



# Conceitos de Lógica de Programação com JavaScript







# Conceitos de Lógica de Programação com JavaScript



Copyright © UNIÃO EDUCACIONAL E TECNOLÓGICA IMPACTA - UNI.IMPACTA

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou

no todo, sem a aprovação prévia, por escrito, da UNIÃO EDUCACIONAL E TECNOLÓGICA IMPACTA - UNI.IMPACTA, estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

*"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."*

# Conceitos de Lógica de Programação com JavaScript

## Coordenação Geral

Alexandre Hideki Chicaoka

## Autoria

Emilio Celso de Souza

## Revisão Ortográfica e Gramatical

Fernanda Monteiro Laneri

## Diagramação

Bruno de Oliveira Santos

## Edição nº 1 | 1896\_0

Fevereiro/ 2021

*Este material constitui uma nova obra e é uma derivação das seguintes obras originais, produzidas por TechnoEdition Editora Ltda.:*

*Jul./2013*

*Introdução à Lógica de Programação*

*Adaptação: André Luís de Vasconcelos*

*Nov./2012*

*JavaScript*

*Autoria: André Ricardo Stern, Cristiana Hoffmann Pavan, José Henrique Cunha Rangel, Maria Rosa Carnicelli Kushnir, Rafael Farinaccio*

<b>Capítulo 1 - Introdução à lógica .....</b>	<b>09</b>
1.1. Lógica .....	10
1.2. Programa .....	11
1.2.1. Tipos de linguagem de programação .....	11
1.3. Tradutores.....	12
1.3.1. Tipos de tradutores .....	12
Pontos principais .....	13
<b>Capítulo 2 - Conceitos de algoritmo.....</b>	<b>15</b>
2.1. Algoritmo .....	16
2.2. Elementos de um algoritmo .....	18
2.2.1. Ação .....	18
2.2.2. Decisão .....	19
2.2.3. Laço ou loop.....	19
2.3. Algoritmo com o comando SE encadeado .....	21
2.4. Algoritmo com o comando CASO.....	21
2.5. Algoritmo com o comando ENQUANTO.....	23
Pontos principais .....	24
<b>Capítulo 3 - Fluxograma.....</b>	<b>25</b>
3.1. Introdução.....	26
3.2. Simbologia .....	26
3.3. Criando fluxogramas .....	27
3.3.1. Estruturas básicas.....	28
3.4. Teste de Mesa .....	33
Pontos principais .....	34
<b>Capítulo 4 - JavaScript.....</b>	<b>35</b>
4.1. A linguagem JavaScript .....	36
4.2. Ambientes de desenvolvimento JavaScript.....	36
4.2.1. Executando instruções JavaScript por meio do browser.....	37
4.2.2. Executando instruções JavaScript por meio de uma página HTML.....	38
4.2.3. Executando instruções JavaScript por meio do Node.js.....	38
4.3. Camadas de desenvolvimento.....	39
4.4. Primeiros códigos em JavaScript .....	39
4.4.1. Escrevendo e executando um programa.....	39
4.4.2. Comentários.....	40
4.5. Orientação a objetos.....	40
Pontos principais .....	41

# Conceitos de Lógica de Programação com JavaScript

<b>Capítulo 5 - Variáveis</b>	<b>43</b>
5.1. O que é uma variável?	44
5.2. Criando variáveis	44
5.2.1. Declarando variáveis	44
5.2.2. Variáveis locais	45
5.2.3. Variáveis globais	45
5.2.4. Aritmética com JavaScript	45
5.3. Palavras reservadas	46
5.4. Tipos de variáveis	47
5.4.1. Convertendo strings em números	48
5.5. Caracteres especiais	50
5.6. Concatenação	51
5.7. Constantes	51
5.8. Objetos globais	51
5.8.1. Variáveis e propriedades dos objetos	52
5.8.2. null	52
5.8.3. undefined	52
5.8.4. NaN	52
5.8.5. Infinity	52
Pontos principais	53
<b>Capítulo 6 - Operadores</b>	<b>55</b>
6.1. Utilizando operadores em JavaScript	56
6.2. Expressões	56
6.3. Tipos de operadores	57
6.3.1. Operadores de atribuição	57
6.3.2. Operadores de comparação	58
6.3.3. Operadores aritméticos	59
6.3.4. Operadores bitwise	60
6.3.4.1. Operadores bitwise lógicos	60
6.3.4.2. Operadores bitwise de deslocamento	62
6.3.5. Operadores lógicos	63
6.3.5.1. Avaliação de curto-circuito	64
6.3.6. Operadores de string	64
6.3.7. Operadores especiais	65
6.3.7.1. Operador condicional	65
6.3.7.2. Operador separador	66
6.3.7.3. delete	66
6.3.7.4. in	67
6.3.7.5. instanceof	68
6.3.7.6. new	68
6.3.7.7. this	68
6.3.7.8. typeof	69
6.4. Precedência dos operadores	70
Pontos principais	71

<b>Capítulo 7 - Declarações</b>	<b>73</b>
7.1. Introdução	74
7.1.1. Declarações em JavaScript	74
7.2. Estruturas condicionais	75
7.2.1. Declaração if/else	75
7.2.2. Declaração switch/case	76
7.3. Estruturas para loops	78
7.3.1. Declaração while	78
7.3.2. Declaração do/while	79
7.3.3. Declaração for	79
7.3.4. Declaração for/in	81
7.3.5. Declaração break	82
7.3.6. Declaração continue	83
Pontos principais	84
<b>Capítulo 8 - Funções</b>	<b>85</b>
8.1. O que é uma função?	86
8.2. Definindo uma função	86
8.2.1. Inserindo funções	87
8.3. Chamando funções	87
8.4. Escopo de uma função	89
8.5. Closures	90
8.6. Inserindo variáveis nos parâmetros	91
8.6.1. Um parâmetro	92
8.6.2. Múltiplos parâmetros	93
8.6.3. Parâmetros passados por valores	93
8.7. Retornando valores	94
8.8. Funções predefinidas	94
8.9. Propriedades das funções	97
8.9.1. arguments.length	97
8.9.2. arguments.callee	97
8.9.3. length	98
Pontos principais	99







# Introdução à lógica

- Conceitos de lógica;
- Programa;
- Tradutores.

Erik Proença, 323.439.688-45



## 1.1. Lógica

**Lógica** é a maneira de raciocinar particular a um indivíduo ou a um grupo, gerando uma sequência coerente, regular e necessária de acontecimentos ou métodos, com a finalidade de obter uma solução prática e eficaz para um problema.

Se observarmos o nosso dia a dia, usamos a lógica frequentemente, em nossas casas, no trabalho, nas compras, no trânsito, ou seja, em tudo. Como exemplo, podemos citar a lógica que fazemos desde a hora que acordamos até quando chegamos ao trabalho:

- **Acordar no horário programado**
- **Tomar banho**
- **Vestir a roupa adequada para trabalhar**
- **Tomar café**
- **Sair de casa**
- **Chegar ao local de trabalho dentro do horário previsto**

Como podemos observar, já acordamos pensando no que temos que fazer, portanto, já acordamos utilizando a lógica e, se a lógica inicial não for a ideal, nós a modificamos para que ela nos leve à melhor solução do problema. Por exemplo, se alguém já estiver tomando banho, então, podemos tomar café antes para que não nos atrasemos para o trabalho.

- **Acordar no horário programado**
- **Verificar se o banheiro está livre**
  - Se sim:
    - **Tomar banho**
    - **Vestir a roupa adequada para trabalhar**
    - **Tomar café**
  - Se não:
    - **Tomar café**
    - **Tomar banho**
    - **Vestir a roupa adequada para trabalhar**
- **Sair de casa**

- **Chegar ao local de trabalho dentro do horário previsto**

Existem diversos tipos de exercícios e jogos que servem para desenvolver o raciocínio lógico. Vamos começar com exercícios de lógica mais comuns e, depois, focaremos a programação.

- **Exemplo**

Vamos supor que exista uma caixa com 15 bolas, sendo 5 verdes, 5 amarelas e 5 azuis. Quantas bolas devem ser retiradas da caixa, sem olhar, para termos certeza de que saíram 2 bolas de cor azul?

- **Resposta**

- 5 bolas verdes
- 5 bolas amarelas
- 2 bolas azuis

Total: 12 bolas

- **Conclusão**

Temos que ter certeza que saíram 2 bolas azuis, portanto consideramos a pior hipótese no caso de retirar bolas aleatoriamente. Foram retiradas as 10 bolas entre verdes e amarelas e, depois, saíram as bolas azuis.

## 1.2. Programa

**Programa** é uma sequência lógica de instruções escritas em uma linguagem de programação, para serem executadas passo a passo, com a finalidade de atingir um determinado objetivo.

### 1.2.1. Tipos de linguagem de programação

Uma **linguagem de programação** é um método padronizado para comunicar instruções para um computador.

As linguagens de programação podem ser de **baixo nível** e de **alto nível**.

As linguagens de baixo nível são aquelas capazes de compreender a arquitetura do computador e que utilizam somente instruções do processador. Exemplos: Linguagem de máquina e Assembly.

As linguagens de alto nível são aquelas com a escrita mais próxima da linguagem humana. Exemplos: Objective-C, C++, C#, Java, JavaScript e VB.

Neste curso, usaremos a linguagem JavaScript por conta da sua facilidade e aderência a diversas plataformas de desenvolvimento.

## 1.3. Tradutores

Os **tradutores** foram criados para tornar mais fácil a interface entre o usuário e a máquina. Como já vimos anteriormente, as linguagens são divididas em baixo e alto nível, cada uma refletindo uma proximidade com a linguagem natural do usuário.

O computador só executa instruções em linguagem de máquina, a qual é composta por dígitos binários. Logo, para que o computador execute instruções escritas em linguagens com estruturas diferentes, é preciso que essas instruções sejam traduzidas para a linguagem de máquina.

O tipo da tradução depende da complexidade da estrutura das linguagens.

### 1.3.1. Tipos de tradutores

Existem quatro tipos básicos de tradutores: **Montador**, **Interpretador**, **Run-time** e **Compilador**.

O **Montador** traduz a linguagem Assembly para a linguagem de máquina. Sua estrutura é relativamente simples e depende diretamente do processador utilizado, pois cada processador tem seu set de instruções característico.

Os outros tradutores são mais complexos, pois necessitam fazer análises mais sofisticadas da estrutura da linguagem para realizar a tradução.

O **Interpretador** realiza a tradução e a execução simultaneamente, não gerando o código-objeto (linguagem de máquina) em disco.

A geração de código em disco é observada no **Run-time** e no **Compilador**. A diferença entre eles é que o **Run-time** trabalha com um código intermediário (pseudocompilado).

O **Compilador** é um programa que traduz uma linguagem de programação de alto nível para linguagem de máquina, gerando um código-objeto independente.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **Lógica** é a maneira de raciocinar particular a um indivíduo ou a um grupo, gerando uma sequência coerente, regular e necessária de acontecimentos ou métodos, com a finalidade de obter uma solução prática e eficaz para um problema;
- **Programa** é uma sequência lógica de instruções escritas em uma linguagem de programação, para serem executadas passo a passo, com a finalidade de atingir um determinado objetivo;
- Os **tradutores** foram criados para tornar mais fácil a interface entre o usuário e a máquina. Como já vimos anteriormente, as linguagens são divididas em baixo e alto nível, cada uma refletindo uma proximidade com a linguagem natural do usuário.



# 2

## Conceitos de algoritmo

- Algoritmo;
- Elementos de um algoritmo;
- Algoritmo com o comando SE;
- Algoritmo com o comando CASO;
- Algoritmo com o comando ENQUANTO.



## 2.1. Algoritmo

**Algoritmo** é a descrição sequencial ordenada dos passos que devem ser executados, de forma lógica e clara, com a finalidade de facilitar a resolução de um problema.

Você viu anteriormente o passo a passo da rotina do início do dia de uma pessoa. Então, é o algoritmo que ajuda a resolver o que fazer desde a hora de acordar até chegar ao local de trabalho.

A seguir, temos um algoritmo, com numeração nas linhas, de **como fritar um ovo**.

Detalhes: Um ovo, uma lata de óleo, um saleiro, um prato, uma colher e uma caixa de fósforos estão ao lado do fogão, em cima de uma pia. Tem gás e a frigideira já está em cima de uma boca do fogão.

1. Início
2. Acender o fogo
3. Colocar óleo na frigideira
4. O óleo está quente?
  - 4.1. Se sim: “Próximo passo”
  - 4.2. Se não: “Esperar o óleo esquentar” e “Vá para o passo 4”
5. Quebrar o ovo
6. O ovo está bom?
  - 6.1. Se sim: “Próximo passo”
  - 6.2. Se não: “Vá para o passo 10”
7. Colocar o ovo na frigideira
8. Colocar uma pitada de sal
9. O ovo está frito?
  - 9.1. Se sim: “Próximo passo”
  - 9.2. Se não: “Esperar fritar” e “Vá para o passo 9”
10. Pegar o ovo com a colher e colocar no prato
11. Desligar o fogo
12. Fim



A seguir, temos um algoritmo de **como trocar um pneu**.

Detalhes: O carro está estacionado em um lugar seguro e quem vai trocar o pneu já está ao lado do porta-malas com as chaves do carro.

1. Início
2. Abrir o porta-malas do carro
3. Tem estepe?
  - 3.1. Se sim: Retirar o estepe
  - 3.2. Se não: Vá para o passo 18
4. Tem ferramentas?
  - 4.1. Se sim: Pegar as ferramentas
  - 4.2. Se não: Vá para o passo 17
5. Desapertar os parafusos da roda
6. Já desapertou todos os parafusos?
  - 6.1. Se sim: Vá para o passo 7
  - 6.2. Se não: Vá para o passo 5
7. Levantar o carro com o macaco
8. Levantou o suficiente?
  - 8.1. Se sim: Próximo passo
  - 8.2. Se não: Vá para o passo 7
9. Retirar os parafusos
10. Retirar a roda
11. Colocar o estepe
12. Recolocar os parafusos
13. Abaixar o carro
14. Apertar os parafusos
15. Já apertou todos os parafusos?
  - 15.1. Se sim: Vá para o passo 16
  - 15.2. Se não: Vá para o passo 14
16. Guardar as ferramentas
17. Guardar o pneu
18. Fechar o porta-malas
19. Fim

Os algoritmos de ações cotidianas geralmente permitem que uma pessoa os execute mesmo que ela nunca tenha feito tal coisa.

Uma receita de um bolo é um exemplo. Você pode fazer o bolo se seguir corretamente a receita. Agora, se a receita não estiver detalhada o suficiente, já não se pode garantir o resultado.

Um algoritmo detalhado mostra ações importantes como **Desligar o fogo**, no primeiro algoritmo, ou **Guardar o pneu**, no segundo algoritmo.

## 2.2. Elementos de um algoritmo

Veremos, a seguir, os elementos que compõem um algoritmo.

### 2.2.1. Ação

- Abrir o porta-malas do carro
- Retirar os parafusos
- Retirar a roda
- Colocar o estepe
- Recolocar os parafusos
- Abaixar o carro
- Guardar as ferramentas
- Guardar o pneu
- Fechar o porta-malas

### 2.2.2. Decisão

**Tem estepe?**

**Se sim: Retirar o estepe**

**Se não: Vá para o passo 18**

**Tem ferramentas?**

**Se sim: Pegar ferramentas**

**Se não: Vá para o passo 17**

**Observação:** Note que, numa decisão, existem duas respostas ou caminhos a seguir: o lado verdadeiro e o lado falso da indagação.

### 2.2.3. Laço ou loop

**5. Desapertar os parafusos da roda**

**6. Já desapertou todos os parafusos?**

**6.1. Se sim: Vá para o passo 7**

**6.2. Se não: Vá para o passo 5**

**Observação:** Note que, se não foram desapertados todos os parafusos, a execução do programa volta para a linha 5, fazendo um loop, que são trechos de uma lógica que podem ser executados várias vezes.

**7. Levantar o carro com o macaco**

**8. Levantou o suficiente?**

**8.1. Se sim: Próximo passo**

**8.2. Se não: Vá para o passo 7**

**Observação:** Note que, se o carro não foi suficientemente levantado, a execução do programa volta para a linha 7, fazendo um loop. Observe, também, que, em caso positivo, a execução do programa vai para o **Próximo passo**, que seria a mesma coisa que **Ir para o passo 9**, que é o próximo passo.

**14. Apertar os parafusos**

**15. Já apertou todos os parafusos?**

**15.1. Se sim: Vá para o passo 16**

**15.2. Se não: Vá para o passo 14**

**Observação:** Note que, se não foram apertados todos os parafusos, a execução do programa volta para a linha 14, fazendo um loop.

## 2.3. Algoritmo com o comando SE encadeado

Já vimos que o comando **SE** tem duas respostas, **SE SIM** e **SE NÃO**, mas pode ser que você precise de mais alternativas como resposta.

A solução é **encadear** o comando **SE**, colocando um dentro do outro.

- Exemplo: Menu principal de um programa

```
1. Início
2. Digite um número de 1 a 6
3. O número digitado é 1?
    3.1. Se sim: Abrir o programa de Inclusão
    3.2. Se não: O número digitado é 2?
        3.2.1. Se sim: Abrir o programa de Exclusão
        3.2.2. Se não: O número digitado é 3?
            3.2.2.1. Se sim: Abrir o programa de Consulta
            3.2.2.2. Se não: O número digitado é 4?
                3.2.2.2.1. Se sim: Abrir o programa de Alteração
                3.2.2.2.2. Se não: O número digitado é 5?
                    3.2.2.2.2.1. Se sim: Abrir o programa de Impressão
                    3.2.2.2.2.2. Se não: O número digitado é 6?
                        3.2.2.2.2.2.1. Se sim: Fechar Menu
                        3.2.2.2.2.2.2. Se não: Exibir "Número Inválido" e
                            "Ir para o passo 2"
4. Fim
```

## 2.4. Algoritmo com o comando CASO

Já vimos que o comando **SE** tem duas respostas, **SE SIM** e **SE NÃO**.

O comando **CASO** pode ter quantas respostas você precisar. Ele é usado como alternativa ao comando **SE** encadeado.

**Observação:** Nem todas as linguagens têm o comando **CASO**, e somente no comando **SE** existe **SE SIM** e **SE NÃO**.

- Exemplo: Menu principal de um programa

1. Início
2. Digite um número de 1 a 6
3. Caso
  - 3.1. Caso tenha digitado 1: Abrir o programa de Inclusão
  - 3.2. Caso tenha digitado 2: Abrir o programa de Exclusão
  - 3.3. Caso tenha digitado 3: Abrir o programa de Consulta
  - 3.4. Caso tenha digitado 4: Abrir o programa de Alteração
  - 3.5. Caso tenha digitado 5: Abrir o programa de Impressão
  - 3.6. Caso tenha digitado 6: Fechar Menu
  - 3.7. Caso contrário: Exibir “Número Inválido” e “Ir para o passo 2”
4. Fim do Caso
5. Fim

**Observação:** Note que, na linha 4, o comando **Fim do Caso** serve para indicar onde termina o comando **CASO**. Depois o programa continua normalmente a partir da próxima linha, que, neste caso, é o comando **Fim**, que serve para identificar o final do algoritmo.

## 2.5.Algoritmo com o comando ENQUANTO

O comando **ENQUANTO**, que, em inglês, significa **WHILE**, serve para que uma parte do programa seja repetida enquanto uma situação for satisfeita. O loop que repete as linhas de programação só é interrompido quando a condição que o faz repetir os comandos não for mais verdadeira.

Exemplo: Vamos supor que exista uma caixa com 30 bolas, sendo 10 verdes, 10 amarelas e 10 azuis. O objetivo é retirar bolas da caixa, sem olhar, até ter certeza de que saíram 3 bolas de cor azul. Para facilitar utilizaremos contadores de bolas, que inicialmente têm o valor zero (0), porque ainda nenhuma bola foi retirada.

**Início**

**Enquanto o contador de bolas azuis for menor que 03**

**Retirar uma bola**

**Caso**

**Caso a bola seja verde: Somar 1 no contador de bolas verdes**

**Caso a bola seja amarela: Somar 1 no contador de bolas amarelas**

**Caso contrário: Somar 1 no contador de bolas azuis**

**Fim do Caso**

**Fim do Enquanto**

**Exibir a mensagem “Três bolas azuis foram retiradas da caixa”**

**Fim**

Note que, dentro do comando **ENQUANTO**, foram colocados os comandos de retirar uma bola e o comando **CASO**, que verifica a cor das bolas somando 1 no contador da respectiva cor. Após o comando **Caso contrário: Somar 1 no contador de bolas azuis**, a execução volta diretamente para o comando **Enquanto o contador de bolas azuis for menor que 03**.

Após essa verificação, caso o contador de bolas azuis não seja, então, menor que 3, a execução vai para **Exibir a mensagem “Três bolas azuis foram retiradas da caixa”**. Caso contrário, a execução entra novamente no loop e retira mais uma bola.



## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **Algoritmo** é a descrição sequencial ordenada dos passos que devem ser executados, de forma lógica e clara, com a finalidade de facilitar a resolução de um problema;
- Os algoritmos são compostos por **ação**, **decisão** e **laço** ou **loop**;
- O comando **ENQUANTO (WHILE)** serve para que uma parte do programa seja repetida enquanto uma situação for satisfeita.





# 3

## Fluxograma

- Simbologia;
- Criação de fluxogramas;
- Teste de Mesa.

Erik Proença, 323.439.6880-45

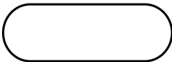
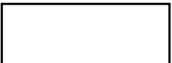
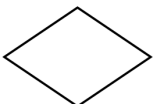

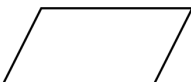








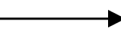
## 3.1. Introdução

**Fluxograma**, ou **Diagrama de Blocos**, é a representação gráfica de um algoritmo, sendo constituído de blocos funcionais que mostram o fluxo dos dados e as operações efetuadas com eles.

## 3.2. Simbologia

O fluxograma utiliza símbolos que permitem a descrição da sequência dos passos a serem executados de forma clara e objetiva. A tabela a seguir descreve a simbologia utilizada em fluxogramas:

Símbolo	Significado	Descrição
	<b>Terminação</b>	Utilizado para indicar o início, fim ou saída de um fluxograma.
	<b>Processamento</b>	Utilizado para indicar uma ação.
	<b>Decisão</b>	Utilizado para comparar dados e desviar o fluxo conforme o resultado seja verdadeiro ou falso.
	<b>Entrada manual</b>	Utilizado para a entrada de dados por meio do teclado.
	<b>Entrada / Saída</b>	Utilizado para indicar operações de leitura e gravação de registros.
	<b>Processamento predefinido ou Módulo</b>	Utilizado para indicar uma chamada a uma sub-rotina.
	<b>Documento</b>	Utilizado para indicar a impressão de dados.
	<b>Exibir</b>	Utilizado para exibir dados na tela.
	<b>Preparação</b>	Utilizado para a execução de looping.

Símbolo	Significado	Descrição
	<b>Conector</b>	Utilizado para continuar o fluxograma em outra parte na mesma página.
	<b>Conector de página</b>	Utilizado para continuar o fluxograma em outra página.
	<b>Seta de fluxo de dados</b>	Utilizado para indicar o sentido do fluxo de dados conectando os símbolos existentes.

### 3.3.Criando fluxogramas

A seguir, temos um algoritmo e o fluxograma de **como lavar as mãos com sabonete de barra**. Detalhes: Suponha que você já está em frente à pia e não falta nada para que possa lavar as mãos:

INÍCIO

Abrir a torneira

Molhar as mãos

Pegar o sabonete

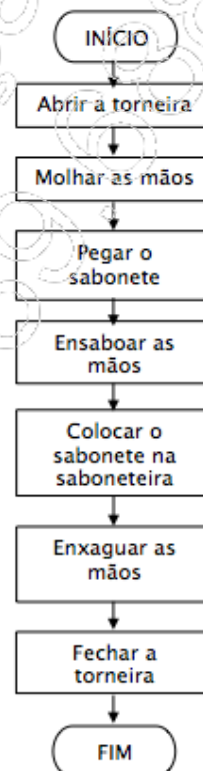
Ensaboar as mãos

Colocar o sabonete na saboneteira

Enxaguar as mãos

Fechar a torneira

FIM



**Observação:** O algoritmo anterior recebeu na primeira linha o texto **INÍCIO**, porém é comum usar o nome do programa. Por exemplo: Neste caso poderia ser **LAVAR AS MÃOS**. O fluxograma teria, então, no primeiro símbolo, que é o símbolo de inicialização, o nome **LAVAR AS MÃOS**. Veja, também, que o algoritmo não recebeu numeração, ou seja, ela não é obrigatória, servindo apenas para facilitar o entendimento.

## 3.3.1. Estruturas básicas

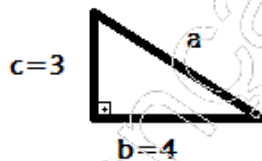
Vejam, a seguir, as estruturas básicas de um fluxograma.

- **Sequência**

No exemplo anterior, de **LAVAR AS MÃOS**, houve uma ação após a outra, portanto chamamos essa estrutura de **SEQUÊNCIA**.

Vejam outro exemplo. De acordo com o Teorema de Pitágoras, em qualquer triângulo retângulo, o quadrado do comprimento da hipotenusa é igual à soma dos quadrados dos comprimentos dos catetos. O triângulo retângulo pode ser identificado pela existência de um ângulo reto, ou seja, um ângulo medindo 90°. O triângulo retângulo é formado pela hipotenusa, que constitui o maior segmento do triângulo e é localizada opostamente ao ângulo reto, e por dois catetos.

No triângulo a seguir, a hipotenusa está representada pela variável **a**, um cateto pela variável **b**, que tem o valor 4, e o outro cateto pela variável **c**, que tem o valor 3.



Veja o algoritmo para efetuar o cálculo  $a^2 = b^2 + c^2$  e exibir o valor da hipotenusa:

### **HIPOTENUSA**

**Declara a, b, c, h numéricas**

**a = 0**

**b = 4**

**c = 3**

**h = 0**

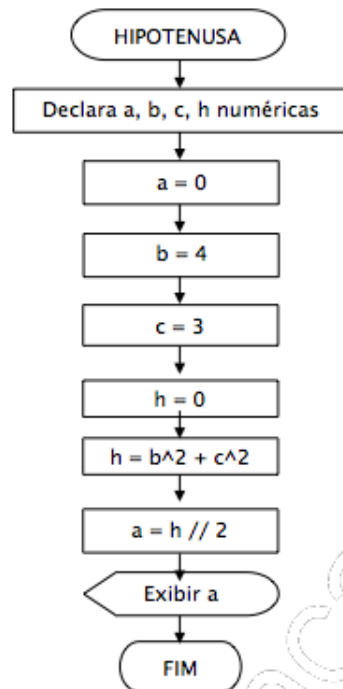
**h = b^2 + c^2**

**a = h // 2 ← Raiz quadrada de h para descobrir o valor da hipotenusa**

**Exibir a**

**FIM**

Note, no algoritmo, que foi executada uma sequência de ações. Veja, a seguir, como fica seu fluxograma:



A seguir, temos a simulação da execução do fluxo para descobrir o valor da hipotenusa:

a = 0  
b = 4  
c = 3  
h = 0  
h = b^2 + c^2  
h = 4^2 + 3^2  
h = 16 + 9  
h = 25  
a = 25 // 2  
a = 5  
Exibir a ← Exibe o valor 5

- **Condição/Seleção**

Esta estrutura permite representar uma condição e selecionar o fluxo a seguir dependendo do resultado da condição, se a condição é verdadeira ou falsa, podendo, assim, executar diferentes instruções. Para isso usaremos o comando **SE** e suas duas respostas, **ENTÃO** e **SENÃO** (IF, THEN, ELSE). O **ENTÃO** é a saída verdadeira, e o **SENÃO** é a saída falsa.

A seguir, temos um algoritmo e o fluxograma para exibir se um número digitado é par ou ímpar:

## NÚMERO

Declara X numérica

X = 0

Ler X

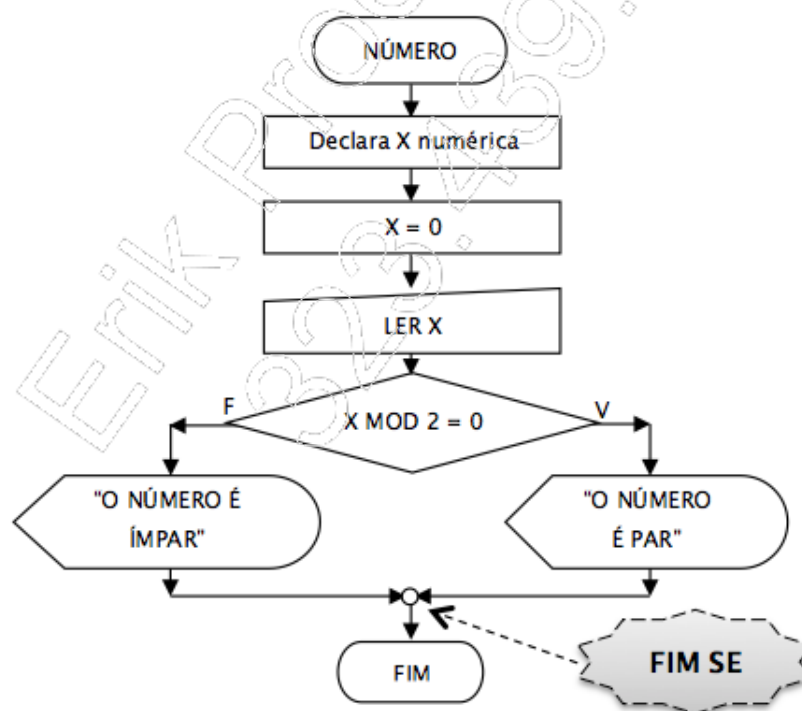
Se  $X \text{ MOD } 2 = 0$  ← Verifica se o resto da divisão de X por 2 é igual a zero

Então exibir mensagem "O número é par"

Senão exibir mensagem "O número é ímpar"

Fim Se

FIM



Note que depois do **FIM SE** só existe uma seta fluxo, ou seja, só existe um caminho a seguir. O comando **FIM SE** é a união das setas dos fluxos que vêm do lado verdadeiro e do lado falso da decisão.

- **Repetição condicional**

Esta estrutura permite representar uma condição e, dependendo do resultado desta, podem-se executar novamente algumas instruções.

Vejamos um exemplo de um algoritmo e de um fluxograma de como atravessar uma rua em um semáforo de pedestres.

## 1. ATRAVERSSAR

## 2. Observar o semáforo de pedestres

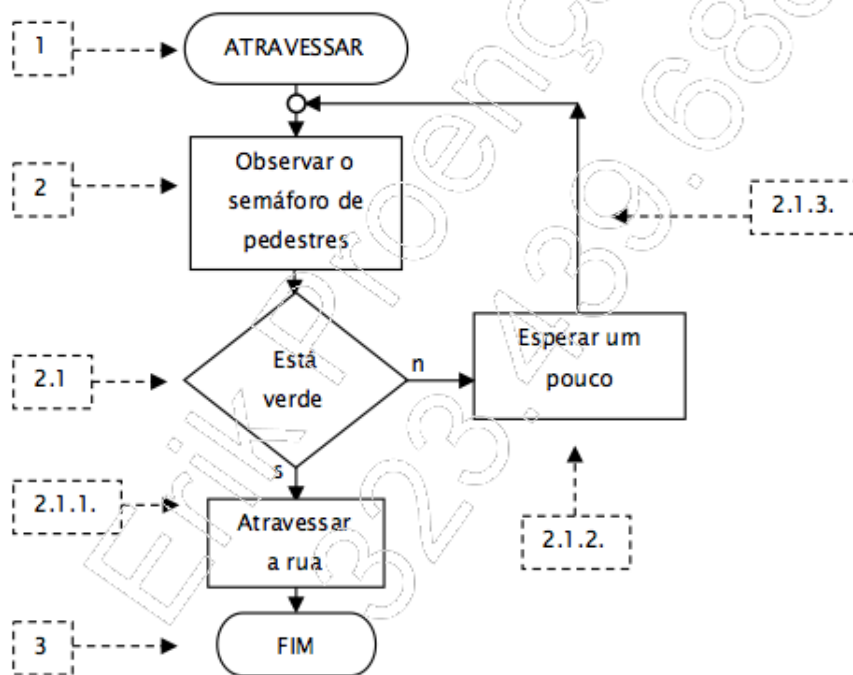
### 2.1. Verifique se está verde

2.1.1. Se sim: Atravessar a rua

2.1.2. Senão: Esperar um pouco

2.1.3. Vá para o passo 2

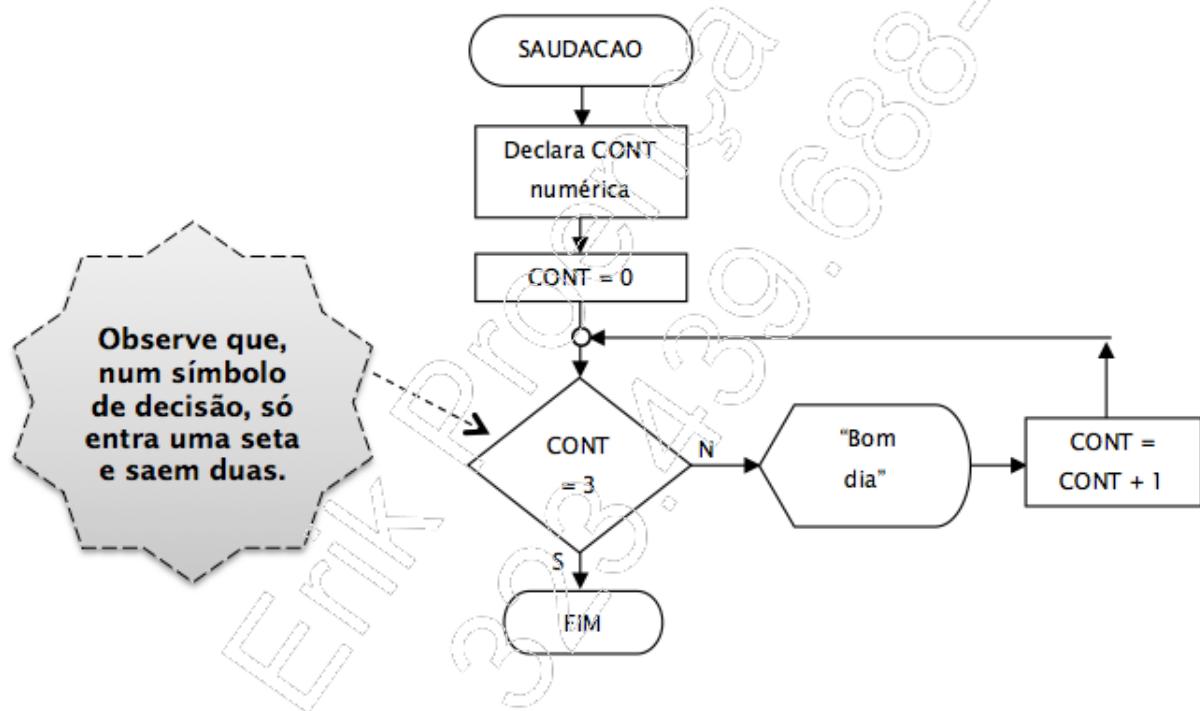
## 3. FIM



Note que o passo 2.1.3, **Vá para o passo 2**, é representado apenas pela seta de fluxo. A seta terminou no caminho que leva ao passo 2, porém poderia ter entrado diretamente no símbolo do passo 2.

A seguir, temos um exemplo de um algoritmo e de um fluxograma com um contador, cujo objetivo é exibir 3 (três) vezes a mensagem “Bom dia”:

1. SAUDACAO
2. Declara CONT numérica
3.  $CONT = 0$
4. Se  $CONT = 3$ 
  - 4.1. Então Vá para o passo 5
  - 4.2. Senão Exibir “Bom dia”
  - 4.3.  $CONT = CONT + 1$
  - 4.4. Vá para o passo 4
5. FIM





### 3.4. Teste de Mesa

**Teste de Mesa** é a simulação da execução de um algoritmo, programa ou fluxograma, sem utilizar o computador, empregando apenas lápis e papel.

A simulação da execução do fluxo para descobrir qual o valor da hipotenusa, conforme vimos anteriormente, representa o Teste de Mesa daquele fluxo.

Note no fluxo anterior que, inicialmente, o contador **CONT** recebeu o valor zero (0) e depois, na execução, recebeu os valores 1, 2 e 3.

**CONT = 0**

**CONT = CONT + 1 → CONT = 1**

**CONT = CONT + 1 → CONT = 2**

**CONT = CONT + 1 → CONT = 3**

O resultado mostrado representa o Teste de Mesa exibindo todos os valores que a variável **CONT** recebeu.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **Fluxograma**, ou **Diagrama de Blocos**, é a representação gráfica de um algoritmo, sendo constituído de blocos funcionais que mostram o fluxo dos dados e as operações efetuadas com eles;
- O fluxograma utiliza símbolos que permitem a descrição, de forma clara e objetiva, da sequência dos passos a serem executados;
- Numa programação, para efetuarmos uma decisão, podemos utilizar os comandos **SE**, **ENTÃO** e **SENÃO** (IF, THEN, ELSE). O **ENTÃO** é a saída verdadeira, e o **SENÃO** é a saída falsa do comando **SE**;
- O comando **FIM SE** é a união das setas dos fluxos que vêm do lado verdadeiro (**ENTÃO**) e do lado falso (**SENÃO**) da decisão. A partir deste ponto, só existe uma seta fluxo, ou seja, só existe um caminho a seguir;
- **Teste de Mesa** é a simulação da execução de um algoritmo, programa ou fluxograma.



# 4 JavaScript

- A linguagem JavaScript;
- Ambientes de desenvolvimento JavaScript;
- Camadas de desenvolvimento;
- Primeiros códigos em JavaScript;
- Orientação a objetos.



## 4.1. A linguagem JavaScript

JavaScript é uma linguagem de script orientada a objeto que pode ser executada em várias plataformas.



Logotipo da plataforma JavaScript (a partir de 2011)

Essa linguagem é leve, concisa e de fácil integração com outras aplicações, mas não é uma boa opção utilizá-la como linguagem independente. É possível, também, conectar a linguagem JavaScript a outros objetos existentes dentro de um ambiente para poder controlá-los de forma programática.

Você encontra um conjunto básico tanto de objetos quanto de elementos de linguagem na JavaScript, mas também pode adicionar objetos. No lado do cliente, é possível estender a linguagem básica ao disponibilizar objetos para controlar um navegador e seu DOM. Assim, elementos podem ser inseridos para executar determinadas ações, por exemplo. Já do lado do servidor, a extensão da linguagem se dá por meio de objetos relevantes para a execução de JavaScript no servidor. Essa extensão pode, por exemplo, viabilizar a comunicação entre uma aplicação e um banco de dados.

## 4.2. Ambientes de desenvolvimento JavaScript

O JavaScript é uma linguagem usada para desenvolvimento de aplicações front-end, ou seja, executadas no browser.

Para o desenvolvimento, podemos usar:

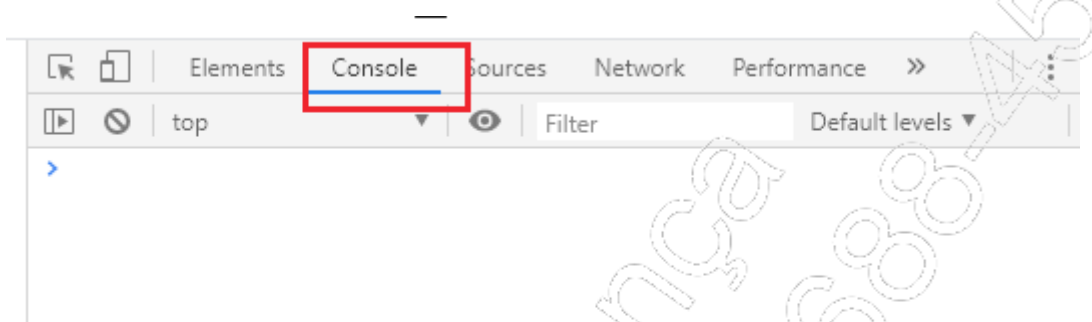
- **O browser (Google Chrome, por exemplo):** Neste caso, usamos o console do browser para testar instruções em JavaScript. É por meio do browser que avaliamos o resultado da execução de um programa em JavaScript quando, por exemplo, algum erro ocorre, ou mesmo para verificar o resultado de uma execução bem sucedida;
- **Página HTML:** Este é, de fato, o cenário mais comum. Uma página HTML é capaz de executar tarefas dinâmicas, cujo programa é interpretado pelo próprio browser. O programa em JavaScript é incorporado à página, permitindo sua interação com os elementos HTML. Via de regra, quando precisamos fornecer dados de entrada para o processamento de um programa em JavaScript, o fazemos por meio de campos de entrada fornecidos em páginas HTML. O processamento ocorre por meio de eventos;

- **Framework Node.js:** O Node.js é uma ferramenta poderosa para desenvolvimento de aplicações front-end, como Angular, React e outras, cuja base é o JavaScript. Com o Node.js é possível executarmos instruções JavaScript, exibindo-as no console, independente do browser.

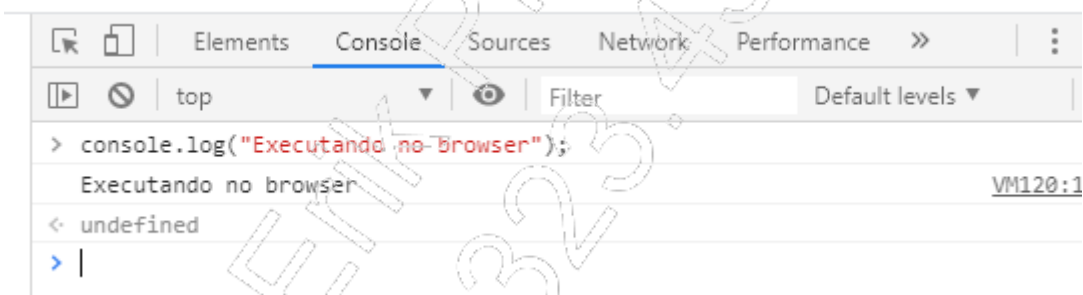
Para estudarmos os fundamentos da lógica de programação com JavaScript, vamos preferir o uso do Node.js. Em exemplos mais específicos, usaremos instruções baseadas em páginas HTML.

### 4.2.1. Executando instruções JavaScript por meio do browser

No browser, pressionando a tecla F12 temos acesso ao console:



No console podemos fornecer instruções e testá-las. O resultado é exibido no próprio console. Esse procedimento não deve ser usado para fins de desenvolvimento; apenas para testes.



## 4.2.2. Executando instruções JavaScript por meio de uma página HTML

Quando as instruções em JavaScript são usadas para interagir com páginas HTML, podemos escrever as instruções na própria página ou em um arquivo separado. Considere o trecho de código a seguir, que consiste em uma página com uma instrução em JavaScript. Veja que usamos o elemento `<script>` do HTML para esta finalidade:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script>

    function exibir(){
      let curso = 'Javascript';
      let duracao = 88;
      let ativo = true;

      console.log('Curso: ' + curso);
      console.log('Duração: ' + duracao);
      console.log('Ativo: ' + ativo);
    }

  </script>
</head>
<body>
  <button onclick="exibir();">Ver variáveis</button>
</body>
</html>
```

## 4.2.3. Executando instruções JavaScript por meio do Node.js

Para usarmos o Node.js, devemos instalá-lo em nosso computador.

O Node.js pode ser obtido por meio do link: [<https://nodejs.org/en/>](https://nodejs.org/en/).

Basta escolher a versão adequada, fazer o download e instalá-lo. Não é objetivo deste curso entrar em detalhes a respeito do framework Node.js. Vamos usá-lo como ferramenta de trabalho para executarmos instruções em JavaScript.

Existem outras ferramentas de desenvolvimento disponíveis no mercado.

Programas em JavaScript, assim como páginas HTML, podem ser elaboradas em qualquer editor de textos, como o Bloco de Notas do Windows, por exemplo. Porém, existem editores mais sofisticados, específicos para desenvolvimento de sistemas. Neste curso usaremos o Visual Studio Code (simplicadamente chamado de VSCode), que pode ser obtido em [<https://code.visualstudio.com/>](https://code.visualstudio.com/).

Durante o desenvolvimento dos programas, escolheremos uma pasta adequada no nosso computador, abriremos o VSCode apontando para esta pasta e criaremos arquivos com a extensão `.js`. Esse será o procedimento em todo o curso, a menos que seja indicado um processo diferente.

## 4.3. Camadas de desenvolvimento

As aplicações de Web hoje são geralmente construídas respeitando o conceito de desenvolvimento em camadas, que nada mais é do que a separação dos códigos de desenvolvimento em três camadas. Basicamente, se quiser seguir o conceito do desenvolvimento em camadas, você vai escrever os códigos de cada camada em arquivos separados e fazer a conexão deles por meio de links. Com isso, fica mais fácil reaproveitar pedaços de códigos em projetos novos, você vai ter menos retrabalho, e os códigos ficarão mais fáceis de entender, corrigir e administrar. As três camadas de desenvolvimento são as seguintes:

- Camada de estruturação de conteúdos, que consiste na marcação HTML;
- Camada de apresentação, representada pelas folhas de estilos;
- Camada de comportamento, onde entram os scripts que definem o comportamento das aplicações. É aqui que entra o JavaScript, e a interatividade com o usuário é inserida.

## 4.4. Primeiros códigos em JavaScript

Apresentaremos, a seguir, os primeiros passos na elaboração de programas em JavaScript. Usaremos o Node.js para executar os programas. Como editor, usaremos o Visual Studio Code (VSCode).

### 4.4.1. Escrevendo e executando um programa

- Crie um arquivo chamado `exemplo01.js`  

```
console.log("Primeiro programa");
```
- No prompt de comando, na mesma pasta onde salvou o arquivo, execute o comando:

```
node exemplo01.js
```

- O resultado mostrado é o seguinte:

```
Primeiro programa
```

Apesar de simples, o procedimento anterior ilustra a criação de um programa usando um editor de textos e sua execução através do Node.js, por meio do comando `node <nome_arquivo.js>`.

## 4.4.2. Comentários

Os códigos de comentários são linhas que não são interpretadas, usadas quando você quiser inserir observações em trechos de código, para facilitar o reconhecimento e a identificação de tarefas por parte do programador. São basicamente uma ou mais linhas, dentro de um código, que servem apenas como comentários para esclarecimentos gerais.

Em um código, você pode definir comentários de uma linha ou de múltiplas linhas. No primeiro caso, o comentário é representado pelos caracteres `//`. Já para múltiplas linhas, o comentário é iniciado com `/*` e finalizado com `*/`.

Por não exigir um limite para a quantidade de linhas, nos comentários de múltiplas linhas, você pode usar o comentário para outras finalidades, como para isolar trechos de códigos que não serão lidos, ajudar na identificação de erros em códigos amplos, entre outros fins.

Muitas vezes os comentários ajudam a interromper a execução de uma instrução sem ter que apagá-la. Quando for necessário recuperá-la basta remover o comentário.

Exemplo:

```
/*  
    Neste exemplo mostramos a instrução para  
    apresentar uma mensagem na tela.  
  
    Observe o comentário na segunda instrução  
*/  
  
console.log("Primeiro programa");  
//console.log("Fim do programa");
```

## 4.5. Orientação a objetos

JavaScript possui suporte a vários aspectos da programação orientada a objetos. Existem internos da linguagem, como é o caso do objeto **console**, a partir do qual chamamos a função **log()**.

É possível definirmos objetos personalizados no JavaScript, quando houver a necessidade de atendermos a tarefas específicas.

Teremos a oportunidade de explorar outros objetos implícitos da linguagem.



## Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- JavaScript é uma linguagem de script orientada a objeto que pode ser executada em várias plataformas. Essa linguagem é leve, concisa e de fácil integração com outras aplicações, mas não é uma boa opção utilizá-la como linguagem independente;
- As aplicações de Web hoje são geralmente construídas respeitando o conceito de desenvolvimento em camadas, que nada mais é do que a separação dos códigos de desenvolvimento em três camadas. Basicamente, se quiser seguir o conceito do desenvolvimento em camadas, você vai escrever os códigos de cada camada em arquivos separados e fazer a conexão deles por meio de links. Com isso, fica mais fácil reaproveitar pedaços de códigos em projetos novos, você vai ter menos retrabalho, e os códigos ficarão mais fáceis de entender, corrigir e administrar;
- JavaScript possui suporte a vários aspectos da programação orientada a objetos. Os objetos podem ser internos da linguagem, do ambiente de hospedagem ou personalizados, ou seja, criados por quem desenvolve a aplicação, e possuem propriedades e métodos.

Erik Proença 323.439.685-223



# 5

## Variáveis

- Definição de variável;
- Criação de variáveis;
- Palavras reservadas;
- Tipos de variáveis;
- Caracteres especiais;
- Concatenação;
- Constantes;
- Objetos globais.



## 5.1. O que é uma variável?

Uma variável é um espaço na memória onde um dado é armazenado. Você pode salvar, em uma variável, qualquer tipo de informação necessária para que os programas possam realizar suas tarefas. Os nomes de variáveis podem ser tão curtos quanto um só caractere ou descritivos como **nomecompleto**.

## 5.2. Criando variáveis

As variáveis no JavaScript podem começar com uma letra, um sublinhado (\_) ou um cifrão (\$) e, a partir do segundo caractere, podem apresentar números também. Podemos, também, usar caracteres ISO 8859-1 ou Unicode, como ü e â. As sequências de escape Unicode \uXXXX também podem ser usadas.

Lembre-se de que o JavaScript diferencia letras maiúsculas de minúsculas, ou seja, **variável** não é a mesma coisa que **Variável**.

Veja alguns exemplos de nomes válidos:

```
var nome = 'João',
    Nome = 'João Henrique',
    $salario = 9999.99,
    _departamento = 'Desenvolvimento de Sistemas'

console.log(nome);
console.log(Nome);
console.log($salario);
console.log(_departamento);
```

A execução do programa anterior produz o seguinte resultado:

```
João
João Henrique
9999.99
Desenvolvimento de Sistemas
```

### 5.2.1. Declarando variáveis

O processo de criar variáveis, no JavaScript, é chamado de **declaração**. Podemos declarar variáveis usando a palavra-chave **var** ou a palavra-chave **let**, como mostrado a seguir:

```
var nome_da_variavel;
let nome_da_variavel;
```

Variáveis declaradas com **var** possuem escopo global no código, e declaradas com **let** possuem escopo somente no bloco onde foram declaradas. Além disso, com **var** podemos redeclarar a variável, ao passo que com **let**, não podemos.

Mas também dá para declarar uma variável já com valores atribuídos. Dê uma olhada:

```
var treinamento = 'Javascript';  
let escola = 'Impacta';
```

Para atribuir um valor textual, não se esqueça de usar aspas antes e depois do valor.

## 5.2.2. Variáveis locais

Chamamos de variáveis locais aquelas declaradas dentro de uma função do JavaScript e só reconhecidas dentro dessa função. Quando a função é completada, as variáveis locais são imediatamente excluídas.

Como só são reconhecidas dentro da função onde são declaradas, você pode usar o mesmo nome para variáveis locais que estiverem dentro de funções diferentes sem problema algum.

## 5.2.3. Variáveis globais

Ao contrário das variáveis locais, as globais são aquelas declaradas fora de uma função, e têm escopo sobre toda a página Web. São excluídas quando você fecha a página.

Outra forma de criar uma variável global é atribuindo valores a variáveis que ainda não existem, ou seja, que ainda não tenham sido declaradas, como você vê no exemplo a seguir:

```
global = 'Impacta';
```

## 5.2.4. Aritmética com JavaScript

É possível realizar operações aritméticas com as variáveis do JavaScript. Dê uma olhada:

```
let salario = 10000;  
let aumento = 1.1;  
let salarioFinal = salario * aumento;  
  
console.log(salarioFinal);
```

## 5.3. Palavras reservadas

As palavras reservadas são palavras do JavaScript que o interpretador (browser) utiliza internamente. Na criação de variáveis, funções, métodos ou identificadores de objetos, você não deve usar palavras reservadas para a nomeação, já que variáveis ou funções com nomes de palavras reservadas podem gerar erros de código. São elas:

- |            |              |          |
|------------|--------------|----------|
| • break    | • finally    | • this   |
| • case     | • for        | • throw  |
| • catch    | • function   | • try    |
| • continue | • if         | • typeof |
| • debugger | • in         | • var    |
| • default  | • instanceof | • void   |
| • delete   | • new        | • while  |
| • do       | • return     | • with   |
| • else     | • switch     |          |

Tem, ainda, algumas palavras que não possuem funcionalidade atualmente, mas que já foram reservadas pelo ECMAScript, caso venham a ter alguma utilidade, e não podem ser usadas como identificadores, tanto no modo estrito quanto não estrito. São elas:

- |         |           |          |
|---------|-----------|----------|
| • class | • export  | • import |
| • enum  | • extends | • super  |

Por fim, estas palavras estão reservadas apenas para o modo estrito:

- |              |             |          |
|--------------|-------------|----------|
| • implements | • package   | • public |
| • interface  | • private   | • static |
| • let        | • protected | • yield  |

As palavras **null**, **true** e **false** são reservadas na ECMAScript para seus usos normais. Já a **const**, que também é reservada, tem uma função atribuída a ela em uma extensão não padrão no Mozilla e na maioria dos navegadores. Essa extensão pode virar padrão no futuro.

## 5.4. Tipos de variáveis

Uma variável pode armazenar diversas classes de informação, como textos, números inteiros, números reais, entre outros. Essas classes são conhecidas como **tipos de variáveis**, cada uma com características e usos próprios.

A seguir, você vai ver cada um dos tipos de variáveis do JavaScript:

- **Numéricas inteiras**

Essas variáveis armazenam números inteiros, como mostrado a seguir:

```
let inteiro = 50;  
console.log(inteiro);
```

- **Numéricas fracionárias (ponto flutuante)**

As variáveis numéricas fracionárias armazenam números fracionários. A casa decimal é marcada pelo ponto (.), como mostrado a seguir:

```
let flutuante = 1.5;  
console.log(flutuante);
```

- **Caractere**

Essas variáveis armazenam apenas um caractere. Ao definir uma variável desse tipo, o conteúdo armazenado precisa estar entre apóstrofes, como no próximo exemplo:

```
let caractere = 'a';  
console.log(caractere);
```

- **String**

Variáveis desse tipo armazenam uma sequência de caracteres. Na definição de strings, o conteúdo precisa estar entre aspas, como no código seguinte:

```
let texto = 'Javascript';  
console.log(texto);
```

Podemos usar aspas ou apóstrofes na definição de strings.

- **Booleanos**

Variáveis desse tipo armazenam estados, isto é, **true** (verdadeiro) ou **false** (falso), como demonstra o próximo código:

```
let booleano = true;
console.log(booleano);
```

- **null**

Os dados nulos são representados pela palavra-chave **null**. Uma variável receberá o valor **undefined** caso não tenha valor atribuído.

```
let nulo = null;
let indefinido;

console.log(nulo);
console.log(indefinido);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
null
undefined
```

## 5.4.1. Convertendo strings em números

A mistura de tipos de dados diferentes pode acontecer quando as variáveis estão sendo manipuladas. Podemos ter strings junto de números. Nesse caso, é uma tendência do JavaScript interpretar esses dados como strings.

```
let numero = 11;
let texto = "Onze " + numero;

console.log(numero);
console.log(typeof numero);

console.log(texto);
console.log(typeof texto);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
11
number
Onze 11
string
```



Na utilização de strings com números, o resultado obtido nem sempre é o esperado. Para evitar erros decorrentes da mistura de dados, existem funções de conversão. As duas principais delas são descritas a seguir:

- **parseInt()**

Para converter uma string em número inteiro, você pode usar essa função. No código, a variável ou string que você queira converter em número deve ser colocada entre parênteses, como mostra o exemplo seguinte:

```
let numero = "11";  
console.log(typeof numero);  
  
numero = parseInt(numero);  
console.log(typeof numero);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
string  
number
```

- **parseFloat()**

Essa função converte uma string em um número de ponto flutuante. A seguir, você vê um exemplo da utilização de **parseFloat()**:

```
let numero = "11.5";  
console.log(typeof numero);  
  
numero = parseFloat(numero);  
console.log(typeof numero);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
string  
number
```

## 5.5. Caracteres especiais

JavaScript faz uso de caracteres especiais em uma string. Um deles é o sinal de aspas, um caractere reservado dentro do JavaScript. Como sabemos, uma string deve ser declarada sempre entre aspas. Mas o programador pode querer inserir, dentro de uma string, uma palavra acompanhada do sinal de aspas. Para isso, deve usar a barra invertida (\) junto das aspas, e isso serve para os demais caracteres especiais.

A tabela a seguir descreve os tipos de caracteres especiais e os respectivos códigos utilizados no script:

Tipo	Código
Aspas (")	\"
Apóstrofo (')	\'
Barra invertida (\)	\\
Alimentação do formulário	\f
Retrocesso	\
Nova linha	\n
Tabulação	\t
Retorno de carro	\r

Veja um exemplo de uso de caracteres especiais:

```
let texto = 'John\'s Phone';  
console.log(texto);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

John's Phone

Alguns caracteres especiais, como o de nova linha, só funcionam dentro de caixas de alerta. O navegador reconhece tags HTML dentro do texto, quando você usa o comando **document.write** (faz a escrita diretamente na página HTML). Em caixas de alerta, as tags HTML não são reconhecidas.

## 5.6. Concatenação

No JavaScript, você pode somar strings e variáveis de todos os tipos. Isso é chamado de concatenação. JavaScript faz a conversão automática na concatenação de uma string com qualquer outro tipo de variável. Comandos do HTML também podem ser concatenados. Para isso, eles devem ser inseridos entre aspas, como mostrado adiante:

```
let texto = 'Javascript';
document.write("<h1>" + texto + "</h1>");
```

O exemplo a seguir demonstra a declaração e a concatenação de dados:

```
let status = true;
let texto = 'Status';
console.log(texto + ": " + status);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
Status: true
```

## 5.7. Constantes

As constantes seguem as mesmas regras das variáveis para declaração e nomeação, com a diferença de usarem a palavra-chave **const**, sempre necessária, mesmo para constantes globais. Se você não incluir a palavra-chave **const**, o JavaScript interpretará que aquilo é uma variável.

O nome escolhido para a constante não pode ser o mesmo de nenhuma variável ou função do mesmo escopo. Além disso, o valor de uma constante não pode ser alterado ou redeclarado enquanto o script estiver sendo executado.

```
const taxa = 6;
```

## 5.8. Objetos globais

Objeto global é onde ficam todas as propriedades e métodos da linguagem e do navegador que você está usando. Um objeto global é automaticamente criado sempre que você iniciar um ambiente de hospedagem do JavaScript. É no objeto global que ficam declaradas as variáveis globais, que você já estudou.

Para fazer referência ao objeto global, você pode usar a palavra-chave **this** (ou **window**), mas preste atenção para que ela esteja fora do corpo de uma função, caso contrário, a referência será feita a outro objeto da própria função, que chamamos genericamente de **call object**.

```
console.log(this);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
{}
```

## 5.8.1. Variáveis e propriedades dos objetos

As propriedades de um objeto são os valores de suas variáveis, ou seja, propriedades e variáveis de objetos são exatamente a mesma coisa.

## 5.8.2. null

A palavra-chave **null**, quando usada, mostra que não existe nenhum valor associado a uma variável, seja ela do tipo que for.

## 5.8.3. undefined

Diferente da **null**, a palavra-chave **undefined**, que é uma propriedade do objeto global, define um valor indefinido a uma variável. Isso acontece quando ela é declarada e não iniciada.

## 5.8.4. NaN

Outra propriedade do objeto global que você precisa conhecer é a **NaN**, que serve para representar um valor que não seja um número.

```
let x = "X";  
console.log(parseInt(x));
```

## 5.8.5. Infinity

A última propriedade do objeto global que você verá neste capítulo é a **infinity**, que representa um valor infinito positivo. O interpretador JavaScript consegue manipular os números encontrados entre  $-1.7976931348623157 \times 10^{308}$  e  $1.7976931348623157 \times 10^{308}$ .

```
let x = 1.7976931348623157 * Math.pow(10, 308);  
console.log(x);
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Uma variável é um espaço na memória onde um dado é armazenado. Você pode salvar, em uma variável, qualquer tipo de informação necessária para que os programas possam realizar suas tarefas. Os nomes de variáveis podem ser tão curtos quanto um só caractere ou descritivos como **nomecompleto**;
- O processo de criar variáveis, no JavaScript, é chamado de declaração. Para declarar uma variável sem valores, você usa a palavra-chave **var**. As variáveis no JavaScript podem começar com uma letra, um sublinhado (\_) ou um cifrão (\$) e, a partir do segundo caractere, podem apresentar números também;
- As palavras reservadas são palavras do JavaScript que o interpretador (browser) utiliza internamente. Na criação de variáveis, funções, métodos ou identificadores de objetos, você não deve usar palavras reservadas para a nomeação, já que variáveis ou funções com nomes de palavras reservadas podem gerar erros de código;
- Uma variável pode armazenar diversas classes de informação, como textos, números inteiros, números reais, entre outros. Essas classes são conhecidas como tipos de variáveis, cada uma com características e usos próprios. As variáveis podem ser numéricas inteiras, numéricas fracionárias, caracteres, strings, booleanos ou null;
- JavaScript faz uso de caracteres especiais em uma string. Um deles é o sinal de aspas, um caractere reservado dentro do JavaScript. Como sabemos, uma string deve ser declarada sempre entre aspas. Mas o programador pode querer inserir, dentro de uma string, uma palavra acompanhada do sinal de aspas. Para isso, deve usar a barra invertida (\) junto das aspas, e isso serve para os demais caracteres especiais;
- No JavaScript, você pode somar strings e variáveis de todos os tipos. Isso é chamado de concatenação. O JavaScript faz a conversão automática na concatenação de uma string com qualquer outro tipo de variável. Comandos do HTML também podem ser concatenados. Para isso, eles devem ser inseridos entre aspas;
- As constantes seguem as mesmas regras das variáveis para declaração e nomeação, com a diferença de usarem a palavra-chave **const**, sempre necessária, mesmo para constantes globais. Se você não incluir a palavra-chave **const**, o JavaScript interpretará que aquilo é uma variável;
- Objeto global é onde ficam todas as propriedades e métodos da linguagem e do navegador que você está usando. Um objeto global é automaticamente criado sempre que você iniciar um ambiente de hospedagem do JavaScript. É no objeto global que ficam declaradas as variáveis globais.





# 6

## Operadores

- Utilização de operadores em JavaScript;
- Expressões;
- Tipos de operadores;
- Precedência dos operadores.



## 6.1. Utilizando operadores em JavaScript

Os operadores são utilizados em praticamente todas as linguagens de programação existentes no mercado. Como você pode ver pelo próprio nome, estes operadores permitem realizar operações e cálculos, assim como você aprendeu em matemática, e são empregados especialmente durante o desenvolvimento dos programas.

O resultado dessas operações permite ao programa variar o seu comportamento conforme as informações obtidas.

Existem diferentes tipos de operadores, e estes podem ser destinados a operações simples ou complexas que envolvem operandos de tipos de dados distintos, por exemplo, textos ou números. A partir de agora, você vai ver mais detalhes sobre os tipos de operadores disponíveis na linguagem JavaScript.

## 6.2. Expressões

Quando tratamos de JavaScript (e, também, de outras linguagens de programação), chamamos de **expressão** qualquer unidade de código que é válida e que resulta em um valor.

Sabendo disso, considere que existam dois tipos de expressões. No primeiro, o valor resultante é atribuído a uma variável. No segundo, você tem apenas o valor que resulta da expressão.

Quando usamos operadores de atribuição, temos, por exemplo, o seguinte caso:

**x=3**

Isso define que a variável **x** possui o valor 3.

Já no caso de uma expressão como **8 - 5**, cujo resultado é **3**, você tem apenas o valor definido, pois ele não é atribuído a nenhuma variável. Nessas expressões são usados operadores simples, que você pode chamar simplesmente de operadores.

As expressões podem ser classificadas pelo tipo de valor que elas resultam. Veja a seguir:

Tipo de expressão	Resultado
<b>Aritmética</b>	Valor numérico, por exemplo, 1,61803399. Geralmente usa operadores aritméticos.
<b>String</b>	String de caracteres, por exemplo, "Impacta". Geralmente usa operadores de string.
<b>Lógica</b>	Valores verdadeiro ( <b>true</b> ) ou falso ( <b>false</b> ). Geralmente usa operadores lógicos.
<b>Objeto</b>	Um objeto. Geralmente usa operadores especiais.



## 6.3. Tipos de operadores

Os tipos de operadores disponíveis para a linguagem de programação JavaScript são os seguintes:

- Operadores de atribuição;
- Operadores de comparação;
- Operadores aritméticos;
- Operadores bitwise;
- Operadores lógicos;
- Operadores de string;
- Operadores especiais.

### 6.3.1. Operadores de atribuição

Com os operadores de atribuição, você pode armazenar informações nas variáveis de memória do sistema.

Veja os diferentes tipos de operadores de atribuição na tabela adiante:

Operadores de atribuição	Descrição
<b>Sinal de igualdade (=)</b>	Este operador, que indica apenas uma atribuição, faz com que a parte direita do sinal de igual seja atribuída à parte da esquerda. Ao passo que a parte da esquerda costuma receber uma variável onde se deseja salvar o dado, a parte da direita geralmente recebe os valores finais.
<b>Sinais de adição e igualdade (+=)</b>	Este operador, que indica uma atribuição com soma, faz com que a parte da direita seja somada com a parte da esquerda. O resultado da operação, por sua vez, é salvo na parte da esquerda.
<b>Sinais de subtração e igualdade (-=)</b>	Este operador é responsável por indicar uma atribuição com subtração.
<b>Sinais de multiplicação e igualdade (*=)</b>	Este operador tem a finalidade de indicar uma atribuição da multiplicação.
<b>Sinais de barra e igualdade (/=)</b>	Este operador é responsável por indicar uma atribuição da divisão.
<b>Sinais de porcentagem e igualdade (%=)</b>	Este operador permite obter o resto e, então, atribuí-lo.

Veja, agora, um exemplo que demonstra a utilização de operadores de atribuição:

```
let a = b = c = d = e = 100;
```

```
a += 5;  
b -= 5;  
c *= 5;  
d /= 5;  
e %= 5;
```

```
console.log('a: ', a);  
console.log('b: ', b);  
console.log('c: ', c);  
console.log('d: ', d);  
console.log('e: ', e);
```

Como resultado, temos o seguinte:

```
a: 105  
b: 95  
c: 500  
d: 20  
e: 0
```

## 6.3.2. Operadores de comparação

Utilizando os operadores de comparação, você pode realizar a comparação entre conteúdos de variáveis de memória.

Veja os diferentes tipos de operadores condicionais na tabela adiante:

Operadores condicionais	Descrição
Dois sinais de igualdade (==)	Este operador permite verificar se dois números são idênticos.
Sinais de exclamação e igualdade (!=)	Por meio deste operador, é possível comprovar se dois números são diferentes.
Sinal de maior (>)	Este operador, que indica "maior que", é responsável por retornar o valor <b>true</b> se o elemento da esquerda for maior que o da direita.
Sinais de maior e igualdade (>=)	Este operador, que indica "maior igual", retorna o valor <b>true</b> se o primeiro elemento for maior ou igual ao segundo.
Sinal de menor (<)	Este operador, que indica "menor que", é responsável por retornar o valor <b>true</b> quando o elemento da esquerda for menor que o da direita.
Sinais de menor e igualdade (<=)	Este operador, que indica "menor igual", retorna o valor <b>true</b> se o primeiro elemento for menor ou igual ao segundo.

Veja, agora, um exemplo que demonstra a utilização de operadores condicionais:

```
let a = 10;
let b = 20;
let c = 30;
let d = 40;

console.log(a == b);
console.log(b != c);
console.log(c > d);
console.log(d >= a);
console.log(a < b);
console.log(c <= d);
```

Como resultado, temos o seguinte:

```
false
true
false
true
true
true
```

### 6.3.3. Operadores aritméticos

Os diferentes tipos de operadores aritméticos disponíveis em JavaScript – os quais são empregados em referências indexadoras, cálculos e manuseio de strings – são descritos na tabela a seguir:

Operadores aritméticos	Descrição
<b>Sinal de adição (+)</b>	Este operador permite obter a soma de dois valores.
<b>Sinal de subtração (-)</b>	Este operador permite obter a diferença de dois valores.
<b>Sinal de multiplicação (*)</b>	Este operador é responsável por multiplicar dois valores.
<b>Sinal de barra (/)</b>	Este operador tem a finalidade de dividir dois valores.
<b>Sinal de porcentagem (%)</b>	Este operador permite obter o resto da divisão de dois valores.
<b>Dois sinais de adição (++)</b>	Este operador é utilizado com um único operando e permite obter o incremento em uma unidade.
<b>Sinal de decremento (--)</b>	Este operador é utilizado com um único operando e permite obter o decremento em uma unidade.

Veja um exemplo que demonstra a utilização dos operadores aritméticos:

```
let a = 10;
let b = 20;
let c = 30;
let d = 40;

console.log(a + b);
console.log(b - c);
console.log(c * d);
console.log(d / a);
console.log(a % b);
console.log(c++);
console.log(d--);
```

Como resultado, temos o seguinte:

```
30
-10
1200
4
10
30
40
```

## 6.3.4. Operadores bitwise

Os operadores bitwise trabalham com valores binários, ou seja, seus operandos são formados por conjuntos de 32 bits, ou dígitos, e cada um deles tem o valor de 1 ou 0. Uma operação bitwise vai sempre fornecer uma resposta em valor numérico padrão, mesmo tendo as operações realizadas em valores binários.

### 6.3.4.1. Operadores bitwise lógicos

Os operadores bitwise lógicos trabalham com pares destes operandos binários de 32 dígitos, sendo que, em um operando, cada dígito corresponde a outro dígito do outro operando, em ordem respectiva.

Os operadores bitwise lógicos são os seguintes:

Operador	Sintaxe	Descrição
<b>AND Bitwise</b>	<code>a &amp; b</code>	Define o valor 1 em cada posição onde ambos os operandos possuem o valor 1.
<b>OR Bitwise</b>	<code>a   b</code>	Define o valor 1 em cada posição onde é encontrado o valor 1, seja em um operando ou no outro, ou em ambos.
<b>XOR Bitwise</b>	<code>a ^ b</code>	Define o valor 1 para cada posição onde, correspondentemente, exista o valor 1 em <b>apenas</b> um dos operandos.
<b>NOT Bitwise</b>	<code>~ a</code>	Define o valor final como o inverso do valor inicial, ou seja, valores 1 tornam-se 0 e vice-versa.

Considere como exemplo os valores binários de 13 e 8, que são, respectivamente, 1101 e 1000. Veja como os operadores bitwise lógicos funcionam:

Expressão	Resultado	Descrição binária
<code>13 &amp; 8</code>	8	<code>1101 &amp; 1000 = 1000</code>
<code>13   8</code>	13	<code>1101   1000 = 1101</code>
<code>13 ^ 8</code>	5	<code>1101 ^ 1000 = 0101</code>
<code>~13</code>	-14	<code>~1101 = 0010</code>
<code>~8</code>	-9	<code>~1000 = 0111</code>

Veja um exemplo que demonstra a utilização dos operadores bitwise lógicos:

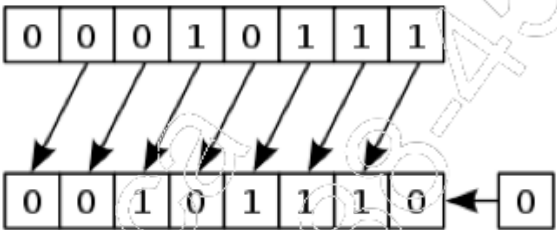
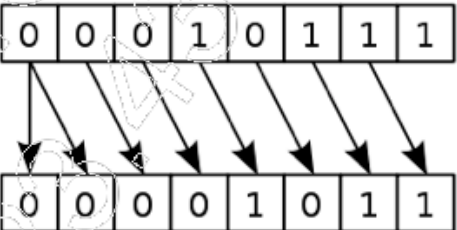
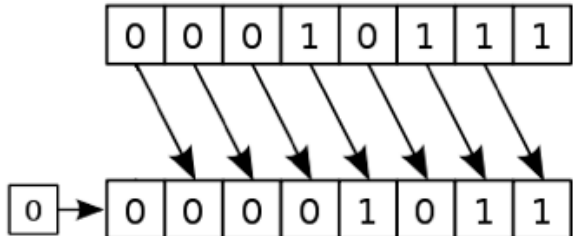
```
let a = 13;
let b = 8;

console.log(a & b);
console.log(a | b);
console.log(a ^ b);
console.log(~a);
console.log(~b);
```

## 6.3.4.2. Operadores bitwise de deslocamento

Você vai encontrar três tipos de operadores bitwise de deslocamento, e seu funcionamento é muito simples.

Nos três tipos de operadores, destacados logo adiante, você deve trabalhar com dois tipos de valores: o valor a ser deslocado e a quantidade de dígitos que este valor sofre de deslocamento. É aí que entram os operadores, que definem para que lado esse deslocamento ocorrerá (direito ou esquerdo) e o que ocorre com os dígitos afetados. Veja na tabela como isso funciona:

<b>Deslocamento para a esquerda</b>	$a \ll b$	<p>Desloca o valor <b>a</b>, em representação binária, <b>b</b> dígitos para a esquerda, preenchendo os dígitos à direita com <b>0</b>.</p> 
<b>Deslocamento para a direita</b>	$a \gg b$	<p>Desloca o valor <b>a</b>, em representação binária, <b>b</b> dígitos para a direita, descartando os valores dos dígitos deslocados e copiando o valor do último dígito da esquerda na esquerda.</p> 
<b>Deslocamento para a direita preenchendo com zeros</b>	$a \ggg b$	<p>Desloca o valor <b>a</b>, em representação binária, <b>b</b> dígitos para a direita, descartando os valores dos dígitos deslocados e inserindo <b>0</b> à esquerda.</p> 

Veja um exemplo que demonstra a utilização dos operadores bitwise de deslocamento:

```
let a = 10;
let b = 20;

console.log(a << b);
console.log(a >> b);
console.log(a >>> b);
```

### 6.3.5. Operadores lógicos

Os operadores lógicos permitem que você realize operações lógicas, que nada mais são do que operações que geram como resultado um valor booleano, ou seja, verdadeiro (**true**) ou falso (**false**). Este operador é muito útil quando é necessário tomar decisões nos scripts.

**Nota:** É importante ressaltar que os operandos booleanos são relacionados pelos operadores lógicos no intuito de gerar como resultado outro operando booleano.

Veja, na tabela adiante, os operadores lógicos e suas respectivas descrições:

Operadores lógicos	Descrição	Exemplo x==50 e y==5
<b>&amp;&amp; (e)</b>	Quando este operador é utilizado em um teste lógico, é necessário que todas as expressões lógicas sejam avaliadas como verdadeiras para que o resultado seja verdadeiro ( <b>true</b> ).	(x==50)&&(y<50) é verdadeiro.
<b>   (ou)</b>	Quando este operador é utilizado em um teste lógico, é necessário que apenas uma expressão lógica seja avaliada como verdadeira para que o resultado seja verdadeiro ( <b>true</b> ).	(x==50)   (y==50) é verdadeiro.
<b>! (não)</b>	Com este operador, a lógica da expressão é invertida, ocasionando uma negação.	x!=y é verdadeiro.

Veja um exemplo que demonstra a utilização dos operadores lógicos:

```
let a = 10;
let b = 20;

console.log(a > 15 && b > 15);
console.log(a > 15 || b > 15);
console.log(!a == 20);
```

Como resultado, temos o seguinte:

```
false  
true  
false
```

## 6.3.5.1. Avaliação de curto-circuito

Existem algumas regras que tornam possível que você avalie uma expressão lógica para ver se ela sofrerá o que chamamos de "curto-circuito". Isso ocorre pois as expressões são processadas da esquerda para a direita, e alguns resultados não precisam passar por certos processos, visto que sempre terão o mesmo resultado segundo a lógica. Essas regras são:

- **falso (false) && qualquer valor:** Resulta sempre em falso (**false**);
- **verdadeiro (true) || qualquer valor:** É sempre verdadeiro (**true**).

Sendo assim, nesses dois casos específicos, qualquer valor que vier depois dos operadores **&&** ou **||** não precisam ser processados, pois a resposta será sempre a mesma.

## 6.3.6. Operadores de string

Para lidar com expressões de string, você pode utilizar os operadores de comparação, já vistos neste mesmo capítulo, porém, com valores de string. Além destes, um operador que é muito utilizado com strings é o concatenador (sinal de +). Com ele, você pode unir diferentes valores de string e ter como resultado uma string que é a junção destes.

Por exemplo: Usando o concatenador nas strings **"Impacta"** + **"Tecnologia"** você obtém como resultado a string **"Impacta Tecnologia"**.

Outra forma de concatenar strings em JavaScript é utilizando o operador **+=**. Nesse caso, ele pega uma string armazenada em uma variável, soma com outra string e associa o valor final à mesma variável da primeira string. Veja o exemplo:

Considere uma variável **novastring** que possui o valor **"Java"**. Usando a variável na expressão **novastring += "Script"**, você obtém como resultado a string **"JavaScript"**, que fica armazenada na mesma variável **novastring**.



### 6.3.7. Operadores especiais

Além dos operadores que você já viu anteriormente neste capítulo, a linguagem JavaScript apresenta alguns operadores chamados especiais. São eles:

- Operador condicional;
- Operador separador;
- delete;
- in;
- instanceof;
- new;
- this;
- typeof;
- void.

#### 6.3.7.1. Operador condicional

Com o operador condicional, você pode definir uma condição e dois valores diferentes, que serão utilizados com base nessa mesma condição. É o único operador da linguagem JavaScript que utiliza três operandos.

Veja a sintaxe:

```
condição ? val1 : val2
```

Em que:

- **condição:** É uma condição que será avaliada pela expressão;
- **val1:** Valor que o operador assume caso a condição definida seja verdadeira;
- **val2:** Valor que o operador assume caso a condição não seja verdadeira.

O operador condicional pode ser usado em qualquer lugar onde possa ser usado um operador padrão.

Veja um exemplo que demonstra a utilização do operador condicional:

```
let resultado = (new Date().getDay() > 3)?"Maior que três":"Menor que três";
console.log(resultado);
```

## 6.3.7.2. Operador separador

Este operador é muito comum quando utilizado dentro de um loop **for**. Ele retorna sempre o valor do seu segundo operando, depois de calcular ambos, funcionando, assim, como um loop. Dentro do loop **for**, ele permite a atualização de múltiplas variáveis.

Veja um exemplo que demonstra a utilização do operador separador:

```
let a = [1, 2, 3, 4];
let b;

for (let i = 0; i < a.length, b = a[i]; i++) {
    console.log(a[i], b);
}
```

## 6.3.7.3. delete

Quando você deseja apagar um objeto, a propriedade de um objeto ou um elemento em um índice específico, utilize o operador **delete**. Ele vai retornar o valor **true** (verdadeiro) caso seja realmente possível realizar a ação, ou **false** (falso) caso não seja permitido.

Ele possui as seguintes sintaxes:

```
delete nomeObjeto;

delete nomeObjeto.propriedade;

delete nomeObjeto[índice];

delete propriedade;
```

Em que:

- **nomeObjeto**: O nome do objeto que você deseja inserir na expressão;
- **propriedade**: Qualquer propriedade existente;
- **índice**: Um número que representa a localização de um elemento em um array.

A última sintaxe pode ser utilizada apenas para deletar uma propriedade de um objeto dentro de uma declaração **with**.

Variáveis declaradas com a declaração **var** não podem ser apagadas com o operador **delete**, apenas as declaradas implicitamente.

Veja um exemplo que demonstra a utilização do operador **delete**:

```
let items = ['a', 'b', 'c'];
delete items[1];
console.log(items);
```

Após salvar e executar o código anterior, temos o seguinte resultado:

```
[ 'a', <1 empty item>, 'c' ]
```

### 6.3.7.4.in

Este operador indica se certa propriedade encontra-se em um objeto definido na expressão. Veja como o operador **in** funciona:

```
propNome in nomeObjeto
```

Em que:

- **propNome**: Pode ser uma string que representa um nome de propriedade ou mesmo um índice de array;
- **nomeObjeto**: É o nome do objeto a ser utilizado.

Veja um exemplo que demonstra a utilização do operador **in**:

```
let Pessoa = {
  nome: 'João',
  idade: 23
};

for(propriedade in Pessoa){
  console.log(propriedade);
}
```

## 6.3.7.5. instanceof

Para verificar se certo objeto faz parte de um tipo definido de objeto, você deve utilizar o operador **instanceof**. Veja sua sintaxe:

```
nomeObjeto instanceof tipoObjeto
```

Em que:

- **nomeObjeto**: Nome do objeto a ser analisado;
- **tipoObjeto**: Tipo do objeto (**Array**, **Date** etc.) com o qual **nomeObjeto** será comparado.

O operador **instanceof** normalmente é utilizado para confirmar o tipo de um objeto durante a execução do código.

Veja um exemplo que demonstra a utilização do operador **instanceof**:

```
let frutas = ['Banana', 'Maçã', 'Abacaxi'];  
console.log(frutas instanceof Array);
```

## 6.3.7.6. new

O operador **new** tem a função de criar uma ocorrência, seja de um dos tipos predefinidos de objetos, seja de tipos criados pelo usuário. Também é possível usar o operador **new** no servidor, com **DbPool**, **Lock**, **File** e **SendMail**. Veja a sintaxe:

```
var nomeObjeto = new tipoObjeto([parametro1,  
parametro2, ..., parametroN]);
```

Veja um exemplo que demonstra a utilização do operador **new**:

```
let navegadores = new Array('Chrome', 'Firefox', 'Safari');
```

## 6.3.7.7. this

Para acessar o objeto atual, utilize o operador **this**, que normalmente se refere ao objeto de chamada em um método. Veja a sintaxe:

```
this["nomePropriedade"]
```

```
this.nomePropriedade
```

Veja um exemplo que demonstra a utilização do operador **this**:

```
let Pessoa = function(nomePessoa){  
    this.nome = nomePessoa;  
}  
  
let a = new Pessoa('Gerson');  
let a = new Pessoa('Lolita');
```

### 6.3.7.8. typeof

O operador **typeof** fornece em forma de string o tipo do operando inserido, que pode ser uma string, variável, palavra-chave ou objeto.

Existem duas sintaxes possíveis para o operador **typeof**:

```
typeof operando
```

```
typeof (operando)
```

Veja um exemplo que demonstra a utilização do operador **typeof**:

```
let navegadores = new Array('Chrome', 'Firefox', 'Safari');  
let x = 100;  
  
console.log(typeof navegadores);  
console.log(typeof x);
```

## 6.4. Precedência dos operadores

A ordem de resolução para uma expressão que possui vários operadores e operandos é definida com base na precedência dos operadores. Portanto, deve ser efetuada a operação que apresenta maior precedência em primeiro lugar, e assim por diante.

A tabela adiante mostra a ordem de prioridade dos operadores:

Ordem de prioridade dos operadores
() [] .
++ -- ~ !
* / %
+ -
> >= < <=
== !=
&
^
&&
? :
=

## Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- Operadores são utilizados em praticamente todas as linguagens de programação existentes no mercado. Permitem realizar operações e cálculos e são empregados especialmente durante o desenvolvimento dos programas;
- Qualquer unidade de código que é válida e que resulta em um valor é chamada de expressão. Considere que existam dois tipos de expressões: no primeiro, o valor resultante é atribuído a uma variável; no segundo, você tem apenas o valor que resulta da expressão;
- Os tipos de operadores disponíveis para a linguagem de programação JavaScript são os seguintes: operadores de atribuição, operadores de comparação, operadores aritméticos, operadores bit a bit, operadores lógicos, operadores de string, operadores especiais;
- A linguagem JavaScript apresenta alguns operadores chamados especiais. São eles: operador condicional, operador separador, delete, in, instanceof, new, this, typeof, void.







# 7

## Declarações

- Estruturas condicionais;
- Estruturas para loops.

Erik Proença, 323.429.6000-45



## 7.1. Introdução

O JavaScript é responsável por grande parte dos recursos interativos nas páginas HTML. Para incorporação desses recursos, faz uso de um grupo de declarações com características e funcionalidades específicas que serão apresentadas no decorrer deste capítulo.

O controle do fluxo de um programa é feito por bloco de declarações, ou seja, uma sequência de declarações entre chaves ({}).

Uma das características dessas declarações em bloco é o aninhamento. Por meio do aninhamento é possível agrupar blocos de declarações em outro bloco.

As estruturas condicionais e as estruturas para loop constituem blocos de declarações, que serão apresentados a seguir.

Ao escrever as declarações ou criar variáveis, objetos e funções, você deve estar atento ao uso de maiúsculas, pois ao contrário da HTML, o JavaScript diferencia maiúsculas de minúsculas.

### 7.1.1. Declarações em JavaScript

Na tabela a seguir, apresentamos as declarações para estruturas JavaScript e suas respectivas finalidades:

Declaração	Finalidade
<b>var</b>	Define uma variável.
<b>function</b>	Define uma função.
<b>return</b>	Retorna um valor.
<b>if/else</b>	Cria uma estrutura condicional.
<b>switch</b>	Cria uma estrutura condicional.
<b>case</b>	Usada na estrutura condicional <b>switch</b> .
<b>break</b>	Usada na estrutura condicional <b>switch</b> .
<b>default</b>	Usada na estrutura condicional <b>switch</b> .
<b>for</b>	Cria um loop.
<b>continue</b>	Reinicia um loop.
<b>while</b>	Cria uma estrutura de repetição.

Declaração	Finalidade
<b>do/while</b>	Cria uma estrutura de repetição.
<b>for/in</b>	Cria um loop em objeto.
<b>throw</b>	Sinaliza erros.
<b>try/catch/finally</b>	Trata erros.
<b>with</b>	Altera escopo.
<b>;</b>	É uma declaração vazia.

## 7.2. Estruturas condicionais

Para que você decida qual ação será realizada em uma parte específica de um programa, é necessário solicitar que o programa avalie uma determinada expressão. Se essa expressão for avaliada como verdadeira, uma sequência de declarações será executada.

Então, as estruturas de condição a serem consideradas são:

- **if/else;**
- **switch/case.**

### 7.2.1. Declaração if/else

A instrução **if** avalia somente expressões com resultados booleanos. Caso a avaliação tenha um resultado verdadeiro, será executado um bloco de declarações localizado entre chaves. Caso contrário, ou seja, se o resultado for falso, o bloco de declarações não será executado ou, ainda, pode ser executado outro bloco de declarações. Veja a sintaxe a seguir:

```
if (Teste Condicional){  
  comando  
}
```

Agora veja um exemplo da utilização de **if**:

```
let idade = 18;  
if (idade >= 18) {  
  console.log('Maior de idade');  
}
```

Ao utilizar **if**, o emprego de chaves não é obrigatório se o bloco de declarações possuir uma única instrução. Mesmo assim, é melhor que você use as chaves, pois proporcionam uma legibilidade melhor. Por esse mesmo motivo, recomenda-se que as instruções **if/else** sejam moderadamente aninhadas.

A cláusula **else** deve ser utilizada ao executar declarações, cuja condição é falsa. Para isso:

1. Digite **else** após as instruções da condição verdadeira;
2. Em seguida, na linha abaixo, insira o bloco de instruções a ser executado. Veja a sintaxe:

```
if (Teste Condicional){  
  comando 1  
}  
else{  
  comando 2  
}
```

Veja um exemplo da utilização das instruções **if/else**:

```
let idade = 18;  
if (idade >= 18) {  
  console.log('Maior de idade');  
} else {  
  console.log('Menor de idade');  
}
```

Ao salvar e executar o código anterior, temos o seguinte:

Maior de idade

## 7.2.2. Declaração **switch/case**

Você também pode adotar a instrução **switch** para simular o uso de diversas instruções **if**. Veja sintaxe de **switch/case**:

```
switch (expressão){  
  case valor1:  
    instruções;  
    break;  
  case valor2:  
    instruções;  
    break;  
}
```

**Switch** verifica uma relação de igualdade. Deste modo, ao executá-lo, o programa pode encontrar uma declaração **break**, a qual indica que a instrução seguinte ao bloco **switch** deve ser executada. Portanto, a instrução **case** será executada enquanto **switch** não for finalizado ou até encontrar uma declaração **break**, caso o código-fonte não a possua.

Assim, enquanto é executado, é possível que o programa passe de uma instrução **case** para outra. Esse procedimento é conhecido como passagem completa, na qual todas as instruções **case** são executadas até o final de **switch**.

Veja um exemplo de **switch/case**:

```
let semana = 3, resultado;
switch(semana){
  case 1: resultado = 'Domingo'; break;
  case 2: resultado = 'Segunda'; break;
  case 3: resultado = 'Terça'; break;
  case 4: resultado = 'Quarta'; break;
  case 5: resultado = 'Quinta'; break;
  case 6: resultado = 'Sexta'; break;
  case 7: resultado = 'Sábado'; break;
}
console.log(resultado);
```

Você visualizará o seguinte resultado:

Terça

No exemplo, a declaração **break** é usada para separação do que for restritamente executado em cada **case**, bem como para finalizar o **case** anterior.

Você ainda pode usar a declaração **default** para concluir a utilização de **switch/case**. Essa declaração será executada sempre que um valor **case** correspondente não for identificado pelo valor assumido pela variável não.

```
let semana = 3, resultado;
switch(semana){
  case 1: resultado = 'Domingo'; break;
  case 2: resultado = 'Segunda'; break;
  case 3: resultado = 'Terça'; break;
  case 4: resultado = 'Quarta'; break;
  case 5: resultado = 'Quinta'; break;
  case 6: resultado = 'Sexta'; break;
  case 7: resultado = 'Sábado'; break;
  default: resultado = 'Dia incorreto'; break;
}
console.log(resultado);
```

## 7.3. Estruturas para loops

As estruturas para loops são declarações usadas para executar um bloco de códigos repetidas vezes, até se atingir certo número de execuções ou até que uma determinada condição seja verdadeira.

Em JavaScript, as estruturas para loops são:

- **while;**
- **do/while;**
- **for;**
- **for/in.**

Também é possível usar **break** e **continue** nas estruturas para loop.

### 7.3.1. Declaração while

Quando você não souber a quantidade de vezes que um bloco de instruções deve ser repetido, use a declaração para repetição **while**.

Com a declaração **while**, uma instrução é continuamente executada até que uma condição seja verdadeira. A condição analisada retorna um valor booleano.

A declaração **while** não será executada se uma condição for falsa na primeira vez em que é verificada. Nesse caso, será executada a instrução seguinte ao loop.

Observe a sintaxe:

```
while (teste condicional){  
  declarações //será executado enquanto teste condicional = true  
}
```

Pela sintaxe, você deve ter notado que somente se a condição for verdadeira o corpo da estrutura será executado.

Veja o exemplo a seguir:

```
let x = 0;  
while (x < 5) {  
  console.log(x++);  
}
```

Veja o resultado:

```
0
1
2
3
4
```

### 7.3.2. Declaração do/while

Apesar de ter fundamentalmente o mesmo funcionamento de **while**, com **do/while**, independentemente de a condição ser verdadeira ou não, as declarações são executadas pelo menos uma vez. Observe a sintaxe:

```
do{
  declaração
}
while(condição)
```

Agora, veja o exemplo:

```
let x = 0;
do {
  console.log(x++);
} while (x < 0);
```

Veja o resultado:

```
0
```

### 7.3.3. Declaração for

Se você souber exatamente a quantidade de vezes que deseja repetir um bloco de instruções, use a declaração **for**.

A declaração **for** é formada pelo corpo do loop, bem como pelas seguintes partes:

- **Declaração e inicialização de variáveis**

Se, na primeira parte de **for**, houver uma ou mais variáveis, estas poderão ser declaradas ou inicializadas. Para isso, você deve inseri-las entre parênteses, após a palavra-chave **for**. Além disso, use vírgulas para separar variáveis do mesmo tipo. Veja:

! A declaração e a inicialização de variáveis em **for** sempre ocorrem antes das outras declarações. Além disso, ocorrem uma única vez no loop, diferentemente da expressão de iteração e do teste booleano, que são executados a cada loop.

- **Expressão condicional**

A expressão condicional se localiza na segunda parte da instrução do loop. Ela é um teste e, quando executada, retorna um valor booleano. Por isso você deve especificar uma expressão lógica como expressão condicional. Porém seja cauteloso com códigos que usam várias expressões lógicas (a complexidade pode ser uma característica de uma expressão lógica).

Veja o exemplo de código seguinte, cuja expressão condicional é válida:

```
let dias = ['Dom', 'Seg', 'Ter', 'Qua', 'Qui', 'Sex', 'Sab'];
for (let i = 0, diaAtual; i < dias.length, diaAtual = dias[i]; i++) {
    console.log(diaAtual);
}
```

Você visualizará o seguinte resultado:

Dom  
Seg  
Ter  
Qua  
Qui  
Sex  
Sab

- **Expressão de iteração**

Embora seja a última a ser executada na instrução do loop **for**, esta expressão é sempre processada após a execução do corpo do loop. É nela que você especificará o que deve ocorrer assim que o corpo do loop de repetição for executado.

Observe a sintaxe completa da instrução **for**:

```
for (declaração e inicialização de variável; condição;
    iteração){
    instrução do corpo do loop for;
}
```

Se você optar por não declarar uma dessas três partes de **for**, o que não é indicado, pode ter como consequências:

- Loop infinito;
- Loop de repetição **for** atuando como um loop de repetição **while** caso não apresente seções de declaração e inicialização.



Veja o próximo exemplo:

```
let dias = ['Dom', 'Seg', 'Ter', 'Qua', 'Qui', 'Sex', 'Sab'];
for (let i = 0, diaAtual; i < dias.length, diaAtual = dias[i]; i++) {
    if(diaAtual == 'Qui'){
        break;
    }
    console.log(diaAtual);
}
```

Você visualizará o seguinte resultado ao usar o código anterior:

Dom  
Seg  
Ter  
Qua

Repare, também, que as seções da instrução **for** são independentes e, por isso, não precisam operar sobre as mesmas variáveis.

Uma expressão de iteração não tem necessariamente como função a configuração ou incrementação de algo. Observe o exemplo a seguir:

```
let dias = ['Dom', 'Seg', 'Ter', 'Qua', 'Qui', 'Sex', 'Sab'];
let i = 2, len = dias.length;
for (; i < dias.length;) {
    console.log(dias[i]);
    i++;
}
```

Veja o resultado:

Ter  
Qua  
Qui  
Sex  
Sab

### 7.3.4. Declaração for/in

A declaração **for/in** repete uma variável especificada em todas as propriedades de um objeto. Para cada uma dessas propriedades, o JavaScript executa as instruções especificadas, ou seja, um código interno ao loop. Veja:

```
for (variável in objeto) {
    declarações
}
```

Agora veja um exemplo da utilização dessa declaração:

```
for (const valor in {a:1, b:2, c:3}) {  
    console.log(valor);  
}
```

Você visualizará a seguinte página ao usar o código anterior:

a  
b  
c

Se você fizer modificações no objeto **array**, adicionar propriedades ou métodos personalizados, a declaração **for/in** retorna o nome de suas propriedades definidas pelo usuário, além de índices numéricos; uma vez que essa declaração também itera sobre propriedades definidas pelo usuário. Por isso, sempre que iterar sobre **array**, recomenda-se usar um loop **for** com um índice numérico.

## 7.3.5. Declaração break

Nos loops de repetição, a declaração **break** interrompe o loop imediatamente, além de manter a execução do programa na linha seguinte ao loop, se a condição imposta for atendida.

Como você pôde ver anteriormente, **break** também pode ser usada com **switch/case**.

Veja um exemplo de utilização de **break**:

```
while(new Date().getDay() < 7){  
    console.log('#');  
    break;  
}
```

Veja o resultado:

#

### 7.3.6. Declaração continue

Esta declaração geralmente é usada em um teste **if** e faz com que o loop retorne imediatamente para o teste de condição do loop de repetição.

Pelo exemplo, você deve ter percebido que o código verifica a existência de erros. Assim, se a condição for verdadeira, a leitura do próximo campo será feita até que se atinja o final do arquivo.

Veja o exemplo a seguir, em que a declaração **continue** é usada com o loop de repetição **while**:

```
let a = 0;
let b = 0;

while(a < 5){
  a++;
  if(a == 3) continue;
  b += a;
}
console.log(a, b);
```

Você visualizará o seguinte resultado:

5 12

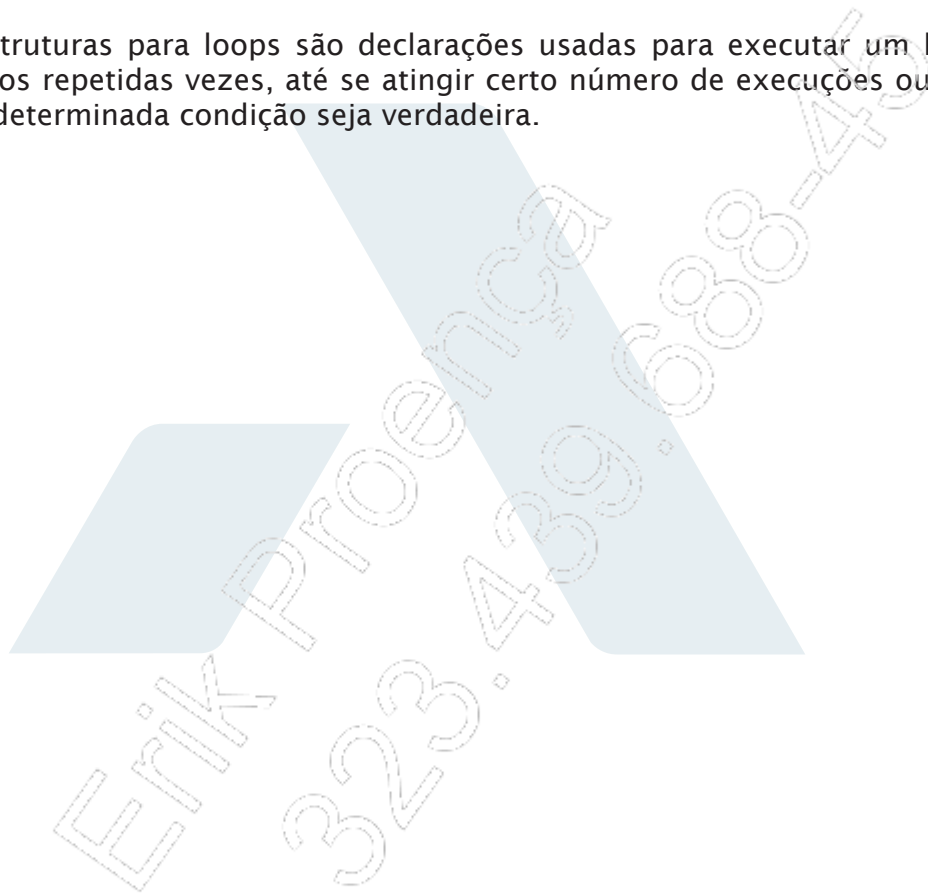
A declaração **continue** é usada somente em estruturas para loops.



## Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- O JavaScript é responsável por grande parte dos recursos interativos nas páginas HTML. Para incorporação desses recursos, faz uso de um grupo de declarações com características e funcionalidades específicas;
- Para que você decida qual ação será realizada em uma parte específica de um programa, é necessário solicitar que o programa avalie uma determinada expressão. Se essa expressão for avaliada como verdadeira, uma sequência de declarações será executada;
- As estruturas para loops são declarações usadas para executar um bloco de códigos repetidas vezes, até se atingir certo número de execuções ou até que uma determinada condição seja verdadeira.





# Funções

- O que é uma função?
- Definindo uma função;
- Chamando funções;
- Escopo de uma função;
- Closures;
- Inserindo variáveis nos parâmetros;
- Retornando valores;
- Funções predefinidas;
- Propriedades das funções.



## 8.1. O que é uma função?

Durante a execução de um programa é comum ser necessário realizar alguns processos repetidas vezes. Para evitar justamente a repetição de códigos nos scripts, podemos agrupá-los em funções. Ao agrupar os processos em uma função, basta apenas chamá-la e ela ficará encarregada de realizar as tarefas necessárias.

Você pode, então, entender uma função como um conjunto de instruções em um mesmo processo, o qual pode ser executado ao ser chamado. Considere, por exemplo, a criação de uma função para mudar a cor de fundo de uma página Web. Nesse caso, se você quisesse mudar a cor de fundo em outra parte da página, não seria necessário escrever todo o código de novo, bastaria apenas chamar a mesma função, já definida.

É importante levar em conta que o uso das funções não se restringe apenas aos desenvolvedores. Além das funções que eles escrevem, também há aquelas já definidas no sistema, que são empregadas em processos rotineiros, como na exibição de mensagens na tela, obtenção de hora, entre outros.

Neste capítulo, você aprenderá a criar e a chamar uma função, sua utilização e manipulação por meio de propriedades e métodos.

## 8.2. Definindo uma função

Para criar uma função, devemos utilizar a seguinte sintaxe:

```
function nome_funcao(){  
  instruções da função  
}
```

Em que:

- **function:** Indica a criação de uma função;
- **nome da função:** Este parâmetro nomeia a função a ser criada. Pode conter números, letras e algum caractere adicional.

Depois de especificado o nome para a variável, basta inserir as instruções da função entre as chaves de abertura e de fechamento.

A utilização das chaves de abertura e de fechamento não é opcional para funções. Elas contribuem, inclusive, para uma visualização mais fácil da estrutura que é formada pelas instruções da função.

### 8.2.1. Inserindo funções

As funções podem ser definidas em qualquer parte de uma página, desde que entre as tags `<script>` e `</script>`. É indiferente, também, se a chamada está antes ou depois da função, desde que esteja no mesmo bloco `<script>`. Em outros termos, a função deve ser definida no mesmo bloco de sua chamada.

Mas, para evitar enganos, o mais comum é inserir a função antes de qualquer chamada, conforme o exemplo a seguir:

```
function saudacoes(){  
    console.log('Olá');  
}  
  
saudacoes();
```

A função também pode ser inserida em um bloco `<script>` em uma página HTML que esteja antes do bloco da chamada. Veja o exemplo adiante:

```
<!DOCTYPE html>  
<html lang="en">  
  
  <head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-  
scale=1.0">  
    <title>Document</title>  
    <script>  
      function saudacoes() {  
        console.log('Olá');  
      }  
    </script>  
  </head>  
  
  <body>  
    <script>  
      saudacoes();  
    </script>  
  </body>  
  
</html>
```

### 8.3. Chamando funções

Para chamar funções, utilize o formato nome da função seguido por parênteses: **nome\_função()**. É possível chamá-las a partir de qualquer parte da página. Dessa maneira, o conjunto de instruções que possui a função entre as duas chaves será executado de forma correta.

Considere o seguinte exemplo, no qual a função é chamada a partir do momento em que um botão é pressionado:

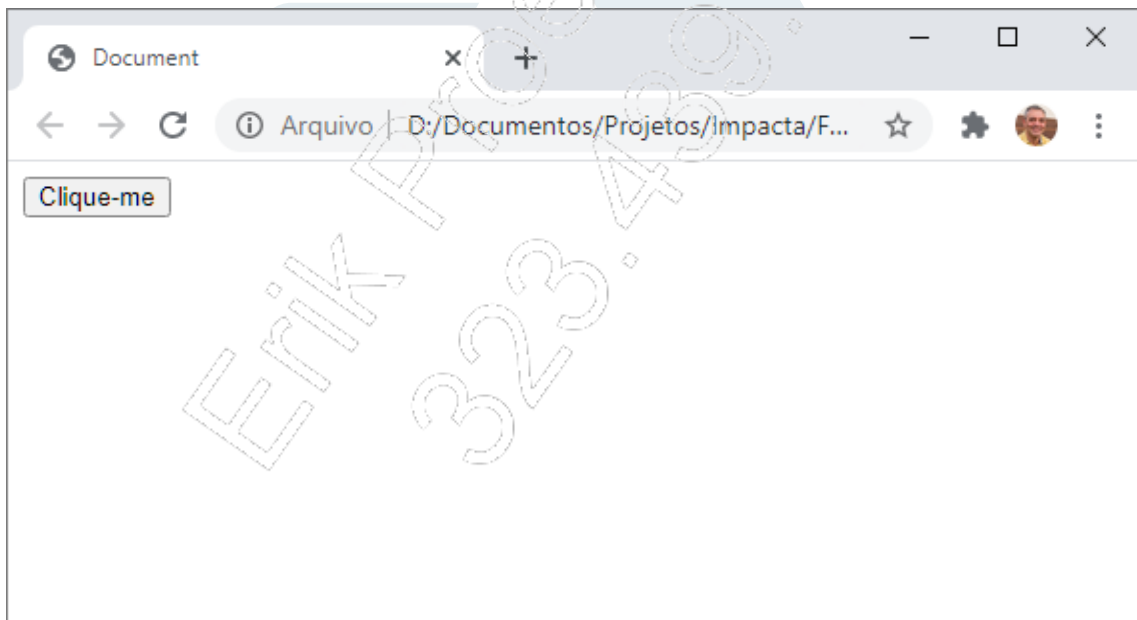
```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
  <script>
    function saudacoes() {
      document.write('Olá');
    }
  </script>
</head>

<body>
  <button onclick="saudacoes()">Clique-me</button>
</body>

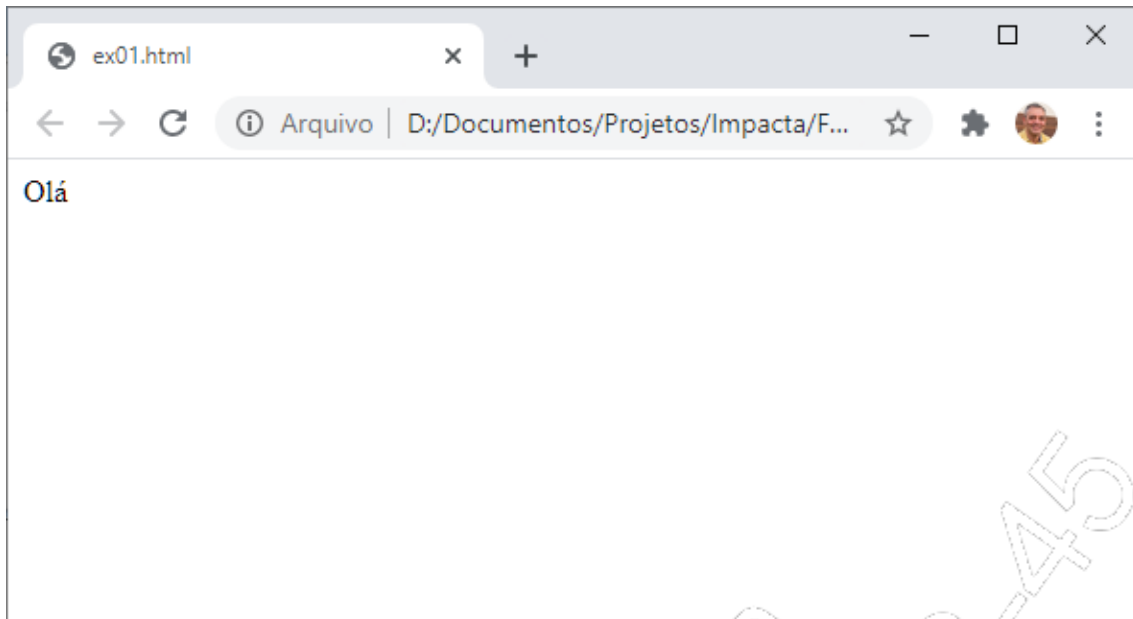
</html>
```

A seguinte página será exibida depois de salvar e executar o código anterior:





Após clicar no botão, a seguinte página será exibida:



## 8.4. Escopo de uma função

Como as variáveis são definidas no escopo da função, elas não podem ser acessadas a partir de qualquer lugar fora da função. Basicamente, uma função pode acessar todas as variáveis e demais funções do mesmo escopo em que foi definida.

Uma função que foi definida no escopo global, por exemplo, poderá acessar todas as variáveis definidas nesse mesmo escopo:

```
var texto = 'Mundo';  
function saudacoes(){  
    console.log('Olá, ' + texto);  
}  
  
saudacoes();
```

Quando uma função é definida dentro de outra função, elas são consideradas como função-pai e função-filha. A função-filha poderá acessar as variáveis da função-pai, assim como as variáveis que esta tiver acessado:

```
var texto = 'Mundo';
function saudacoes(){

    function funcao_filha() {
        return texto;
    }

    console.log('Olá, ' + funcao_filha());
}

saudacoes();
```

## 8.5. Closures

Um dos recursos mais interessantes do JavaScript são os **closures**. Eles representam o aninhamento de funções, isto é, uma função é definida dentro de outra função. Um closure é criado quando a função interna passa a ser disponibilizada fora do escopo da função externa.

Com isso, a função interna possui acesso total às variáveis e funções dentro da função em que foi definida, além do acesso às variáveis e funções que esta tiver acessado.

Considere o seguinte exemplo:

```
function multiplicarPor(n) {
    return function(x) {
        return x * n;
    }
}

let por10 = multiplicarPor(10);
console.log(por10(2));
console.log(por10(3));
```

Como resultado, temos o seguinte:

```
20
30
```

As variáveis e funções nela incluídas podem ter a sua vida útil estendida em relação à própria função externa, uma vez que as funções internas podem acessar o seu escopo e são destinadas a uma vida útil maior.

Já a função externa, em contrapartida, não tem acesso às variáveis e funções que estão dentro da função interna, garantindo segurança a elas:

```
function multiplicarPor(n) {  
  console.log('1. x existe? ', (typeof x));  
  return function(x) {  
    return x * n;  
  }  
}  
  
let por10 = multiplicarPor(10);  
console.log(por10(4));  
console.log(por10(5));  
  
console.log('2. x existe? ', (typeof x));
```

Veja o resultado:

```
1. x existe?  undefined  
40  
50  
2. x existe?  undefined
```

## 8.6. Inserindo variáveis nos parâmetros

Para receber e devolver valores, as funções apresentam uma entrada e uma saída. Por meio dos parâmetros, é possível definir valores de entrada para a função, os quais serão empregados por ela para realizar as ações.

Para uma função que deve realizar a soma de dois números, por exemplo, os parâmetros serão esses dois números, ou seja, ambos serão a entrada. O resultado, por sua vez, será a saída. Veja o seguinte exemplo:

```
function somarCom(n) {  
  return function(x) {  
    return x + n;  
  }  
}  
  
let com2 = somarCom(2);  
  
for (let i = 0; i <= 10; i++) {  
  console.log(i + ' + 2 = ' + com2(i));  
}
```

Como resultado, teremos o seguinte:

```
0 + 2 = 2
1 + 2 = 3
2 + 2 = 4
3 + 2 = 5
4 + 2 = 6
5 + 2 = 7
6 + 2 = 8
7 + 2 = 9
8 + 2 = 10
9 + 2 = 11
10 + 2 = 12
```

A utilização dos parâmetros será explicada nos próximos subtópicos.

## 8.6.1. Um parâmetro

Para especificar um parâmetro na função, é necessário inserir o nome da variável que contém o dado a ser passado para a função quando a chamarmos.

**!** O tempo de vida útil da variável dura até o término da execução da função. Assim que a função é executada, a variável deixa de existir.

Quando você quer chamar a função, o valor do parâmetro deve ser colocado entre parênteses. Veja o exemplo adiante, em que chamamos a função do exemplo anterior:

```
let com2 = somarCom(2);
```

Os parâmetros aceitam quaisquer tipos de dados, sejam eles textuais, numéricos ou mesmo booleanos. Quando não definimos o tipo de parâmetro, é necessária atenção especial ao especificar as ações que serão efetuadas dentro da função.

Também é importante ter cuidado ao passar valores para a função, para que os valores passados não entrem em conflito com os tipos já definidos para as variáveis ou parâmetros.

### 8.6.2. Múltiplos parâmetros

Podemos inserir a quantidade desejada de parâmetros dentro de uma função. Para defini-los, basta separá-los por vírgulas dentro dos parênteses, conforme o exemplo adiante:

```
function calcularRegra(n1, n2) {  
  return function(x) {  
    return (x * n1) + n2;  
  }  
}  
  
let calc = calcularRegra(10, 5);  
  
console.log(calc(3));
```

### 8.6.3. Parâmetros passados por valores

Os parâmetros passados por valores fazem com que a variável original não tenha seu valor modificado mesmo quando você altera um parâmetro em uma função.

Para compreender melhor a utilização desses parâmetros, veja o exemplo adiante:

```
function calcularRegra(n1) {  
  let n2 = 0;  
  
  return function(x) {  
    n2++;  
    return (x * n1) + n2;  
  }  
}  
  
let calc = calcularRegra(10);  
  
console.log(calc(10));  
console.log(calc(10));  
console.log(calc(10));
```

O resultado é o seguinte:

```
101  
102  
103
```

## 8.7. Retornando valores

Na linguagem JavaScript, há funções que têm como objetivo retornar valores, ou seja, atribuí-los às variáveis. Isso amplia as possibilidades de utilização dos aplicativos.

Podemos retornar valores por meio da palavra-chave **return**, conforme utilizado no exemplo anterior:

```
return function(x) {  
    n2++;  
    return (x * n1) + n2;  
}
```

## 8.8. Funções predefinidas

Existem algumas funções predefinidas disponíveis para utilização na linguagem JavaScript, são elas: **eval**, **isFinite**, **isNaN**, **parseInt**, **parseFloat**, **number**, **string**, **encodeURIComponent**, **decodeURI**, **encodeURIComponent** e **decodeURIComponent**.

A seguir, essas funções predefinidas serão descritas:

- **eval**

Com a utilização desta função, é possível retornar um número a partir de uma operação que contém uma string.

Veja o seguinte exemplo:

```
eval('var resultado = 10 + 3');  
console.log(resultado);
```

- **isFinite**

Com a utilização desta função, disponível na linguagem JavaScript a partir da versão 1.3, é possível verificar se um número é finito ou não. Caso o número seja finito, o valor **true** será retornado; caso contrário, o valor **false** será retornado.

Veja o exemplo adiante:

```
let resultado = 10 * Math.pow(10, 309);  
console.log(isFinite(resultado));
```

- **isNaN**

Com a utilização desta função, é possível verificar se um argumento é ou não um número. O argumento pode ser tanto um objeto quanto uma variável.

O retorno da função será um valor booleano: caso o valor não seja um número, ela retorna **true**; se for um número, ela retorna **false**.

Veja o seguinte exemplo:

```
let numero = parseInt('A1');  
console.log(isNaN(numero));
```

- **parseInt**

Por meio desta função, é possível transformar uma string em um número inteiro. O número inteiro é retornado mesmo quando a string possui um valor com casa decimal.

Observe o exemplo adiante:

```
let texto = '1A';  
console.log(texto);  
texto = parseInt(texto);  
console.log(texto);
```

Veja o resultado:

1A

1

- **parseFloat**

Com a utilização desta função, você consegue transformar uma string em um ponto flutuante.

Veja um exemplo demonstrando a utilização da função **parseFloat**:

```
let texto = '1.2A';  
console.log(texto);  
  
texto = parseFloat(texto);  
console.log(texto);
```

- **number**

Por meio desta função, é possível converter um objeto em um número.

Observe o exemplo adiante:

```
let data = new Date();
let numero = Number(data);
console.log(numero);
```

- **string**

Por meio desta função, é possível converter um objeto em uma string.

Observe o exemplo adiante:

```
let data = new Date();
let numero = String(data);
console.log(numero);
```

- **encodeURIComponent**

Com a utilização desta função, você consegue codificar um determinado texto. Com isso, os caracteres são substituídos por caracteres Unicode 8 bits.

Veja o exemplo demonstrando a utilização da função **encodeURIComponent**:

```
let texto = "João Conceição";
console.log(encodeURIComponent(texto));
```

- **decodeURI**

Com a utilização desta função, que atua de forma oposta à função **encodeURIComponent**, é possível decodificar um determinado texto que possua caracteres UTF-8.

Veja um exemplo demonstrando a utilização da função **decodeURI**:

```
let texto = "Jo%C3%A3o%20Concei%C3%A7%C3%A3o";
console.log(decodeURI(texto));
```

- **encodeURIComponent**

Por meio desta função, é possível codificar para caracteres UTF-8 os caracteres especiais de um componente URL.

Observe o seguinte exemplo:

```
console.log(encodeURIComponent(window.location));
```



- **decodeURIComponent**

Por meio desta função, oposta à função **encodeURIComponent**, é possível decodificar um texto que possua caracteres UTF-8.

Observe o seguinte exemplo:

```
console.log(decodeURIComponent(encodeURIComponent(window.location)));
```

## 8.9. Propriedades das funções

As funções possuem algumas propriedades. São elas: **arguments.length**, **arguments.callee**, **length**, **constructor** e **prototype**.

Os próximos subtópicos descrevem as propriedades das funções.

### 8.9.1. arguments.length

Trata-se da propriedade **length** do objeto **arguments**. Por meio dela, você consegue saber quantos argumentos a função possui, isto é, o retorno é o número de argumentos da função.

Veja um exemplo de utilização desta propriedade:

```
function calcular(a, b, c){  
    console.log(arguments.length);  
}  
calcular(1,2,3);  
calcular();
```

### 8.9.2. arguments.callee

Trata-se da propriedade **callee** do objeto **arguments**. Por meio desta propriedade, é possível fazer referência a uma função anônima, sem que seja necessário criar um nome para ela. Com isso, você consegue chamar a função anônima dentro de si mesma.

Para compreender melhor o funcionamento desta propriedade, veja o exemplo a seguir:

```
function fatorial(){
    return function(n){
        if(n <= 1){
            return 1;
        }
        return n * arguments.callee(n - 1);
    }
}

let resultado = fatorial()(5);
console.log(resultado);
```

A chamada da função dentro de si mesma é denominada mecanismo recursivo, o que qualifica a função como recursiva.

## 8.9.3. length

Trata-se da propriedade **length** do objeto **função**. Por meio dela, é possível verificar quantos argumentos a função espera receber, isto é, o retorno é o número de argumentos que podem ser passados à função. É diferente da propriedade **arguments**. **length**.

A tabela a seguir ajuda a identificar a diferença entre essas propriedades:

TABELA COMPARATIVA	
Propriedade <b>length</b>	Propriedade <b>arguments.length</b>
Retorna o número de argumentos que podem ser passados para a função.	Retorna o número de argumentos passados para a função.
Pode ser acessada a partir de fora da função.	Só existe dentro do corpo da função.

Veja o exemplo de utilização da propriedade **length**:

```
function fatorial(){
    return function(n){
        if(n <= 1){
            return 1;
        }
        return n * arguments.callee(n - 1);
    }
}

console.log(fatorial.length);
console.log(fatorial().length);
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Funções podem ser consideradas um conjunto de instruções em um mesmo processo, que pode ser executado ao ser chamado. Com isso, não precisamos executar tais processos repetidas vezes, basta chamar a mesma função;
- É necessário prestar atenção à sintaxe e aos formatos utilizados para a criação de uma função. Por exemplo, ela pode ser definida em qualquer parte de uma página, desde que entre as tags `<script>` e `</script>`;
- Para chamar funções, utilize o formato nome da função seguido por parênteses: **nome\_função ()**;
- Com relação ao escopo de uma função, ela poderá acessar todas as variáveis e demais funções do mesmo escopo em que foi definida;
- Os **closures**, isto é, quando uma função é definida dentro de outra função, permitem que a função interna acesse as variáveis e funções da função externa, além do acesso às variáveis e funções que esta tiver acessado. Já a função externa, em contrapartida, não tem acesso às variáveis e funções que estão dentro da função interna, garantindo segurança a elas;
- Para especificar um parâmetro na função, é necessário inserir o nome da variável que contém o dado a ser passado para a função quando ela for chamada. Quando você quer chamar a função, o valor do parâmetro deve ser colocado entre parênteses e a variável deixará de existir;
- Para retornar valores, ou seja, atribuí-los às variáveis, utilize a palavra-chave **return**;
- As seguintes funções são predefinidas na linguagem JavaScript: **eval**, **isFinite**, **isNaN**, **parseInt**, **parseFloat**, **number**, **string**, **encodeURIComponent**, **decodeURI**, **encodeURIComponent** e **decodeURIComponent**.

