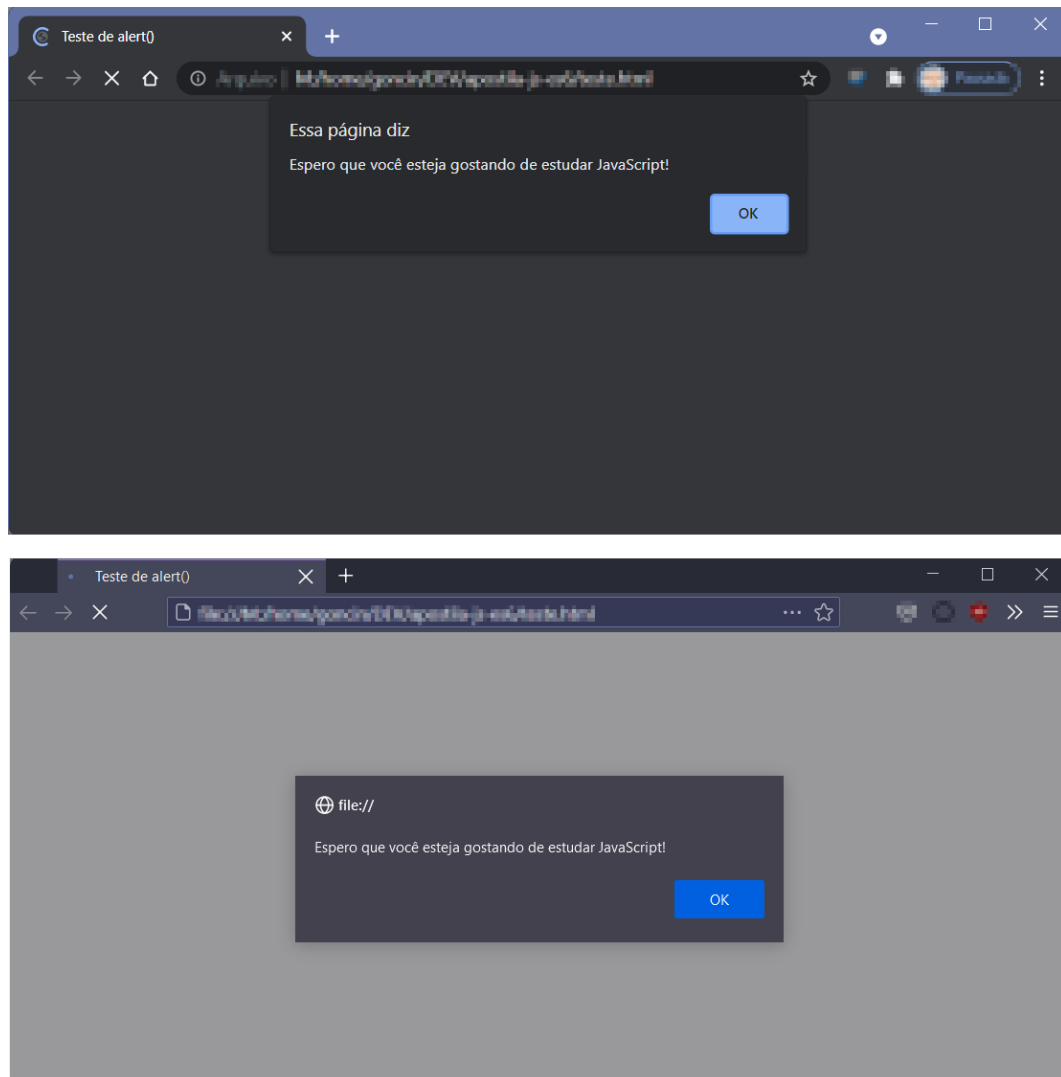


Figura 4.1: Mensagem de `alert()` sendo exibida no Google Chrome (em cima) e no Mozilla Firefox (embaixo)

4.1.2 `document.write()`

Outra forma de mostrar uma informação textual é escrevendo na seção `<body>` do documento HTML usando `document.write()`. **O resultado será exibido no sempre no início da página**, mesmo que esta já tenha outros conteúdos.

```
document.write('JavaScript é bem legal!')
```

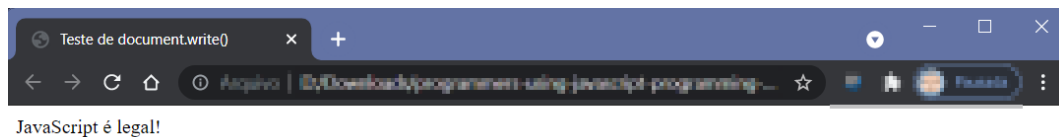


Figura 4.2: Texto escrito com `document.write()`

Uma peculiaridade de `document.write()` é que ela **não efetua quebras automáticas de linha**. Ou seja, se emitirmos duas chamadas consecutivas à função, o resultado será exibido em uma única linha (Figura 4.3).

```
document.write('JavaScript é bem legal!')  
document.write('Quero aprender mais sobre a linguagem.')
```

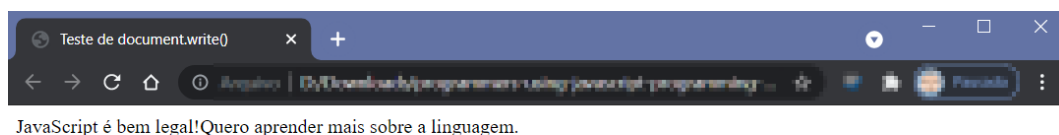


Figura 4.3: Resultado de duas chamadas seguidas a `document.write()`

Para resolver esse problema, precisamos escrever a tag HTML `
` (de *break row*, quebra de linha, em inglês) no final da primeira mensagem, e usá-la sempre que for necessário pular a linha (Figura 4.4).

```
document.write('JavaScript é bem legal!<br>') // Note o <br> aqui  
document.write('Quero aprender mais sobre a linguagem.')
```



Figura 4.4: Resultado do uso de `
` entre chamadas a `document.write()`

4.1.3 `console.log()`

Este é o método de exibição de informações mais curioso, pois o seu destinatário final não é o usuário, e sim o próprio desenvolvedor. O que `console.log()` faz não aparece em lugar algum para quem está vendo a página, **exceto** se a pessoa teclar `F12` e acessar a aba Console das Ferramentas de Desenvolvedor.

Por isso, `console.log()` é muito usado para testes e depuração, como mostra o exemplo da Listagem 4.1.

Listagem 4.1 Exemplo de uso de `console.log()`

```
1 let a = 7, b = 3, c
2 c = 7 ** 3
3 console.log('Valor de c: ' + c)
```

NOTE BEM: na linha 3, o sinal de `+` não representa uma soma, e sim uma concatenação, já que o primeiro operando é uma *string*.

Confira o resultado na Figura 4.5.

4.2 Coletando informações

Um dos motivos da criação do JavaScript foi permitir que as páginas Web não apenas exibissem informações aos usuários, como também fossem capazes de coletar informações que eles forneçam.

Para isso, a linguagem dispõe de dois métodos, os quais iremos aprender na sequência.

4.2.1 `prompt()`

A função `prompt()` exibe uma caixa de diálogo, com uma mensagem de instrução e um campo de texto onde o usuário pode digitar o que se pede. O valor informado pelo usuário pode

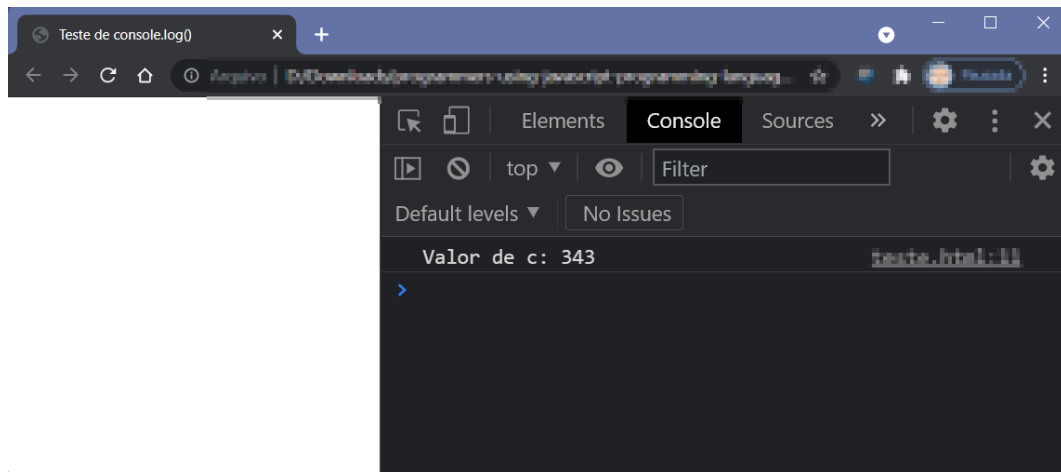


Figura 4.5: Resultado do uso de console.log()

ser coletado em uma variável para uso posterior no código.

Veja um exemplo de uso:

```
let nome  
nome = prompt('Informe o seu nome:')
```

O resultado desse código é semelhante ao retratado na Figura 4.6.

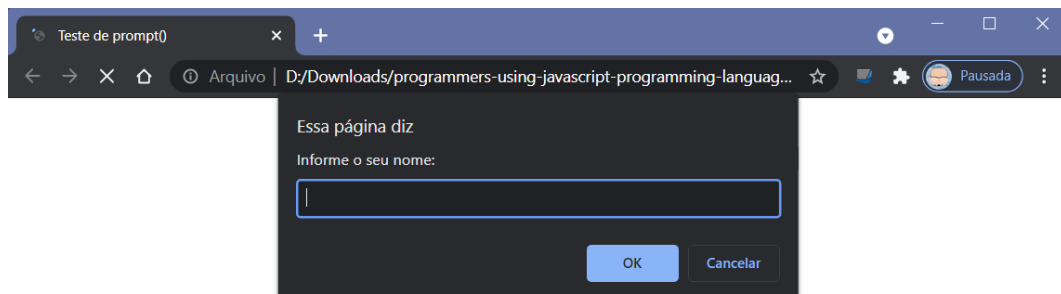


Figura 4.6: A função prompt() sendo executada

Caso o usuário digite a informação pedida e clique sobre o botão OK, a variável `nome` receberá o que for informado.

OBSERVAÇÃO: a função `prompt()` também pode ser escrita como `window.prompt()`.

`prompt()` é muito utilizada em um ciclo que envolve:

1. a coleta da informação (via `prompt()`);

2. o processamento da informação coletada; e
3. a exibição do resultado do processamento, com `alert()`.

Experimente executar o código da Listagem 4.2:

Listagem 4.2 Exemplo de uso combinado de `prompt()` e `alert()`

```
1 let nome, mensagem
2 nome = console.log('Informe o seu nome:')
3 mensagem = 'Olá, ' + nome + '! Tudo bem?'
4 alert(mensagem)
```

4.2.2 `confirm()`

Outra forma de solicitar a interação do usuário é usando `confirm()`. Será exibida uma caixa de diálogo com uma mensagem (normalmente uma pergunta) e os botões `OK` e `Cancelar`. O resultado pode ser coletado em uma variável, que terá um dos seguintes valores:

- `true`, se o usuário tiver pressionado o botão `OK` (ou teclar `Enter`);
- `false`, caso o usuário tiver pressionado o botão `Cancelar` (ou teclar `Esc`).

Observe o exemplo da Listagem 4.3:

Listagem 4.3 Exemplo de uso de `confirm()`

```
1 let resposta
2 resposta = confirm('Deseja realmente continuar?')
3 alert('Sua resposta foi: ' + resposta)
```

OBSERVAÇÃO: a função `confirm()` também pode ser escrita como `window.confirm()`.

Confira a aparência da caixa de diálogo do `confirm` na Figura 4.7.

Nossos programas em JavaScript estão ganhando forma. Já sabemos como interagir com o usuário. Nos próximos capítulos, vamos aprender a **controlar** as informações que estão ao nosso dispor.

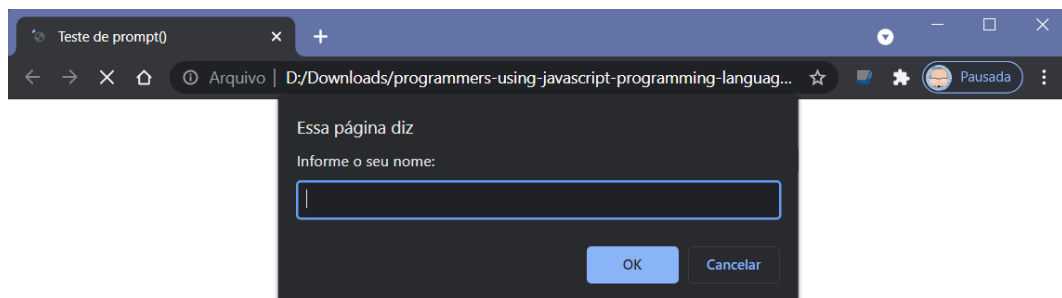


Figura 4.7: A função `confirm()` sendo executada

Capítulo 5

ESTRUTURAS CONDICIONAIS

Instruções em programas de computador são normalmente executadas em sequência, uma após as outras. No entanto, existem situações nas quais existe mais de uma possibilidade, e precisamos fazer um **desvio condicional**: seguir por um ou por outro caminho de acordo com o resultado de uma decisão. Para fazer isso, precisamos das estruturas condicionais.

5.1 A estrutura `if`

A estrutura condicional mais simples é `if`. Seu formato é o seguinte:

```
if(condição) {  
    Instrução ou instruções a serem executadas  
}
```

Algumas notas:

1. `if` é uma palavra reservada do JavaScript, e, como todas elas, é escrita **em minúsculas**.
2. Os **parênteses** em torno da condição **são obrigatórios**.
3. Condição é um valor booleano, ou que se resolve em um valor booleano (`true` ou `false`).
4. Depois do parêntese de fechamento da condição, segue-se uma chave de abertura (`{`). Ele indica o início do bloco de instruções que será executado **apenas se a condição for verdadeira**.
5. Uma chave de fechamento (`}`) marca o final do bloco de instruções.

OBSERVAÇÃO: as chaves são opcionais quando há uma única linha de código atrelada à estrutura `if`. No entanto, até que você esteja bem familiarizado(a) com a sintaxe do JavaScript, recomendo que sempre use as chaves.

Vejamos um exemplo prático na Listagem 5.1.

Listagem 5.1 Exemplo (1) da estrutura condicional 'if'

```
1 let resposta
2
3 resposta = confirm('Deseja realmente continuar?')
4
5 if(resposta) {
6     alert('Você decidiu continuar.')
7 }
```

Recorde que `confirm()` retorna um valor booleano, de acordo com a resposta dada pelo usuário na caixa de diálogo. O `alert()` da linha 6 será executado somente se a resposta tiver sido `OK`.

Mais um exemplo (Listagem 5.2):

Listagem 5.2 Exemplo (2) da estrutura condicional 'if'

```
1 let nota1 = 7, nota2 = 8, media
2
3 media = (nota1 + nota2) / 2
4
5 if(media >= 6) {
6     document.write('Média final: ' + media + ', maior ou igual a 6.')
7     document.write('Situação: APROVADO')
8 }
```

Neste outro exemplo, a condição é uma expressão que se resolve em um valor booleano. Afinal, ou a média é maior ou igual a 6 (`true`) ou é menor que 6 (`false`). As instruções das linhas 6 e 7 serão executadas apenas no primeiro caso.

5.1.1 `if..else`

Se você for observador(a), deve ter notado que tínhamos **duas** possibilidades em cada uma das listagens anteriores. E pode estar se perguntado: que mensagem o usuário verá se decidir não continuar ou se a média tiver ficado abaixo de 6?

Na verdade, nenhuma mensagem será exibida naqueles casos, pois tratamos apenas uma das possibilidades: aquela em que a condição se resolve em `true`. Para lidar também com as possibilidades falsas, precisamos acrescentar o bloco `else` abaixo do `if`.

```
if(condição) {
    Instruções a serem executadas se a condição for verdadeira
}
else {
    Instruções a serem executadas se a condição for falsa
}
```



```
}
```

Quero chamar a sua atenção para o fato de que tanto o **if** quanto o **else** têm seu próprio par de chaves. Ou seja, antes de escrever o **else**, você deve fechar a chave do bloco do **if**.

Vejamos como fica o primeiro exemplo anterior, agora com **else** (Listagem 5.3):

Listagem 5.3 Exemplo (1) da estrutura condicional 'if' com 'else'

```
1 let resposta
2
3 resposta = confirm('Deseja realmente continuar?')
4
5 if(resposta) {
6     alert('Você decidiu continuar.')
7 }
8 else {
9     alert('Pedi para parar, parou.')
10 }
```

E também o segundo exemplo (Listagem 5.4):

Listagem 5.4 Exemplo (2) da estrutura condicional 'if' com 'else'

```
1 let nota1 = 7, nota2 = 8, media
2
3 media = (nota1 + nota2) / 2
4
5 if(media >= 6) {
6     document.write('Média final: ' + media + ', maior ou igual a 6.')
7     document.write('Situação: APROVADO')
8 }
9 else {
10    document.write('Média final: ' + media + ', menor que 6.')
11    document.write('Situação: REPROVADO')
12 }
```

PROBLEMA REAL: conversão de tipos

Agora que já aprendemos a estrutura condicional básica, temos condições de resolver alguns problemas práticos. Tente executar o *script* da Listagem 5.5.

Se você tiver feito o teste, talvez esteja decepcionado(a) agora. Isso porque, não importa quais números tenham sido informados, não veremos uma soma – e sim a concatenação dos dois valores informados.

São dois os motivos desse resultado:

Listagem 5.5 'Script' simples para somar dois números inteiros

```
1 let num1, num2, soma
2
3 num1 = prompt('Informe um número inteiro:')
4 num2 = prompt('Informe outro número inteiro:')
5
6 soma = num1 + num2
7
8 document.write('A soma dos dois inteiros informados é ' + soma)
```

1. A função `prompt()` sempre vai retornar seu valor como *string*, mesmo que o usuário tenha informado valores numéricos.
2. O operador `+`, como sabemos, concatena seus operandos caso pelo menos um deles seja *string*.

Portanto, precisamos converter as *strings* informadas em números antes de efetuar a soma. A forma mais comum de fazer essa conversão é usando a função `parseInt()`. Podemos usá-la em conjunto com a função `prompt()`, como mostrado na Listagem 5.6.

Listagem 5.6 'Script' simples para somar dois números inteiros, usando 'parseInt()'

```
1 let num1, num2, soma
2
3 // Note o uso da função parseInt() nas duas linhas a seguir
4 num1 = parseInt(prompt('Informe um número inteiro:'))
5 num2 = parseInt(prompt('Informe outro número inteiro:'))
6
7 soma = num1 + num2
8
9 document.write('A soma dos dois inteiros informados é ' + soma)
```

ATENÇÃO ao nome da função. `parseInt` é escrita com inicial **minúscula**, e o **I** de **Int** é **maiúsculo**. JavaScript é uma linguagem sensível à diferença entre letras maiúsculas e minúsculas, você se lembra?

Essa nova versão funcionará corretamente **caso o usuário informe *strings* que possam ser convertidas para números**. Mas, o que acontecerá se o usuário informar, por exemplo, `batata` ou `gsfdjghfdljgh`? A função `parseInt()` **não conseguirá efetuar a conversão, e retornará o valor especial `NaN` (Not a Number)**.

Podemos detectar se a tentativa de conversão falhou usando uma outra função, `isNaN()`, que, como seu próprio nome indica, retorna `true` se o valor testado for igual a `NaN`. Com o uso dessa função e de uma estrutura condicional `if..else`, chegamos à versão definitiva e robusta do nosso *script* de soma de números inteiros (Listagem 5.7).

Listagem 5.7 'Script' simples para somar dois números inteiros, usando 'parseInt()' e 'isNaN()'

```
1 let num1, num2, soma
2
3 num1 = parseInt(prompt('Informe um número inteiro:'))
4 num2 = parseInt(prompt('Informe outro número inteiro:'))
5
6 // Se a conversão de num1 OU se a conversão de num2 tiver falhado:
7 if(isNaN(num1) || isNaN(num2)) {
8     alert('Algum dos valores informados não é um número!')
9 }
10 // Se conversão tiver sido bem sucedida
11 else {
12     soma = num1 + num2
13     document.write('A soma dos dois inteiros informados é ' + soma)
14 }
```

ATENTE-SE- ao uso de letras maiúsculas e minúsculas no nome da função `isNaN()`.

5.1.2 if..else if..else

Há casos em que apenas duas saídas em uma estrutura condicional não são suficientes. Suponha que precisemos converter a nota de um aluno, um valor numérico, em um conceito alfabético, segundo a Tabela 5.1.

Tabela 5.1: Tabela de conversão entre notas e conceitos

Faixa de notas	Conceito
9 a 10	A
7 a 8,9	B
5 a 6,9	C
3 a 4,9	D
0 a 2,9	E

Suponha, ainda, que é o usuário quem irá digitar a nota a ser convertida para conceito (usando `prompt()`), e, obviamente, ele tem a possibilidade de digitar algo que não é um valor numérico ou mesmo fora dos limites entre 0 e 10.

Logo, temos nada menos do que seis possibilidades, uma para cada conceito e mais uma para o caso de nota inválida. Para lidar com essa situação, usamos a estrutura `if..else if..else`, como demonstra a Listagem 5.8.

Listagem 5.8 Exemplo de uso da estrutura condicional 'if..else if..else'

```
1  let nota, conceito
2
3  /* Number() efetua conversão de string para números,
4     aceitando valores fracionários */
5  nota = Number('Informe a nota (entre 0 e 10)')
6
7  if(nota >= 9) {                // Possibilidade nº 1
8      conceito = 'A'
9  }
10 else if(nota >= 7) {           // Possibilidade nº 2
11     conceito = 'B'
12 }
13 else if(nota >= 5) {           // Possibilidade nº 3
14     conceito = 'C'
15 }
16 else if(nota >= 3) {           // Possibilidade nº 4
17     conceito = 'D'
18 }
19 else if(conceito >= 0) {       // Possibilidade nº 5
20     conceito = 'E'
21 }
22 // Nota não numérica ou fora da faixa entre 0 e 10
23 else {                          // Possibilidade nº 6
24     alert('ERRO: nota inválida')
25 }
26
27 document.write('Conceito: ' + conceito)
```

Vamos analisar os principais pontos dessa listagem.

- Na linha 5, usamos a função `Number()` (note o `N` maiúsculo) para fazer a conversão do valor que usuário digitar no `prompt()` para número. Ela trabalha de modo semelhante à `parseInt()`, mas aceita valores fracionários, como é o caso das notas.

Existe também a função `parseFloat()`, que produz o mesmo efeito que `Number()`.

- Nas linhas 10, 13, 16 e 19, encadeamos um `else` imediatamente a um novo `if`. Note que, ao contrário do que acontece em outras linguagens de programação, `else if` são **duas palavras separadas**.
- A última possibilidade (linha 23) conta apenas com um `else`, recaindo sobre ela qualquer valor não manipulado pelas possibilidades anteriores.

5.2 A estrutura `switch..case`

A estrutura `if..else if..else` nos oferece uma forma de lidar com várias possibilidades. Observe este outro *script*, que recebe um número entre 1 e 7 e retorna o dia da semana correspondente (Listagem 5.9).

Nesse *script*, temos algumas características bastante peculiares. Vejamos:

- **todos os testes feitos foram de igualdade**; e
- **a mesma variável foi testada em todos os casos** (a variável `diaNum`).

Ou seja, em cada `if` ou `else if`, escrevemos o nome da mesma variável e vamos testando-a contra diferentes valores.

Uma outra de testar uma única variável contra uma série de valores diferentes é usando a estrutura `switch..case`. Uma primeira tentativa de uso dessa estrutura resultaria na Listagem 5.10.

Listagem 5.9 Outro exemplo de uso da estrutura condicional 'if..else if..else'

```
1  let diaNum
2
3  diaNum = parseInt(prompt('Informe o dia da semana (entre 1 e 7):'))
4
5  if(diaNum === 1) {
6      alert('domingo')
7  }
8  else if(diaNum === 2) {
9      alert('segunda-feira')
10 }
11 else if(diaNum === 3) {
12     alert('terça-feira')
13 }
14 else if(diaNum === 4) {
15     alert('quarta-feira')
16 }
17 else if(diaNum === 5) {
18     alert('quinta-feira')
19 }
20 else if(diaNum === 6) {
21     alert('sexta-feira')
22 }
23 else if(diaNum === 7) {
24     alert('sábado')
25 }
26 else {
27     alert('ERRO: dia inválido')
28 }
```

Listagem 5.10 A estrutura 'switch..case', ainda sem 'breaks'

```
1  let diaNum
2
3  diaNum = parseInt(prompt('Informe o dia da semana (entre 1 e 7):'))
4
5  switch(diaNum) {
6      case 1:
7          alert('domingo')
8      case 2:
9          alert('segunda-feira')
10     case 3:
11         alert('terça-feira')
12     case 4:
13         alert('quarta-feira')
14     case 5:
15         alert('quinta-feira')
16     case 6:
17         alert('sexta-feira')
18     case 7:
19         alert('sábado')
20     default:
21         alert('ERRO: dia inválido')
22 }
```

Observemos os detalhes.

1. A estrutura `switch..case` inicia-se com a palavra `switch`, seguida do nome da variável a ser testada entre parênteses e uma chave de abertura (linha 5). Essa chave só será fechada no final da estrutura (no nosso *script*, na linha 22).
2. Há um `case` para cada valor a ser testado, seguido de dois pontos (linhas 6, 8, 10, 12, 14, 16 e 18).
3. Por fim, temos o caso `default`, que funciona como o “else” do `switch..case`. Ou seja, se a execução não entrar em nenhum `case`, cairá ali.

No entanto, ao executar esse *script* como apresentado, veremos um resultado estranho. Se o usuário informar o número 4, por exemplo, verá não apenas a mensagem “quarta-feira”, mas todas as que se seguem, inclusive a mensagem de erro do `default`.

Fique tranquilo(a), você não fez nada de errado. Mas é preciso entender um pouco melhor como funciona a estrutura `switch..case`.

Cada `case` é um ponto de entrada dentro da estrutura. Quando a execução encontra um `case` compatível, ela entra no `switch..case` naquele ponto e prossegue executando linha a linha até o término da estrutura. Pode parecer estranho, mas os projetistas da linguagem a conceberam para se comportar exatamente assim. E isso tem suas vantagens, como veremos mais adiante.

Mas... como fazer com que a estrutura execute apenas o código associado ao `case` do valor informado?

5.2.1 `break`

O que precisamos fazer é interromper o fluxo da execução uma vez que o código associado ao `case` tenha sido processado. Isso é feito colocando-se a palavra-chave `break` ao final de cada `case`, como mostrado na Listagem 5.11.

Note a instrução `break` nas linhas 8, 11, 14, 17, 20, 23 e 26. Ela faz com que a execução, uma vez que a encontre, interrompa o processamento do `switch..case` e prossiga na primeira linha após a chave de fechamento da estrutura.

O caso `default` não precisa de `break` porque, estando por último, o processamento da estrutura termina logo após, de qualquer forma.

5.2.2 `case` s vazios

Imaginemos uma situação em que o usuário tenha que escolher entre três alternativas, digamos, A, B e C. Podemos usar `prompt()` para obter a escolha, mas esta pode ser informada tanto em letras maiúsculas quanto em letras minúsculas. Ou seja, tanto `A` quando `a` representam a mesma opção.

Listagem 5.11 A estrutura 'switch..case', agora com 'breaks'

```
1 let diaNum
2
3 diaNum = parseInt(prompt('Informe o dia da semana (entre 1 e 7):'))
4
5 switch(diaNum) {
6     case 1:
7         alert('domingo')
8         break
9     case 2:
10        alert('segunda-feira')
11        break
12    case 3:
13        alert('terça-feira')
14        break
15    case 4:
16        alert('quarta-feira')
17        break
18    case 5:
19        alert('quinta-feira')
20        break
21    case 6:
22        alert('sexta-feira')
23        break
24    case 7:
25        alert('sábado')
26        break
27    default:
28        alert('ERRO: dia inválido')
29 }
```

Se tivéssemos usando uma sequência `if..else if..else`, deveríamos escrever algo como:

```
// (...)
if(opcao === 'A' || opcao === 'a') {
    // (...)
}
else if(opcao === 'B' || opcao === 'b') {
    // (...)
}
// (...)
else {
    // (...)
}
```

```
// (...)
```

A estrutura `switch..case` traz uma solução mais elegante quando mais de um valor leva à mesma ação. São os chamados `case` s vazios. Veja como funciona na Listagem 5.12.

Listagem 5.12 A estrutura 'switch..case' com 'case's vazios

```
1 let opcao
2
3 opcao = prompt('Qual sua escolha? (A, B ou C)')
4
5 switch(opcao) {
6     case 'A': // case vazio
7     case 'a':
8         alert('Você escolheu a opção A.')
9         break
10    case 'B': // case vazio
11    case 'b':
12        alert('Você escolheu a opção B.')
13        break
14    case 'C': // case vazio
15    case 'c':
16        alert('Você escolheu a opção C.')
17        break
18    default:
19        alert('ERRO: opção não reconhecida.')
20 }
```

Observe os `case` s das linhas 6, 10 e 14. Não há nenhuma ação (nenhum `alert()`) associada a eles, **nem mesmo possuem um `break`**. Ainda assim, eles constituem pontos de entrada da execução na estrutura `switch..case`, prosseguindo a partir daí linha a linha até atingir a próxima instrução `break` ou o final da estrutura.

Se o usuário, por exemplo, informar a opção `B` (maiúscula), a execução entrará no `switch..case` pela linha 10 e continuará até a linha 13, passando pelo `alert()` da opção B na linha 12.

Enfim, os `case` s vazios constituem o motivo pelo qual o projeto da linguagem exige o `break` ao final de um `case` que contenha, efetivamente, uma ação. Essa é uma característica que JavaScript herdou da linguagem C, estando presente também em outras linguagens da família, como C++, C#, PHP e Java.

5.2.3 Limitação

A estrutura `switch..case` é muito prática quando se trata de comparar o conteúdo de uma única variável contra vários valores diferentes. No entanto, a **única comparação possível**

na estrutura **é a de igualdade**. Colocar operadores após a palavra `case` nem sempre é erro de sintaxe em JavaScript, mas é erro de lógica, porque o *script* não se comportará como esperado.

Em vista disso, problemas como o da Listagem 5.8 (conversão de nota em conceito) não podem ser resolvidos com `switch..case`.

5.3 O operador ternário

Uma ocorrência bastante comum em programação é o resultado de uma condição determinar o valor de uma variável. Por exemplo:

```
let media, status

// (...)

if(media >= 6) {
  status = 'APROVADO'
}
else {
  status = 'REPROVADO'
}
```

Ou, então, o texto de uma saída depender de uma condição:

```
let pais

// (...)

if(pais === 'Brasil') {
  document.write('brasileiro(a)')
}
else {
  document.write('estrangeiro(a)')
}
```

Em ambas podemos detectar:

1. **Uma situação com duas saídas**, uma para o caso de a condição ser verdadeira (`if`) e outra se ela for falsa (`else`).
2. Os blocos associados tanto ao `if` quanto ao `else` têm, cada qual, **apenas uma linha de código**.

Nessas ocasiões, podemos escrever o código de forma mais sucinta usando o operador

ternário, cuja sintaxe é a seguinte:

condição ? ação se verdadeiro : ação se falso

O nome **operador ternário** deve-se ao fato de ser o único operador da linguagem que exige **três** operandos.

Para usá-lo, devemos obedecer à sequência:

1. Primeiramente, escrevemos a condição, a mesma que seria informada no bloco **if** (1º *operando*).
2. Em seguida, colocamos o caractere **?** (ponto de interrogação), como se, de fato, estivéssemos fazendo uma pergunta baseada na condição.
3. Logo em seguida, vem a ação ou valor que o operador assumirá caso a condição seja verdadeira (2º *operando*).
4. Segue-se o caractere **:** (dois pontos), que serve como separador entre a parte “verdadeira” e a parte “falsa” do operador.
5. Por fim, colocamos a ação ou valor assumido se a condição restar falsa (3º *operando*).

Usando o operador ternário, poderíamos escrever os exemplos anteriores conforme mostram a Listagem 5.13 e a Listagem 5.14.

Listagem 5.13 Exemplo (1) de uso do operador ternário

```
1 let media, status
2
3 // (...)
4
5 status = media >= 6 ? 'APROVADO' : 'REPROVADO'
```

Listagem 5.14 Exemplo (2) de uso do operador ternário

```
1 let media, status
2
3 // (...)
4
5 document.write(pais === 'Brasil' ? 'brasileiro(a)' : 'estrangeiro(a))'
```

Bem menos código, não é mesmo? ;)

Com as estruturas condicionais, temos agora um maior **controle** sobre nossos programas, e conseguimos também fazer coisas bastante úteis. Eu não quero ser **repetitivo**, mas seu controle sobre o código aumentará ainda mais no próximo capítulo.

Capítulo 6

ESTRUTURAS DE REPETIÇÃO

Computadores, sem dúvida, são máquinas fascinantes. Todavia, por mais inteligentes que pareçam, tudo o que fazem baseia-se em apenas dois fundamentos: **repetir e contar**.

Neste capítulo, vamos aprender como instruir computadores a fazer repetições (também chamadas **iterações**) e contagens. As estruturas de repetição usadas para isso são chamadas, também, de **laços** ou **loops**.

6.1 while

O laço mais genérico é o **while**, cuja sintaxe é a que segue:

```
while(condição) {  
  Instrução ou instruções a serem executadas  
}
```

A sintaxe de **while** é semelhante à da estrutura condicional **if**. Tudo o que dissemos em relação a esta também se aplica agora, ou seja:

1. **while** é uma palavra reservada do JavaScript, e, como todas elas, é escrita **em minúsculas**.
2. Os **parênteses** em torno da condição **são obrigatórios**.
3. Condição é um valor booleano, ou que se resolve em um valor booleano (**true** ou **false**).
4. Depois do parêntese de fechamento da condição, segue-se uma chave de abertura (**{**). Ele indica o início do bloco de instruções que será executado **apenas se a condição for verdadeira**.
5. Uma chave de fechamento (**}**) marca o final do bloco de instruções.

OBSERVAÇÃO: as chaves são opcionais quando há uma única linha de código atrelada à estrutura `while`. No entanto, até que você esteja bem familiarizado(a) com a sintaxe do JavaScript, recomendo que sempre use as chaves.

A finalidade do laço `while` é repetir a instrução ou instruções no bloco a ele associado, enquanto a condição for verdadeira. O exemplo da Listagem 6.1 exibirá uma contagem de 1 a 10.

Listagem 6.1 Exemplo (1) da estrutura de repetição 'while'

```
1 let contagem = 1
2
3 while(contagem <= 10) {
4     document.write(contagem + '<br>')
5     contagem++ // Equivale a: contagem = contagem + 1
6 }
```

Na linha 1, iniciamos a variável com o valor 1. Como esse valor é menor ou igual a 10 (teste do loop `'while'` na linha 3), o código do bloco associado à estrutura de repetição será executado uma vez, ou seja: na linha 4, será exibido o valor atual da variável e, na linha 5, esse valor será incrementado. Seguindo, a execução retorna para a linha 3, na qual a condição será reavaliada. Enquanto o valor da variável se mantiver menor ou igual a 10, o bloco será executado novamente. Quando o valor da variável atinge 11, a condição torna-se falsa, e o laço é encerrado.

Neste outro exemplo (Listagem 6.2), queremos exibir apenas os números múltiplos de 7 e que sejam menores que 100. Para tanto, combinamos a estrutura de repetição `while` com a estrutura condicional `if`.

Listagem 6.2 Exemplo (2) da estrutura de repetição 'while'

```
1 let num = 1
2
3 while(num < 100) {
4     if(num % 7 === 0) {
5         document.write(contagem + '<br>')
6     }
7     num++
8 }
```

DICA: para determinar se um número **a** é divisível por um número **b**, use o operador `%` (resto da divisão inteira), como foi feito na linha 4, e verifique se o valor retornado foi `0`. Ou seja, **a** será divisível por **b** se não houver resto na divisão.

Se a condição inicial do laço `while` for falsa, ele não será executado nenhuma vez (Listagem 6.3). Por isso, dizemos que `while` **não tem garantia de execução**.

Listagem 6.3 Exemplo (3) da estrutura de repetição 'while'

```
1 let x = 20
2
3 /* A condição inicial é falsa;
4    o loop não será executado */
5 while(x < 10) {
6     document.write(x + '<br>')
7     x-- // Equivale a: x = x - 1
8 }
```

6.2 do..while

Como vimos, o bloco de instruções associado a um *loop while* pode nem ser executado se a condição for falsa no começo. Esse comportamento se dá porque **while** verifica a condição logo no início, antes mesmo de executar qualquer instrução de seu bloco.

Contudo, há situações em que desejamos que o bloco de instruções seja **executado pelo menos uma vez**. Imagine, por exemplo, que estamos implementando uma rotina na qual o usuário precise informar um valor obrigatoriamente numérico. Já tivemos situações assim nesta apostila, e o que fizemos foi informar uma mensagem de erro caso o valor informado não fosse numérico, encerrando o *script* em seguida. Caso o usuário quisesse fazer uma nova tentativa, deveria recarregar a página para forçar a reexecução do código JavaScript.

Com os laços de repetição, podemos fazer melhor. Podemos dar a oportunidade de o usuário informar o valor numérico **uma primeira vez** e, em caso de erro, oferecer novas chances até que ele acerte.

Isso é perfeitamente possível com o uso do *loop do..while*, cuja sintaxe é mostrada a seguir:

```
do {
    Instrução ou instruções a serem executadas
} while(condição)
```

Vamos analisar cada uma das partes:

1. O laço começa com a palavra-chave **do**, seguida da chave de abertura (**{**) do bloco de instruções.
2. No meio do laço, são colocadas as instruções que devem ser executadas por ele, dentro de um bloco.
3. No final, temos a chave de fechamento do bloco (**}**) seguida da palavra-chave **while** e da condição do laço entre parênteses (obrigatórios).

Veja que, com essa estrutura, o laço **do..while** avalia a condição **apenas no final**. A consequência disso é que, até que a condição seja avaliada, **o bloco de instruções já foi executado**.

Com isso, mesmo que a condição se prove falsa, é garantido que o bloco de instruções seja executado pelo menos uma vez.

Usando o *loop* **do..while**, a rotina de entrada de um valor obrigatoriamente numérico pode ser escrita como mostra a Listagem 6.4.

Listagem 6.4 Exemplo de uso do laço 'do..while'

```
1 let num
2
3 do {
4   num = Number(prompt('Informe um valor numérico:'))
5 } while(isNaN(num))
```

Dessa forma, o usuário será convidado a inserir o valor numérico uma vez, na linha 4. Veja que tentamos converter a entrada informada para número usando `Number()`. Já vimos que, quando a conversão falha, o valor resultante é `NaN` e podemos verificar isso usando a função `isNaN()`. Portanto, caso ele informe algo que não seja numérico, `isNaN(num)` retornará `true`, fazendo com que um novo `prompt()` seja exibido ao usuário, dando-lhe uma nova chance de informar um valor numérico. Esse processo irá se repetir enquanto o valor informado não for válido.

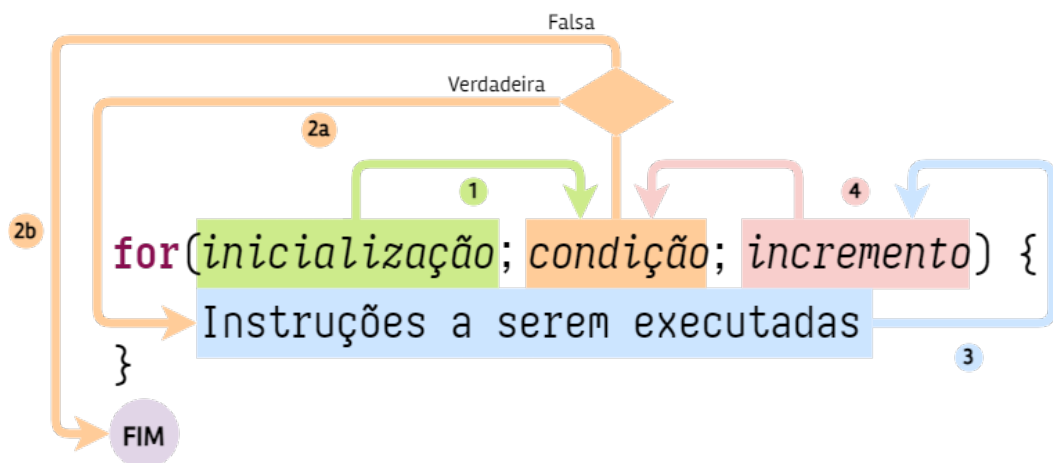
Quando o usuário informa, de fato, um valor numérico, a conversão com `Number()` é bem-sucedida e a verificação de `isNaN(num)` retorna `false`, dando fim à execução do *loop*.

6.3 for

Tanto **while** quanto **do..while** foram projetados para serem executados, no mais das vezes, um número **indeterminado** de vezes; ou seja, as iterações acontecem enquanto a condição se mantiver verdadeira.

Entretanto, uma outra função dos laços de repetição é efetuar contagens, situação em que a quantidade de repetições é **determinada ou determinável**. JavaScript possui o laço **for**, especializado em contagens, cuja sintaxe básica é mostrada na Figura 6.1.

1. O laço começa com a palavra-chave **for**, seguida de um parêntese de abertura. Dentro do parêntese, há três partes distintas, separadas entre si por ponto-e-vírgula (;):
 - a. A primeira parte, chamada **inicialização**, serve para ajustar o valor inicial da variável que servirá como contadora do laço.
 - b. A segunda parte contém uma **condição** que, enquanto se mantiver verdadeira, fará com que o *loop* se repita.
 - c. Por fim, temos a parte de **incremento**, na qual o valor da variável contadora é atualizado.



O laço começa pela **inicialização**, que é executada uma única vez para ajustar o valor inicial da variável de controle.

Em seguida **1**, avalia a **condição**.

- Se a **condição** for verdadeira **2a**, prossegue para a execução das **instruções do bloco**.
- Se a **condição** for falsa **2b**, o laço é encerrado.

No caso de **2a**, após a execução **3** das **instruções do bloco**, processa-se o **incremento** para mudar o valor da variável de controle, retornando em seguida para a reavaliação **4** da **condição**, prosseguindo (**2a**) ou não (**2b**) com uma nova repetição.

Figura 6.1: Fluxograma do laço 'for'

2. Após o parêntese de fechamento (`)`), deve-se colocar a chave de abertura (`{`) do bloco de instruções do laço.
3. Dentro do bloco, naturalmente, vão as instruções que devem ser repetidas pelo laço.
4. O laço é encerrado com a chave de fechamento (`}`) do bloco de instruções.

O exemplo real de código da Listagem 6.5 exibe os números de 1 a 10.

Listagem 6.5 Laço 'for' para exibir os números de 1 a 10

```
1 for(let i = 1; i <= 10; i++) {  
2   document.write(i + '<br>')  
3 }
```

OBSERVAÇÃO: é comum declarar a variável de controle do *loop* `for` dentro da seção de inicialização, com a palavra-chave `let`. Note, no entanto, que variáveis declaradas assim deixam de existir tão logo o laço termine, isto é, não é possível saber o valor de `i` na linha de código seguinte ao final do *loop*.

Note, na linha 1:

- a **inicialização**, com `let i = 1`.
- a **condição**, com `i <= 10`.
- o **incremento**, com `i++`.

Temos nesse código, portanto, um laço `for` que é controlado pela variável `i`. Esta recebe, na inicialização, seu valor inicial (`1`). Em seguida, como esse valor cumpre a condição (`i` é menor ou igual a `10`), o bloco de instruções é executado, com `document.write()` (linha 2) mostrando o valor de `i`. Prosseguindo, é processado o incremento, no qual o valor de `i` é aumentado em uma unidade, passando a valer `2`. A condição é reavaliada; como `2` é maior ou igual a `10`, o bloco de instruções será executado novamente, depois o incremento, e, por fim, a reavaliação da condição.

Quando `i` atingir o valor `11`, a condição tornar-se-á falsa, encerrando a execução do *loop*.

O laço `for` é bastante flexível, permitindo diversas variações. No exemplo da Listagem 6.6, temos uma contagem regressiva, de 10 a 1.

Listagem 6.6 Laço 'for' para exibir os números de 10 a 1, em ordem inversa

```
1 for(let i = 10; i >= 1; i--) {  
2   document.write(i + '<br>')  
3 }
```

O que fizemos nesse caso foi inicializar a variável `i` com o valor máximo pretendido (`10`), ajustando a condição para que o laço seja executado enquanto `i` seja **maior ou igual** a `1`. A

terceira parte do `for`, a rigor, aqui, é um **decremento**, já que precisamos **diminuir** o valor da variável de controle a cada iteração para conseguir o efeito desejado.

Uma outra possibilidade é criar laços cujo incremento (ou decremento) ocorre em intervalos diferentes de 1. O exemplo da Listagem 6.7 exibe os números entre 0 e 100, contando de 5 em 5.

Listagem 6.7 Laço 'for' com incremento em intervalo diferente de 1

```
1 for(let i = 0; i <= 100; i += 5) {  
2   document.write(i + '<br>')  
3 }
```

Repare que, na seção de incremento, a variável `i` ganha 5 unidades a cada repetição do *loop*, usando um operador composto de atribuição.

Outra situação: nem sempre sabemos, no momento da codificação, quantas iterações serão executadas pelo laço `for`. Isso pode ser decidido, por exemplo, pelo usuário. Assim, o número de repetições, embora não seja determinado, é **determinável**. Observe a Listagem 6.8.

Listagem 6.8 Laço 'for' com uma variável determinando a quantidade máxima de iterações

```
1 let quant  
2  
3 do {  
4   quant = parseInt(prompt('Informe a quantidade de asteriscos:'))  
5 } while(isNaN(quant))  
6  
7 for(let i = 1; i <= quant; i++) {  
8   document.write('*')  
9 }
```

Veja que, em vez de cravarmos um número fixo de repetições (do caractere `*`, no exemplo), é o valor da variável `quant`, informado pelo usuário, que determina esse limite. Note, ainda, como foram usados os *loops* `do..while` (linhas 3 a 5) e `for` em um mesmo *script*, cada qual com sua finalidade inerente.

Por fim, vale comentar que, assim como o *loop* `while`, `for` também **não tem garantia de execução**, isto é, caso a condição seja falsa logo de início, nada acontecerá.

A Tabela 6.1 oferece um comparativo dos três laços de repetição da linguagem JavaScript.

Tabela 6.1: Comparativo entre as estruturas de repetição da linguagem JavaScript

Laço	Local de verificação da condição	Garantia de execução	Quantidade de iterações
<code>while</code>	No início	0 vezes	Indeterminada
<code>do..while</code>	No final	1 vez	Indeterminada
<code>for</code>	No início	0 vezes	Determinada ou determinável

As estruturas condicionais e de repetição são realmente poderosas. Nossos códigos estão ficando maiores e mais complexos, e isso pode demandar um pouco de organização. No próximo capítulo, vamos aprender como **funções** podem nos ajudar a ter programas mais fáceis de ler e de dar manutenção.

Capítulo 7

FUNÇÕES (básico)

Usualmente, *scripts* JavaScript (ou programas escritos na maioria das linguagens de programação) são executados em sequência, linha a linha. Vimos que essa sequência pode ser quebrada por meio das estruturas de controle condicionais, que permitem executar ou não determinado trecho de código caso uma condição seja verdadeira ou falsa. As estruturas de repetição, por seu turno, passam reiteradas vezes pelo mesmo bloco de código também dependendo da avaliação de uma condição.

Uma outra estrutura da linguagem que também “quebra” a execução sequencial são as funções. Trata-se de trechos de código, aos quais normalmente se dá um nome, que são escritos uma vez e podem ser invocados posteriormente uma ou mais vezes. A própria linguagem JavaScript já tem definidas várias funções, algumas das quais já usamos, como `alert()`, `prompt()` e `isNaN()`. De fato, como você irá aprender ao se aprofundar no conhecimento da linguagem, funções são o coração e a alma do JavaScript.

Além das funções prontas fornecidas pela linguagem, podemos também criar nossas próprias funções. É que vamos aprender a partir de agora.

7.1 Declaração e chamada de função

O trabalho com funções, em geral, divide-se em duas partes: **declaração** e **chamada** de função. Observe a Listagem 7.1.

OBSERVAÇÃO: lembre-se de que, para ver a saída da função `console.log()`, você precisa pressionar a tecla `F12` e ativar a aba Console no navegador.

Vamos analisar o código por partes.

1. A **declaração** de uma função inicia-se pela palavra reservada `function` (linha 2), seguida do nome que queremos atribuir à função. A nomeação de funções segue as mesmas regras

Listagem 7.1 Exemplo simples de declaração e chamadas de função

```
1 // Declaração de função
2 function elevarAoCubo(n) {
3     return n * n * n
4 }
5
6 // Chamada de função (1)
7 let res = elevarAoCubo(3)
8
9 console.log(res)           // Mostrará 27
10
11 // Chamada de função (2)
12 console.log(elevarAoCubo(5)) // Mostrará 125
```

usadas para nomear variáveis.

2. Após o nome da variável, segue-se um par de parênteses, **que são obrigatórios**. Dentro dos parênteses pode haver os chamados **parâmetros**, também chamados **atributos**, que constituem **informações de entrada** para a função, ou seja, informações que a função recebe e que são necessárias para que ela execute o seu trabalho. Dentro da função, os parâmetros são usados como se fossem variáveis.
3. Aos parênteses, segue-se um bloco de código, delimitado por chaves (`{ }`). Esse bloco pode ter uma ou mais linhas de instruções. Uma função pode, também, produzir uma **informação de saída**, que é indicada pelo uso da palavra-chave `return` (linha 3).

Ou seja, entre as linhas 2 e 4 da listagem, temos declarada uma função, à qual demos o nome de `elevarAoCubo` e que recebe uma informação de entrada (o parâmetro `n`). No bloco de código da função, é calculado o valor do parâmetro `n` elevado ao cubo, o qual constitui a informação de saída da função, indicada por `return`.

É importante notar que uma função **não é executada** no ponto onde foi declarada. Em outras palavras, é como se o JavaScript ignorasse o trecho das linhas 2 a 4, iniciando a execução desse *script* somente na linha 7. Mesmo porque, no momento da declaração da função, não sabemos ainda o valor do parâmetro `n` e, portanto, não seria efetuar o cálculo de elevação ao cubo.

1. Uma função é acionada efetivamente quando é feita uma **chamada** a ela, o que, no exemplo da Listagem 7.1, ocorreu duas vezes, nas linhas 7 e 12. Para fazer a chamada a uma função, usamos seu nome e, nos parênteses que se seguem, o valor dos parâmetros, caso esses tenham sido previstos na declaração.
2. No caso da chamada feita na linha 7, a informação passada para a função e que será recebida no parâmetro `n` é o valor `3`. Logo, a função irá calcular 3 elevado ao cubo e o valor resultante (`27`), a informação de saída, pode ser capturada em uma variável (`res`).
3. Já na chamada efetuada na linha 12, o valor passado para o parâmetro `n` foi `5` que,

elevado ao cubo, resulta em `125`. Agora, em vez de capturar o resultado em uma variável para só então exibi-lo com `console.log()`, fizemos a chamada da função diretamente dentro de `console.log()`. Em suma, a **saída** da função `elevarAoCubo()` tornou-se a **entrada** de `console.log()`, e essa é apenas uma das muitas possibilidades de uso de funções.

Neste momento, talvez você esteja se perguntando: não seria mais simples escrever algo como `let res = 3 * 3 * 3` na linha 7 ou `console.log(5 * 5 * 5)` na linha 12, eliminando a necessidade de declarar e chamar uma função? De fato, esse é um exemplo simples, e, em uma situação prática, o uso de uma função não se justificaria.

Todavia, precisamos ter em mente que funções podem fazer tarefas mais complexas, e é nesses casos que elas se mostram mais úteis. Veja agora o exemplo da Listagem 7.2.

Listagem 7.2 Função para o cálculo do fatorial de um número inteiro

```
1 // Declaração de função
2 function fatorial(x) {
3     let resultado = x
4     for(let i = x - 1; i > 1; i--) {
5         resultado *= i
6     }
7     return resultado
8 }
9
10 // Chamadas de função
11 console.log(fatorial(5))    // 120
12 console.log(fatorial(7))    // 5040
```

Caso você não se lembre, o fatorial de um número inteiro `n` (representado por `n!`) é igual a ele mesmo multiplicado por seus antecessores até o número `1`. Ou seja, `5! = 5 * 4 * 3 * 2 * 1` e `7! = 7 * 6 * 5 * 4 * 3 * 2 * 1`. Repare que, no corpo da função (bloco de código), foram necessárias 5 linhas de código (de 3 a 7) para chegar ao resultado. Sem a função, para calcular o fatorial dos números 5 e 7, como foi feito nas linhas 10 e 11, precisaríamos escrever essas 5 linhas duas vezes, com pequenas modificações de valores, resultado em um *script* consideravelmente maior. Em situações assim, podemos constatar outra vantagem da utilização de funções, que é **evitar repetição de código**.

7.1.1 Múltiplos parâmetros

Até agora, os exemplos que vimos mostraram funções que recebem apenas um parâmetro. No entanto, funções podem receber mais de um parâmetro, como mostra a Listagem 7.3.

O exemplo da Listagem 7.3 serve também para ilustrar que funções não se aplicam apenas a tarefas de cálculos matemáticos. Nela, usamos uma função para gerar uma *string* de acordo

Listagem 7.3 Função com dois parâmetros

```
1  /* Declaração de uma função que gera um traço composto pelo
2     caractere indicado, repetido quantas vezes for a quantidade
3     especificada
4  */
5  function gerarTraco(caract, quant) {
6      let resultado = ''
7      for(let i = 1; i <= quant; i++) {
8          resultado += caract    // += aqui é concatenação
9      }
10     resultado += '<br>' // Adiciona quebra de linha
11     return resultado
12 }
13
14 // Chamada de função que gera um traço com 20 asteriscos
15 document.write(gerarTraco('*', 20))
16
17 // Chamada de função que gera um traço com 50 sublinhados
18 document.write(gerarTraco('_', 50))
```

com um padrão predeterminado pelos dois parâmetros. Note, na chamada de função da linha 15, que o valor `'*'` está sendo passado na primeira posição e, portanto, será recebido pela função no parâmetro `caract`. De forma semelhante, o valor `20`, passado na segunda posição, será recebido pela função no parâmetro `quant`. A conclusão a que chegamos é que **valores são passados a funções e recebidos nos parâmetros POR ORDEM**. Guarde essa informação, ela é importante.

Veja, agora, uma função com três parâmetros (Listagem 7.4).

Nesse novo exemplo, não sabemos as informações que iremos passar à função no momento que escrevemos o código, pois é o usuário quem irá fornecê-las. Para tanto, temos primeiramente a declaração de três variáveis e seu respectivo preenchimento pelo usuário (linhas 19 a 21) e, depois, a chamada à função (linha 24) usando essas variáveis para passar os valores informados. Não é necessário que as variáveis tenham o mesmo nome dos parâmetros da função; afinal, já aprendemos que, na passagem de parâmetros, o que importa é a **ordem** dos valores.

7.1.2 Funções sem parâmetros

É possível a existência de funções sem parâmetro, ou seja, funções que não necessitam de informações externas para fazer seu trabalho. Quando isso acontece, os parênteses que delimitam os parâmetros ficam vazios, mas não podem ser omitidos, seja na declaração, seja na chamada da função.

Listagem 7.4 Função com três parâmetros

```
1  /* Declaração de função que calcula a área de uma forma
2     geométrica: retângulo (tipo 'R') ou triângulo (tipo 'T') */
3  function areaForma(base, altura, tipo) {
4      let area
5      switch(tipo) {
6          case 'R':    // Retângulo
7              area = base * altura
8              break
9          case 'T':    // Triângulo
10             area = base * altura / 2
11             break
12         default:     // Forma inválida
13             area = 0
14     }
15     return area
16 }
17
18 // O usuário informará a base, a altura e o tipo
19 let infoBase = parseFloat(prompt('Informe a base:'))
20 let infoAltura = parseFloat(prompt('Informe a altura:'))
21 let infoTipo = prompt('Informe o tipo (R ou T):')
22
23 // Chamada da função
24 let resArea = areaForma(infoBase, infoAltura, infoTipo)
25
26 alert('A área da forma é ' + resArea)
```

Aliás, **os parênteses são a marca registrada das funções**, tanto que é comum fazer referência a elas pelo nome seguido de parênteses, como já fizemos várias vezes nesta apostila. Dessa forma, em vez de escrever “função `alert`” em um texto, escrevemos “função `alert()`”, com os parênteses, para melhor diferenciá-la do nome de uma variável.

Observe, na Listagem 7.5, a declaração de uma função sem parâmetros que retorna a hora atual do computador em formato de *string*. Para produzir o resultado, essa função faz chamadas a outras funções da linguagem JavaScript, que igualmente não levam parâmetros.

7.1.3 Funções sem valor de retorno

Funções também podem ser usadas para isolar trechos de códigos que desempenham uma tarefa, geralmente repetitiva, mas não retornam valor algum. Veja como isso ocorre nos exemplos da Listagem 7.6.

Nessa listagem, temos declaradas duas funções sem valor de retorno, `desenharTraco()` (linha 1) e `exibirSaudacao()` (linha 12). Ambas se caracterizam pela ausência da palavra-

Listagem 7.5 Exemplos de funções sem parâmetros

```
1 // Declaração de função: os parênteses dos parâmetros ficam vazios
2 function agora() { // Retorna a hora atual do computador
3     let hoje = new Date()
4     // Veja, na linha abaixo, outras três funções que também não recebem
5     // ↪ parâmetros
6     return hoje.getHours() + ':' + hoje.getMinutes() + ':' + hoje.getSeconds()
7 }
8
9 // Chamada à função: os parênteses da função se mantêm, mesmo vazios
10 alert('A hora atual é ' + agora())
```

chave **return** em seus blocos de código. Como consequência, ao efetuarmos chamadas a essas funções (linhas 9, 10 e 27), não faz sentido atribuir seu resultado a uma variável, já que não retornam (explicitamente) nenhum valor.

OBSERVAÇÃO: tecnicamente, uma função que não possui a palavra-chave **return** ainda assim retorna o valor **undefined**.

7.2 Expressão de função

Em JavaScript, funções podem ser atribuídas a variáveis e constituem um tipo de dados próprio. Com isso, é possível escrever uma **expressão de função** no lugar de uma declaração de função na maioria dos casos.

Uma expressão de função nada mais é do que uma função com bloco de código atribuída a uma variável, como demonstra a Listagem 7.7.

A listagem mostra uma declaração de função (**imc1()**, linha 2) e, logo abaixo, uma expressão de função equivalente, **imc2()** (linha 7). Na expressão de função, esta é atribuída a uma **const** ante – uma “variável” que recebe um valor inicial que não pode mais ser modificado (para evitar erros de lógica). Note também que, na expressão de função, não há um **nome** entre a palavra-chave **function** e os parênteses dos parâmetros. Em outras palavras, a função em uma expressão de função é **anônima**.

OBSERVAÇÃO: sempre que, em seus programas, você tiver uma variável que recebe um valor inicial que não irá mais mudar, prefira declará-la com **const** em vez de **let**. Isso evita que você mude o valor dessa variável acidentalmente e previne o surgimento de *bugs* no código.

A chamada a uma função que foi criada a partir de uma expressão de função é feita pelo nome da **const** ante que recebeu a função. Com isso, as chamadas de função das linhas 11 e 12

Listagem 7.6 Exemplos de funções sem valor de retorno

```
1 // Desenha na página um traço do tamanho especificado
2 function desenharTraco(tamanho) {
3     for(let i = 1; i <= tamanho; i++) {
4         document.write('-')
5     }
6     document.write('<br>') // Quebra de linha
7 }
8
9 desenharTraco(30) // Desenha um traço com 30 hífens
10 desenharTraco(50) // Desenha um traço com 50 hífens
11
12 function exibirSaudacao(horaDia) {
13     if(horaDia < 12) {
14         alert('Bom dia!')
15     }
16     else if(horaDia < 18) {
17         alert('Boa tarde!')
18     }
19     else if(horaDia <= 23) {
20         alert('Boa noite!')
21     }
22     else {
23         alert('ERRO: hora do dia inválida!')
24     }
25 }
26
27 exibirSaudacao(14) // Exibirá 'Boa tarde!'
```

produzem o mesmo resultado.

Expressões de função são muito usadas na programação profissional em JavaScript. Portanto, mesmo que você não entenda seu sentido ou utilidade neste momento, esforce-se em aprender o conceito para quando for necessário utilizá-lo no futuro.

7.3 Onde declarar ou colocar expressar funções no código

Como vimos anteriormente, o trabalho com funções envolve duas etapas distintas: 1. Declarar a função ou criar a expressão de função; e 2. Efetuar chamada(s) à função.

No código, **a declaração de função pode vir ANTES ou DEPOIS da(s) respectiva(s) chamadas**. Em outras palavras, as chamadas podem vir acima ou abaixo da declaração da função, indistintamente, como exemplificado pela Listagem 7.8.

As chamadas de função efetuadas nas linhas 2 e 14 funcionarão normalmente, uma vez

Listagem 7.7 Declaração de função vs. expressão de função

```
1 // Declaração de função
2 function imc1(peso, altura) {
3     return peso / (altura ** 2)
4 }
5
6 // Expressão de função
7 const imc2 = function(peso, altura) {
8     return peso / (altura ** 2)
9 }
10
11 alert(imc1(80, 1.76))
12 alert(imc2(80, 1.76))
```

Listagem 7.8 Posicionamento de declarações e chamadas de função no código

```
1 // Chamada da função ANTES da declaração
2 alert(fatorial(4)) // 24
3
4 // Declaração da função
5 function fatorial(n) {
6     let res = 1
7     for(let i = n; i > 1; i--) {
8         res *= i
9     }
10    return res
11 }
12
13 // Chamada da função DEPOIS da declaração
14 alert(fatorial(5)) // 120
```

que o JavaScript organiza internamente o código e consegue encontrar a função no momento da chamada mesmo que a respectiva declaração só apareça posteriormente. Apesar disso, desenvolvedores profissionais têm o hábito de posicionar declarações de função **antes** das chamadas.

No entanto, **chamadas a funções criadas com expressões de função só podem ser efetuadas POSTERIORMENTE**. Expressões de função nada mais são do que funções que foram atribuídas a uma variável ou constante e, portanto, seguem o comportamento destas. Uma variável ou constante passa a existir somente a partir do ponto em que é declarada, e isso se estende às expressões de função, como é possível observar na Listagem 7.9.

7.4 Variáveis e funções

Listagem 7.9 Posicionamento de expressões e chamadas de função no código

```
1 // Chamada da função ANTES da criação da expressão de função
2 alert(fatorial(4)) // NÃO FUNCIONARÁ - a função ainda não existe!
3
4 // Criação da expressão da função
5 // A função passa a existir a partir deste ponto
6 const fatorial = function(n) {
7     let res = 1
8     for(let i = n; i > 1; i--) {
9         res *= i
10    }
11    return res
12 }
13
14 // Chamada da função DEPOIS da criação da expressão de função
15 alert(fatorial(5)) // Funcionará normalmente
```

Capítulo 8

MANIPULAÇÃO DE *STRINGS*

Capítulo 9

DOCUMENT OBJECT MODEL (DOM)

Capítulo 10

VETORES

Capítulo 11

OBJETOS

Capítulo 12

REVISITANDO O DOM

Capítulo 13

FUNÇÕES (avançado)

Licença

Esta obra é disponibilizada a você sob a licença [Creative Commons BY-NC 4.0](#).

Créditos

Imagem da capa: [Technology vector created by vectorjuice – www.freepik.com](#)