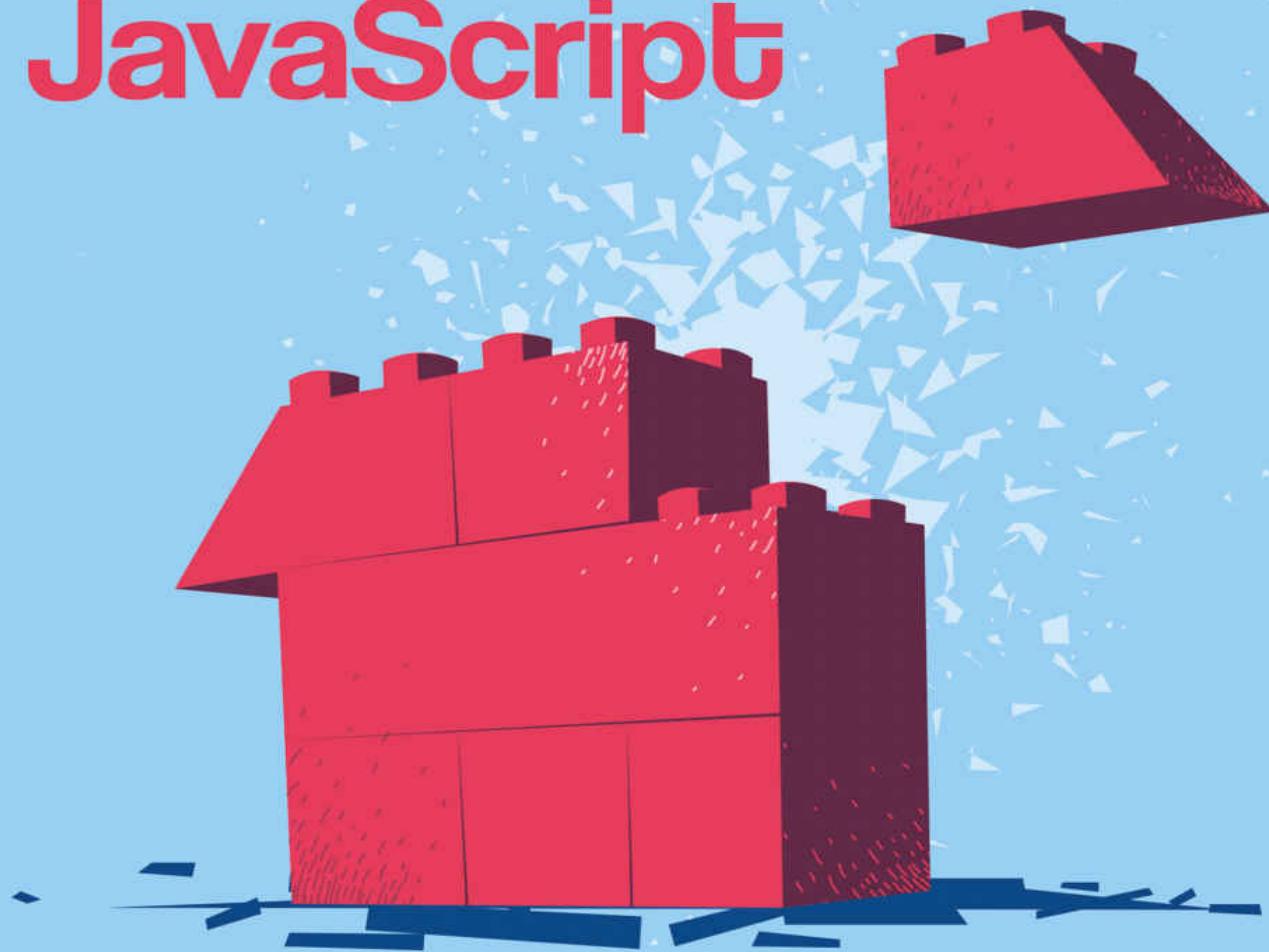


2^a EDIÇÃO Revisada e ampliada

UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES

Lógica de Programação e Algoritmos com **JavaScript**



novatec

Edécio Fernando Iepsen

Lógica de Programação e Algoritmos com JavaScript

**UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES**

2^a Edição

Edécio Fernando Iepsen

Novatec

© Novatec Editora Ltda. 2018, 2022.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates GRP20220302

Revisão gramatical: Tássia Carvalho

Ilustrações: Carolina Kuwabata

Capa: Carolina Kuwabata

ISBN do ebook: 978-65-86057-91-1

ISBN do impresso: 978-65-86057-90-4

Histórico de impressões:

Março/2022 Segunda edição

Setembro/2019 Primeira reimpressão

Março/2018 Primeira edição (ISBN: 978-85-7522-656-8)

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: <https://novatec.com.br>
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec
GRP20220302

*Aos meus filhos, Henrique e Miguel:
fontes de alegria e esperança.*

*A minha esposa, Delair, e a minha mãe, Leonídia:
fontes de incentivo em todas as jornadas.*

*E à amizade de meus irmãos, sobrinhos e enteada:
Eduardo, Elenice, Gabriel, Sabrine e Daniela;*

e dos meus afilhados:

Cândida, Elsa, Gabriel, Murilo, Lauren, Rafaela, Marina e Âgata.

Sumário

[Agradecimentos](#)

[Sobre o autor](#)

[Prefácio](#)

[capítulo 1 Introdução](#)

[1.1 Lógica de programação](#)

[1.1.1 Compreender o que é pedido](#)

[1.1.2 Realizar deduções na construção do programa](#)

[1.1.3 Enumerar as etapas a serem realizadas](#)

[1.1.4 Analisar outras possibilidades de solução](#)

[1.1.5 Ensinar ao computador uma solução](#)

[1.1.6 Pensar em todos os detalhes](#)

[1.2 Entrada, processamento e saída](#)

[1.3 JavaScript](#)

[1.4 Editores de código JavaScript](#)

[1.5 Saída de dados com alert\(\) e console.log\(\)](#)

[1.6 Variáveis e constantes](#)

[1.7 Entrada de dados com prompt\(\)](#)

[1.8 Comentários](#)

[1.9 Tipos de dados e conversões de tipos](#)

[1.10 Exemplos de entrada, processamento e saída](#)

[1.11 Exercícios](#)

[1.12 Considerações finais do capítulo](#)

[capítulo 2 Integração com HTML](#)

[2.1 Estrutura básica de um documento HTML](#)

[2.2 Cabeçalhos, parágrafos e campos de formulário](#)

[2.3 Criação do programa JavaScript](#)

[2.4 Métodos querySelector\(\) e getElementById\(\)](#)

[2.5 Introdução a eventos e funções](#)

[2.6 Propriedades innerText, innerHTML e value](#)

- [2.7 Método preventDefault\(\)](#)
- [2.8 Operadores aritméticos e funções matemáticas](#)
- [2.9 Exemplos de programas JavaScript integrados com HTML](#)
- [2.10 Exercícios](#)
- [2.11 Considerações finais do capítulo](#)

capítulo 3 Construção de algoritmos com Node.js

- [3.1 Instalação do Node.js](#)
- [3.2 Adição de pacote para entrada de dados](#)
- [3.3 Criação e execução de programas com o Node.js](#)
- [3.4 Exemplos de algoritmos com Node.js](#)
- [3.5 Considerações finais do capítulo](#)

capítulo 4 Condições

- [4.1 If... else](#)
- [4.2 Operadores relacionais](#)
- [4.3 Operadores lógicos](#)
- [4.4 Operador ternário](#)
- [4.5 Switch... Case](#)
- [4.6 Exemplos com HTML e JavaScript](#)
- [4.7 Exemplos com Node.js](#)
- [4.8 Exercícios](#)
- [4.9 Considerações finais do capítulo](#)

capítulo 5 Repetições

- [5.1 Repetição com variável de controle: laços for](#)
- [5.2 Repetição com teste no início: laços while](#)
- [5.3 Repetição com teste no final: laços do.. while](#)
- [5.4 Interrupções nos laços \(break e continue\)](#)
- [5.5 Contadores e acumuladores](#)
- [5.6 Depurar programas \(detectar erros\)](#)
- [5.7 Exemplos de Algoritmos de Repetição com Node.js](#)
- [5.8 Exercícios](#)
- [5.9 Considerações finais do capítulo](#)

capítulo 6 Vetores

- [6.1 Inclusão e exclusão de itens](#)
- [6.2 Tamanho do vetor e exibição dos itens](#)

- [6.3 For..of e forEach\(\)](#)
- [6.4 Localizar conteúdo](#)
- [6.5 Vetores de objetos](#)
- [6.6 Desestruturação e operador Rest/Spread](#)
- [6.7 Pesquisar e filtrar dados](#)
- [6.8 Map, Filter e Reduce](#)
- [6.9 Classificar os itens do vetor](#)
- [6.10 Manipulação de vetores com Node.js](#)
- [6.11 Exercícios](#)
- [6.12 Considerações finais do capítulo](#)

capítulo 7 Strings e datas

- [7.1 Percorrer os caracteres de uma string](#)
- [7.2 Converter para letras maiúsculas ou minúsculas](#)
- [7.3 Cópia de caracteres e remoção de espaços da string](#)
- [7.4 Localizar um ou mais caracteres na string](#)
- [7.5 Dividir a string em elementos de vetor](#)
- [7.6 Validar senhas com o método match\(\)](#)
- [7.7 Substituição de caracteres](#)
- [7.8 Manipulação de datas](#)
- [7.9 Strings e datas com Node.js](#)
- [7.10 Exercícios](#)
- [7.11 Considerações finais do capítulo](#)

capítulo 8 Funções e eventos

- [8.1 Functions e Arrow Functions](#)
- [8.2 Funções com passagem de parâmetros](#)
- [8.3 Funções com retorno de valor](#)
- [8.4 Funções com parâmetros Rest](#)
- [8.5 Funções anônimas](#)
- [8.6 Eventos JavaScript](#)
- [8.7 Funções com Node.js](#)
- [8.8 Exercícios](#)
- [8.9 Considerações finais do capítulo](#)

capítulo 9 Persistência de dados com localStorage

- [9.1 Salvar e recuperar dados](#)

[9.2 Uma “pitada” de Bootstrap](#)
[9.3 Remover dados do localStorage](#)
[9.4 Uso do querySelectorAll\(\)](#)
[9.5 Manipular listas no localStorage](#)
[9.6 Exercícios](#)
[9.7 Considerações finais do capítulo](#)

[capítulo 10 Inserir elementos HTML via JavaScript](#)

[10.1 Inserir e manipular elementos de texto](#)
[10.2 Inserir imagens](#)
[10.3 Manipular tabelas HTML](#)
[10.4 Exercícios](#)
[10.5 Considerações finais do capítulo](#)

[capítulo 11 É muito bom programar... Programe!](#)

[11.1 Programa Jockey Club](#)
[11.2 Programa Reserva de Poltronas em Teatro](#)
[11.3 Jogo “Descubra a Palavra”](#)
 [11.3.1 Cadastro de palavras](#)
 [11.3.2 Listagem de palavras](#)
 [11.3.3 Programação do jogo](#)
[11.4 Considerações finais do capítulo](#)

[capítulo 12 Criação de um App: Back-end com Express](#)

[12.1 Express](#)
[12.2 Nodemon](#)
[12.3 Rotas POST e Formato JSON](#)
[12.4 Middlewares](#)
[12.5 Use o Knex e escolha o banco de dados](#)
[12.6 Criação de tabelas com Migrations](#)
[12.7 Seeds: “semeando” dados iniciais](#)
[12.8 Database Config e express.Router](#)
[12.9 Async e await](#)
[12.10 Status Code HTTP](#)
[12.11 Rotas para a realização do CRUD](#)
[12.12 Filtros, totalizações e agrupamentos](#)
[12.13 CORS](#)

12.14 Considerações finais do capítulo

capítulo 13 Criação de um App: Front-end com React

13.1 Variáveis de Estado: useState()

13.2 Organização em componentes

13.3 Criação de um novo projeto React

13.4 Simplificando o gerenciamento do form com React Hook Form

13.5 Axios para comunicar com o Web Service

13.6 Criação de Rotas com o React Router

13.7 useEffect()

13.8 Filtrando os registros da listagem

13.9 Exclusão, alteração e passagem de funções como props

13.10 Resumo com gráfico ou dashboard do sistema

13.11 Considerações finais

Referências

Agradecimentos

Agradeço primeiro a Deus o dom da vida, por guiar os meus passos e por Jesus Cristo, meu Senhor e Salvador.

A toda a minha família, por ser o meu porto seguro, onde encontro paz, carinho e incentivo para os desafios do dia a dia.

Aos diretores, coordenadores, professores e funcionários da Faculdade de Tecnologia Senac Pelotas, instituição comprometida com a qualidade de ensino, onde muito me orgulho de lecionar.

À editora Novatec, pelo apoio na criação deste livro. As publicações da Novatec têm sido muito importantes para a minha atuação profissional, e agora é uma honra publicar este livro pela editora.

Aos meus professores, que, a partir de exemplos, ensinamentos e cuidados, me ajudaram a ser uma pessoa melhor.

Aos meus alunos, com os quais tenho a honra de compartilhar a minha história, fazer parte de suas conquistas, ensinar e aprender – a cada nova aula.

A todas as pessoas que acreditam na Educação como fonte propulsora para a construção de uma sociedade mais justa, com oportunidades de vida digna para todos.

Sempre contei com o auxílio de muitas pessoas para atingir cada um dos meus objetivos. Desde aquelas que contribuíram de uma forma mais significativa até as mais singelas, meu muito obrigado. Gratidão a todos vocês.

Sobre o autor

Edécio Fernando Iepsen graduou-se em Tecnologia em Processamento de Dados pela UCPel (Universidade Católica de Pelotas) em 1992. Atuou como programador por diversos anos, desenvolvendo sistemas comerciais nas linguagens Clipper, Delphi e PHP. De forma conjunta à atividade de programador, atuou como professor do Colégio Santa Margarina, no curso Técnico em Informática, até 2004. Também lecionou na Universidade Católica de Pelotas, durante seis anos. Em 2005, passou a lecionar na Faculdade de Tecnologia Senac Pelotas, onde atua até a presente data. Concluiu os cursos de Licenciatura Plena para Professor de 2º grau (UTFPR/IFSul) e Especialização em Informática com Ênfase em Planejamento (UCPel).

Em 2008, obteve o título de Mestre em Ciência da Computação, também pela UCPel. Em 2010, ingressou no Doutorado em Informática na Educação pela UFRGS (Universidade Federal do Rio Grande do Sul). Sua tese de Doutorado abordou o ensino de Algoritmos apoiado por técnicas de Computação Afetiva – uma subárea da Inteligência Artificial, tendo diversos artigos publicados sobre o tema. Concluiu o Doutorado em 2013.

Dentre as diversas disciplinas que já lecionou, destaca-se a de Algoritmos e Lógica de Programação, tema deste livro. Atua também com frequência nas Unidades Curriculares de Programação para Internet, Estrutura de Dados, Linguagens de Programação e Ciência de Dados.

Prefácio

Compartilhar boas experiências é uma tarefa muito agradável. O tempo em que venho lecionando a disciplina de Algoritmos e Lógica de Programação me permitiu aprender muito com meus alunos, pois o conhecimento não é uma via de mão única. A partir dessa experiência, foi possível perceber diversos aspectos, como qual sequência de assuntos se mostra mais adequada para facilitar a compreensão dos conceitos e práticas da disciplina, quais conteúdos geram as maiores dúvidas e necessitam de um tempo maior de discussão e quais correlações se mostram mais significativas para o aprendizado. Além disso, ao longo desses anos, elaborei diversos novos exemplos e exercícios e verifiquei quais deles despertam maior interesse por parte dos estudantes. Compartilhar esses exemplos e exercícios com um número maior de pessoas, por meio da publicação deste livro, é um sonho que há tempos venho pensando em realizar. Reitero meu agradecimento à Editora Novatec por possibilitar a concretização desse sonho.

Os conteúdos abordados em Lógica de Programação e Algoritmos são essenciais a todos que desejam ingressar no universo da programação de computadores. Esses conteúdos, no geral, exigem certo nível de abstração, normalmente ainda não experimentado pelos estudantes nas disciplinas do ensino médio. Por isso, a importância de abordar tais conteúdos passo a passo, como é a proposta deste livro.

Para os iniciantes, construir um novo programa pode se constituir uma árdua tarefa. É necessário descrever os comandos seguindo regras rígidas de sintaxe, além de compreender conceitos de entrada, processamento e saída de dados, variáveis, condições, repetições, entre outros. Tudo isso deve ser tratado de modo a não sobrecarregar os aspectos cognitivos de quem pretende ingressar na área. Os exercícios propostos nesta obra foram cuidadosamente planejados para que os avanços de complexidade ocorram

da forma mais linear possível, facilitando a absorção desses diversos quesitos por parte do leitor.

Se pensarmos na importância que a tecnologia pode assumir na vida das pessoas e no dia a dia de empresas, vamos perceber o quanto é significativo que os sistemas desenvolvidos executem corretamente cada uma de suas tarefas. Esses sistemas estão presentes em praticamente todas as áreas, e você pode ajudar a construir um mundo melhor para diversas pessoas com a construção de novos aplicativos. Aprender Lógica de Programação constitui um grande passo rumo ao futuro. E é praticamente impossível pensar no futuro, sem pensar na evolução tecnológica e sem imaginar o quanto novos sistemas computacionais poderão beneficiar áreas fundamentais como saúde, educação, agricultura e segurança.

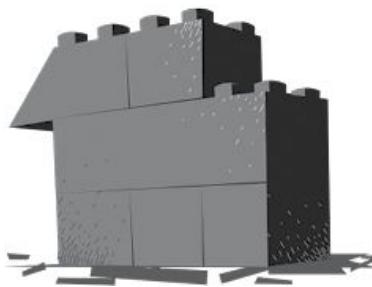
Escrever um livro compartilhando a experiência acumulada no ensino de Algoritmos, com os exemplos em JavaScript – uma linguagem em constante ascensão no mercado, foi para mim uma tarefa muito agradável. Espero que gostem do livro da mesma forma como eu gostei de escrevê-lo. Desejo que ele seja uma importante fonte de construção de conhecimentos para que você, leitor, adquira as habilidades necessárias para ingressar na área da programação de computadores.

Os códigos-fonte do livro estão disponíveis para download em:

<https://novatec.com.br/livros/logica-programacao-algoritmos-com-javascript-2ed/>

CAPÍTULO 1

Introdução



Muito bem-vindo ao universo da programação de computadores! Neste universo, você será capaz de construir coisas incríveis. Ajudar a melhorar o mundo. Desenvolver um sistema para uma empresa aperfeiçoar o gerenciamento de seu negócio. Talvez, criar um aplicativo a fim de auxiliar pessoas a superarem suas dificuldades. Seja qual for o seu objetivo, você precisa começar pelo estudo de Algoritmos e Lógica de Programação. Organizando os programas com lógica, eles vão executar corretamente as tarefas necessárias para automatizar um processo.

E como se aprende lógica de programação? Estudando as técnicas de programação e exercitando-as muito a partir da criação de pequenos programas. Isso mesmo. Resolvendo os exercícios de programação, começamos a pensar da forma como um programa funciona. É quase como o processo de alfabetização de uma criança. Ela não começa pela leitura de um livro, mas pelo aprendizado das vogais, consoantes. Em seguida, pela formação das sílabas para, então, formar palavras e frases. Assim, estará apta para realizar algo maior, a leitura de um livro. Ou melhor, de qualquer livro que ela queira ler.

O processo de aprendizado de Algoritmos e Lógica de Programação é semelhante. Vamos começar pela compreensão dos conceitos de variáveis. Depois veremos as partes que compõem um programa (entrada,

processamento e saída). E vamos avançando, com o acréscimo de estruturas de condições, repetições, manipulação de listas de dados (vetores) e outras técnicas.

Os exemplos de programas vão começar pela realização de tarefas simples, como calcular o dobro de um número. Depois de entender as etapas da programação sequencial, avançaremos um pouco. Usar condições para calcular a média de notas e indicar se um aluno foi aprovado ou reprovado. Criar repetições, gerenciar listas de dados e muitos outros exemplos. No final, você verá que estará apto a criar programas maiores e propor soluções tecnológicas para os mais diversos problemas do dia a dia de uma empresa ou de processos que, em seu julgamento, poderiam ser aperfeiçoados com a ajuda de um aplicativo.

No entanto, para atingir seus objetivos, você deve ser determinado. Caso o seu programa não funcione corretamente logo na primeira (ou segunda, terceira...) vez em que o executar, não desanime. Concentre-se. Verifique os detalhes. Faltou um ponto e vírgula? Alguma letra maiúscula ou minúscula ficou incorreta? Às vezes, depois de ajustar um simples detalhe, a mágica acontece. E o seu programa funciona! Que legal! Você vai vibrar como se tivesse marcado um gol!

No processo de construção de um sistema, é necessário ter conhecimento de lógica. Esse conhecimento aplicado a uma linguagem de programação vai produzir um sistema. E as linguagens de programação são muitas. Java, PHP, C#, Python, JavaScript, Delphi – só para citar algumas. No geral, elas focam em um segmento ou em uma plataforma. PHP, por exemplo, é mais utilizado para o desenvolvimento de sistemas web – no lado do servidor. Delphi e C#, por sua vez, possuem versões com foco no desenvolvimento de sistemas Windows Desktop. Mas, para construir um sistema em qualquer uma dessas linguagens, é essencial dominar as técnicas trabalhadas em lógica de programação. Essas técnicas são semelhantes, com pequenas variações ou particularidades em cada linguagem. Dessa forma, se você entender corretamente como criar programas em JavaScript, será muito mais fácil estudar outra linguagem – pois as estruturas e os conceitos

básicos são os mesmos. A Figura 1.1 ilustra o processo de construção de um aplicativo.

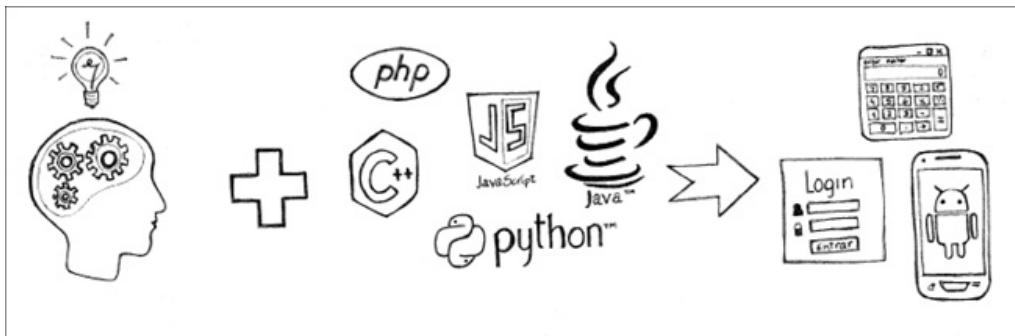


Figura 1.1 – Os conhecimentos de lógica de programação aplicados em uma linguagem permitem criar um aplicativo.

Para começar a pensar como um programa funciona, vamos utilizar um exemplo: o sistema de um caixa eletrônico de um banco, claro que de um modo genérico, apenas para ilustrar as etapas e os fluxos básicos de programação. O programa começa pela solicitação dos dados de identificação do cliente (entrada de dados). Estes podem ser fornecidos a partir da leitura de um cartão magnético do banco, de uma senha ou de uma característica biométrica do cliente. Em seguida, o cliente deve informar qual operação deseja realizar, como o cálculo do saldo da sua conta (processamento). Por fim, ocorre a impressão do saldo ou a exibição desses dados na tela (saída de dados). Observe que temos as três etapas: entrada, processamento e saída.

Vamos analisar agora os fluxos de programação disponíveis nesse sistema. Inicialmente ocorre o que chamamos de programação sequencial. Uma tarefa após a outra. O programa solicita a identificação do cliente e, depois, a sua senha. O passo seguinte é a execução de uma condição, que consiste em verificar se a senha está ou não correta. Temos, portanto, a programação condicional. Se a senha estiver correta, o programa exibe um menu com as opções disponíveis para o cliente. Caso contrário, é exibida uma mensagem indicando senha incorreta. Então o programa executa instruções de repetição – que é o terceiro fluxo de programação disponível nas linguagens. Se a senha estiver incorreta, ele repete a leitura. E esse processo é infinito? Não. O cliente, no geral, pode repetir até três vezes a digitação

da senha.

Ao estruturar os passos de um programa, estamos montando um algoritmo. Portanto, um algoritmo é uma sequência de passos (comandos) a serem executados para a realização de uma tarefa, em um tempo finito. Organizar essa sequência de passos de forma lógica é a nossa atribuição enquanto programadores de sistemas.

1.1 Lógica de programação

Os estudos de lógica são bastante antigos. Consultando os livros de lógica matemática, encontramos citações a estudos de Aristóteles, nascido em 384 a.C. Como filosofia, a lógica busca entender por que pensamos de uma maneira e não de outra. Organizar o pensamento e colocar as coisas em ordem são tarefas de lógica de que necessitamos para resolver problemas com o uso do computador.

Para fazer com que um problema seja resolvido corretamente por um sistema computacional, deve-se dar atenção especial a diversos aspectos. Muitos deles diretamente relacionados com as questões de lógica. Esses cuidados servem tanto para resolver os exercícios de Algoritmos propostos neste livro, quanto para solucionar problemas maiores, com os quais você vai deparar no exercício da profissão de programador. Alguns exemplos serão utilizados para facilitar o entendimento. Os pontos são os seguintes:

1.1.1 Compreender o que é pedido

Percebo, principalmente nas aulas iniciais de Algoritmos e Lógica de Programação, que alguns alunos estão ansiosos para resolver logo todos os exercícios propostos. Rapidamente escrevem o programa e realizam os testes para verificar se os dados de saída são os mesmos dos exemplos apresentados no enunciado de cada exercício. No entanto, se o aluno não compreender corretamente o que o exercício pede, não poderá chegar à resposta correta. E como é o próprio aluno que testa e verifica se um programa está ou não exibindo a resposta correta, ele pode ficar com uma sensação de frustração. Por que não está funcionando?

Alguns desses alunos solicitam a ajuda do professor. Outros são mais tímidos. E, se essa sensação de frustração permanecer por algum tempo, ela pode contribuir para a desistência do aluno e manter alta as taxas de evasão tradicionalmente verificadas nos cursos da área de computação.

Destaco isso para reforçar algo fundamental para o processo do aprendizado de Algoritmos: a compreensão do problema a ser solucionado. Perca alguns minutos na leitura dos enunciados dos exercícios que você vai resolver. Eles serão valiosos no final e talvez economizem horas do seu tempo.

1.1.2 Realizar deduções na construção do programa

Na construção das soluções dos exercícios de Algoritmos, faz-se necessário realizar pequenas deduções. Esse assunto é trabalhado na disciplina de Lógica Matemática (ou Lógica Formal) e contribui para o aprendizado de Lógica de Programação. Vamos ver alguns exemplos de dedução lógica que podem ser extraídos a partir da veracidade de proposições preliminares. Observe:

1. O carro está na garagem ou na frente da casa.
2. O carro não está na frente da casa.

Considerando que as afirmações 1 e 2 são verdadeiras, podemos deduzir que:

3. O carro está na garagem.

Observe outro exemplo:

1. Se chover, Silvana irá ao cinema.
2. Choveu.

Logo, novamente considerando que as afirmações 1 e 2 são verdadeiras, podemos concluir que:

3. Silvana foi ao cinema.

Utilizamos a lógica para deduzir as conclusões nos exemplos anteriores. Ela também será necessária para a resolução dos exercícios, que vão avançar de forma gradativa em níveis de complexidade. Com treinamento, aperfeiçoaremos a nossa lógica para raciocinar como os sistemas

computacionais funcionam e realizar deduções sobre quais controles devem ser utilizados para melhor solucionar um problema.

1.1.3 Enumerar as etapas a serem realizadas

Algumas ações realizadas em um programa seguem uma lógica sequencial, ou seja, um comando (ação) é realizado após o outro. Vamos continuar no exemplo do carro na garagem. Para sair com o carro, é necessário:

1. Abrir a porta do carro.
2. Entrar no carro.
3. Ligar o carro.
4. Abrir o portão da garagem.
5. Engatar a marcha ré.
6. Sair com o carro da garagem.
7. Fechar o portão.
8. Engatar a primeira marcha.
9. Dirigir ao destino.

Algumas dessas ações poderiam exigir a criação de condições. Por exemplo, o que deve ser feito se o carro não ligar? A inclusão de condições e repetições será abordada na Seção 1.1.6.

1.1.4 Analisar outras possibilidades de solução

Você tem três amigos e precisa somar a idade dos dois amigos que possuem as maiores idades. Como resolver esse problema? Você pode:

1. Descobrir a maior idade.
2. Descobrir a segunda maior idade.
3. Somar as duas idades maiores.

Mas será que essa é a única solução para esse problema? Pense um pouco... Talvez exista outro modo para organizar nossos passos. Observe outra solução:

1. Descobrir quem tem a menor idade.

2. Somar a idade dos outros dois.

Esse passos também resolvem o problema. Talvez até com menor esforço. Ou seja, estamos pensando formas de solucionar um problema utilizando a lógica. Quando deparar com um problema que você está com dificuldades para resolver de uma forma, respire um pouco... Tome uma água... Tente pensar se poderia existir outra forma de solucioná-lo.

1.1.5 Ensinar ao computador uma solução

Um exemplo simples: você precisa calcular o número total de horas de uma viagem, expressa em dias e horas. Uma viagem para Florianópolis dura 2 dias e 5 horas, por exemplo. Qual é a duração total dessa viagem em número de horas?

Para resolver esse problema, precisamos pegar o número de dias, multiplicar por 24 (já que um dia tem 24 horas) e somar com o número de horas. Na resolução de um algoritmo, é necessário ensinar ao computador quais operações devem ser realizadas para se chegar a uma solução correta para o problema. Ou seja, deve-se primeiro entender como solucionar o problema para depois passá-lo para o algoritmo.

É importante salientar que na resolução de algoritmos, no geral, existem diversas formas de se chegar a um resultado satisfatório. Por exemplo, para calcular o dobro de um número, pode-se resolver esse problema multiplicando o número por 2, ou, então, somando o número com ele mesmo. Ambas as soluções produzem um resultado correto.

1.1.6 Pensar em todos os detalhes

Uma analogia geralmente realizada pelos autores de livros de Algoritmos e Lógica de Programação é a de que criar um algoritmo pode ser comparado com o processo da criação de uma receita de bolo. Esse exemplo é muito interessante. Na montagem de uma receita de bolo, temos os ingredientes (como os dados de entrada), as ações a serem realizadas sobre os ingredientes (processamento) e o resultado esperado, que é o bolo em si (como os dados de saída).

Esquecer algum ingrediente ou o detalhe de alguma ação certamente fará com que o bolo não fique conforme o planejado. Na construção de algoritmos vale a mesma regra. Vamos imaginar uma tarefa simples a ser convertida em um algoritmo: acender um palito de fósforo. Quais etapas são necessárias para realizar essa tarefa? Vamos pensar, agora avançando um pouco além do que foi visto na Seção 1.1.3:

1. Pegar uma caixa de fósforo.
2. Abrir a caixa de fósforo.
3. Verificar se tem palito. Se Sim:
 - 3.1 Retirar um palito.
 - 3.2 Fechar a caixa.
 - 3.3 Riscar o palito.
 - 3.4 Verificar se acendeu. Se Sim:
 - 3.4.1 Ok! Processo Concluído.
 - 3.5 Se não: Retornar ao passo?
4. Se não: Descartar a caixa e retornar ao passo 1.

Para qual passo o item 3.5 deve retornar? Poderia ser o 3.3? Mas quantas vezes? Talvez o palito estivesse úmido... Isso não pode ser infinito. E, após a ocorrência desse número de repetições, o que fazer? Descartar o palito e voltar ao passo 3.1 para pegar outro palito? Cuidado, depois de “Retirar o palito”, a caixa foi fechada. Portanto, precisaríamos retornar ao passo 2.

Esse exemplo poderia ainda conter muitas outras verificações. Mas, da forma como foi organizado, já nos permite extrair alguns pontos a serem relacionados ao processo de criação de um programa.

Na montagem de um programa, utilizamos comandos sequenciais, comandos para definição de condições e comandos para criação de estruturas de repetição. Os comandos sequenciais são os mais simples. Uma ação realizada após a outra (passos 1, 2, 3). As condições servem para determinar quais comandos serão executados a partir da análise de uma condição. Se a condição retornar verdadeiro, o programa segue por um caminho, se falso, por outro (3. Verificar se tem palito? ou 3.4, Verificar se

acendeu?). Já as estruturas de repetições indicam que uma ação ou conjunto de ações devem ocorrer várias vezes (retornar ao passo 2 e retornar ao passo 1). Nessas estruturas, é preciso indicar quantas vezes a repetição vai ocorrer ou criar algum ponto de saída no laço.

Outro aspecto importante que pode ser observado no exemplo é que os passos (comandos) estão formatados na sua margem esquerda. Esse processo se chama indentação do código (relacionada com a palavra inglesa *indentation* – com referência a recuar). Ele serve para facilitar a compreensão do código e é uma prática que deve ser seguida tanto em programas simples quanto em programas maiores e complexos.

1.2 Entrada, processamento e saída

No processo de aprendizado de algo novo, é importante memorizar um roteiro de etapas a serem seguidas. Para dar os primeiros passos, esse roteiro nos auxilia a organizar o nosso pensamento. Em lógica de programação, o roteiro para resolver a maioria dos programas iniciais é:

- a) Leia os dados de *entrada*.
- b) Realize o *processamento* dos dados.
- c) Apresente a *saída* dos dados.

Em programas maiores, essas etapas vão se intercalar e outras ações serão necessárias, como salvar os dados em um banco de dados. No entanto, depois de concluir o aprendizado inicial, a nossa compreensão terá avançado e estará apta para lidar com essas situações.

A etapa da entrada de dados consiste em solicitar ao usuário alguma informação. Nome, idade, salário – por exemplo. Após, deve acorrer a etapa do processamento. Calcular o novo salário, calcular um desconto ou verificar a idade são exemplos dessa etapa. E, por fim, o nosso programa deve apresentar a saída de dados. A exibição do novo salário, do desconto ou se a pessoa é maior ou menor de idade são exemplos de respostas para os dados de um programa.

Para realizar cada uma dessas etapas, as linguagens de programação

utilizam comandos. Há um ou vários comandos para realizar a entrada de dados. A sintaxe (palavra e formato) do comando difere de uma linguagem para outra. Mas todas as linguagens dispõem de comandos para realizar essa etapa. Ou seja, o conceito é o mesmo, mas o comando é diferente.

Já a etapa de processamento, no geral, implica trabalhar com variáveis para realizar um cálculo. De igual forma, as linguagens têm sintaxes um pouco diferentes para declarar e realizar a atribuição de dados para uma variável. Em JavaScript, PHP, Java, Python, a atribuição é realizada a partir do símbolo “=”. Em Pascal e Delphi, pelos símbolos “:=”.

O mesmo ocorre para a etapa de saída de dados para o usuário. `alert`, `print` e `echo` são exemplos de comandos utilizados nas linguagens a fim de exibir uma mensagem em um programa.

É importante ressaltar que nos programas de teste que estaremos desenvolvendo é o próprio programador que faz o papel de usuário. Então, vamos escrever o código, executar as ações para rodar o programa, informar os dados de entrada e conferir se a resposta fornecida pelo programa está correta. Ufa... são muitas tarefas. Mas, no final, você verá que vale a pena. Que a profissão de programador compensa pelas oportunidades que ela proporciona.

1.3 JavaScript

A linguagem JavaScript foi criada pela Netscape Communications Corporation junto com a Sun Microsystems. Sua primeira versão foi lançada em 1995. Na época, quem dominava o mercado de browsers era o Netscape Navigator. Após, tivemos um período bastante conturbado de guerra entre browsers, no qual códigos da linguagem funcionavam em um navegador e não funcionavam em outro. Em 1996, a fim de evitar esse problema, a Netscape decidiu entregar o JavaScript para a ECMA (European Computer Manufacturers Association). A ECMA é uma associação dedicada à padronização de sistemas de informação. Em 1997, foi lançada a primeira edição da linguagem gerenciada por essa associação. Por isso, a linguagem JavaScript também é chamada de ECMAScript e as

versões da linguagem estão associadas a esse nome. Então, temos a ECMAScript 6, ECMAScript 7, ECMAScript 2018 etc. No endereço <https://www.ecma-international.org/> você pode acompanhar as novidades da linguagem, conforme ilustra a Figura 1.2.

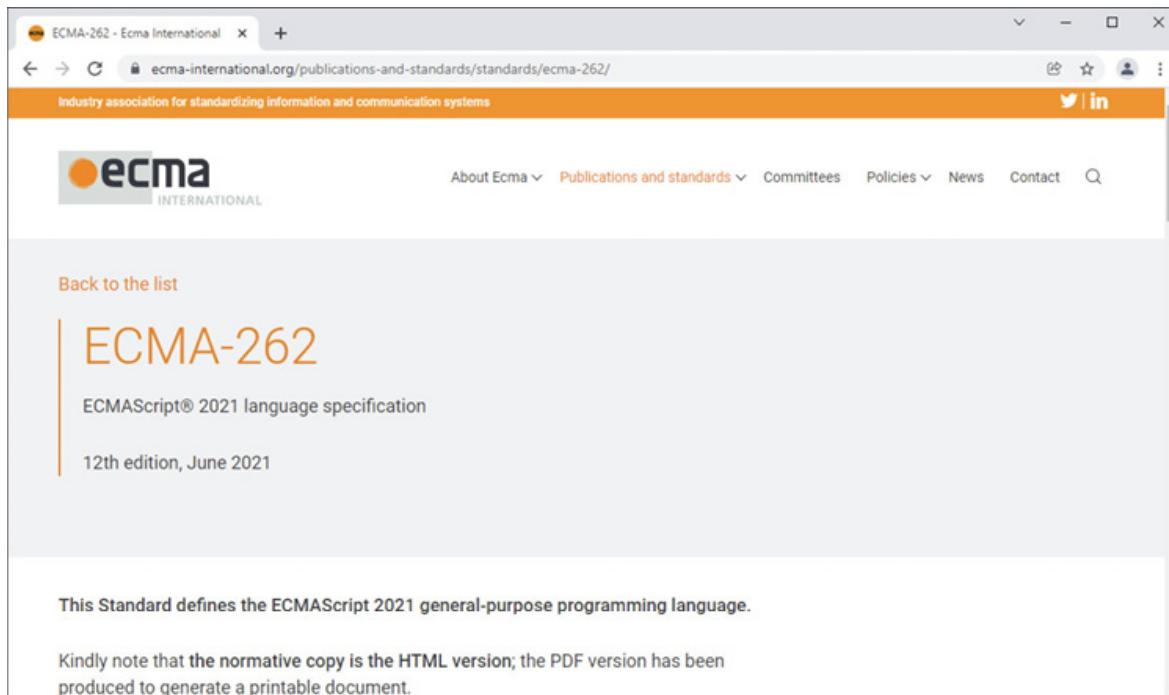


Figura 1.2 – Últimas novidades das versões ECMAScript.

JavaScript possui um importante papel no processo de desenvolvimento de páginas para internet, junto com HTML (HyperText Markup Language) e CSS (Cascading Style Sheets). O HTML serve para descrever o conteúdo de uma página web e definir a marcação semântica (significado) dos elementos que compõem a página. O CSS determina os estilos e a formatação dos elementos, ou seja, utiliza-se o CSS para definir a aparência do site – cores, bordas, espaçamentos etc. É a apresentação da página em si, cuja implementação geralmente fica aos cuidados do Web Designer. Completando a tríade, a linguagem JavaScript é utilizada para definir o comportamento dos elementos da página. Os códigos escritos em JavaScript, também chamados de scripts, são interpretados diretamente pelos navegadores web.

As funcionalidades que podem ser inseridas em páginas web a partir da

linguagem JavaScript são inúmeras. Com JavaScript, podemos interagir com os visitantes de uma página a partir de campos de formulário, acessar e modificar o conteúdo e as características de uma página, salvar informações no navegador do usuário, auxiliar na construção de layouts complexos, exibir opções de compras e alterar características de produtos de acordo com as escolhas dos clientes ou, ainda, criar jogos interativos que rodam no browser do internauta em qualquer plataforma (Windows, Mac ou Linux) e em qualquer dispositivo (computador, tablet ou celular).

Apenas para citar alguns exemplos de páginas com a implementação de funcionalidades JavaScript, podemos recorrer aos programas desenvolvidos neste livro:

- Em um site de uma loja de esportes, trocar o símbolo e a cor de fundo do título da página para a cor do clube pelo qual o cliente torce, a partir da seleção de um campo de formulário. Salvar a escolha no browser do cliente, a fim de manter esses dados para as próximas visitas do cliente ao site (Exemplo 9.1).
- Em um site de uma pizzaria, criar um aplicativo para o garçom informar os itens de um pedido. O aplicativo pode conter funcionalidades visando agilizar a seleção dos itens de acordo com o tipo do produto a ser inserido no pedido e listar os produtos já selecionados (Exemplo 8.4).
- Em um site de uma empresa de eventos, montar o layout com as centenas de poltronas de um teatro disponíveis para reserva em um determinado show musical. Permitir a seleção de uma poltrona pelos usuários e trocar as figuras das poltronas ocupadas (Exemplo 11.2).
- Em um site de uma escola infantil, criar um jogo de “Descubra a Palavra” com links para um adulto cadastrar palavras (e dicas sobre as palavras) e ver a relação das palavras salvas no browser. O jogo seleciona uma palavra aleatória para a criança adivinhar – letra a letra. A criança também pode ver as dicas e acompanhar a troca das imagens de acordo com as chances que lhe restam (Exemplo 11.3).

Algumas pesquisas de abrangência mundial destacam a linguagem de programação JavaScript, colocando-a no topo de suas classificações. Essas

pesquisas utilizam diversos fatores para medir o ranking das linguagens, como análise dos arquivos publicados no repositório GitHub ou pesquisas realizadas em sites como o Stack Overflow. Essa discussão sobre linguagens, geralmente, produz longos debates. As pesquisas também diferem, uma vez que, dependendo da plataforma, os índices mudam. Contudo, esteja tranquilo quanto à importância de aprender JavaScript. É uma linguagem muito valorizada no mercado de trabalho atual. A Figura 1.3 destaca uma dessas pesquisas sobre linguagens e tecnologias.

No processo de programação de sistemas web, há linguagens que rodam no lado cliente e que rodam no lado do servidor. JavaScript é utilizada principalmente para rodar scripts no lado do cliente, embora também seja crescente o número de aplicações desenvolvidas com a linguagem para rodar no lado do servidor. Mas o que significa dizer que uma linguagem roda no lado do cliente? Significa que o próprio navegador web (Chrome, Internet Explorer/Edge, Firefox, Safari, Opera) deve conter funcionalidades capazes de interpretar o código JavaScript e executá-lo. Já as linguagens que rodam no lado do servidor são executadas em um programa instalado no servidor web (Apache, IIS) e retornam para a máquina do cliente apenas o código resultante dessa execução. PHP, ASP, Java, C#, Python e também JavaScript são exemplos de linguagens que rodam no lado do servidor.

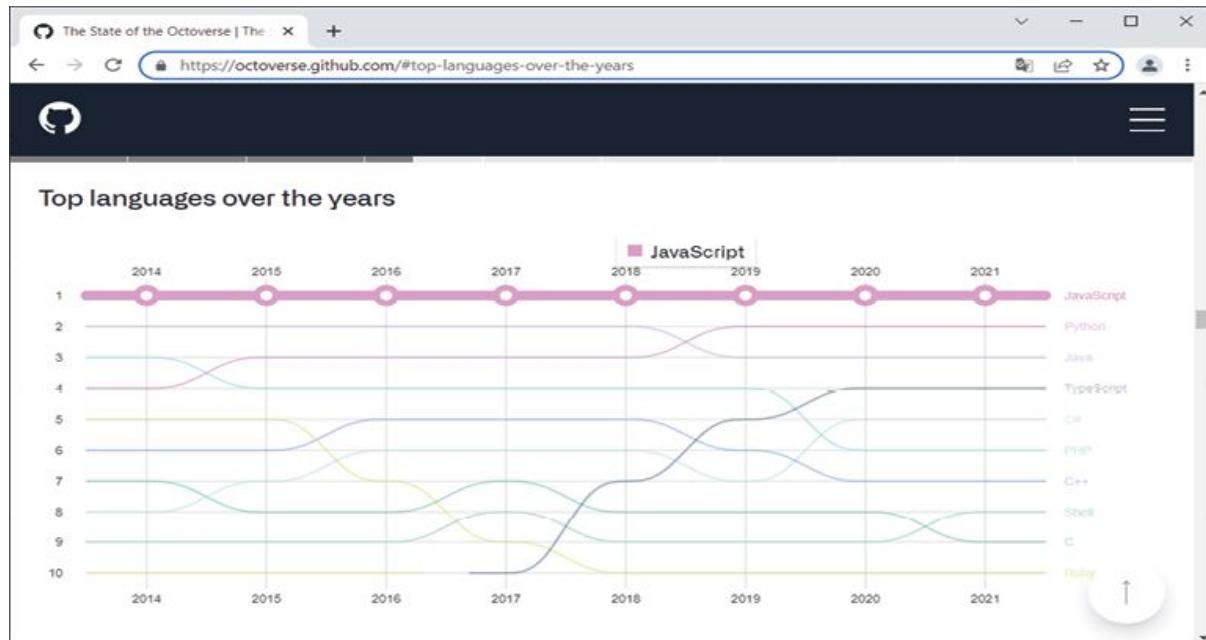


Figura 1.3 – JavaScript é destaque no ranking das linguagens do GitHub
[\(https://octoverse.github.com/#top-languages-over-the-years\)](https://octoverse.github.com/#top-languages-over-the-years).

Este livro não pretende ser uma referência sobre JavaScript, mas utilizar os recursos da linguagem para o ensino de Lógica de Programação. Na prática, ao finalizar a leitura e desenvolver os exemplos construídos ao longo dos capítulos, você estará apto a implementar os próprios programas JavaScript e explorar os demais recursos da linguagem. Para um aprofundamento maior na linguagem JavaScript, recomendo os livros *JavaScript – Guia do Programador*, de Maurício Samy Silva, e *Aprendendo JavaScript*, de Shelley Powers – da editora Novatec.

JavaScript também é uma linguagem orientada a objetos. Esse paradigma de programação é essencial para o desenvolvimento de aplicações profissionais já há algum tempo. O objetivo deste livro é explicar os passos iniciais para você se tornar um programador de computadores. Esteja ciente de que vários outros passos ainda precisam ser dados para você se tornar um profissional completo. A fim de avançar nos estudos em programação, deixo a sugestão dos livros *Princípios de Orientação a Objetos em JavaScript*, de Nicholas C. Zakas, e *Estruturas de dados e algoritmos em JavaScript*, de Loiane Groner – ambos publicados pela Novatec.

Entendo que o aprendizado de Algoritmos se dá na sua essência pela prática de exercícios de programação. Por isso, este livro foca nos exemplos e exercícios. Espero que o leitor acompanhe a explicação dos exemplos e crie os códigos de cada exemplo. Também é possível baixar os exemplos no site da editora Novatec. Nos exercícios, dedique um tempo para criar cada programa. Mesmo que pareçam simples, eles são fundamentais para o processo de aprendizagem. Em programas maiores, dominar esses recursos simples, como criar condições ou repetições, por exemplo, serão fundamentais para que você consiga resolver o problema da maneira correta. No site da editora, estão disponíveis os programas com uma forma de resolução para cada exercício proposto (visto que, no geral, existem várias formas de se resolver um algoritmo).

1.4 Editores de código JavaScript

Para criar programas JavaScript, podemos utilizar editores simples que já estão instalados no computador, editores online disponíveis na internet ou instalar programas profissionais de edição de código. Editores simples, como o bloco de notas no Windows, não são muito recomendados pela ausência de recursos de auxílio ao programador. Os editores online contêm alguns recursos extras e a vantagem de poder acessar e compartilhar os códigos na internet. Já os editores profissionais contêm diversos recursos que facilitam o desenvolvimento dos programas.

Os editores online possibilitam testar diretamente os programas em um navegador web. São exemplos dessa classe de editores JavaScript os sites w3schools.com e js.do. No site w3schools.com – da companhia Refsnes Data, há uma descrição dos recursos da linguagem e de seus comandos (em inglês). Nos exemplos apresentados, há o botão **Try It Yourself** (tente você mesmo), que permite a visualização e a alteração do código. Ao clicar no botão **Run**, você pode verificar o que cada alteração implica no resultado do programa. A Figura 1.4 apresenta um exemplo de tela do site da w3schools.com. No quadro da esquerda, está o código HTML/JavaScript que pode ser editado e, no quadro da direita, o resultado da execução do programa.

Já os editores profissionais apresentam uma série de recursos que nos auxiliam no desenvolvimento de aplicações. Autocompletar os comandos (IntelliSense), alertas de erros de sintaxe, formatação (indentação) de código, cores diferentes para os comandos e integração com GitHub são alguns exemplos. E o melhor, há versões gratuitas de ótimos editores de código disponíveis na internet, como Visual Studio Code, Netbeans IDE e Sublime – que são multiplataforma (ou seja, possuem versões para Windows, Mac e Linux). Sugiro que você baixe e instale o Visual Studio Code. Ele pode ser obtido no endereço code.visualstudio.com/download, conforme ilustra a Figura 1.5. Esse software será utilizado nos exemplos deste livro.

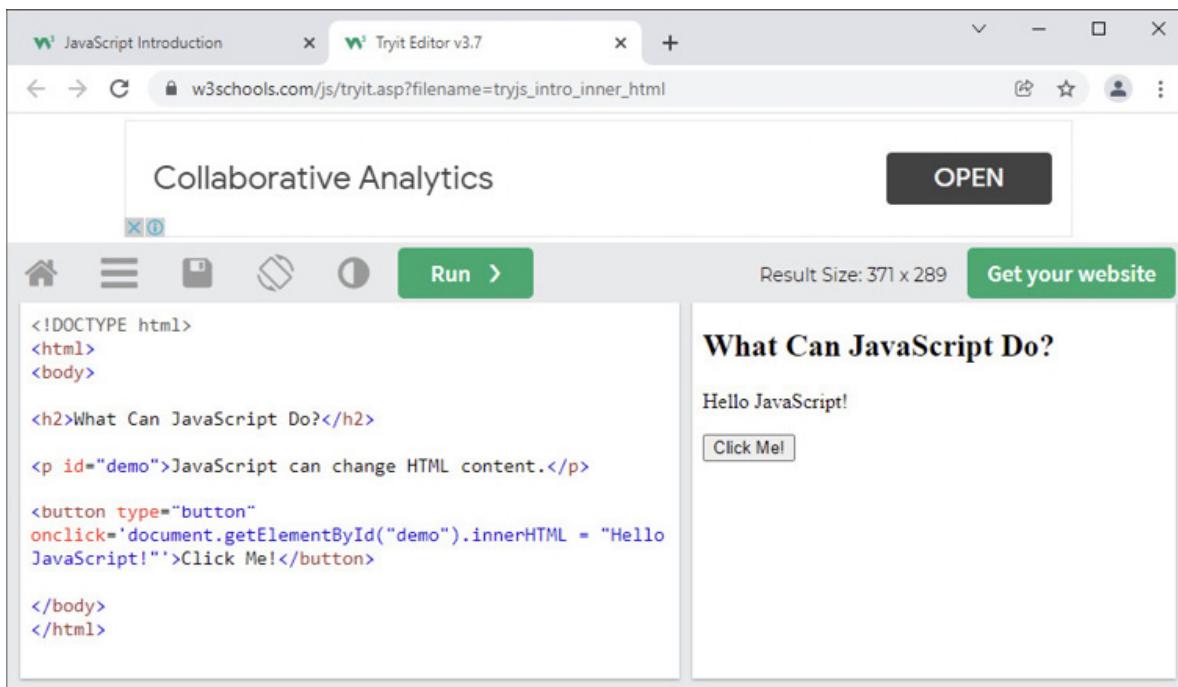


Figura 1.4 – Site da *w3schools.com*.

Para realizar a instalação do Visual Studio Code, selecione a versão de acordo com o seu sistema operacional e baixe o instalador. O processo de instalação é simples. Leia e aceite os termos de licença, informe o local onde o Visual Studio Code será instalado, **Próximo > Próximo > Instalar > Concluir**.

Na primeira execução do Visual Studio Code, é apresentada a tela com as opções iniciais de configuração, junto com uma caixa de alerta que permite a instalação do pacote de idiomas (Figura 1.6). Faça a instalação desse pacote caso prefira utilizar o editor com as opções de menu e mensagens em português.

As opções de “Passo a passo” são exibidas na página inicial, conforme ilustra a Figura 1.7. Com elas é possível ajustar a aparência do editor de códigos, aprender sobre os conceitos básicos de VS Code e explorar recursos visando aumentar sua produtividade na edição dos programas. Para começar um novo arquivo, clique no menu **Arquivo > Novo Arquivo (Ctrl+N ou Cmd+N)**. Uma janela com um documento em branco e sem nome é exibida.

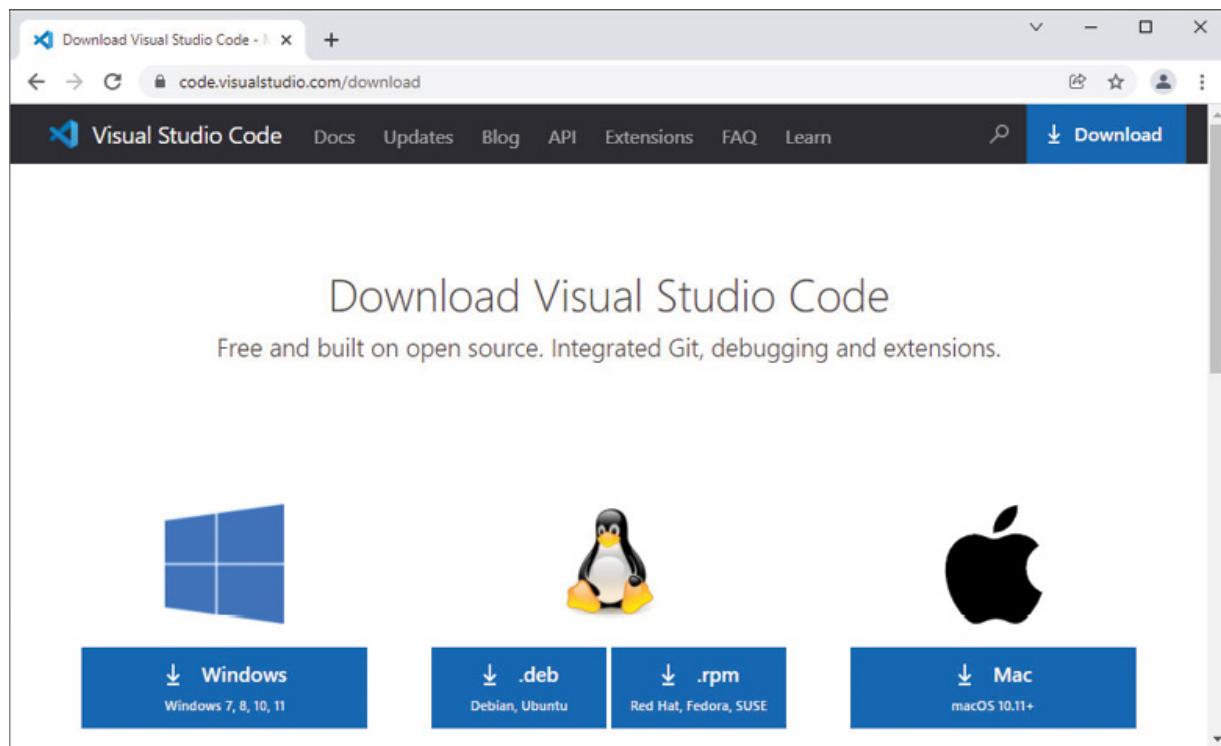


Figura 1.5 – Página de Download do Visual Studio Code.

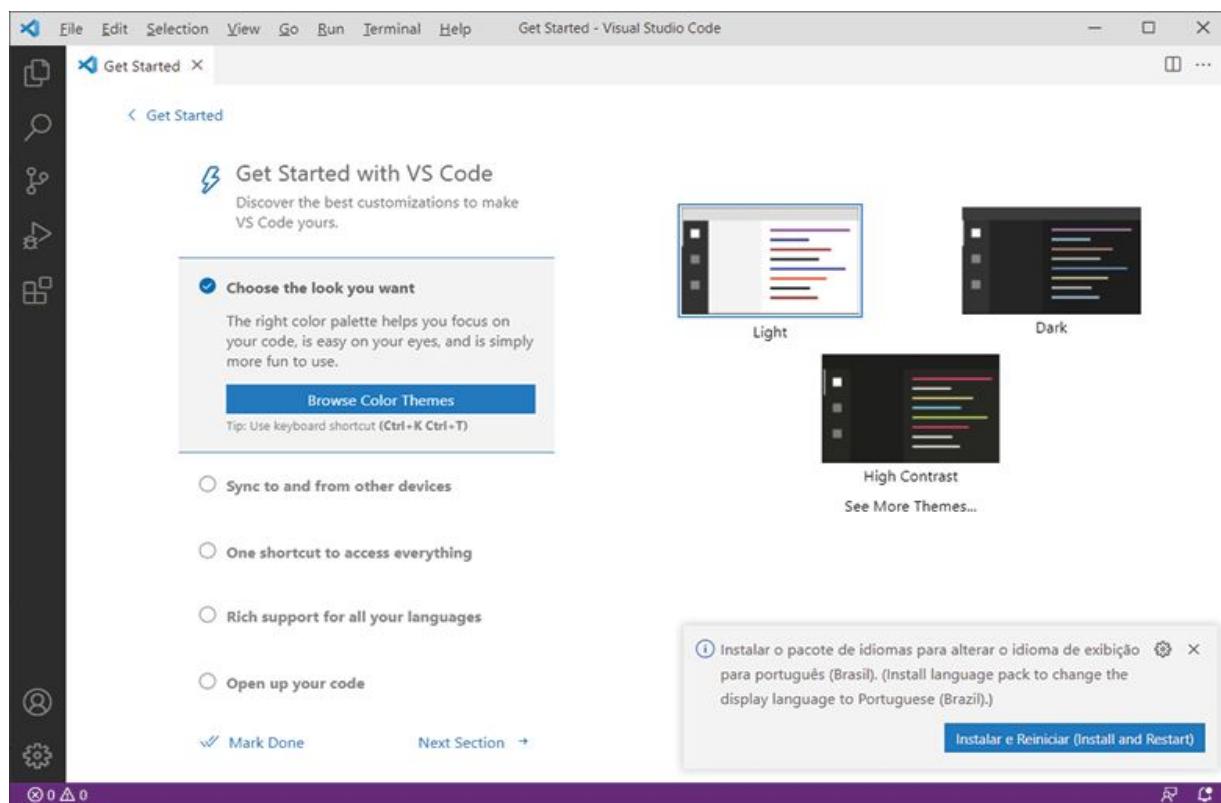


Figura 1.6 – Tela inicial do Visual Studio Code com link para instalar

pacote de idiomas.

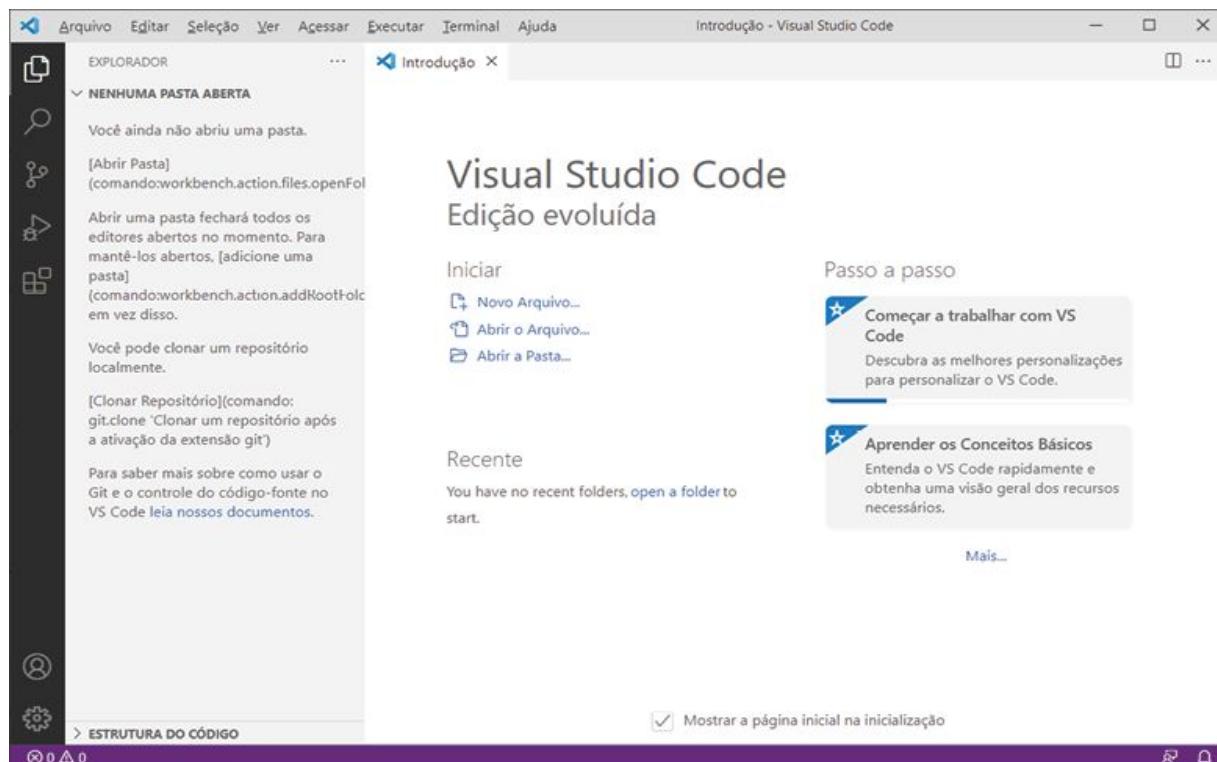


Figura 1.7 – Página inicial do Visual Studio Code.

1.5 Saída de dados com alert() e console.log()

Pronto para realizar seus primeiros testes de programação? Antes de começarmos a digitar os códigos de alguns exemplos, vamos definir a estrutura das pastas em que os arquivos serão salvos. Recomendo que você crie uma pasta específica para os exemplos deste livro, como `livrojs` na raiz do disco C:. Dentro dessa pasta, crie uma nova pasta para cada capítulo. Iniciamos, portanto, pela criação da pasta `cap01`. Dessa forma, nossos programas ficam organizados e fáceis de localizar. Para os nomes de programas, também vamos adotar um padrão: `ex1_1.html`, `ex1_2.html` e assim sucessivamente para os exemplos do Capítulo 1. Já os arquivos contendo os exemplos de resposta dos exercícios finais de cada capítulo são salvos com o nome `resp1_a.html`, `resp1_b.html` etc.

Caso esteja utilizando um computador em que não gostaria de instalar o Visual Studio Code, você pode rodar os programas deste capítulo, acessando um dos sites de editores online destacados anteriormente. No Visual Studio Code, para que o editor disponibilize alguns auxílios que serão discutidos ao longo do livro, é necessário indicar o tipo de documento que está sendo criado. Isso pode ser feito a partir da barra de status do editor. Ao clicar sobre **Texto sem Formatação**, uma lista de tipos de arquivos é apresentada, conforme ilustra a Figura 1.8. Nossos primeiros documentos devem ser do tipo HTML. Outra forma de indicar o tipo do arquivo é salvá-lo logo no início da edição, selecionando o seu tipo. Faça isso e já salve o documento em branco com o nome ex1_1.html (dentro da pasta c:\livrojs\cap01).

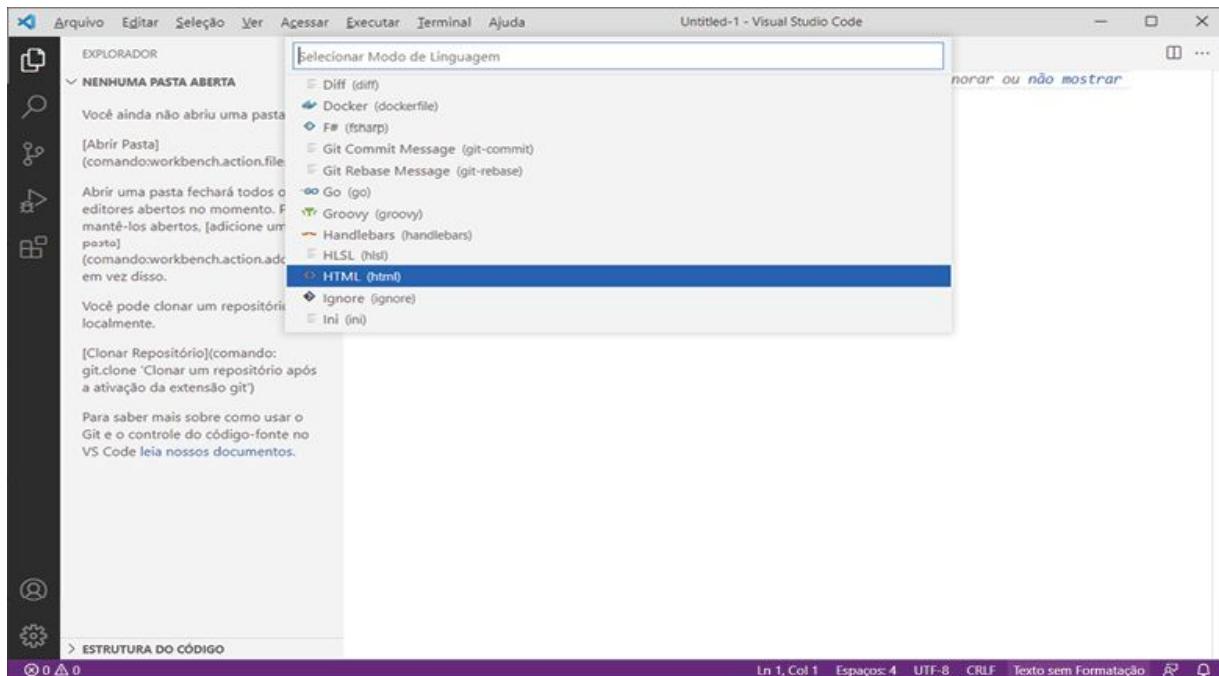


Figura 1.8 – Ao clicar em “Texto sem Formatação” na barra de status, pode-se selecionar o tipo do arquivo.

Vamos então ao primeiro exemplo. Começamos com algo bem simples: apresentar uma mensagem ao usuário. Na tela em branco do Visual Studio Code, informe os seguintes comandos:

Exemplo 1.1 – Saída de dados com alert (ex1_1.html)

```
<script>
```

```
alert("Bem-Vindo ao Mundo JavaScript!")
console.log("Meu primeiro programa...")
</script>
```

Depois de salvar o arquivo, é necessário executá-lo, ou, no caso do JavaScript, renderizá-lo em um navegador. Para isso, abra o seu navegador favorito e na barra de endereços informe o caminho onde você salvou o arquivo c:\livrojs\cap01\ex1_1.html. Você também pode digitar apenas uma parte desse caminho e depois selecionar o arquivo. Outra opção é ir até a pasta em que a página foi salva, clicar com o botão direito do mouse sobre o arquivo, selecionar **Abrir com** e escolher um dos navegadores instalados no computador. Ao executar o programa, a mensagem que você escreveu dentro das aspas “Bem-Vindo ao Mundo JavaScript!” é exibida em uma caixa no centro da tela, conforme ilustra a Figura 1.9.

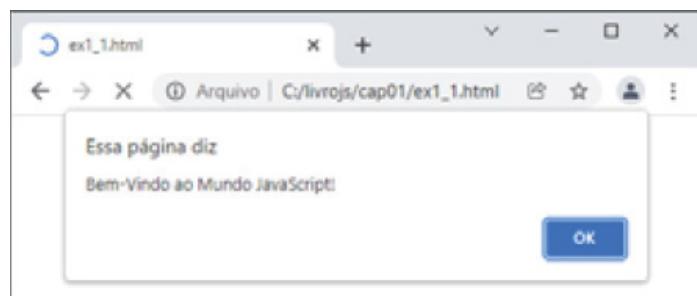


Figura 1.9 – Exemplo de saída de dados em uma caixa de alerta.

Para visualizar a saída gerada pelo comando `console.log()`, você deve procurar por **Ferramentas do Desenvolvedor** no seu browser e selecionar o item **console**. No Google Chrome, você pode obter o mesmo resultado ao pressionar a tecla **F12**. A Figura 1.10 exibe a saída gerada pelo `console.log()`.

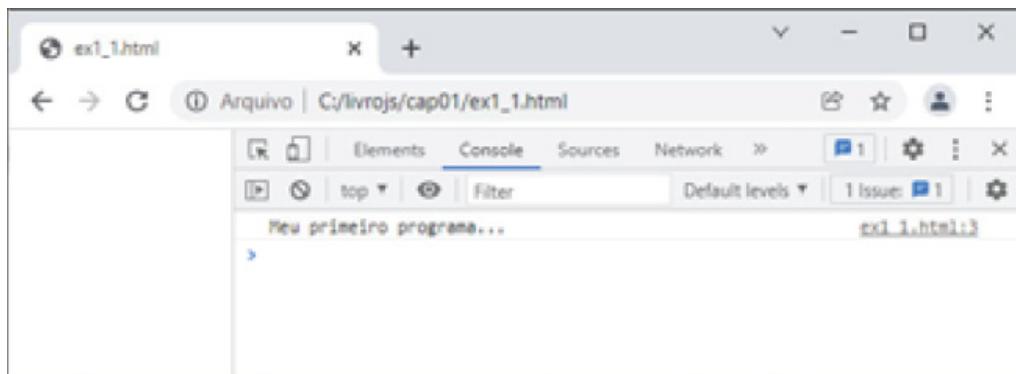


Figura 1.10 – Saída de dados gerada por um comando `console.log()`.

Certifique-se de ter digitado os comandos da mesma forma como no exemplo. As linguagens de programação são rígidas quanto à sintaxe dos comandos. Muitas delas, como o JavaScript, são *case sensitive*, ou seja, diferenciam letras maiúsculas de minúsculas. Se a caixa de alerta não foi exibida quando você mandou abrir o arquivo no navegador, verifique cuidadosamente se os comandos digitados estão corretos. Após os ajustes, salve o arquivo e abra novamente a página no navegador (ou pressione **F5**). Você vai se acostumar a ser cuidadoso com a escrita do código com o passar do tempo. No Capítulo 5, será apresentada uma forma de localizar possíveis erros no código – processo conhecido como depuração de programas.

Todos os exemplos e exercícios deste livro estão disponíveis para download no site da Editora Novatec. Recomendo que escreva o código exemplificado no livro. Caso aconteça algum problema, recorra ao código disponível no site da editora.

Os programas deste capítulo são dedicados a apresentar os primeiros passos de programação. Por isso, utilizaremos o comando/método `alert()` para exibir mensagens em uma caixa no navegador. Este e os demais métodos JavaScript precisam estar delimitados por `<script>` e `</script>`. No segundo capítulo, vamos avançar um pouco mais e trabalhar as formas de integração do código JavaScript com HTML. Os termos comando, função, método ou procedimento podem ter pequenas diferenças quanto ao conceito, dependendo da linguagem. Por enquanto, entenda que eles são palavras-chaves da linguagem que servem para executar uma ação no programa.

1.6 Variáveis e constantes

Uma das principais funções de um programa é interagir com os usuários. Uma das formas de realizar essa interação é pela solicitação de informações e, a partir dessas informações, implementar ações e apresentar respostas. Imagine um terminal de caixa eletrônico que possibilite saques de apenas 100 reais. Pouco útil, já que os clientes do banco que possuem saldo inferior a 100 reais não poderiam utilizá-lo e aqueles que precisam de 500 reais, por exemplo, teriam de realizar 5 vezes a mesma operação. Haveria ainda

aqueles que precisam de um valor não múltiplo de 100. Ou seja, o melhor é solicitar ao cliente o valor a ser sacado. Para isso é que existem as variáveis. As variáveis são espaços alocados na memória do computador que permitem guardar informações e trabalhar com elas – como o valor que o cliente deseja sacar no terminal do caixa eletrônico. Como o nome sugere, os valores armazenados em uma variável podem ser alterados durante a execução do programa. São exemplos de variáveis manipuladas em um programa: a descrição, a quantidade e o preço de um produto ou, então, o nome, o salário e a altura de uma pessoa.

As variáveis declaradas em um programa devem possuir um nome, seguindo algumas regras de nomenclatura. Em JavaScript, os nomes de variáveis não podem:

- Conter espaços.
- Começar por número.
- Conter caracteres especiais, como +,-,* , /, %, (,),{,},!,@,#.
- Utilizar nomes de palavras reservadas da linguagem, como function, var, new, for OU return.

Variáveis escritas com letras maiúsculas são diferentes de variáveis escritas com letras minúsculas. O uso do caractere “_” é válido, porém dê preferência para declarar variáveis com o nome em letras minúsculas e o uso de uma letra maiúscula para destacar palavras compostas (padrão denominado *camelcase*). São exemplos de nomes válidos de variáveis: cidade, nota1, primeiroCliente, novoSalario, precoFinal, dataVenda. Procure usar nomes que indicam o conteúdo que a variável vai armazenar.

Para declarar uma variável em JavaScript, podemos utilizar os comandos var, let ou const. Nas últimas versões do JavaScript, passou-se a recomendar o uso de const ou let. Uma variável criada a partir de um desses comandos possui um escopo local (de bloco), o que pode evitar desperdício de memória – pois, ao final do bloco, a variável deixa de existir. Além disso, const e let impedem que um programa rode com alguns problemas no código, como declarar duas variáveis com o mesmo nome no script.

Para fazer com que uma variável receba um dado, utiliza-se o conceito de atribuição. Em JavaScript, a atribuição de valor para uma variável é feita com o sinal “=”. É possível declarar uma variável e atribuir-lhe diretamente um valor com o uso da palavra reservada `const` (ou `let`) e do sinal de atribuição da seguinte forma:

```
const idade = 18
```

Em algumas linguagens, o uso de constantes é restrito apenas à declaração de variáveis de controle ou de configuração do sistema, como o número máximo de alunos a serem manipulados pelo programa ou os dados de conexão com um banco de dados.

No JavaScript moderno, o uso de `const` passou a ter um novo significado e tem se tornado o padrão da linguagem. Declarar uma variável com `const` serve para indicar que essa variável deve possuir uma única atribuição de valor e não será alterada no decorrer do programa. Então, caso o conteúdo da variável possa ser alterado, declare-a com `let`. Em todos os demais casos, opte pelo `const`.

1.7 Entrada de dados com `prompt()`

Vamos avançar um pouco? Já apresentamos uma mensagem na tela em nosso primeiro exemplo. Vamos agora receber uma informação e apresentar uma mensagem utilizando a informação recebida. Para isso, vamos utilizar o conceito de variável visto na seção anterior e aprender um novo comando JavaScript. Para receber dados do usuário, uma das formas possíveis em JavaScript é utilizar o comando(método) `prompt()`, que exibe uma caixa com um texto e um espaço para digitação. Crie um segundo programa, com os códigos do Exemplo 1.2.

Exemplo 1.2 – Entrada de dados e uso de variáveis (ex1_2.html)

```
<meta charset="UTF-8">
<script>
  const nome = prompt("Qual é o seu nome?")
  alert("Olá " + nome)
</script>
```

Observe a execução do programa nas figuras 1.11 e 1.12. Inicialmente, é solicitado o nome do usuário, a partir do método prompt (Figura 1.11). Após, a mensagem de “Olá “ seguida do nome digitado é apresentada (Figura 1.12).

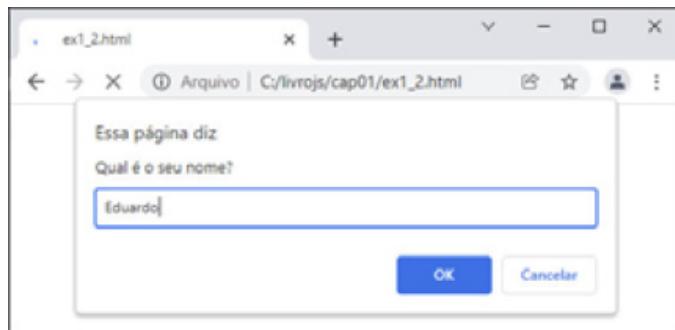


Figura 1.11 – Execução do método prompt.

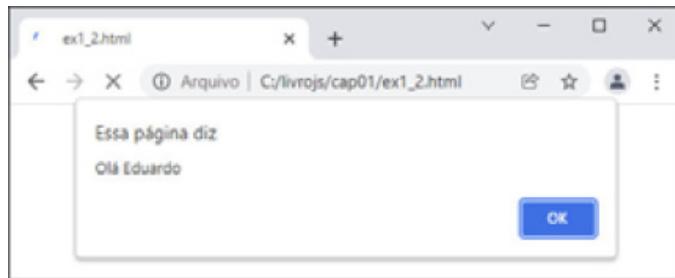


Figura 1.12 – Execução do alert com um texto concatenado com uma variável.

Vamos aos detalhes desse programa. Como destacado no primeiro exemplo, os códigos JavaScript devem ser delimitados por `<script>` e `</script>`. Antes deles foi adicionado um comando HTML, também chamado de tag, ou nesse caso metatag, que serve para ajustar os caracteres de acentuação a serem exibidos pela página. A linha 3 contém o comando:

```
const nome = prompt("Qual é o seu nome?")
```

Esse comando realiza as tarefas de declarar uma variável e executar o método `prompt()`. O nome digitado pelo usuário na caixa de diálogo do `prompt` é atribuído à variável `nome`.

A linha 4 contém o método `alert()`, também visto no primeiro exemplo. No entanto, há um importante detalhe nesse comando.

```
alert("Olá " + nome)
```

Há um texto entre aspas, que é um texto fixo a ser exibido na execução do comando, concatenado (+) com a variável nome. O fato de nome estar fora das aspas indica que naquele local deve ser apresentado o conteúdo da variável, e não um texto fixo. O resultado será a exibição da palavra Olá, seguida do nome digitado pelo usuário, conforme ilustrado na Figura 1.12.

Outra forma de exibir mensagens que contenham um texto fixo e o conteúdo de uma variável em JavaScript é com o uso da chamada Template Strings. Para isso, deve-se delimitar a mensagem entre ` (crases) e inserir o nome das variáveis usando a sintaxe: \${nomeVar}. Observe a linha a seguir:

```
alert(`Olá ${nome}`);
```

O uso do ponto e vírgula ";" no final dos comandos em programas JavaScript é opcional.

1.8 Comentários

No exercício profissional de programador de sistemas, você vai atuar em diversos projetos. Provavelmente em várias linguagens. E, eventualmente, terá de dar manutenção em algum sistema que você não mexe já há um bom tempo. Talvez, ainda, em uma linguagem diferente da que você está trabalhando no momento. Nessa hora, você vai perceber a importância de adicionar comentários aos seus programas. Um comentário é uma observação inserida pelo programador no código, com o objetivo de explicar algum detalhe do programa. Explicar para quem? Provavelmente para ele mesmo, no futuro, quando algum cliente ligar pedindo uma alteração em um sistema antigo.

No cenário das grandes empresas, que desenvolvem projetos em equipes, a inserção de comentários nos códigos de um programa assume um papel ainda mais relevante.

Os comentários não afetam a execução do programa. Eles são ignorados pela linguagem, mas essenciais para o programador. É fundamental adicionar comentários para explicar os detalhes de alguma fórmula utilizada em um sistema, como o cálculo do décimo terceiro salário em um sistema de folha de pagamento. Também é de fundamental importância adicionar

comentários para salientar a função de algum comando não utilizado com frequência. Ou, então, alguma particularidade da linguagem ou sistema em desenvolvimento.

Os comentários também são importantes quando estamos programando um sistema maior, que será concluído em outro momento ou após alguma outra tarefa. Dessa forma, ao retomar a implementação do programa, a leitura dos comentários vão nos auxiliar a manter a linha de raciocínio, além de facilitar a compreensão do ponto em que havíamos parado.

Entenda que gastar alguns segundos digitando um breve comentário no seu sistema poderá economizar um bom tempo quando você precisar dar manutenção nesse sistema no futuro. Então, não poupe nas linhas de comentário nos sistemas que você vai desenvolver.

Em JavaScript, os comentários podem ser inseridos para uma linha ou várias, utilizando os seguintes caracteres:

```
// para comentários de uma linha
```

```
/*
  para comentários de duas
  ou mais linhas
*/
```

1.9 Tipos de dados e conversões de tipos

As variáveis manipuladas em um programa são de um determinado tipo. Em JavaScript, os tipos principais de dados são strings (variáveis de texto), números e valores booleanos (true ou false). Saber o tipo de uma variável nos permite identificar quais operações são possíveis para essa variável. Ou, então, qual o comportamento dessa variável nas fórmulas em que elas estão inseridas. Nesse contexto, há algumas particularidades na linguagem JavaScript. Vamos apresentar uma dessas particularidades no Exemplo 1.3, no qual o resultado do cálculo é exibido ao lado de cada variável como um comentário (//).

Exemplo 1.3 – Operações envolvendo strings e números

(ex1_3.html)

```
<script>
  const a = "20"
  const b = a * 2 // b = 40
  const c = a / 2 // c = 10
  const d = a - 2 // d = 18
  const e = a + 2 // e = 202 ???
  alert("e: " + e) // exibe o valor de uma variável
</script>
```

Nesse exemplo, temos uma variável do tipo string que recebe “20” (const a = “20”). Ela é entendida como sendo do tipo string por estar delimitada por aspas. Nas operações de multiplicação, divisão e subtração, a linguagem converte esse texto em número e o valor retornado está de acordo com o esperado. Contudo, quando realizamos a adição, o valor de retorno é diferente do padrão, pois a linguagem concatena (+) o texto com o número, algo semelhante ao que foi feito no Exemplo 1.2.

Para resolver esse problema, precisamos converter o texto em número. Isso pode ser feito, em JavaScript, pelos métodos Number(), parseInt() e parseFloat(). Vamos utilizar o método Number() para facilitar o processo de aprendizagem. Dessa forma, modifique a linha que declara a variável “e” para conter o método Number():

```
const e = Number(a) + 2 // e = 22
```

Agora, a variável “a” é inicialmente convertida em número, e, após, ocorre a soma do valor 2, gerando como resultado o valor 22.

Caso seja necessário converter um número em string, para utilizar métodos como os de preenchimento de espaços que estão disponíveis apenas para variáveis do tipo string – por exemplo, devemos utilizar o método toString(). Ele será utilizado em alguns exemplos do livro.

Em JavaScript não é obrigatório definir o tipo da variável na sua declaração. Ela assume um tipo no momento da atribuição de valor à variável. A atribuição de um conteúdo entre aspas (simples ou dupla, mas sempre aos pares) cria uma variável do tipo String. Para variáveis numéricas, não devem ser utilizadas as aspas. As variáveis booleanas podem conter os valores true ou false. Elas serão exploradas em nossos exemplos, nos

próximos capítulos. As entradas de dados realizadas com o método `prompt()` criam variáveis do tipo `String`, exceto se houver uma função de conversão de dados como `Number()`. Exibir uma variável que não recebeu uma atribuição de valor vai gerar uma saída “`undefined`”. O Exemplo 1.4 declara variáveis e exibe seus tipos a partir do comando `typeof`.

Exemplo 1.4 – Tipos de variáveis (ex1_4.html)

```
<script>
  const fruta = "Banana"
  const preco = 3.50
  const levar = true
  let novoValor
  console.log(typeof fruta)    // string
  console.log(typeof preco)   // number
  console.log(typeof levar)    // boolean
  console.log(typeof novoValor) // undefined
</script>
```

A declaração de uma variável/constante com `const` exige uma atribuição de valor – que se manterá fixo durante o programa ou bloco de código onde foi declarada. Além da possibilidade de verificar se uma variável é do tipo `number`, também é possível verificar se o número é inteiro ou possui decimais: `Number.isInteger()` faz essa verificação:

```
console.log(Number.isInteger(12)) // true
console.log(Number.isInteger(3.50)) // false
```

1.10 Exemplos de entrada, processamento e saída

Já vimos como realizar a entrada de dados em JavaScript, os conceitos de variáveis e tipos de dados, além de duas formas de exibir uma resposta ao usuário (saída de dados). Vimos também que, no geral, para elaborar um programa simples, é necessário realizar três etapas: entrada, processamento e saída. Vamos agora implementar alguns exemplos de algoritmos de programação sequencial, os quais realizam essas etapas.

a) Elaborar um programa que leia um número. Calcule e informe o dobro

desse número.

- Entrada de dados: ler um número
- Processamento: calcular o dobro
- Saída: informar o dobro

Os comandos necessários para realizar essas operações são apresentados no Exemplo 1.6. Os comentários são opcionais.

Exemplo 1.5 – Cálculo do dobro de um número (ex1_5.html)

```
<meta charset="utf-8">
<script>
    const num = prompt("Número: ") // lê um dado de entrada
    const dobro = num * 2        // calcula o dobro
    alert("Dobro é: " + dobro)   // exibe a resposta
</script>
```

Como estamos realizando uma operação de multiplicação, não é necessário converter a entrada de dados realizada pelo método `prompt()` – que retorna sempre um texto, em número. Contudo, se quisermos calcular o dobro do número a partir de uma operação de adição, a conversão é necessária. Para isso, pode-se utilizar o método `Number()` direto na entrada de dados, como demonstra a linha a seguir:

```
const num = Number(prompt("Número: ")) // lê dado de entrada e converte para número
```

b) Elaborar um programa que leia dois números. Calcule e informe a soma desses números.

- Entrada de dados: ler dois números
- Processamento: calcular a soma
- Saída: informar a soma

Para realizar a leitura dos dois números, vamos declarar as variáveis `num1` e `num2`. Lembre-se de que os nomes de variável não devem conter espaços e não podem começar por número. O exemplo implementa uma forma de resolução para esse programa. Observe que o método `prompt()` é utilizado duas vezes e, como deve ser realizada uma soma, é necessário converter a entrada em número.

Exemplo 1.6 – Soma de dois números (ex1_6.html)

```
<meta charset="utf-8">
<script>
const num1 = Number(prompt("1º Número: ")) // lê os números
const num2 = Number(prompt("2º Número: "))
const soma = num1 + num2 // calcula a soma
alert(`Soma é: ${soma}`) // exibe o resultado
</script>
```

c) Elaborar um programa que leia o valor de um jantar. Calcule e informe o valor da taxa do garçom (10%) e o valor total a ser pago.

Vamos avançar um pouco. Neste e nos demais exercícios, vamos apresentar um exemplo de possíveis dados de entrada e dos respectivos dados de saída exibidos pelo programa (e deixar para você enumerar as etapas).

```
Valor do Jantar R$: 80.00
Taxa do Garçom R$: 8.00
Total a Pagar R$: 88.00
```

O exemplo de dados do programa serve apenas para ilustrar e reforçar o que é solicitado na descrição do exercício. Tenha cuidado de digitar valores com decimais separados por ponto, e não vírgula. Outro detalhe refere-se à compreensão do enunciado do exercício e ao uso correto de variáveis. Utilizando variáveis, o programa vai receber o valor do jantar, calcular corretamente a taxa do garçom e o valor total a ser pago. O Exemplo 1.7 apresenta uma maneira de resolver esse problema. Observe que o método `toFixed(2)` é adicionado às variáveis de saída. Ele serve para indicar que o valor a ser apresentado deve conter 2 casas decimais.

Exemplo 1.7 – Cálculo do valor do jantar (ex1_7.html)

```
<meta charset="utf-8">
<script>
const jantar = Number(prompt("Valor do Jantar R$: ")) // lê o valor do jantar
const garcom = jantar * 0.10 // calcula variáveis de saída
const total = jantar + garcom
alert(`Taxa Garçom R$: ${garcom.toFixed(2)}\nTotal R$: ${total.toFixed(2)}`)
</script>
```

A Figura 1.13 ilustra os valores exibidos para um jantar de R\$ 80.00. A quebra de linha é gerada pela “`\n`”, inserida no método `alert()`, antes de “Total

R\$: ”.

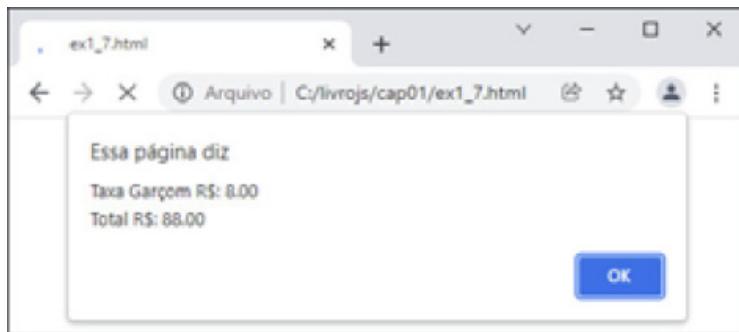


Figura 1.13 – Uso do toFixed() para determinar o número de casas decimais das respostas.

Nesse exemplo, o cálculo da taxa do garçom é uma multiplicação. Porém, para calcular o total, utilizamos a adição. Portanto, a entrada precisa ser convertida. Para facilitar, vamos padronizar nossos programas, definindo que todas as entradas numéricas devem ser convertidas em número.

Para calcular a taxa do garçom, realizamos uma operação de multiplicação do valor do jantar por 0.10. Mas por que por 0.10? Por que 10% é, como falamos: 10 por 100. Ou seja, 10 / 100, que resulta em 0.10. Há outras formas de calcular 10%, como dividir o valor por 10. Mas, se quisermos calcular 12%, por exemplo, a fórmula de multiplicar o valor por 0.12 continua válida.

Também o cálculo do valor total pode ser feito a partir do uso de outras fórmulas. Como:

```
const total = jantar + (jantar * 0.10) // ou então,  
const total = jantar * 1.10           // desta forma
```

Como o valor total é o valor do jantar acrescido de 10% (taxa do garçom), essas duas fórmulas estão igualmente corretas. Na primeira fórmula, o valor total recebe o valor do jantar + o valor do jantar multiplicado pelos 10%. Na segunda, o valor total recebe o valor do jantar multiplicado por 1.10. Esse 1.10 é calculado a partir da seguinte ideia: multiplicar qualquer valor por 1 resulta no próprio valor. Como queremos adicionar 10%, devemos somar 0.10 ao 1, resultando em 1.10.

O importante é você entender uma dessas maneiras. Com a realização dos

exercícios, você vai aprimorar o entendimento sobre as formas de calcular os dados de saída de um programa e aplicá-las de acordo com os problemas a serem solucionados.

d) Elaborar um programa que leia a duração de uma viagem em dias e horas. Calcule e informe a duração total da viagem em número de horas.

Exemplo de dados de entrada e saída do programa (para uma viagem que dura 2 dias + 5 horas).

Nº Dias: 2

Nº Horas: 5

Total de Horas: 53

Observe que o programa deve ler duas variáveis. Com base nessas variáveis, precisamos pensar uma forma de calcular o valor total das horas e converter isso em uma fórmula matemática que esteja correta para quaisquer valores válidos de entrada. Sabendo que um dia tem 24 horas, precisamos multiplicar o número de dias por 24 e adicionar o número de horas. O código 1.10 descrito a seguir apresenta um exemplo de resolução para esse problema.

Exemplo 1.8 – Cálculo da duração de horas de uma viagem (ex1_8.html)

```
<meta charset="utf-8">
<script>
    const dias = Number(prompt("Nº Dias: ")) // lê os dados de entrada
    const horas = Number(prompt("Nº Horas: "))
    const total = (dias * 24) + horas      // calcula a duração
    alert(`Total de Horas: ${total}`)       // exibe o total
</script>
```

1.11 Exercícios

Vamos realizar alguns exercícios de programação sequencial com o uso das caixas de diálogo `prompt()` e `alert()`. Lembre-se de que a construção de exercícios é fundamental para o aprendizado de algoritmos. Então, reserve algum tempo para essa atividade. Em seguida, confira os exemplos de respostas disponíveis no site da editora. Os arquivos estão na pasta `cap01`,

com os nomes resp1_a.html, resp1_b.html, resp1_c.html e resp1_d.html. Como destacado neste capítulo, um algoritmo pode ser solucionado de várias maneiras, desde que exiba corretamente as respostas para todos os possíveis valores de entrada.

a) Elaborar um programa que leia um número. Calcule e informe os seus vizinhos, ou seja, o número anterior e posterior.

Exemplo:

Número: 15

Vizinhos: 14 e 16

b) Elaborar um programa para uma pizzaria, o qual leia o valor total de uma conta e quantos clientes vão pagá-la. Calcule e informe o valor a ser pago por cliente.

Exemplo:

Valor da Conta R\$: 90.00

Número de Clientes: 3

Valor por cliente R\$: 30.00

c) Elaborar um programa para uma loja, o qual leia o preço de um produto e informe as opções de pagamento da loja. Calcule e informe o valor para pagamento à vista com 10% de desconto e o valor em 3x.

Exemplo:

Preço R\$: 60.00

À Vista R\$: 54.00

Ou 3x de R\$: 20.00

d) Elaborar um programa que leia 2 notas de um aluno em uma disciplina. Calcule e informe a média das notas.

Exemplo:

1^a Nota: 7.0

2^a Nota: 8.0

Média: 7.5

1.12 Considerações finais do capítulo

Este capítulo objetivou destacar os pontos essenciais necessários para o aprendizado de lógica de programação com JavaScript, os quais são:

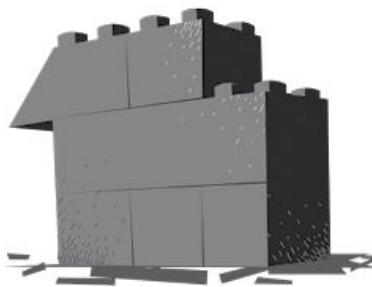
- Para você se tornar um programador, deve ser persistente e cuidadoso com os detalhes da codificação. Sua organização lógica do programa pode estar correta, mas, se um pequeno erro de grafia (sintaxe) existir no código, ele não é executado.
- Lógica de programação é algo que se aprende com treinamento. Ao criar um algoritmo, vamos “ensinar” o computador a realizar uma tarefa. Com a compreensão e a prática dos exercícios, passamos a assimilar a forma como as estruturas de um programa devem estar organizadas. Usamos a lógica para montar corretamente essas estruturas.
- Uma boa regra a seguir para resolver os primeiros exercícios de programação é que todo o programa tem três etapas: entrada, processamento e saída. Ou seja, ele recebe alguns dados, executa alguma operação sobre esses dados e apresenta a resposta.
- JavaScript é uma linguagem de destaque no cenário atual. É muito valorizada por empresas de desenvolvimento web pela capacidade de adicionar inúmeros recursos nas páginas, visando, por exemplo, à interação com os usuários e à criação de layouts profissionais.
- Existem diversas opções de editores de código JavaScript. Alguns estão disponíveis online e são úteis para realizarmos pequenos testes. Para criar programas maiores, instale um editor profissional que contém diversos recursos visando auxiliar o trabalho do programador. Há ótimas alternativas de editores gratuitos na internet. O Visual Studio Code é um deles.
- A linguagem JavaScript dispõe dos comandos (métodos) `prompt()` e `alert()` para realizar pequenas interações com os usuários. Eles nos permitem praticar as etapas de entrada e saída de dados de um algoritmo. Também se pode utilizar o método `console.log()` para exibir uma saída de dados que é apresentada no console, nas Ferramentas do Desenvolvedor, dos browsers web.
- Variável é um conceito fundamental para a criação de programas. São as

variáveis que permitem guardar os dados de entrada, armazenar um cálculo ou outro processamento e, a partir delas, exibir dados de saída personalizados para cada interação de um usuário do sistema.

Para nos tornarmos desenvolvedores de sistemas, é preciso construir um conhecimento sólido sobre a base, assim como na construção de um grande edifício, em que o alicerce é fundamental para as demais estruturas. Construir e entender os exemplos e exercícios deste capítulo são igualmente fundamentais para que você avance nos estudos sobre lógica de programação.

CAPÍTULO 2

Integração com HTML



Para desenvolver uma página web, devemos criar um arquivo HTML (HiperText Markup Language) contendo as tags (comandos) HTML que definem o conteúdo e a semântica dos elementos que constituem a página. Depois de salvar o arquivo, ele deverá ser aberto em um navegador web que vai renderizar (visualizar) esse documento. Nenhum processo adicional é necessário. Os códigos de programas JavaScript são desenvolvidos para adicionar um comportamento à página. Igualmente, não é preciso compilar o programa ou outra ação adicional. O próprio navegador web contém um interpretador para os programas JavaScript. Eles são inseridos nas páginas web em uma seção delimitada pelas tags `<script>` e `</script>` ou em um arquivo `.js` que deve ser referenciado pelo documento HTML.

Com JavaScript, podemos interagir de diversas formas com os usuários de páginas web. Os conceitos vistos no Capítulo 1 serão agora empregados para recuperar informações digitadas em campos de formulário de uma página e para exibir os resultados em parágrafos do documento HTML.

Nosso objetivo no livro não é produzir páginas bonitas, mas abordar os conceitos de lógica de programação utilizando a linguagem JavaScript. Caso você já possua conhecimentos em HTML e deseje melhorar o visual dos exemplos, ótimo. Caso você ainda não possua esses conhecimentos, fique tranquilo. Entenda que deve dar um passo de cada vez e que o passo

que você está dando ao ler este livro é muito importante para construir uma carreira profissional na área de TI (Tecnologia da Informação).

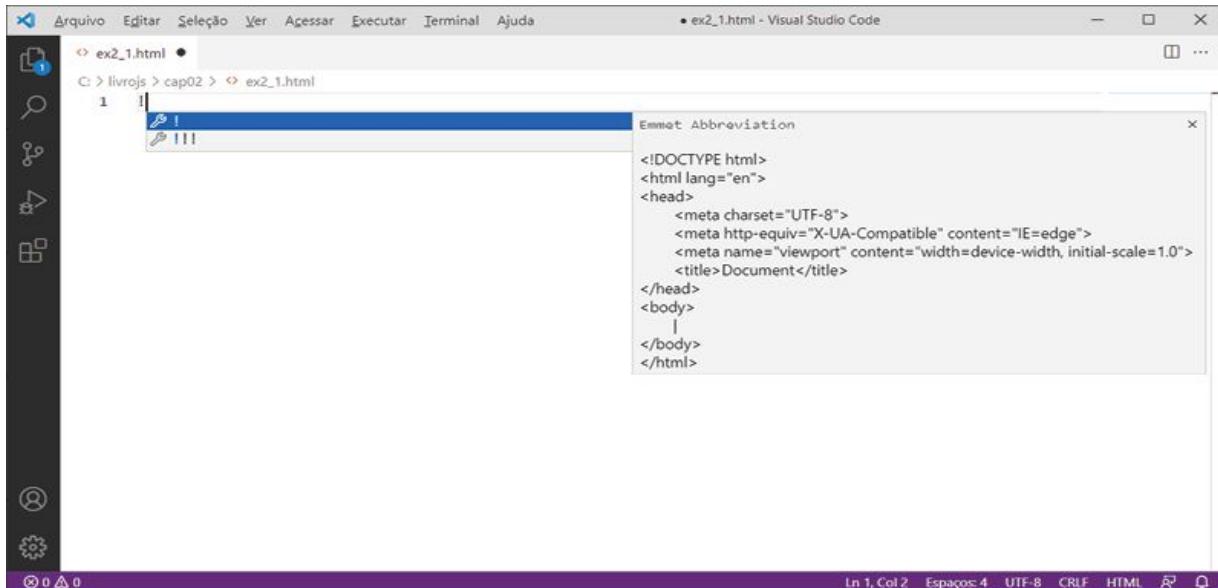
Vamos acrescentar diversos comandos HTML. Alguns servem para definir a estrutura básica de uma página web. Outros, para exibir textos e campos de formulário. Novos comandos JavaScript também serão abordados. Neste contexto, é importante o uso de um editor profissional, como o Visual Studio Code, cujo processo de instalação foi apresentado no Capítulo 1. Um dos benefícios de um editor profissional é nos auxiliar na digitação dos comandos, que exigem cuidados quanto a sua sintaxe. Ao digitar as letras iniciais de um comando, o editor apresenta as opções de comandos que há àquelas letras. Ao pressionar a tecla **Tab** ou **Enter**, o editor conclui a digitação do comando selecionado. Além disso, há diversos atalhos que inserem um conjunto de códigos, como o que define a estrutura básica de um documento HTML de uma página web.

Vamos agora criar uma nova pasta, chamada `cap02`, dentro da pasta `livrojs`, a fim de manter organizados os exemplos e exercícios deste livro. Em seguida, abra o Visual Studio Code.

2.1 Estrutura básica de um documento HTML

Para criar um novo arquivo HTML, devem ser inseridas algumas tags que definem as seções e configurações básicas do documento. Você pode digitá-las uma vez, salvar o documento e recorrer a ele para copiar e colar essas linhas. Mas também pode fazer isso de uma forma bem mais simples.

No Visual Studio Code, inicie um novo arquivo (**Arquivo > Novo Arquivo**), salve o documento como sendo do tipo HTML (dentro da pasta `cap02`, com o nome `ex2_1.html`) e depois digite `!.` O editor vai apresentar um recurso que permite inserir um modelo de códigos no documento (Emmet Abbreviation), conforme ilustra a Figura 2.1. Pressione **Tab** ou **Enter** para que os comandos básicos de uma página HTML sejam inseridos, como mostra a Figura 2.2. Há outros “atalhos” para facilitar a digitação de códigos que serão vistos no decorrer do livro.



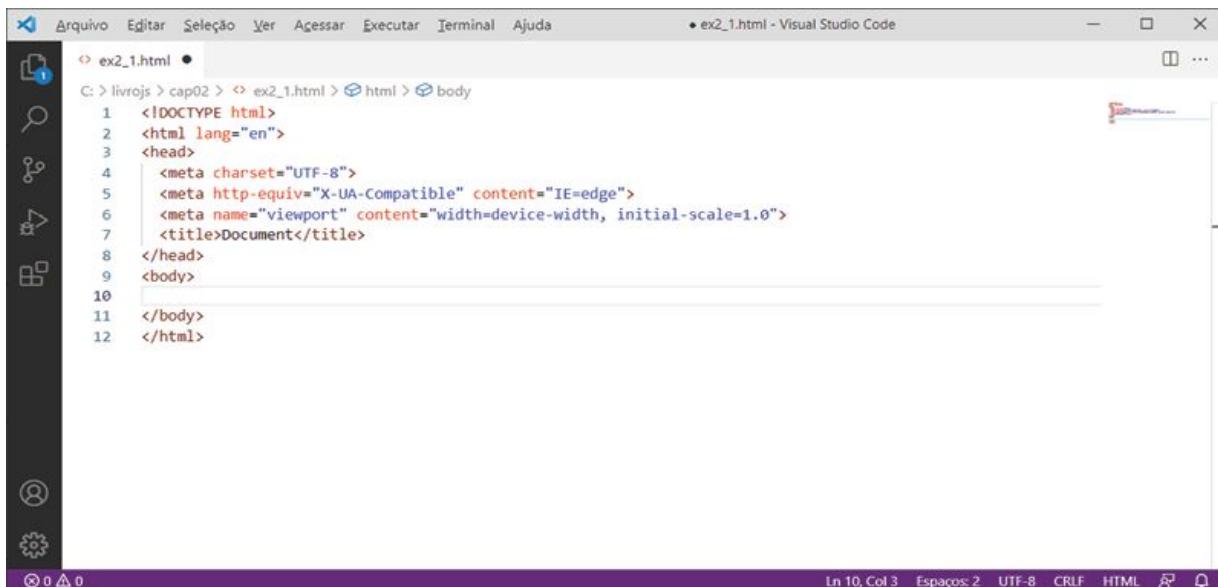
The screenshot shows the Visual Studio Code interface with the title bar "Arquivo Editar Seleção Ver Acessar Executar Terminal Ajuda" and "ex2_1.html - Visual Studio Code". The left sidebar shows the file path "C:\livros\cap02\ex2_1.html". The main editor area has the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
</body>
</html>
```

A tooltip window titled "Emmet Abbreviation" is open over the opening tag of the body element, showing the expanded HTML code.

Figura 2.1 – Depois de salvar o documento como tipo HTML, digite !.

Observe que na linha 2 é especificado o idioma da sua página. Troque o "en" (English) para "pt-br" (Português do Brasil). Definir corretamente o idioma do documento é importante por diversos aspectos, como permitir uma melhor pronúncia por um software de leitura de tela (para portadores de necessidades especiais) e indicar ao browser o dicionário a ser utilizado para a correção gramatical de textos digitados em campos de formulário.



The screenshot shows the Visual Studio Code interface with the title bar "Arquivo Editar Seleção Ver Acessar Executar Terminal Ajuda" and "ex2_1.html - Visual Studio Code". The left sidebar shows the file path "C:\livros\cap02\ex2_1.html". The main editor area has the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
</body>
</html>
```

The code is identical to Figure 2.1, but the cursor is now positioned at the end of the first line of the body content, ready for input.

Figura 2.2 – Ao pressionar Tab ou Enter, os comandos básicos de um documento HTML são inseridos.

Outro detalhe sobre as tags HTML é que elas geralmente são declaradas aos pares. Há `<html>` e `</html>`, `<head>` e `</head>`, `<body>` e `</body>`. As tags `<head>` e `<body>` definem as seções principais da página. Na seção de cabeçalho (`head`), foram inseridas três metatags e o título do documento que você pode alterar conforme o exemplo. O título define o texto a ser exibido em uma aba na barra superior do navegador. `<meta charset="utf-8">` já foi utilizada nos exemplos do Capítulo 1 e serve para definir a página de códigos do documento. A metatag `<meta name="viewport" ...>` está relacionada ao processo de criação de páginas responsivas, ou seja, que respondem adequadamente aos diversos tipos de dispositivos utilizados pelos usuários, como computadores, tablets e smartphones. Já a metatag `<meta http-equiv...>` tem relação com os aspectos de compatibilidade entre navegadores.

2.2 Cabeçalhos, parágrafos e campos de formulário

Vamos acrescentar outras tags HTML ao corpo (`body`) do documento. Digite as seguintes linhas:

```
<h1> Programa Olá Você! </h1>
<form>
  <p> Nome:
    <input type="text" id="inNome">
    <input type="submit" value="Mostrar">
  </p>
</form>
<h3></h3>
```

A tag `<h1>` serve para destacar um texto com um conteúdo relevante no site. Para realizar a entrada de dados, vamos criar campos de formulário, que são a principal forma de interagir com os usuários do site. Os campos de formulário devem estar entre as tags `<form>` e `</form>`. Para receber um texto do usuário, devemos utilizar a tag `<input type="text">`. Cada campo deve possuir um identificador (`id`) a ser utilizado no código JavaScript para obter o conteúdo do campo. E a tag `<input type="submit" ...>`, como o nome sugere, serve para enviar os dados para um destino. Vamos gerenciar esse botão a partir de um programa JavaScript. Esses dois últimos comandos estão dentro de

um parágrafo criado com as tags `<p>` e `</p>`. Já as tags `<h3></h3>` criam uma nova linha, que será utilizada para exibir a mensagem de resposta do programa.

O código completo desse primeiro documento HTML com o acréscimo das linhas do corpo do documento pode ser visto no Exemplo 2.1 destacado a seguir.

Exemplo 2.1 – Documento HTML com campos de formulário (ex2_1.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exemplo 2.1</title>
</head>
<body>
  <h1> Programa Olá Você! </h1>
  <form>
    <p> Nome:
      <input type="text" id="inNome">
      <input type="submit" value="Mostrar">
    </p>
  </form>
  <h3></h3>
  <script src="js/ex2_1.js"></script>
</body>
</html>
```

A parte destacada em negrito, referente ao cabeçalho e rodapé, será o padrão utilizado nos exemplos deste capítulo. Depois de concluir a digitação, salve o arquivo. Da mesma forma como visto no Capítulo 1, o próximo passo é renderizar o documento no browser de sua preferência. Para isso, abra o navegador e digite na barra de endereços o caminho do arquivo. Ou, então, vá até a pasta em que você salvou o arquivo, selecione-o, clique com o botão direito do mouse, encontre a opção **Abrir com** e escolha o seu navegador preferido. A Figura 2.3 apresenta a tela do

documento HTML, criada no Exemplo 2.1, aberta no navegador Google Chrome.

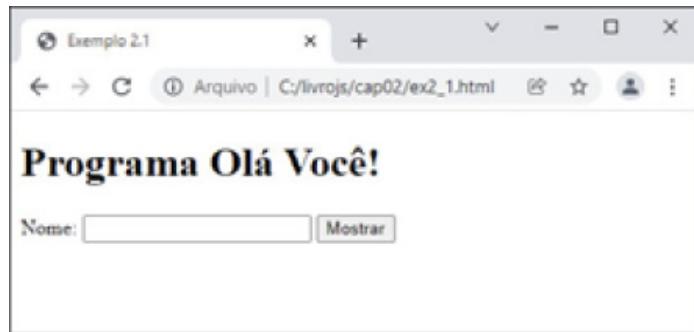


Figura 2.3 – Renderização do código HTML com campos de formulário.

2.3 Criação do programa JavaScript

Concluído o código HTML, responsável por renderizar a página no browser, vamos agora criar o arquivo JavaScript – que, por sua vez, será responsável por adicionar um comportamento à página. Observe que o código HTML já contém um link para esse programa JavaScript na linha:

```
<script src="js/ex2_1.js"></script>
```

Crie, portanto, uma pasta js, dentro da pasta cap02, e nela insira os seguintes comandos, a serem detalhados nos próximos subcapítulos:

Programa JavaScript que exibe o nome informado pelo usuário no campo de edição (js/ex2_1.js)

```
// cria referência ao form e ao elemento h3 (onde será exibida a resposta)
const frm = document.querySelector("form")
const resp = document.querySelector("h3")

// cria um "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    const nome = frm.inNome.value // obtém o nome digitado no form
    resp.innerText = `Olá ${nome}` // exibe a resposta do programa
    e.preventDefault() // evita envio do form
})
```

Existem também outras formas de adicionar programação JavaScript a uma página HTML. Contudo, a forma atualmente recomendada, e que será utilizada nos exemplos do livro, segue o padrão apresentado nos programas

ex2_1.html e ex2_1.js.

2.4 Métodos querySelector() e getElementById()

Para referenciar um elemento HTML identificado no documento, podemos utilizar os métodos `querySelector()` ou `getElementById()`. Esses métodos permitem referenciar qualquer elemento da página - como um campo de formulário, um parágrafo, um botão, uma imagem, entre outros. Com o método `getElementById()`, para que um elemento HTML seja referenciado, ele precisa conter um atributo `id`. Já o método `querySelector()` é mais amplo, e permite criar uma referência a um elemento HTML pela sua tag `name`, `id` ou classe. Observe os exemplos:

```
const resp = document.querySelector("h3") // primeiro elemento h3 da página
const cor = document.querySelector("#inCor") // elemento com id=inCor
const lista = document.querySelector(".lista") // elemento da class=list
```

Uma desvantagem que havia no `querySelector()` em relação ao `getElementById()` é que o `querySelector()` não é suportado por versões antigas dos navegadores. Ou seja, seu programa não funcionará corretamente se você tentar executá-lo em algum desses navegadores (com versões lançadas antes de 2008 ou 2009, conforme o navegador).

Para resolver esse problema, ou seja, permitir que façamos uso de recursos modernos de programação JavaScript, sem nos preocupar com essa questão, surgiram os chamados “transpiladores” de código. Caso o seu programa precise funcionar em versões antigas dos browsers, você recorre ao transpiler e ele converterá o seu código moderno para instruções equivalentes compatíveis com esses browsers. O Babel (babeljs.io) é um dos principais softwares desse segmento para JavaScript.

Além do `querySelector()`, que cria uma referência ao primeiro elemento da página que corresponda ao seletor, existe também o método `querySelectorAll()`. Ele será discutido no capítulo 9, após abordarmos vetores, visto que cria uma lista com todos os elementos que correspondam ao seletor.

Observe ainda, nos exemplos, que podemos armazenar a referência a um

elemento em uma variável e depois obter a sua propriedade, como descrito a seguir:

```
const frm = document.querySelector("form")
const nome = frm.inNome.value
```

Ou, então, utilizar um único comando, acessando diretamente a propriedade que queremos obter, como a seguir.

```
const nome = document.querySelector("form").inNome.value
```

Se o programa trabalhar com o mesmo elemento mais de uma vez, é recomendado armazenar a localização dele em uma variável (exemplo de duas linhas). Caso contrário, você pode fazer uma referência direta a sua propriedade (exemplo de uma linha). Nos exemplos do livro, daremos preferência pelo uso dos comandos em duas linhas, a fim de padronizar um formato (facilita o aprendizado) e também para evitar linhas longas que (no livro) dificultariam a leitura e a compreensão do código.

Outra questão importante está relacionada à padronização dos nomes de variáveis a serem utilizadas no programa. Nos exemplos do livro, os campos de formulário serão precedidos pelas letras `in` (`input`), e espaços de saída, quando identificados, serão precedidos por `out` (`output`).

2.5 Introdução a eventos e funções

Ao carregar a página HTML do Exemplo 2.1, o browser apresentou o nome do programa, um campo de texto precedido pela palavra `Nome` e um botão. A página está ali estática, esperando que você digite um nome e clique no botão. Ao clicar sobre o botão, uma função deve ser executada. Muito da programação JavaScript construída em páginas web é desenvolvida desta forma: elas são acionadas a partir da ocorrência de um evento. Quando o usuário executa uma ação, o programa responde ao evento do usuário com uma ou mais ações. O evento mais comum de ser programado para um formulário é o clique no botão `submit`. Mas há diversos outros, como modificar o conteúdo de um campo, clicar sobre um elemento da página, sair de um campo, carregar a página, entre outros. Mais detalhes sobre os eventos serão apresentados no Capítulo 8.

Para criar um evento e definir o que será executado quando esse evento ocorrer, deve-se utilizar uma palavra reservada para indicar para qual evento a linguagem ficará “na escuta”. A palavra reservada pode ser, por exemplo, submit, change, click, blur ou load. Para adicionar um ouvinte de evento a um elemento da página, utiliza-se o método addEventListener(), com o evento e o nome de uma função ou uma arrow function (função de seta) com os comandos a serem executados. Observe a sintaxe desse método:

```
frm.addEventListener("submit", (e) => { comandos })
```

Portanto, os exemplos do Capítulo 1, de calcular o dobro de um número ou a média de notas de um aluno serão agora inseridos dentro de um bloco e executados quando um evento ocorrer. Não utilizaremos caixas para solicitar dados e exibir respostas, mas campos de formulário e mensagens em parágrafos do documento.

2.6 Propriedades innerText, innerHTML e value

As propriedades value e innerText serão utilizadas em praticamente todos os programas desenvolvidos neste e nos demais capítulos do livro. A propriedade value obtém ou altera o conteúdo de um campo de formulário HTML. Portanto, para obter o nome do usuário informado no Exemplo 2.1, é preciso utilizar essa propriedade. Já a propriedade innerText serve para alterar ou obter o conteúdo de elementos de texto do documento identificados no código HTML. É possível, portanto, alterar o texto de qualquer parágrafo ou texto de cabeçalho em uma página web utilizando essa propriedade. Há também a propriedade innerHTML, semelhante a innerText quanto aos elementos em que atua, porém renderiza os códigos HTML existentes no seu conteúdo. A Tabela 2.1 destaca a diferença entre as propriedades innerText, innerHTML e value.

Tabela 2.1 – Comparativo entre as propriedades innerText, innerHTML e value

innerText	Consulta ou altera o texto exibido por elementos HTML como parágrafos (p), cabeçalhos (h1, h2,...) ou containers (span, div).
-----------	---

innerHTML	Consulta ou altera o conteúdo de elementos HTML como parágrafos (p), cabeçalhos (h1, h2,...) ou containers (span, div). Códigos HTML presentes no conteúdo são renderizados pelo navegador.
value	Consulta ou altera o conteúdo de campos de formulário.

A propriedade `innerHTML` pode apresentar algum risco relacionado à segurança na construção de páginas web em um tipo de ataque denominado XSS (Cross-Site Scripting). Essa vulnerabilidade explora a exibição de dados contendo códigos que poderiam ser enviados por usuários maliciosos. Para evitar esse problema, é necessário filtrar os dados de entrada de um site. Caso o conteúdo a ser exibido na página pelo programa não contenha dados informados pelo usuário, não há riscos em utilizar o `innerHTML`.

As propriedades `textContent` ou `outerText` são semelhantes ao `innerText`. Sinta-se à vontade, se preferir utilizá-las no lugar de `innerText`.

2.7 Método `preventDefault()`

Por padrão, quando o usuário clica sobre o botão submit de um formulário, uma ação de envio dos dados desse form é executada. Isso faz um reload da página, e tanto o conteúdo dos campos do form quanto das respostas exibidas pelo programa são perdidas. O método `preventDefault()` previne esse comportamento default do botão submit. Observe que ele é aplicado sobre um event passado por parâmetro (e) na construção da arrow function. Esse assunto de passagem de parâmetros será abordado no Capítulo 8.

Neste capítulo, novos termos foram utilizados na descrição dos programas, como objeto, método e propriedade. Observe a seguir, de uma forma resumida, o que cada um deles representa em um programa:

- **Objeto** – representa uma instância de uma classe.
- **Método** – representa uma instrução ou um conjunto de instruções que executam uma tarefa.
- **Propriedade** – representa uma característica (atributo) de um objeto.

No Exemplo 2.1, utilizamos o objeto `document`, que a partir da execução do método `querySelector()` pode referenciar as tags `form` e `h3` da página. Já a propriedade `value` é utilizada para obter o conteúdo digitado no campo de

formulário. A propriedade `innerText`, por sua vez, altera um atributo do documento, que faz com que a resposta seja exibida na página.

Não se preocupe caso algum desses conceitos ainda lhe pareça estranho, eles serão revisados em diversos exercícios neste e nos demais capítulos do livro. Lembre-se de explorar os recursos do editor. Observe que, ao iniciar a digitação de um comando, o Visual Studio Code apresenta uma caixa com sugestões de comandos ou métodos contendo as letras já digitadas. Na Figura 2.4, é apresentado o funcionamento desse recurso denominado IntelliSense na digitação do método `addEventListener()`. Para concluir a digitação do método, pressione **Tab** ou **Enter**.

Outro auxílio importante proporcionado pelo uso de um editor profissional como o Visual Studio Code refere-se à formatação (indentação) do documento. Esta também é considerada uma boa prática de programação, pois facilita a compreensão das estruturas utilizadas no programa – seja no arquivo HTML, seja no programa JavaScript. O atalho do Visual Studio Code para aplicar essa formatação aos comandos do seu programa é **Alt + Shift + F**. Você pode obter um resumo com as teclas de atalho disponíveis no editor acessando o menu **Ajuda > Referência de Atalhos do Teclado**.

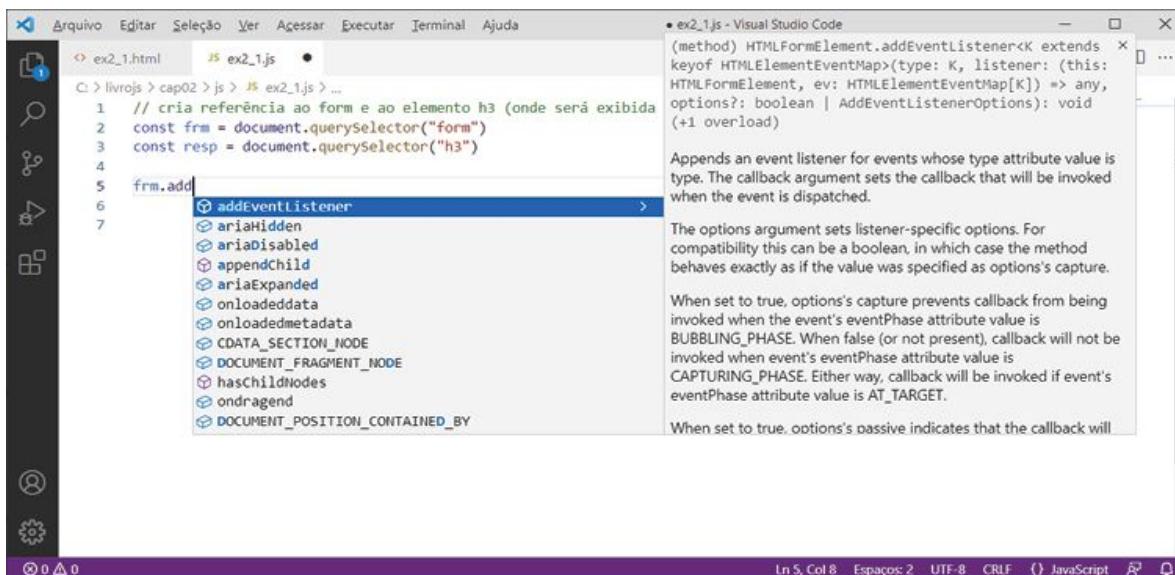


Figura 2.4 – IntelliSense no Visual Studio Code.

Vamos testar o nosso programa! Depois de digitar todos os comandos

conforme os exemplos destacados anteriormente, salve os arquivos e recarregue a página no seu browser favorito. A Figura 2.5 exibe a tela com a execução do programa.



Figura 2.5 – Ao clicar no botão Mostrar, a mensagem Olá seguida do nome é exibida.

2.8 Operadores aritméticos e funções matemáticas

Além dos tradicionais operadores de adição(+), subtração(-), multiplicação(*), divisão (/) e exponenciação (**), as linguagens de programação dispõe também do operador módulo (%). O módulo é utilizado para obter o resto da divisão entre dois números. Observe as seguintes expressões matemáticas:

```
const a = 5 % 2 // a = 1  
const b = 7 % 4 // b = 3
```

Na primeira expressão, a variável a recebe 1 porque 5 dividido por 2 é 2 e o resto é 1. Na segunda expressão, a variável b recebe 3 porque 7 dividido por 4 é 1 e o resto da divisão é 3. O exemplo 2.2 explora um dos usos do operador módulo.

Outros cálculos como raiz quadrada, seno e cosseno podem ser obtidos em JavaScript com o uso das funções matemáticas da classe Math. A Tabela 2.2 apresenta as principais funções matemáticas da linguagem e destaca um exemplo explicado no livro no qual cada função é utilizada.

Tabela 2.2 – Principais funções matemáticas da classe Math

Math.abs(num)	Retorna o valor absoluto de um número, ou seja, se o valor for negativo, ele
---------------	--

	<p>será convertido para positivo. Se positivo, o valor permanece o mesmo.</p> <p>Exemplo: <code>Math.abs(-3) => 3</code></p> <p>Veja uma aplicação de <code>Math.abs()</code> no Exemplo 8.3</p>
<code>Math.ceil(num)</code>	<p>Arredonda o valor para cima. Dessa forma, se o valor possuir decimais, retorna o próximo número inteiro do valor analisado.</p> <p>Exemplo: <code>Math.ceil(4.2) => 5</code></p> <p>Veja uma aplicação de <code>Math.ceil()</code> no Exemplo 9.2</p>
<code>Math.floor(num)</code>	<p>Arredonda o valor para baixo, retornando a parte inteira do número.</p> <p>Exemplo: <code>Math.floor(7.9) => 7</code></p> <p>Veja uma aplicação de <code>Math.floor()</code> no Exemplo 2.2</p>
<code>Math.pow(base, exp)</code>	<p>Retorna a base elevada ao expoente.</p> <p>Exemplo: <code>Math.pow(3, 2) => 9</code></p> <p>Veja uma aplicação de <code>Math.pow()</code> no Exemplo 3.2</p>
<code>Math.random()</code>	<p>Retorna um número aleatório entre 0 e 1, com várias casas decimais. O número aleatório possível inicia em 0 e vai até um valor inferior a 1.</p> <p>Exemplo: <code>Math.random() => 0.6501314074022906</code></p> <p>Veja uma aplicação de <code>Math.random()</code> no jogo do Exemplo 5.2</p>
<code>Math.round(num)</code>	<p>Arredonda o valor para o inteiro mais próximo. A partir de .5 na parte fracionária, o valor é arredondado para cima. Anterior a .5, é arredondado para baixo.</p> <p>Exemplo: <code>Math.round(2.7) => 3</code></p> <p>Veja uma aplicação de <code>Math.round()</code> no Exemplo 6.10</p>
<code>Math.sqrt(num)</code>	<p>Retorna a raiz quadrada do número (square root).</p> <p>Exemplo: <code>Math.sqrt(16) => 4</code></p> <p>Veja uma aplicação de <code>Math.sqrt()</code> no Exemplo 3.5</p>

Ao criar expressões matemáticas, devemos ter o cuidado com a ordem de precedência dos operadores. Observe as duas fórmulas a seguir:

```
const media1 = (nota1 + nota2) / 2
const media2 = nota1 + nota2 / 2
```

O valor das variáveis `media1` e `media2` será o mesmo? Não. Na primeira linha, como foram utilizados os parênteses, a soma de `nota1` e `nota2` terá prioridade sobre a divisão. Na segunda linha, primeiro será realizada a divisão de `nota2` por 2 e o resultado será, então, adicionado a `nota1`. Observe, a seguir, os valores de `media1` e `media2` caso `nota1` receba o valor 7, e `nota2` receba o valor 8.

```
media1
(7 + 8) / 2
15 / 2
7.5
```

media2

7 + 8 / 2

7 + 4

11

Ao montar uma expressão matemática, fique atento à ordem hierárquica de execução dos operadores. Como visto no cálculo do exemplo anterior da média, eles podem alterar significativamente o resultado obtido. A seguir, são destacadas as principais regras matemáticas aplicáveis às fórmulas que podem ser criadas para manipular os dados de um sistema. Os exemplos são utilizados para ilustrar cada situação.

1. Os parênteses redefinem a ordem das prioridades. Podem ser utilizados vários conjuntos de parênteses em uma mesma expressão.

Exemplo...: $10 * (6 - (2 * 2))$

Cálculo(1): $10 * (6 - 4)$

Cálculo(2): $10 * 2$

Resultado.: 20

2. As funções matemáticas ou funções criadas pelo usuário têm prioridades sobre os demais operadores aritméticos.

Exemplo...: Math.sqrt(9) * 8 / 2

Cálculo(1): 3 * 8 / 2

Cálculo(2): 24 / 2

Resultado.: 12

3. Os operadores de multiplicação, subtração e módulo têm prioridade sobre os operadores de adição e subtração.

Exemplo...: $2 + 5 * 2$

Cálculo(1): $2 + 10$

Resultado.: 12

4. Caso uma expressão contenha operadores de mesmo nível de hierarquia, o resultado é calculado da esquerda para a direita.

Exemplo...: $5 / 2 * 3$

Cálculo(1): $2.5 * 3$

Resultado.: 7.5

Naturalmente, você pode modificar a ordem de execução de qualquer fórmula com a inserção de parênteses. Os parênteses também podem ser utilizados em algumas expressões para auxiliar na compreensão do cálculo. Observe a fórmula que foi utilizada no Exemplo 1.8, do primeiro capítulo:

```
const total = (dias * 24) + horas
```

Conforme destacado nas regras anteriores, a multiplicação do número de dias por 24 (horas) vai, naturalmente, ter prioridade na resolução da fórmula. Os parênteses, nesse caso, servem apenas para facilitar a interpretação do cálculo.

Caso tenhamos uma expressão matemática complexa em um programa, podemos dividi-la em fórmulas menores, atribuídas a variáveis que armazenam os valores de cada parte da expressão. Como cada variável ocupa um espaço de memória, isso também implica um consumo maior de memória. Contudo, deve-se analisar cada caso e, dependendo da situação, optar pela forma que privilegia um melhor entendimento dos cálculos de um sistema.

Para realizar pequenos testes e verificar o resultado de fórmulas que utilizam operadores aritméticos de diferentes níveis de hierarquia, pode-se recorrer a um editor online, como visto no Capítulo 1, e utilizar o método `alert()`. Observe o exemplo a seguir.

```
<script>
  const x = 10 / 2 * Math.sqrt(9) - 4
  alert(x)
</script>
```

Faça alguns testes de fórmulas e verifique o valor exibido pelo `alert()` para sanar possíveis dúvidas.

2.9 Exemplos de programas JavaScript integrados com HTML

Vamos criar alguns exemplos de programa para explorar o que foi abordado neste capítulo. Além da parte da integração com HTML, vamos utilizar as funções matemáticas e o operador módulo em diversos programas. Observe os exemplos de dados de entrada e saída de cada programa.

a) Elaborar um programa para um Cinema, que leia o título e a duração de um filme em minutos. Exiba o título do filme e converta a duração para horas e minutos, conforme destacado na Figura 2.6.

Crie um novo arquivo e salve-o com o nome ex2_2.html (arquivo tipo HTML). Lembre-se do atalho ! Tab para criar a estrutura do documento. As tags `<h1> ... </h1>` e `<p> ... </p>` também podem ser criadas a partir de atalhos que simplificam a digitação desses comandos. Para o `h1`, digite `h1` e pressione **Tab** (ou **Enter**). Todas as tags HTML com abertura e fechamento podem ser criadas a partir do uso desse padrão de atalho. As demais tags podem ser criadas a partir da digitação das letras iniciais e pela seleção do complemento correspondente, semelhante ao ilustrado na Figura 2.4. Para finalizar a seleção de um comando, pressione **Tab** (ou **Enter**).

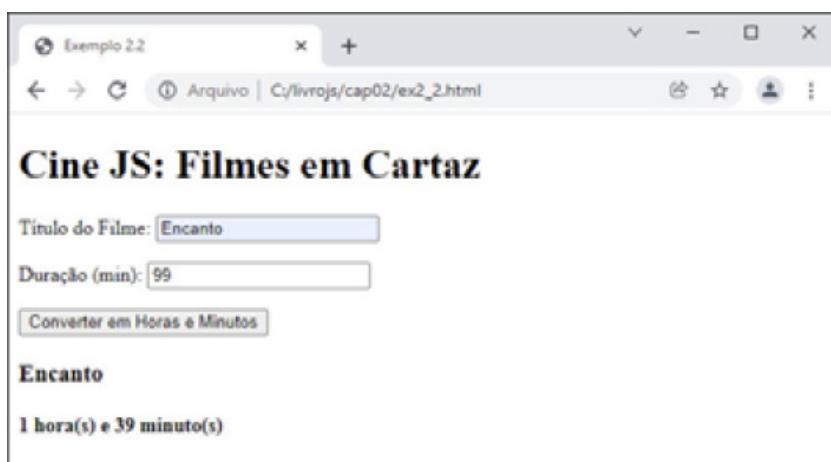


Figura 2.6 – Programa deve converter a duração do filme.

Exemplo 2.2 – Código HTML do programa Vídeo Locadora (ex2_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 2.1) -->
<h1>Cine JS: Filmes em Cartaz</h1>
<form>
  <p> Título do Filme:
```

```

<input type="text" id="inTitulo" required>
</p>
<p> Duração (min):
<input type="number" id="inDuracao" required>
</p>
<input type="submit" value="Converter em Horas e Minutos">
</form>
<h3></h3>
<h4></h4>
<script src="js/ex2_2.js"></script>
<!-- /body e /html -->

```

Os códigos HTML do cabeçalho e rodapé são semelhantes ao do Exemplo 2.1, exceto pelo título, que você pode modificar para Exemplo 2.2. Portanto, substitua as linhas <!-- --> (que servem para criar um comentário em HTML) pelo conteúdo em negrito do Exemplo 2.1. Observe que aos campos de formulário foi adicionado um parâmetro `required`, que, como o nome sugere, indica que esse campo é de preenchimento obrigatório.

Programa JavaScript que converte a duração de um filme (js/ex2_2.js)

```

// cria referência ao form e aos elementos h3 e h4 (resposta)
const frm = document.querySelector("form")
const resp1 = document.querySelector("h3")
const resp2 = document.querySelector("h4")

// cria um "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    const titulo = frm.inTitulo.value // obtém conteúdo dos campos
    const duracao = Number(frm.inDuracao.value)

    const horas = Math.floor(duracao / 60) // arredonda para baixo resultado
    const minutos = duracao % 60 // obtém o resto da divisão

    resp1.innerText = titulo // exibe as respostas
    resp2.innerText = `${horas} hora(s) e ${minutos} minuto(s)`

    e.preventDefault() // evita envio do form
})

```

Para converter a duração em horas e minutos, foi utilizada a função

`Math.floor()` e o operador módulo `%`. Como destacado neste capítulo, `Math.floor` arredonda um valor para baixo e `%` retorna o resto da divisão entre dois números. Eles são necessários nesse programa. Vamos usar os dados de entrada do exemplo: $108 / 60$ resulta em 1.8 ; `Math.floor(1.8)` retorna 1 , que é o número de horas do filme. Para obter os minutos, usamos $108 \% 60$, que resulta em 48 , que são os minutos restantes da duração. Insira dados de outros filmes e observe os valores de retorno.

b) Elaborar um programa para uma revenda de veículos. O programa deve ler modelo e preço do veículo. Apresentar como resposta o valor da entrada (50%) e o saldo em 12x. A Figura 2.7 ilustra uma execução desse programa.

Exemplo 2.3 – Código HTML do programa Revenda de Veículos (ex2_3.html)

```
<!-- doctype, html, head e body (conf. exemplo 2.1) -->
<h1>Revenda de Veículos JS</h1>
<form>
  <p> Veículo:
    <input type="text" id="inVeiculo" required>
  </p>
  <p> Preço R$:
    <input type="number" id="inPreco" required>
  </p>
  <input type="submit" value="Ver Promoção">
</form>
<h3 id="outResp1"></h3>
<h3 id="outResp2"></h3>
<h3 id="outResp3"></h3>
<script src="js/ex2_3.js"></script>
<!-- /body e /html -->
```

Figura 2.7 – Exemplo de dados do programa revenda de veículos.

No código HTML do Exemplo 2.3, acrescentamos três comandos `<h3>` para exemplificar a captura dos elementos de resposta a partir do seu id, como demonstra o programa JavaScript a seguir.

Código JavaScript do programa Revenda de Veículos (js/ex2_3.js)

```
// cria referência ao form e aos elementos de resposta (pelo seu id)
const frm = document.querySelector("form")
const resp1 = document.querySelector("#outResp1")
const resp2 = document.querySelector("#outResp2")
const resp3 = document.querySelector("#outResp3")

// cria um "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    const veiculo = frm.inVeiculo.value // obtém o conteúdo dos campos
    const preco = Number(frm.inPreco.value)

    const entrada = preco * 0.50 // calcula valor da entrada
    const parcela = (preco * 0.50) / 12 // ... e das parcelas

    resp1.innerText = `Promoção: ${veiculo}` // exibe as respostas
    resp2.innerText = `Entrada de R$: ${entrada.toFixed(2)}`
    resp3.innerText = `+12x de R$ ${parcela.toFixed(2)}`

    e.preventDefault() // evita envio do form
})
```

Os cálculos para obter o valor da entrada e das parcelas podem ser feitos de várias maneiras. Considerando que a entrada é 50%, poderíamos também dividir preço por 2. Para o cálculo das parcelas, também poderíamos utilizar a própria variável entrada e dividir por 12.

Observe que os programas seguem um padrão. Os exemplos HTML devem conter as tags básicas de estruturação da página, o formulário com os campos de entrada de dados e as linhas para a resposta. Os programas JavaScript iniciam pela captura dos elementos a serem manipulados pelo programa e pela adição do ouvinte de evento para o form. A seguir, ocorrem as etapas destacadas no capítulo anterior: entrada, processamento e saída, com a exibição da resposta ao usuário.

c) *Elaborar um programa para um restaurante que leia o preço por kg e o consumo (em gramas) de um cliente. Exiba o valor a ser pago, conforme ilustra a Figura 2.8.*

The screenshot shows a web browser window with the title 'Exemplo 2.4'. Inside the window, there is a form titled 'Restaurante JS'. The form contains two input fields: one for 'Buffet por Quilo R\$:' with the value '58.50' and another for 'Consumo do Cliente (gr):' with the value '500'. Below these fields is a button labeled 'Calcular Preço'. At the bottom of the form, the calculated result 'Valor a pagar RS: 29.25' is displayed.

Figura 2.8 – Programa Restaurante: dados de exemplo.

Exemplo 2.4 – Código HTML do programa Restaurante JS (ex2_4.html)

```
<!-- doctype, html, head e body (conf. exemplo 2.1) -->
<h1>Restaurante JS</h1>
<form>
  <p>Buffet por Quilo R$:
    <input type="number" id="inQuilo" min="0" step="0.01" required>
  </p>
```

```

<p>Consumo do Cliente (gr):
<input type="number" id="inConsumo" min="0" required>
</p>
<input type="submit" value="Calcular Preço">
</form>
<h3></h3>
<script src="js/ex2_4.js"></script>
<!-- /body e /html -->

```

No código HTML do Exemplo 2.4 foram adicionados os parâmetros `min` e `step` no campo `<input type="number">`. Eles servem para indicar o valor mínimo a ser aceito por esse campo e o valor de incremento ou decremento – que, como se refere a um valor monetário, pode variar em centavos.

Código JavaScript do programa Restaurante JS (js/ex2_4.js)

```

// cria referência ao form e ao elemento h3 (onde será exibida a resposta)
const frm = document.querySelector("form")
const resp = document.querySelector("h3")

// cria um "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    const quilo = Number(frm.inQuilo.value) // obtém conteúdo dos campos
    const consumo = Number(frm.inConsumo.value)

    const valor = (quilo / 1000) * consumo // calcula valor a ser pago
    resp.innerText = `Valor a pagar R$: ${valor.toFixed(2)}` // exibe resposta

    e.preventDefault() // evita envio do form
});

```

O programa JavaScript inicia pela referência aos elementos da página. Após, obtêm-se os valores informados nos campos `inQuilo` e `inConsumo`. Como os valores estão em grandezas diferentes (quilo e gramas), é necessário converter quilo para gramas (`quilo/1000`). Esse valor é então multiplicado pelo consumo do cliente. O próximo passo é apresentar o valor a ser pago pelo cliente. Observe a ocorrência das etapas de entrada (obtenção do preço do quilo e consumo do cliente), processamento (cálculo do valor) e saída (exibição do valor a pagar), discutidas no Capítulo 1. Elas também estarão presentes nos exercícios propostos a seguir.

2.10 Exercícios

Vamos praticar o processo de integração HTML x JavaScript! Observe os exemplos desenvolvidos e adapte para o que é solicitado nos exercícios. Um exemplo de correção para cada programa está disponível no site da editora.

- a) **Uma farmácia está com uma promoção** – Na compra de duas unidades de um mesmo medicamento, o cliente recebe como desconto os centavos do valor total. Elaborar um programa que leia descrição e preço de um medicamento. Informe o valor do produto na promoção. A Figura 2.9 apresenta a tela com um exemplo de dados de entrada e saída do programa.

The screenshot shows a web browser window with the title bar 'Exercicio 2.a'. The main content area is titled 'Farmácia JS'. It contains two input fields: 'Medicamento:' with the value 'Aspirina' and 'Preço R\$:' with the value '12.45'. Below these fields is a button labeled 'Mostrar Promoção'. Underneath the button, the text 'Promoção de Aspirina' is displayed in bold. At the bottom, the text 'Leve 2 por apenas R\$: 24.00' is shown in a dark font.

Figura 2.9 – Exemplo dos dados de entrada e saída do programa Farmácia.

- b) **Elaborar um programa para uma Ian house de um aeroporto** – O programa deve ler o valor de cada 15 minutos de uso de um computador e o tempo de uso por um cliente em minutos. Informe o valor a ser pago pelo cliente, sabendo que as frações extras de 15 minutos devem ser cobradas de forma integral. A Figura 2.10 exibe um exemplo com dados do programa.
- c) **Um supermercado está com uma promoção** – Para aumentar

suas vendas no setor de higiene, cada etiqueta de produto deve exibir uma mensagem anunciando 50% de desconto (para um item) na compra de três unidades do produto. Elaborar um programa que leia descrição e preço de um produto. Após, apresente as mensagens indicando a promoção – conforme o exemplo ilustrado na Figura 2.11.

The screenshot shows a web browser window titled "Exercício 2.b". The address bar indicates the file is located at "C:/livrojs/cap02/resp2_b.html". The main content area displays the title "Lan House JS". Below it, there are two input fields: "Valor por 15min de Uso R\$:" with the value "3.00" and "Tempo de Uso do Cliente:" with the value "25". A button labeled "Calcular Valor a Pagar" is present. The result is displayed below the button: "Valor a Pagar R\$: 6.00".

Figura 2.10 – Dados de exemplo do programa Lan House JS.

The screenshot shows a web browser window titled "Exercício 2.c". The address bar indicates the file is located at "C:/livrojs/cap02/resp2_c.html". The main content area displays the title "Supermercado JS". There are two input fields: "Produto:" with the value "Creme Dental" and "Preço R\$:" with the value "6.00". A button labeled "Ver Promoção" is present. Below the button, promotional text is displayed: "Creme Dental - Promoção: Leve 3 por R\$: 15.00" and "O 3º produto custa apenas R\$: 3.00".

Figura 2.11 – Promoção do Supermercado JS.

2.11 Considerações finais do capítulo

Neste capítulo, foram destacadas as técnicas para integrar os programas JavaScript com uma página web. Para rodar um programa na página, não é

necessário realizar qualquer processo adicional, pois o próprio navegador possui um interpretar para os códigos JavaScript. Ou seja, basta seguir as regras de sintaxe para a criação da página HTML e do arquivo JavaScript a fim de rodá-los diretamente no navegador. A entrada dos dados é agora realizada a partir de campos de formulário HTML, e a saída, pela exibição de dados em linhas do documento. Dessa forma, podemos interagir com os usuários de um site.

Para criar um documento HTML, é necessário inserir algumas tags básicas que definem a estrutura da página. Compete ao HTML determinar o conteúdo e a semântica (significado) dos elementos que compõem um site. O uso de um editor profissional de código facilita esse processo, pois ele contém atalhos que adicionam essas tags ao documento, além de nos auxiliar na digitação de comandos e formatação do arquivo.

A programação JavaScript em uma página web é geralmente acionada a partir da ocorrência de um evento, como carregar a página, alterar um campo de formulário, pressionar uma determinada tecla. Um evento comumente utilizado para executar um programa é o click sobre um botão exibido na página. Os programas, por sua vez, devem ser criados contendo uma função declarada pelo programador. Assim, quando o evento ocorre, a função contendo um conjunto de ações a serem executadas é acionada.

Para referenciar um elemento HTML da página, seja um campo de formulário ou um parágrafo de texto, é necessário identificá-lo no código HTML e, em seguida, utilizar os métodos `querySelector()` ou `getElementById()` no programa JavaScript. A identificação consiste em adicionar o atributo `id="nome"` à tag HTML do elemento. A partir da criação da referência ao elemento, é possível recuperar o conteúdo de um campo de formulário (propriedade `value`), bem como alterar um texto exibido em um parágrafo ou linha de cabeçalho do documento (propriedade `innerText`).

A integração do programa JavaScript com o documento HTML pode ocorrer a partir de três formas: a) inserir uma seção `<script>` no próprio documento HTML; b) criar um novo arquivo JavaScript e usar as rotinas de tratamento de eventos DOM; ou c) criar um novo arquivo JavaScript e

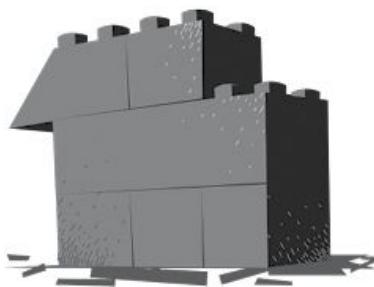
register um ouvinte de evento, também chamado modelo de eventos DOM nível 2. Esta última (c) é a forma atualmente recomendada e será utilizada nos exemplos do livro. Separar o código HTML do código JavaScript é considerado uma boa prática de programação, pois HTML e JavaScript têm papéis diferentes no processo de construção de um site. A ligação do HTML com o JavaScript, nesse modelo, ocorre a partir da inserção da tag `<script src="nomearq.js"></script>` no documento HTML.

Neste capítulo, também foram apresentadas as principais funções matemáticas disponíveis na classe `Math`. Elas nos auxiliam na realização de cálculos em que é necessário, por exemplo, arredondar um valor com decimais para cima ou para baixo. Ou seja, para obter o próximo valor inteiro de um número (para cima) ou para obter apenas a parte inteira do número (para baixo). Essas funções serão utilizadas em diversos exemplos do livro.

As técnicas trabalhadas neste capítulo nos permitem avançar de forma significativa no processo do aprendizado de lógica de programação com JavaScript. Nossos programas realizam as operações de entrada, processamento e saída discutidas no primeiro capítulo, agora interagindo com o usuário a partir de campos de formulário e com as informações de resposta sendo exibidas em parágrafos do documento HTML. Um avanço importante que pode ser mais bem compreendido com a realização dos exercícios propostos ao final do capítulo. Eles também serão o ponto de partida para o que será abordado na sequência do livro.

CAPÍTULO 3

Construção de algoritmos com Node.js



Nos exemplos construídos nos dois capítulos iniciais deste livro, foram elaborados Algoritmos em JavaScript que realizam a interação com o usuário de duas formas: a) a partir de caixas de diálogo exibidas no browser; b) a partir de páginas web, onde o usuário preenche alguns campos de um formulário e obtém uma resposta exibida na própria página. Para rodar esses programas, não é necessário instalar nenhum software adicional na sua máquina, apenas digitar os códigos em um editor de texto e carregar a página no browser. O browser já contém um interpretador para executar os programas JavaScript.

Existe também outro modo de construir algoritmos com JavaScript, que é utilizando o Node.js. Nesse modo, semelhante ao que acontece com linguagens de programação tradicionais como Python, Java e C#, é necessário instalar a linguagem na sua máquina e executar os programas a partir de um prompt de linha de comandos.

O Node.js é, conforme o que está documentado em sua página oficial (nodejs.dev), um ambiente de execução de programas JavaScript gratuito, de código aberto e multiplataforma que permite aos desenvolvedores escrever programas de linha de comando e scripts do lado do servidor fora de um

navegador. Os recursos e comandos utilizados para a criação de estruturas de condições, repetições e vetores, que serão apresentadas nos capítulos seguintes são os mesmos. O que muda é que você não estará rodando o seu programa a partir de um navegador web, e sim a partir de um prompt de comandos.

Este capítulo tem por objetivo apresentar o processo de construção de algoritmos utilizando o Node.js. Caso você não pretenda instalar o Node.js na sua máquina, pode pular para o capítulo seguinte e prosseguir seus estudos em JavaScript com a construção de Algoritmos que rodam no navegador web. Caso você goste desse formato de desenvolver seus estudos de técnicas de programação em JavaScript, utilizando o Node.js – que se assemelha ao formato tradicional de ensino de programação, siga os passos descritos nas próximas seções.

3.1 Instalação do Node.js

Para instalar o Node.js, vá até o site do ambiente nodejs.dev e selecione a opção **download**. Uma figura semelhante à 3.1 é apresentada. Escolha a opção **LTS Recomended for most users** e baixe o instalador de acordo com o sistema operacional de sua máquina.

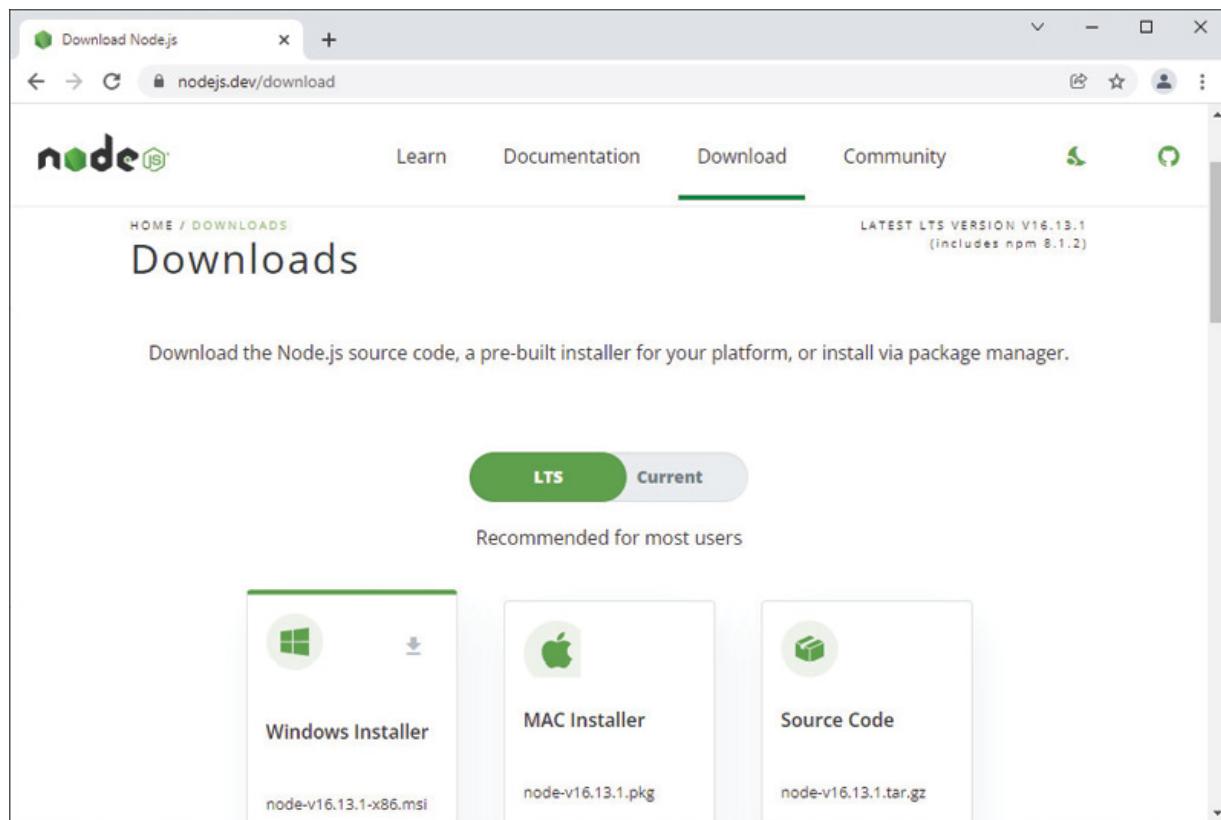


Figura 3.1 – Opções de download do Node.js.

Após baixar o arquivo de instalação, execute-o conforme o seu sistema operacional. No Windows, dê um duplo clique sobre o arquivo, leia e aceite os termos de licença e indique a pasta onde o software deve ser instalado, conforme ilustra a Figura 3.2.

Nas próximas telas, **Next > Next > Install > Finish**, concluída a instalação, acesse o prompt de comandos e execute o comando `node -v`. Ele serve para exibir a versão do Node.js instalada, além de indicar que o processo de instalação foi realizado com sucesso.

```
C:\Users\edecio>node -v
v16.13.1
```

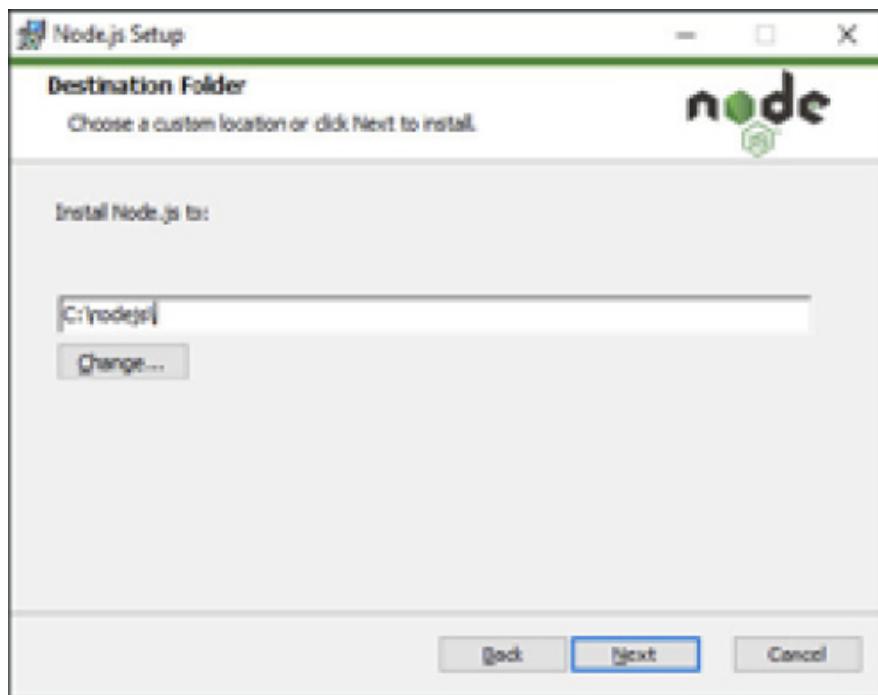


Figura 3.2 – Nas etapas de instalação, indique a pasta onde o Node.js deve ser instalado.

3.2 Adição de pacote para entrada de dados

O passo seguinte à instalação do Node.js é adicionar um pacote que permitirá que os nossos programas façam a entrada de dados via linha de comando. Vamos utilizar o `prompt-sync()`. Existem diversos outros, mas esse contém os recursos básicos necessários para o desenvolvimento de Algoritmos – que é a proposta deste livro. Vamos inicialmente criar a pasta `cap03`, em `livrojs` (como nos exemplos anteriores). Podemos fazer a criação da pasta via linha de comandos. Nas linhas de código a seguir, acessamos a pasta `livrojs`, instalamos o pacote `prompt-sync`, criamos a pasta `cap03` e, na sequência, entramos nesta pasta:

```
C:\Users\edecio>cd\livrojs  
C:\livrojs>npm i prompt-sync  
C:\livrojs>md cap03  
C:\livrojs>cd cap03
```

Após a execução desses comandos, foi adicionada ao diretório (pasta) `livrojs` a pasta `node-modules`. Ela contém os pacotes que foram adicionados ao

projeto. Além dela, também foram criados arquivos .json – tema abordado no Capítulo 12. É importante destacar que os programas a serem desenvolvidos com o Node.js (e que utilizam o prompt-sync) precisam estar em pastas que ficam dentro de livrojs, uma vez que o pacote prompt-sync foi instalado nela.

3.3 Criação e execução de programas com o Node.js

Vamos agora criar e executar o nosso primeiro programa em JavaScript a partir do Node.js. Abra o Visual Studio Code (você pode fazer isso digitando code . na linha de comando), crie um novo arquivo com o nome ex3_1.js. Vamos ajustar um programa de cada um dos capítulos anteriores para rodar neste modelo. Veja e insira o seguinte código no arquivo:

Código JavaScript do programa Soma 2 Números (ex3_1.js)

```
const prompt = require("prompt-sync")() // adiciona pacote para entrada de dados
const num1 = Number(prompt("1º Número: ")) // lê os números
const num2 = Number(prompt("2º Número: "))
const soma = num1 + num2 // calcula a soma
console.log(`Soma é: ${soma}`) // exibe o resultado
```

Compare com o Exemplo 1.6 e observe a semelhança! Neste modelo, não é necessário inserir os comandos entre as tags <script> e </script>. Contudo, precisamos acrescentar o comando const prompt = require("prompt-sync")(). O comando console.log() exibe a resposta do programa e também pode ser utilizado nos programas que rodam no browser, como indicado na Seção 1.5.

Após salvar o arquivo, volte ao prompt de comando e execute a instrução node ex3_1. Não é necessário informar a extensão do nome do arquivo (.js). A seguir um exemplo de execução desse programa.

```
C:\livrojs\cap03>node ex3_1
1º Número: 5
2º Número: 8
Soma é: 13
```

O exemplo do Capítulo 2 que vamos adaptar para rodar a partir do Node.js

é o programa ex2_3.js. Crie o arquivo ex3_2.js com o seguinte código:

Código JavaScript do programa Revenda (ex3_2.js)

```
const prompt = require("prompt-sync")() // adiciona o pacote ao programa
const veiculo = prompt("Veículo: ") // lê os dados de entrada
const preco = Number(prompt("Preço R$: "))
const entrada = preco * 0.50 // calcula valor da entrada
const parcela = (preco * 0.50) / 12 // ... e das parcelas
console.log(`Promoção: ${veiculo}`) // exibe as respostas
console.log(`Entrada de R$: ${entrada.toFixed(2)}`)
console.log(`+12x de R$ ${parcela.toFixed(2)}`)
```

Novamente, após salvar o arquivo, retorne ao prompt para rodar este programa:

```
C:\livrojs\cap03>node ex3_2
Veículo: Palio
Preço R$: 36000
Promoção: Palio
Entrada de R$: 18000.00
+12x de R$ 1500.00
```

3.4 Exemplos de algoritmos com Node.js

Vamos praticar o desenvolvimento de programas utilizando o Node.js? Leia o enunciado e tente construir o Algoritmo. Após, veja o exemplo de correção com os comentários. Lembre-se de que há várias formas de elaborar um algoritmo; portanto, se o seu programa funciona e gera as saídas de acordo com o enunciado (para valores válidos de entrada), não há necessidade de você alterar o seu código.

- Elaborar um programa para uma empresa que leia o salário e o tempo que um funcionário trabalha na empresa. Sabendo que a cada 4 anos (quadriênio) o funcionário recebe um acréscimo de 1% no salário, calcule e informe o número de quadriênios a que o funcionário tem direito e o salário final.

Leia com bastante atenção o enunciado do exercício e siga os passos dos exemplos anteriores: crie, edite e salve o arquivo. Para verificar se o seu programa está funcionando corretamente, volte ao prompt e rode o

comando node nomeprog. Caso ocorra algum erro, retorno ao editor, ajuste a linha indicada, salve o arquivo e rode-o novamente. A seguir, um exemplo de resposta para este exercício.

Código JavaScript do programa Quadriênios (ex3_3.js)

```
const prompt = require("prompt-sync")()      // adiciona o pacote ao programa
const salario = Number(prompt("Salário R$: ")) // lê os dados de entrada
const tempo = Number(prompt("Tempo (anos): "))
const quadrienios = Math.floor(tempo / 4)      // calcula quadriênios
const acrescimo = salario * quadrienios / 100 // ... e acréscimo
console.log(`Quadriênios: ${quadrienios}`)    // exibe as respostas
console.log(`Salário Final R$: ${(salario+acrescimo).toFixed(2)}`)
```

Observe que as etapas de entrada de dados, processamento e saída se mantêm como padrão. Um detalhe no exemplo de resposta do programa ex3_3.js é que o cálculo do salário final é realizado na saída dos dados `$(salario+acrescimo).toFixed(2)`, sendo mais um recurso que pode ser utilizado na construção de nossos algoritmos. Veja este programa em execução:

```
C:\livrojs\cap03>node ex3_3
Salário R$: 4500
Tempo (anos): 10
Quadriênios: 2
Salário Final R$: 4590.00
```

Caso tivéssemos cometido algum erro na escrita deste programa, ao executá-lo, o Node.js nos indicaria o tipo de erro e em qual linha ele ocorreu. Por exemplo, se na linha do primeiro `console.log()` tivéssemos indicado o nome da variável como quadrienio (no singular), o seguinte erro seria exibido:

```
C:\livrojs\cap03>node ex3_3
Salário R$: 4500
Tempo (anos): 10
C:\livrojs\cap03\ex3_3.js:6
  console.log(`Quadriênios: ${quadrienio}`) // exibe as respostas
                                         ^
ReferenceError: quadrienio is not defined
```

Ou seja, há um erro na linha 6, onde a variável quadrienio não está definida. No exercício a seguir, vou apresentar o enunciado do programa e um

exemplo de execução. Tente fazê-lo. Após, há um exemplo de resolução do algoritmo.

b) Elaborar um programa para uma veterinária, que leia o peso de uma ração em kg e o quanto um gato consome por dia da ração, em gramas. Informe quantos dias irá durar a ração e o quanto sobra da ração (em gramas).

Após finalizado, o programa deve rodar da seguinte forma:

```
C:\livrojs\cap03>node ex3_4
Peso da Ração (kg): 2
Consumo Diário (gr): 300
Duração: 6 dias
Sobra: 200gr
```

Código JavaScript do programa Veterinária (ex3_4.js)

```
const prompt = require("prompt-sync")() // adiciona o pacote ao programa
const pesoKg = Number(prompt("Peso da Ração (kg): ")) // lê dados de entrada
const consumo = Number(prompt("Consumo Diário (gr): "))
const pesoGr = pesoKg * 1000 // cria variável auxiliar pesoGr
const duracao = Math.floor(pesoGr / consumo) // cálculo das respostas
const sobra = pesoGr % consumo
console.log(`Duração: ${duracao} dias`) // dados de saída
console.log(`Sobra: ${sobra}gr`)
```

No exemplo de resolução do `ex3_4.js`, criamos uma variável auxiliar para calcular o peso em gramas (`pesoGr`). Como o cálculo do peso da ração em gramas é utilizado nas duas fórmulas, torna-se interessante atribuir esse cálculo a uma variável.

3.5 Considerações finais do capítulo

Desenvolver algoritmos utilizando a linguagem JavaScript pode se dar de inúmeras formas. Nestes três primeiros capítulos do livro são apresentados três modelos – sendo que os dois primeiros rodam diretamente no navegador web e não necessitam da instalação de nenhum software adicional. No modelo apresentado neste capítulo, os exemplos são construídos para rodar a partir do Node.js, um software que deve ser instalado na sua máquina e é responsável por rodar os programas JavaScript

em um servidor web. Vamos conversar mais sobre esse processo no Capítulo 12.

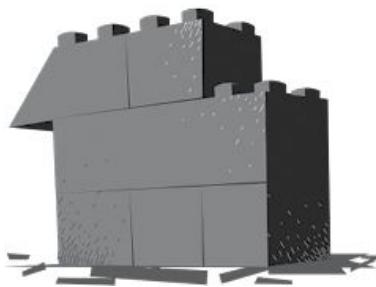
O Node.js permite que vários pacotes adicionais sejam acrescentados aos nossos programas, tornando-os aptos a realizar tarefas além das básicas disponíveis na linguagem, como pacotes que permitem configurar e enviar e-mails, se comunicar com bancos de dados, manipular imagens etc. Neste capítulo, adicionamos o pacote `prompt-sync()`, que permite a entrada de dados via prompt de comandos.

O interessante do Node.js é que a sintaxe das instruções JavaScript é a mesma da utilizada para rodar os scripts no browser. Então, essa é uma ótima forma de treinar a escrita de códigos JavaScript, principalmente para quem está dando os seus primeiros passos na área de programação de computadores.

Nos capítulos sobre condições, repetições, vetores, strings, datas e funções será acrescida uma seção com exercícios comentados de programas para serem executados com o Node.js.

CAPÍTULO 4

Condições



Diversas são as situações em um programa em que é necessário criar uma condição para indicar qual tarefa deve ser executada. Vamos retornar ao exemplo do caixa eletrônico, destacado no primeiro capítulo. O programa precisa verificar se a senha do cliente está ou não correta. Se estiver, um menu com opções é apresentado. Caso contrário, outras operações devem ocorrer, como ler novamente a senha e adicionar 1 ao número de erros da senha. Caso o cliente selecione a opção de saque, novamente vão surgir condições para serem verificadas. Há saldo na conta? O valor solicitado pode ser pago com as notas disponíveis no terminal?

Para definir uma condição em um programa, as linguagens de programação utilizam instruções próprias para essa finalidade. Os comandos utilizados para a leitura de dados (entrada), realização de operações (processamento) e apresentação de mensagens (saída) são os mesmos. O que vamos modificar agora é que alguns desses comandos serão inseridos no programa dentro de estruturas condicionais. Ou seja, a execução desses comandos vai depender da condição estabelecida e dos dados fornecidos pelo usuário. Por exemplo, se no terminal do caixa eletrônico houver apenas notas de 10 reais e o cliente solicitar um valor múltiplo de 10, o pagamento pode ser realizado. Caso contrário, uma mensagem de advertência deve ser emitida.

Conforme a situação, poderemos utilizar as estruturas condicionais nos

trechos de código que realizam a entrada, o processamento ou a saída de dados do programa. Em um programa que calcula a média das notas de um aluno e exibe se o aluno foi aprovado ou reprovado, as etapas de entrada de dados (leitura das notas) e processamento (cálculo da média) são as mesmas. A condição será utilizada apenas na saída, para mostrar a mensagem “Você foi Aprovado(a)” ou “Você foi Reprovado(a)”.

Já em um programa que calcula o peso ideal, no qual o cálculo do peso é diferente para homens e mulheres, as etapas correspondentes à entrada e saída de dados são as mesmas. Apenas o cálculo do peso (processamento) é que deve ficar dentro da estrutura condicional. Outras situações podem exigir que a condição seja inserida na entrada de dados do programa. Em uma rotina de cadastro, realizar a leitura do número do cartão caso o cliente possua um determinado convênio é um exemplo tradicional para a situação. Ler e compreender corretamente o objetivo do programa continua essencial, como discutido no Capítulo 1, para a correta construção de um algoritmo.

Portanto, será necessário agora adicionar às etapas de entrada, processamento e saída do programa estruturas que definem condições. Uma condição será definida com base no conteúdo de uma variável. Se a idade for maior que 18 anos, se o bairro do cliente for “Centro”, entre outros. Depois de definirmos condições simples, vamos avançar um pouco e utilizar os operadores lógicos, que permitem criar condições para analisar duas ou mais comparações em uma mesma instrução. E o raciocínio lógico continua essencial para a montagem dessas estruturas.

Os comandos `if...else` e `switch...case` são os responsáveis por criar condições em JavaScript (e na maioria das linguagens).

4.1 If... else

Uma das representações tradicionalmente utilizadas para o ensino de Algoritmos e Lógica de Programação são os fluxogramas. Eles são úteis para facilitar a compreensão do fluxo dos comandos em uma estrutura de controle. A Figura 4.1 exibe o fluxograma que representa uma estrutura condicional clássica. As setas no início e no final do fluxograma indicam

que há comandos antes e após a estrutura condicional. Ou seja, uma condição é parte de um programa. Ela pode ser criada para controlar apenas se um cálculo deve ser feito com uma fórmula ou outra, por exemplo, e a entrada e a saída de dados poderiam não pertencer a essa estrutura condicional.

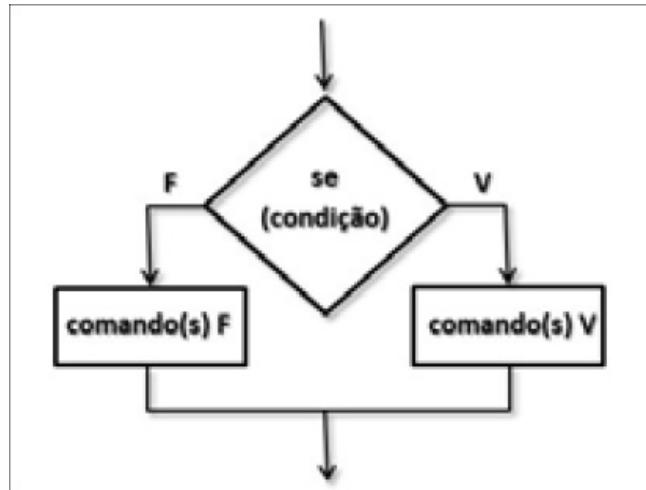


Figura 4.1 – Representação do fluxo dos comandos em uma estrutura condicional.

Para criar essa estrutura clássica, utilizamos os comandos `if... else` (`se... senão`). Eles possuem algumas variações. É possível utilizar apenas o `if` (para apresentar uma mensagem caso o cliente seja menor de idade, por exemplo). E também criar vários comandos `else` (para verificar a classificação etária de um aluno de natação, que poderia ser infantil, juvenil ou adulto).

Exemplos da sintaxe do comando `if`.

```
// define uma condição simples
if (condição) {
    comandos
}
```

```
// define uma condição de if... else
if (condição) {
    comandos V
} else {
    comandos F
}
```

```
// define múltiplas condições
if (condição 1) {
    comandos 1
} else if (condição 2) {
    comandos 2
} else {
    comandos 3
}
```

Quando houver apenas um comando que pertence à condição, o uso das chaves não é obrigatório. Contudo, para facilitar a compreensão, recomenda-se utilizar as {} em todas as ocorrências das estruturas condicionais de um programa.

4.2 Operadores relacionais

Para definir as condições utilizadas nas estruturas condicionais, deve-se fazer uso dos operadores relacionais. A Tabela 4.1 apresenta os operadores relacionais utilizados em JavaScript. Quando inseridas em um programa, cada comparação deve retornar true (verdadeiro) ou false (falso).

Tabela 4.1 – Operadores relacionais

Símbolo	Significado
==	Igual. Retorna verdadeiro caso os dados contenham o mesmo conteúdo.
!=	Diferente. Retorna verdadeiro caso os dados contenham conteúdos diferentes.
>	Maior. Pode ser utilizado para comparar números ou palavras. Na comparação de palavras, a classificação alfabética é avaliada.
<	Menor. Também podem ser realizadas comparações de números ou palavras.
>=	Maior ou igual. Os símbolos devem estar nesta ordem (>=)
<=	Menor ou igual. Tenha cuidado com a ordem dos símbolos (<=)

Existem ainda os símbolos de === (estritamente igual) e !== (estritamente diferente). Eles compararam também o tipo do dado em análise. Assim, '5' === 5 retorna falso; e '5' !== 5 retorna verdadeiro.

Vamos construir um exemplo de uso das condições e operadores relacionais. A página exibida na Figura 4.2 faz a leitura do nome e das

notas de um aluno, apresenta a média e uma mensagem para o aluno: “Parabéns ... Você foi aprovado(a)!” ou, então, “Ops... Você foi reprovado(a).”. A situação de aprovado ou reprovado é definida pela média das notas, que deve ser 7.0 ou superior para aprovação. Caso a nota seja inferior a 7.0, a mensagem indicando a reprovação deve ser exibida. Também faremos nesse exemplo a aplicação de um estilo na mensagem que indica a situação do aluno. Esse recurso refere-se à camada CSS (Cascading Style Sheets) da construção de páginas web e também pode ser manipulado por programas JavaScript. A mensagem de aprovação é exibida em azul e de reprovação, em vermelho.

Exemplo 4.1

Arquivo | C:/livrojs/cap04/ex4_1.html

Programa Situação do Aluno

Nome do Aluno:

1ª Nota:

2ª Nota:

Média das Notas 7.50

Parabéns Silvana! Você foi aprovado(a)

Figura 4.2 – Uso das condições para exibir a situação do aluno.

Como nos capítulos anteriores, vamos agora criar a pasta `cap04` dentro de `livrojs`. Crie também a pasta `js` que vai conter os arquivos dos programas JavaScript do capítulo. O código descrito a seguir deve ser salvo dentro da pasta `cap04` com o nome `ex4_1.html`. Lembre-se de utilizar o atalho **Alt+Shift+F** para indentar as linhas do programa a fim de facilitar a leitura e a compreensão do código.

Exemplo 4.1 – Código HTML do programa Situação do Aluno (ex4_1.html)

```
<!-- doctype, html, head padrão e body (conf. exemplo 2.1) -->
```

```

<h1>Programa Situação do Aluno</h1>
<form>
  <p>Nome do Aluno:
    <input type="text" id="inNome" required>
  </p>
  <p>1a Nota:
    <input type="number" step="0.1" min="0" max="10" id="inNota1" required>
  </p>
  <p>2a Nota:
    <input type="number" step="0.1" min="0" max="10" id="inNota2" required>
  </p>
  <input type="submit" value="Exibir Média e Situação">
</form>
<h3></h3>
<h4></h4>
<script src="js/ex4_1.js"></script>
<!-- /body e /html -->

```

O código HTML contém as tags básicas de estruturação da página, do form com os campos de entrada e das linhas h3 e h4 que são utilizadas para exibição das respostas do programa. O programa que vai manipular esses dados é descrito a seguir e deve ser salvo dentro da pasta js (livrojs/cap04/js/ex4_1.js).

Código JavaScript do programa Situação do Aluno (js/ex4_1.js)

```

// cria referência ao form e elementos de resposta do programa
const frm = document.querySelector("form")
const resp1 = document.querySelector("h3")
const resp2 = document.querySelector("h4")

// cria um "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
  e.preventDefault()          // evita envio do form
  const nome = frm.inNome.value // obtém valores do form
  const nota1 = Number(frm.inNota1.value)
  const nota2 = Number(frm.inNota2.value)
  const media = (nota1 + nota2) / 2 // calcula a média das notas
  resp1.innerText = `Média das Notas ${media.toFixed(2)}`
  // cria as condições
  if (media >= 7) {
    // altera o texto e estilo da cor do elemento resp2
  }
})

```

```

resp2.innerHTML = `Parabéns ${nome}! Você foi aprovado(a)`
resp2.style.color = "blue"
} else {
  resp2.innerHTML = `Ops ${nome}... Você foi reprovado(a)`
  resp2.style.color = "red"
}
})

```

Nesse programa, a condição foi definida na saída de dados, visto que a mensagem a ser exibida pelo programa é diferente para cada situação. Observe que o código inicia com a criação de referências aos elementos manipulados pelo programa. Na sequência, cria-se um listener (ouvinte) para o evento submit do form. Vamos manter esse padrão em todos os programas do livro para facilitar a compreensão.

Poderíamos, ainda, ter uma terceira situação para o aluno, “Em Exame”, por exemplo. Nesse caso, faríamos uso do else if. Observe a estrutura condicional para acrescentar essa situação. O aluno está em exame se possuir uma média maior ou igual a 4 e menor que 7.

```

if (media >= 7) {
  resp2.innerHTML = `Parabéns ${nome}! Você foi aprovado(a)`
  resp2.style.color = "blue"
} else if (media >= 4) {
  resp2.innerHTML = `Atenção ${nome}. Você está em exame`
  resp2.style.color = "green"
} else {
  resp2.innerHTML = `Ops ${nome}... Você foi reprovado(a)`
  resp2.style.color = "red"
}

```

O uso dos comandos else é recomendado, pois ele simplifica as condições. No teste para verificar se o aluno está em exame, apenas a condição (media >= 4) é necessária, pois foi utilizado um else if. Ele significa “senão se”, ou seja, se a média não é maior ou igual a 7 (que é verificado no if inicial) e é maior ou igual a 4.

Em alguns casos, é necessário criar uma condição dentro de outra. Por exemplo, caso o aluno tenha ficado em exame, pode-se realizar a leitura da nota da prova do exame e então definir uma nova condição para verificar o resultado dessa prova. É comum termos estruturas de condição ou de

repetição (a serem discutidas no próximo capítulo) dentro de outras estruturas em um mesmo programa.

4.3 Operadores lógicos

No exercício anterior, foi criada uma condição com base em uma comparação ($media \geq 7$). Há situações em que mais do que uma condição deve ser analisada. Alguns exemplos: a) um cliente quer um carro da cor azul ou cinza; b) o preço do carro deve ser inferior a R\$ 20.000,00 e o ano maior ou igual a 2010; c) o modelo do carro deve ser “fusca” e o preço menor que R\$ 8.000,00.

Para definir mais de uma condição em um programa, devemos utilizar os operadores lógicos. A Tabela 4.2 apresenta os principais operadores lógicos disponíveis em JavaScript.

Tabela 4.2 – Operadores lógicos

Símbolo	Significado
!	Not. Indica negação. Inverte o resultado de uma comparação.
&&	And. Indica conjunção. Retorna verdadeiro quando todas as comparações forem verdadeiras.
	Or. Indica disjunção. Retorna verdadeiro se, no mínimo, uma das condições definidas for verdadeira.

Vamos comparar os operadores lógicos utilizando tabelas que indicam os valores que cada comparação pode assumir, a fim de relacionar esses operadores. Para isso, faremos uso de uma tabela verdade – que é uma tabela com todas as possíveis combinações dos valores lógicos (verdadeiro ou falso) das proposições e dos conectivos (operadores lógicos) utilizados. A seguir, temos a representação de duas variáveis, cor e ano, e um valor que ela pode assumir na execução do programa. Nas tabelas verdade, utilizamos uma letra, geralmente p e q, para representar uma proposição (“o carro é da cor azul”, “o ano do carro é 2017”).

```
const cor = "Azul" // (p)
const ano = 2017 // (q)
```

A Tabela 4.3 apresenta a tabela verdade da negação, representada pelo

símbolo (!). Ela pode ser aplicada a apenas uma proposição.

Tabela 4.3 – Negação

p	!p
V	F
F	V

A negação é o mais simples dos operadores relacionais. Ela inverte o resultado (verdadeiro ou falso) de uma condição. Equivale ao sinal de diferente (!=) quando puder ser aplicada. Por exemplo, os testes a seguir realizam a mesma verificação.

```
if (!cor == "Azul") { ... }
if (cor != "Azul") { ... }
```

A conjunção, representada pelos símbolos `&&`, reflete a ideia da simultaneidade. A Tabela 4.4 expressa os valores resultantes para as proposições p e q, usando a conjunção.

Tabela 4.4 – Conjunção (&&)

p	q	p && q
V	V	V
V	F	F
F	V	F
F	F	F

Na conjunção, como observado na Tabela 4.4, a expressão só retorna verdadeiro se todas as comparações forem verdadeiras. Se um cliente quer um carro azul e de 2017, ele só será atendido se as duas condições forem satisfeitas. São exemplos de condições utilizando `&&`.

```
if (cor == "Azul" && ano == 2017) { ... }
if (cor == "Cinza" && ano < 2017) { ... }
if (ano >= 2012 && ano <= 2017) { ... }
if (cor != "Azul" && cor != "Vermelho") { ... }
```

Observe a sintaxe dos dois últimos exemplos. Quando uma mesma variável é utilizada na condição, ela deve ser repetida em cada comparação.

Já a disjunção (`||`) reflete uma noção de que pelo menos uma das condições

deve ser verdadeira, para que o resultado seja verdadeiro. A Tabela 4.5 representa a disjunção.

Tabela 4.5 – Disjunção (\parallel)

p	q	$p \parallel q$
V	V	V
V	F	V
F	V	V
F	F	F

Na disjunção, no mínimo uma das condições deve ser verdadeira. Agora, nosso cliente do exemplo quer um carro de cor azul ou de 2017. Qualquer carro em que uma dessas condições for verdadeira serve para esse cliente. Para criar a disjunção, utilize dois símbolos \parallel .

São exemplos de condições que utilizam o operador \parallel .

```
if (cor == "Azul" || ano == 2017) { ... }  
if (cor == "Azul" || cor == "Branco") { ... }  
if ((cor == "Azul" || cor == "Branco") && ano == 2017) { ... }  
if (cor == "Azul" && (ano == 2016 || ano == 2017)) { ... }
```

Nos dois últimos exemplos, foram utilizados juntos os operadores And ($&&$) e Or (\parallel). É fundamental o uso dos parênteses para indicar a ordem de precedência das comparações e evitar problemas de interpretação. Observe a última linha dos exemplos: se os parênteses não fossem utilizados, qualquer veículo de 2017 seria selecionado (pois a interpretação iria ocorrer da esquerda para a direita: veículos da cor “azul” e de 2016, ou de 2017).

Vamos construir um novo exemplo para explorar o uso dos operadores lógicos na construção do script. O nosso programa vai calcular o peso ideal de uma pessoa. Para isso, foram pesquisados alguns sites sobre o assunto. Em um deles, há a indicação de que o peso ideal de um adulto pode ser calculado a partir das fórmulas: $22 * \text{altura}^2$ (para homens); e $21 * \text{altura}^2$ (para mulheres).

A página deve ficar semelhante à ilustrada na Figura 4.3. Para criar o campo de formulário sexo, foram utilizadas as tags `<input type="radio" ...>`. A propriedade `name` serve para indicar que os campos de seleção pertencem a

um mesmo grupo e, dessa forma, marcar um item implica desmarcar automaticamente o outro. A página também apresenta uma imagem a fim de embelezá-la de uma forma simples, já que o foco do livro não é construir sofisticados layouts. Todas as figuras inseridas nos programas do livro foram criadas por Carolina Kuwabata, da Editora Novatec, a quem registro meu especial agradecimento.



Figura 4.3 – Página para exemplificar o uso dos operadores lógicos.

Para alinhar a figura à esquerda da página e definir alguns outros estilos a serem aplicados aos elementos da página, vamos criar um arquivo chamado estilos.css, dentro de uma nova pasta de nome css. Como são poucos os estilos e seguem um mesmo modelo em todos os exemplos, vamos criar um novo arquivo de estilos para cada capítulo. Assim, mantemos organizados os códigos de nossos programas. Inclua as seguintes linhas no arquivo estilos.css a ser aplicado aos exemplos deste capítulo.

```
img { float: left; height: 300px; width: 300px; }  
h1 { border-bottom-style: inset; }
```

Essas regras CSS definem que as tags img da página devem ficar alinhadas à esquerda e possuir o tamanho 300 x 300px. As tags h1, por sua vez, são apresentadas com uma borda inferior do estilo inset. Isso cria uma linha horizontal que separa o cabeçalho do restante da página, como pode ser observado na Figura 4.3. A tag <link rel="stylesheet" href="css/estilos.css"> inserida na seção head do código HTML, conforme ex4_2.html, define a ligação da página web com um arquivo de estilos CSS.

Para inserir uma imagem em uma página HTML, é necessário primeiro obter a imagem. Elas não são incorporadas ao documento, como ocorre com um documento de texto. Na construção de páginas web, as imagens são apenas referenciadas no código. Elas devem continuar existindo no local indicado para serem exibidas no momento em que a página é renderizada pelos navegadores. Vamos salvar as imagens de nossos exemplos e exercícios em uma nova pasta `img`, dentro da pasta de cada capítulo.

A partir desse exemplo, vamos acrescentar um favorite icon (favicon) para cada página do livro. É uma pequena imagem, exibida na aba do navegador, e serve como um logotipo para representar um site. A tag HTML para exibir um favicon é a `<link rel="icon"...>` inserida na seção `head` do `ex4_2.html`.

Você pode utilizar as imagens exibidas no livro, disponíveis no site da editora, ou realizar uma pesquisa no Google sobre um tema relacionado. Salve a imagem com o nome do arquivo a ser referenciado nas tags `` ou `<link rel="icon"...>`. O código HTML da página exibida na Figura 4.3 pode ser conferido no Exemplo 4.2. Lembre-se de utilizar os atalhos do editor (! Tab, img Tab, input:r Tab, input:b Tab, script:src Tab...) para facilitar a digitação dos comandos.

Exemplo 4.2 – Código HTML (ex4_2.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" type="image/png" href="img/livrojs.png">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 4.2</title>
</head>
<body>
  
  <h1>Programa Cálculo do Peso Ideal</h1>
  <form>
    <p>Nome:<br/>
      <input type="text" id="inNome" required>
    </p>
    <p>Sexo:<br/>
      <input type="radio" id="inSexoM" value="Masculino" checked="">
      <label for="inSexoM">Masculino</label>
      <input type="radio" id="inSexoF" value="Feminino" checked="">
      <label for="inSexoF">Feminino</label>
    </p>
    <table border="1">
      <thead>
        <tr>
          <th>Peso (kg)</th>
          <th>Altura (m)</th>
          <th>Cálculo (kg)</th>
        </tr>
      <tbody>
        <tr>
          <td>60</td>
          <td>1.75</td>
          <td>65.5</td>
        </tr>
        <tr>
          <td>70</td>
          <td>1.75</td>
          <td>75.5</td>
        </tr>
        <tr>
          <td>80</td>
          <td>1.75</td>
          <td>85.5</td>
        </tr>
        <tr>
          <td>90</td>
          <td>1.75</td>
          <td>95.5</td>
        </tr>
      </tbody>
    </table>
  </form>
</body>
```

```

<input type="radio" name="sexo" id="inMasculino" required> Masculino
<input type="radio" name="sexo" id="inFeminino" required> Feminino
</p>
<p>Altura:<br/>
<input type="number" min="0" step="0.01" id="inAltura" required>
</p>
<input type="submit" value="Calcular Peso Ideal">
<input type="reset" value="Limpar Campos">
</form>
<h3></h3>
<script src="js/ex4_2.js"></script>
</body>
</html>

```

Para interagir com o usuário a partir do clique nos botões disponibilizados na página, crie um novo arquivo do tipo JavaScript na pasta js e digite os comandos a seguir.

Código JavaScript do programa Cálculo do Peso Ideal (js/ex4_2.js)

```

// cria referência ao form e elemento onde será exibida a resposta
const frm = document.querySelector("form")
const resp = document.querySelector("h3")

// "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    e.preventDefault()          // evita envio do form

    const nome = frm.inNome.value // obtém valores do form
    const masculino = frm.inMasculino.checked
    const altura = Number(frm.inAltura.value)

    let peso      // declara a variável peso
    if (masculino) { // se masculino (ou, if masculino == true)
        peso = 22 * Math.pow(altura, 2) // Math.pow eleva ao quadrado
    } else {
        peso = 21 * Math.pow(altura, 2)
    }

    // apresenta a resposta (altera o conteúdo do elemento h3 da página)
    resp.innerText = `${nome}: Seu peso ideal é ${peso.toFixed(3)} kg`
})

```

Vamos analisar o código do Programa ex4_2.js. Observe que no if, para verificar o sexo, é utilizado o valor booleano (true ou false) da propriedade checked do campo RadioButton. Essa propriedade verifica se o campo está ou não selecionado. Nesse if há uma particularidade. Quando uma variável contém o valor true ou false, não é necessário realizar uma comparação, pois a própria variável já contém um valor que é true ou false.

É possível perceber também que apenas um RadioButton foi analisado. Como indicamos que esse campo contém o atributo required no código HTML, um dos dois itens será selecionado pelo usuário. Logo, se o RadioButton masculino não estiver selecionado, é certo que o feminino estará.

Outro detalhe: nesse programa apenas o cálculo da fórmula está dentro da condição. Ela é que é diferente, conforme o sexo do usuário. A entrada e saída de dados não devem estar dentro de condições.

Nesse programa utilizamos pela primeira vez a declaração de uma variável com let no lugar de const. Como a variável peso está dentro do if, ela só existe dentro desse bloco. E como precisamos dela fora do if, é necessário primeiro declarar a variável, que após a declaração terá o seu conteúdo alterado dentro do if e utilizado no contexto do programa principal. O seguinte trecho de código também poderia ser utilizado neste programa (sem declarar let peso):

```
if (masculino) {  
    const peso = 22 * Math.pow(altura, 2)  
    resp.innerHTML = `${nome}: Seu peso ideal é ${peso.toFixed(3)} kg`  
} else {  
    const peso = 21 * Math.pow(altura, 2)  
    resp.innerHTML = `${nome}: Seu peso ideal é ${peso.toFixed(3)} kg`  
}
```

Como o programa executa apenas os comandos que estão dentro do if ou do else, poderíamos utilizar const. Contudo, observe que a saída que apresenta a variável peso precisaria estar repetida dentro de cada condição (o que não é muito adequado). Fora do contexto do bloco, essa variável não existe. Nosso formulário HTML contém ainda uma tag <input type="reset">. O clique nesse botão limpa o conteúdo dos campos do formulário, mas não limpa o espaço onde é exibida a resposta do programa. Para fazer isso, acrescente

um ouvinte para o evento `reset` do form e atribua um “” (vazio) para a propriedade `innerText` do elemento `h3`, como demonstrado a seguir:

```
frm.addEventListener("reset", () => {
  resp.innerText = "" // limpa o conteúdo do elemento h3 que exibe a resposta
})
```

4.4 Operador ternário

Existe ainda uma forma abreviada para criar as instruções `if.. else` conhecida como operador ternário (três operandos) ou operador condicional. Consiste em realizar uma atribuição para uma variável com base na análise de uma condição. Observe o exemplo a seguir:

```
const categoria = idade >= 18 ? "Adulto" : "Juvenil"
```

A condição deve ser inserida após o sinal de atribuição (`=`). O primeiro valor após a interrogação (`?`) é atribuído à variável caso a condição seja verdadeira. E o segundo, após os `:`, caso a condição seja falsa. A instrução anterior equivale à seguinte estrutura `if... else`.

```
let categoria
if (idade >= 18) {
  categoria = "Adulto"
} else {
  categoria = "Juvenil"
}
```

Em razão da sua simplicidade, o operador ternário é bastante utilizado e está disponível na maioria das linguagens de programação da atualidade. Podemos ajustar o nosso programa `ex4_2.js`, com a substituição do `if` pelo operador ternário, a partir do seguinte comando:

```
const peso = masculino ? 22 * Math.pow(altura, 2) : 21 * Math.pow(altura, 2)
```

4.5 Switch... Case

As linguagens de programação dispõem de outra estrutura que permite criar condições. Trata-se do comando `switch... case`. Ele é útil quando tivermos várias alternativas definidas a partir do conteúdo de uma variável. Vamos recorrer aos métodos `prompt()` e `alert()` utilizados no Capítulo 1, para construir

um exemplo. O script vai informar o valor da taxa de entrega de um medicamento em uma farmácia, conforme o bairro do cliente. A partir dele, fica melhor de destacar os detalhes e as regras de sintaxe dessa estrutura condicional.

Exemplo 4.3 – Condições com switch... case (ex4_3.html)

```
<script>
const bairro = prompt("Bairro de Entrega: ")
let taxaEntrega
switch (bairro) {
  case "Centro":
    taxaEntrega = 5.00
    break
  case "Fragata":
  case "Três Vendas":
    taxaEntrega = 7.00
    break
  case "Laranjal":
    taxaEntrega = 10.00
    break
  default:
    taxaEntrega = 8.00
}
alert(`Taxa R$: ${taxaEntrega.toFixed(2)})`)</script>
```

O comando `switch` inicia pela definição da variável que escolhe a condição a ser executada. Cada instrução `case` deve conter um valor de comparação (seguida pelos “:”). Os comandos devem ser finalizados por `break`. Nesse exemplo, o valor da taxa de entrega para os bairros “Fragata” e “Três Vendas” é o mesmo, então eles devem ser colocados um abaixo do outro. Caso o bairro seja diferente dos conteúdos indicados nas instruções `case`, o fluxo do programa direciona para a execução dos comandos inseridos na instrução `default` (que significa: por falta ou ausência).

4.6 Exemplos com HTML e JavaScript

Vamos construir novos exemplos para discutir sobre a aplicação das

condições em um programa? Começamos com um exemplo simples, que utiliza apenas o if (sem o else). Após, exemplos com if... else e também else if.

a) Sabendo que o fuso horário da França em relação ao Brasil é de + 5 horas (no horário de verão na França), elaborar um programa que leia a hora no Brasil e informe a hora na França. A Figura 4.4 ilustra a tela com dados de entrada e saída do programa.



Figura 4.4 – Exemplo de dados de entrada e saída do programa fuso horário.

Exemplo 4.4 – Código HTML do programa Fuso Horário (ex4_4.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa Fuso Horário</h1>
<form>
  <p>Hora no Brasil (h.m.):
    <input type="number" id="inHoraBrasil" min="0" max="23.59" step="0.01" required>
  </p>
  <input type="submit" value="Exibir Hora na França">
</form>
<h3></h3>
<script src="js/ex4_4.js"></script>
<!-- /body e /html -->
```

Observe que na tag img há o atributo alt. Ele está relacionado com questões de acessibilidade e deve ser utilizado para exibir um texto alternativo em

navegadores não gráficos e em leitores de página para pessoas portadoras de necessidades especiais. Crie, na sequência, o programa JavaScript que vai calcular o horário na França, conforme o enunciado do exemplo.

Programa JavaScript do exemplo 4.4 (js/ex4_4.js)

```
// cria referência ao form e elemento onde será exibida a resposta
const frm = document.querySelector("form")
const resp = document.querySelector("h3")

// "ouvinte" de evento, acionado quando o botão submit for clicado
frm.addEventListener("submit", (e) => {
    e.preventDefault()          // evita envio do form
    // obtém e converte o conteúdo do campo inHoraBrasil
    const horaBrasil = Number(frm.inHoraBrasil.value)
    let horaFranca = horaBrasil + 5 // calcula o horário na França
    if (horaFranca > 24) {        // se passar das 24 horas na França
        horaFranca = horaFranca - 24 // ... subtrai 24
    }
    // exibe a resposta (altera o conteúdo do elemento h3 da página)
    resp.innerText = `Hora na França ${horaFranca.toFixed(2)}`
})
```

O programa contém os passos e comandos já destacados nos exemplos anteriores. Para calcular o horário na França, adicionamos 5 ao horário do Brasil. Esse cálculo fica correto para a maioria dos horários. Contudo, caso no Brasil sejam 20 horas, por exemplo, o programa não pode informar que na França são 25 horas. Então, caso o horário da França ultrapasse as 24 horas, significa que já é o próximo dia e deve-se subtrair 24 desse valor obtido. Como essa variável pode ter o seu conteúdo alterado, deve-se declará-la com let.

b) Elaborar um programa que leia um número e calcule sua raiz quadrada. Caso a raiz seja exata (quadrados perfeitos), informá-la, caso contrário, informe: ‘Não há raiz exata para ..’. A Figura 4.5 ilustra uma execução desse programa.



Figura 4.5 – Exemplo de execução do programa raiz quadrada.

Exemplo 4.5 – Programa Raiz Quadrada – código HTML (ex4_5.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa Raiz Quadrada</h1>
<form>
<p>Número:
    <input type="number" id="inNumero" required>
</p>
    <input type="submit" value="Exibir Raiz Quadrada">
</form>
<h3></h3>
<script src="js/ex4_5.js"></script>
<!-- /body e /html -->
```

Programa Raiz Quadrada – código JavaScript (js/ex4_5.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => {
    e.preventDefault() // evita envio do form
    const numero = Number(frm.inNumero.value) // obtém número digitado no form
    const raiz = Math.sqrt(numero) // calcula raiz quadrada do número
    if (Number.isInteger(raiz)) { // se valor da raiz for um número inteiro
        resp.innerText = `Raiz: ${raiz}` // ...mostra a raiz
    } else { // senão,
        resp.innerText = `Não há raiz exata para ${numero}` // ...mostra mensagem
```

```
    }  
})
```

O programa `ex4_5.js` contém as instruções padrões já discutidas nos exemplos anteriores. Em seguida, realiza-se o cálculo da raiz quadrada a partir da função `Math.sqrt()`. Agora há duas possibilidades. O cálculo vai retornar um número inteiro ou com decimais. Por exemplo, a raiz de 16 é 4, já a raiz de 15 é 3.872983...

Como indicado na Seção 1.9, o método `Number.isInteger()` verifica se um número é inteiro. Então, podemos utilizá-lo na condição para apresentar a resposta solicitada. Há outras formas para solucionar esse exercício. Por exemplo, o teste condicional do `if()` poderia ser substituído pelo comando exibido a seguir:

```
if (raiz % 1 == 0) { ... }
```

Pois, seguindo no exemplo da raiz de 16, $4 \% 1$ retorna 0. Já a raiz de 15, que é $3.872983... \% 1$, retorna $0.872983...$. A regra vale para os demais números que podem ser informados pelo usuário.

c) *Em um determinado momento do dia, apenas notas de 10, 50 e 100 estão disponíveis em um terminal de caixa eletrônico. Elaborar um programa que leia um valor de saque de um cliente, verifique sua validade (ou seja, se pode ser pago com as notas disponíveis) e informe o número mínimo de notas de 100, 50 e 10 necessárias para pagar esse saque. A Figura 4.6 exemplifica uma execução do programa.*



Figura 4.6 – Programa caixa eletrônico: exemplo de saque.

Exemplo 4.6 – Programa Caixa Eletrônico – código HTML (ex4_6.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa Caixa Eletrônico</h1>
<form>
  <p>Valor do Saque R$:
    <input type="number" id="inSaque" min="10" required>
  </p>
  <input type="submit" value="Exibir Notas para Saque">
</form>
<h3 id="outResp1"></h3>
<h3 id="outResp2"></h3>
<h3 id="outResp3"></h3>
<script src="js/ex4_6.js"></script>
<!-- /body e /html -->
```

Observe que, no código HTML do Exemplo 4.6, foram inseridas três linhas h3 para a exibição das respostas. Ou seja, o número de notas de 100, 50 e 10 reais será exibido nesses locais da página. Crie agora o arquivo ex4_6.js, na pasta js, para validar e apresentar o número de notas necessárias para pagar o valor solicitado.

Programa Caixa Eletrônico – código JavaScript (js/ex4_6.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp1 = document.querySelector("#outResp1")
const resp2 = document.querySelector("#outResp2")
const resp3 = document.querySelector("#outResp3")

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form
  const saque = Number(frm.inSaque.value) // obtém valor do saque
  if (saque % 10 != 0) { // se saque não é múltiplo de 10
    alert("Valor inválido para notas disponíveis (R$ 10, 50, 100)")
    frm.inSaque.focus()
    return
  }
  const notasCem = Math.floor(saque / 100) // calcula notas de 100
```

```

let resto = saque % 100           // quanto sobra para pagar
const notasCinquenta = Math.floor(resto / 50) // calcula notas de 50
resto = resto % 50               // quanto ainda sobra
const notasDez = Math.floor(resto / 10) // calcula notas de 10
if (notasCem > 0) {              // exibe as notas se houver
    resp1.innerText = `Notas de R$ 100: ${notasCem}`
}
if (notasCinquenta > 0) {
    resp2.innerText = `Notas de R$ 50: ${notasCinquenta}`
}
if (notasDez > 0) {
    resp3.innerText = `Notas de R$ 10: ${notasDez}`
}
})

```

Nesse exemplo, recorremos novamente ao operador módulo (%). A partir dele é possível verificar se o valor solicitado para o saque pode ser pago com as notas disponíveis. Se o valor solicitado não for múltiplo de 10, o resto da divisão do valor por 10 vai produzir um valor diferente de zero, e a mensagem de advertência será exibida.

A realização de testes de validação contendo o comando `return` são uma forma de evitar a criação de diversos comandos `else` no programa. Assim, caso algum campo apresente um erro de validação, a mensagem é exibida e o programa retorna à página. Após as validações, a programação é realizada sem os possíveis problemas que dados inválidos poderiam causar, como uma divisão por zero.

Para calcular o número mínimo de notas de 100, 50 e 10 necessárias para pagar um saque, começamos pelo cálculo do número de notas de 100. Utilizamos a função `Math.floor()` para arredondar para baixo o resultado da divisão do valor solicitado por 100. Imagine alguns valores: $490/100 \Rightarrow 4.9$, com `Math.floor()` resulta 4; $1240/100 \Rightarrow 12.4$, com `Math.floor()` resulta 12.

O próximo passo é obter o valor que ainda não foi pago com as notas de 100. Para isso, podemos utilizar o operador módulo (%) outra vez. Com os valores exemplificados no parágrafo anterior: $490 \% 100$, resulta 90; $1240 \% 100$, resulta 40. Ou seja, os valores que precisam ser pagos com as notas de 50 e 10. E o processo para o cálculo das notas de 100 e para o cálculo do

resto é então aplicado novamente para as notas de 50 e 10. Aliás, ele poderia ser empregado para outras notas (20 ou 5, por exemplo), caso necessário.

Mas por que foi criada uma condição antes de exibir cada número de notas? Apenas para evitar que as mensagens com o número de notas com zero seja exibida. Por exemplo, se alguém quiser sacar R\$ 50,00, não fica bem o programa apresentar Notas de 100: 0 e Notas de 10: 0. Ele apenas deve apresentar as saídas com o número de notas se houver notas do valor exibido.

4.7 Exemplos com Node.js

Vamos construir mais três exemplos de programas com condições, agora usando o Node.js, para você praticar. A fim de manter organizados os nossos programas, vamos criar e inserir esses programas em uma pasta nodejs dentro de cap04.

a) A entrada para um clube de pesca custa R\$ 20,00 por pessoa e cada pessoa tem direito a levar um peixe. Peixes extras custam 12,00. Elabore um programa que leia o número de pessoas de uma família que foram ao clube e o número de peixes obtidos na pescaria. Informe o valor a pagar.

Exemplo 4.7 – Programa Pescaria (nodejs/ex4_7.js)

```
const prompt = require("prompt-sync")() // adiciona pacote prompt-sync
const pessoas = Number(prompt("Nº Pessoas: ")) // lê dados de entrada
const peixes = Number(prompt("Nº Peixes: "))
let pagar // declara variável pagar
if (peixes <= pessoas) { // condição definida no exemplo
    pagar = pessoas * 20
} else {
    pagar = (pessoas * 20) + ((peixes - pessoas) * 12)
}
console.log(`Pagar R$: ${pagar.toFixed(2)}`) // exibe o valor a ser pago
```

A seguir, um exemplo de execução desse programa:

```
C:\livrojs\cap04\nodejs>node ex4_7
```

```
Nº Pessoas: 4
```

```
Nº Peixes: 5
```

Pagar R\$: 92.00

b) Uma farmácia necessita de um programa que leia o total de uma compra. Exiba como resposta o nº máximo de vezes que o cliente pode parcelar essa compra e o valor de cada parcela. Considere as seguintes condições: a) cada parcela deve ser de, no mínimo, R\$ 20,00; b) o número máximo de parcelas permitido é 6.

Exemplo 4.8 – Programa Farmácia (nodejs/ex4_8.js)

```
const prompt = require("prompt-sync")() // adiciona pacote prompt-sync
const valor = Number(prompt("Valor da Compra R$: ")) // lê valor da compra
const aux = Math.floor(valor / 20) // aux = nº de parcelas sem condições
const parcelas = aux == 0 ? 1 : aux > 6 ? 6 : aux // operador ternário
const valorParcela = valor / parcelas // cálculo do valor da parcela
console.log(`Pode pagar em ${parcelas}x de R$: ${valorParcela.toFixed(2)}`)
```

Para calcular o número de parcelas, no Exemplo 4.8, foi utilizado o operador ternário. Ele equivale às seguintes linhas de código:

```
let parcelas
if (aux == 0) {
  parcelas = 1
} else if (aux > 6) {
  parcelas = 6
} else {
  parcelas = aux
}
```

Ou seja, economizamos várias linhas com o operador condicional. Em atribuições simples de valor, como nesse caso, dê preferência por utilizá-lo.

A seguir, a execução desse programa.

```
C:\livrojs\cap04\nodejs>node ex4_8
Valor da Compra R$: 90.00
Pode pagar em 4x de R$: 22.50
```

c) Elaborar um programa que leia um número – que deve ser uma centena. Calcule e exiba a centena invertida. Caso o número não seja uma centena, exiba mensagem.

Exemplo 4.9 – Programa Inverte Centena (nodejs/ex4_9.js)

```
const prompt = require("prompt-sync")() // adiciona pacote prompt-sync
```

```

const numero = Number(prompt("Número (centena): ")) // lê o número
if (numero < 100 || numero >= 1000) {
    console.log("Erro... deve ser uma centena")
    return
}
const num1 = Math.floor(numero / 100) // obtém 1º número
const sobra = numero % 100          // o que sobra (dezena)
const num2 = Math.floor(sobra / 10)   // obtém 2º número
const num3 = sobra % 10             // obtém 3º número
console.log(`Invertido: ${num3}${num2}${num1}`) // exibe o número invertido

```

Como nos exemplos com HTML e JavaScript, o comando `return` serve para fazer com que o programa retorne ao ponto de origem, sem executar o restante dos comandos. Nesse caso, a origem é a linha de comandos. Faça alguns testes e confira o funcionamento do programa.

```

C:\livrojs\cap04\nodejs>node ex4_9
Número (centena): 278
Invertido: 872

```

4.8 Exercícios

Que tal você agora criar alguns programas utilizando as estruturas de condições discutidas neste capítulo? A seguir, alguns exercícios que exigem a inserção dos comandos condicionais. Você pode construir no modelo HTML com JavaScript ou a partir do Node.js. Exemplos de respostas estão disponíveis no site da editora. Mas lembre-se: tente resolver primeiro os programas com os cuidados destacados nesses primeiros capítulos antes de ver os exemplos de resolução. A prática é fundamental para o processo de aprendizado de Algoritmos.

a) Elaborar um programa que leia um número. Informe se ele é par ou ímpar. Faça com o if... else... tradicional e, após, tente criar a condição com o operador ternário. A Figura 4.7 ilustra a tela de execução do programa. Para os exercícios, foi utilizada uma figura padrão, mas você pode substituí-la caso tenha interesse.

Exercício 4.a

Arquivo | C:/livrojs/cap04/resp4_a.html

Programa Par ou Ímpar

Número:

5 é ímpar

Figura 4.7. – Programa Par ou Ímpar.

b) Elaborar um programa que leia a velocidade permitida em uma estrada e a velocidade de um condutor. Se a velocidade for inferior ou igual à permitida, exiba “Sem Multa”. Se a velocidade for de até 20% maior que a permitida, exiba “Multas Leve”. E, se a velocidade for superior a 20% da velocidade permitida, exiba “Multas Grave” – conforme ilustra a Figura 4.8.

Exercício 4.b

Arquivo | C:/livrojs/cap04/resp4_b.html

Programa Verifica Velocidade

Velocidade Permitida:

Velocidade do Condutor:

Situação: Multa Leve

Figura 4.8 – Exemplo de dados de entrada e saída do programa verifica velocidade.

c) Elaborar um programa para simular um parquímetro, o qual leia o valor de moedas depositado em um terminal de estacionamento rotativo. O programa deve informar o tempo de permanência do veículo no local e o troco (se existir), como no exemplo da Figura 4.9. Se o valor for inferior ao tempo mínimo, exiba a mensagem: “Valor Insuficiente”. Considerar os valores/tempos da Tabela 4.6 (o máximo é 120 min).

Tabela 4.6 – Valores do Parquímetro.

Valor R\$	Tempo (min)
1,00	30
1,75	60
3,00	120



Figura 4.9 – Depois de confirmar o depósito, o programa deve exibir tempo e troco (se houver).

d) Elaborar um programa que leia três lados e verifique se eles podem ou não formar um triângulo. Para formar um triângulo, um dos lados não pode ser maior que a soma dos outros dois. Caso possam formar um triângulo, exiba também qual o tipo do triângulo: Equilátero (3 lados iguais), Isósceles (2 lados iguais) e Escaleno (3 lados diferentes). A Figura 4.10 exibe um exemplo de execução do exercício.

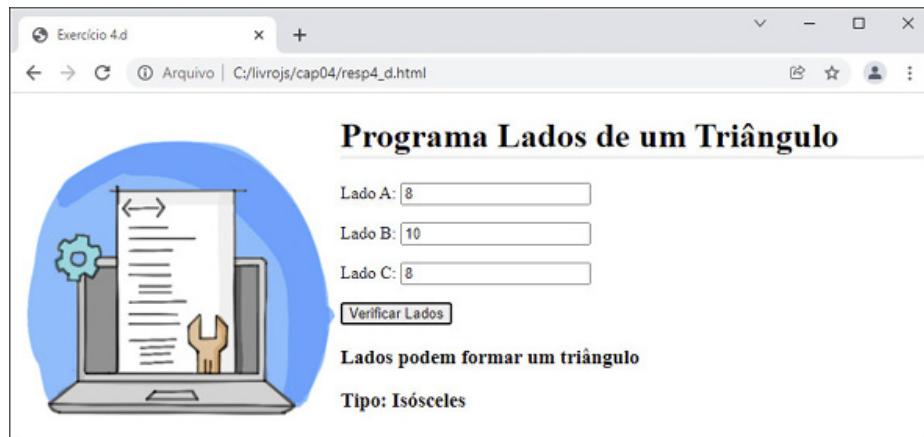


Figura 4.10 – Exemplo dos dados do programa lados de um triângulo.

4.9 Considerações finais do capítulo

As estruturas condicionais cumprem um importante papel no processo de construção de programas. Com as condições, é possível, por exemplo, liberar ou negar o acesso a uma área restrita de um site, validar o saque de um cliente em um terminal de caixa eletrônico ou, então, exibir uma mensagem de “Você foi Aprovado” ou “Você foi Reprovado”.

Para definir uma condição em JavaScript, podem ser utilizados os comandos `if.. else` ou `switch.. case`. Quando o número de opções for grande e baseado no conteúdo de uma variável, é recomendável utilizar o `switch.. case`. Os demais casos podem ser resolvidos com a estrutura condicional `if.. else`. As condições são definidas a partir do uso dos operadores relacionais: igual (`==`), diferente (`!=`), maior (`>`), menor (`<`), maior ou igual (`>=`) e menor ou igual (`<=`). Cada condição deve retornar um valor lógico de verdadeiro ou falso, como `if (idade >= 18)`.

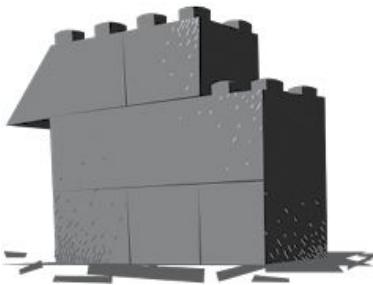
Algumas situações nos programas exigem que várias comparações sejam definidas. Obter os veículos com preço inferior a R\$ 20.000,00 e de uma determinada marca, por exemplo. Para que uma condição seja definida a partir de duas ou mais comparações, é necessário utilizar os operadores lógicos, que são: `not (!)`, `and (&&)` e `or (||)`. Quando é necessário que duas ou mais comparações sejam simultaneamente verdadeiras, deve-se utilizar o `and (&&)`. E se, com apenas uma comparação verdadeira, a condição deva ser aceita, utiliza-se o operador lógico `or (||)`.

Existe ainda o operador ternário, útil para definir condições de uma forma abreviada – no geral, em uma única linha. Para quem está iniciando, o recomendável é entender bem as estruturas condicionais, implementá-las com o `if... else` e, aos poucos, ir explorando essa forma alternativa de criação de condições. Nos exemplos deste capítulo, foi possível destacar a importância do uso das estruturas condicionais. Também foi apresentado um exemplo com a programação de duas funções, uma para realizar um cálculo a partir dos dados inseridos pelo usuário e outra para preparar o programa para uma nova interação com o usuário. Com os operadores lógicos, foi possível realizar validações para os dados de entrada do usuário.

Ou seja, nossos exemplos avançaram em diversos aspectos. Sinta-se motivado para realizar os exercícios propostos neste capítulo e mostrar aos seus amigos os programas legais que você já está desenvolvendo!

CAPÍTULO 5

Repetições



As estruturas de repetição permitem fazer com que um ou mais comandos em um programa sejam executados várias vezes. Essas estruturas, também denominadas laços de repetição ou loops, complementam a programação sequencial e a programação condicional, vistas anteriormente.

Quando discutimos sobre o funcionamento geral de um programa instalado em um terminal de caixa eletrônico (de novo ele...), destacamos que algumas ações ocorrem de forma sequencial, como solicitar a identificação do cliente e ler a sua senha. Outras ficam vinculadas a uma condição: se o cliente informou a senha correta, se o valor informado é múltiplo de 10, se há saldo suficiente na conta... Vamos agora abordar as estruturas que permitem fazer com que algumas ações sejam executadas diversas vezes no programa, como solicitar novamente a senha, pagar várias contas ou exibir todas as movimentações financeiras do cliente em um intervalo de datas. É importante ressaltar, contudo, que essas repetições necessitam de um ponto de interrupção, que pode ocorrer a partir de uma ação do usuário ou a partir de uma configuração do sistema, como a indicação de um limite para a digitação de senhas incorretas.

As estruturas de repetição são muito utilizadas para manipular listas de dados, como a lista de produtos em promoção em um site de comércio eletrônico. Monta-se o layout de apresentação de cada produto, com

imagem, descrição, marca e preço, e, a partir de um laço de repetição, percorrem-se todos os itens para exibi-los no site. Outro exemplo de uso dos loops é a apresentação dos valores das parcelas de um financiamento de um imóvel. Imagine calcular e exibir no programa o valor e a data de vencimento das 360 parcelas do financiamento... Com um laço de repetição, o processo se torna muito mais simples.

No contexto do desenvolvimento de interfaces para páginas web, o uso das estruturas de repetição é igualmente imprescindível. No Exemplo 11.2, vamos construir o layout para representar um teatro com uma imagem a fim de ilustrar cada uma das suas 240 poltronas. Organizamos os comandos para exibir cada poltrona e os envolvemos em uma estrutura de repetição com o intuito de representar essa exibição 240 vezes. Certo, mas e como indicar as poltronas ocupadas? A partir da inclusão de um `if` condicional dentro da repetição. As estruturas de programação sequencial, condicional e repetição são utilizadas em conjunto na construção dos nossos algoritmos. A cada capítulo, um novo assunto é abordado e as estruturas anteriormente trabalhadas continuam importantes e são utilizadas conforme as particularidades de cada programa. Por exemplo, ainda neste exercício de montagem do layout das poltronas do teatro, para fazer uma quebra de fila, ou seja, para não exibir todas as poltronas em uma única linha, será preciso utilizar o operador módulo (%). A cada 20 poltronas, por exemplo, outra fila deve ser criada e, se o número da poltrona exibida % 20, resultar 0, é sinal de que uma nova quebra deve ser inserida.

Para construir as estruturas de repetição em um programa, a linguagem JavaScript dispõe dos comandos `for`, `while` e `do... while`. Pequenas particularidades fazem com que o uso de cada um deles seja mais adequado para uma ou outra situação. Começamos analisando o funcionamento do comando `for`.

5.1 Repetição com variável de controle: laços `for`

A sintaxe do comando `for` é composta de três instruções, que definem: a) o

valor inicial da variável de controle; b) a condição que determina se a repetição deve ou não continuar; c) o incremento ou decremento da variável de controle. A Figura 5.1 destaca a sintaxe de um comando `for` com valores de exemplo atribuídos a uma variável de controle `i`.

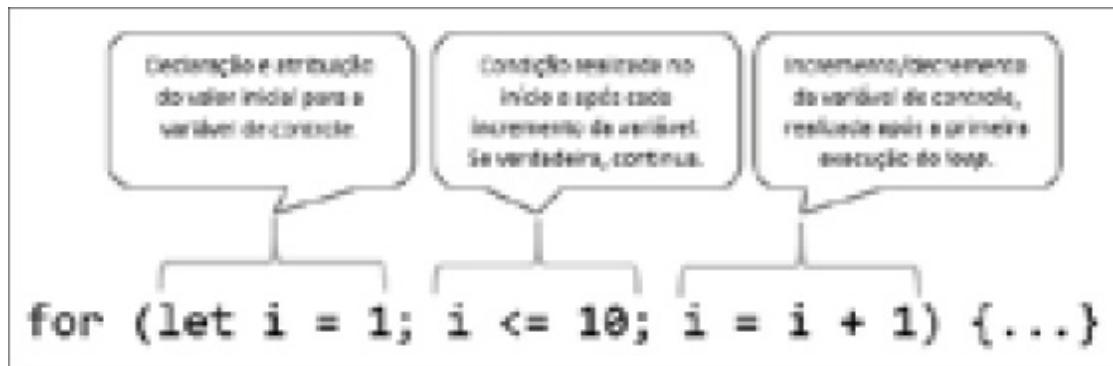


Figura 5.1 – Sintaxe do comando `for` com o significado de cada parte.

Entre as chaves {} devem ser inseridos os comandos que serão executados repetidas vezes. O incremento `i = i + 1` pode ser abreviado por `i++`. A repetição é controlada por uma variável, que, no exemplo da Figura 5.1, inicia em 1 e aumenta até 10. Outros valores, porém, podem ser indicados, como a variável de controle iniciar em 10 e decrescer até 1. Nesse último caso, utiliza-se `i = i - 1`, ou `i--`, na terceira instrução.

Poderíamos comparar o loop executado pelo comando `for` a um maratonista percorrendo as voltas de uma pista de atletismo. Na execução do `for`, quando se realiza uma volta completa, o valor da variável é incrementado e essa sequência de passos é repetida até o final. O maratonista realiza um processo semelhante, pois, ao passar pelo ponto de partida, tem o seu número de voltas incrementado em 1. Os loops do `for` contêm, na sua variável `i`, o valor indicativo da volta sendo realizada. A Figura 5.2 ilustra o funcionamento de um loop `for`, que contém inclusive uma condição que avalia o valor da variável `i`, retornando verdadeiro nas voltas de número par.

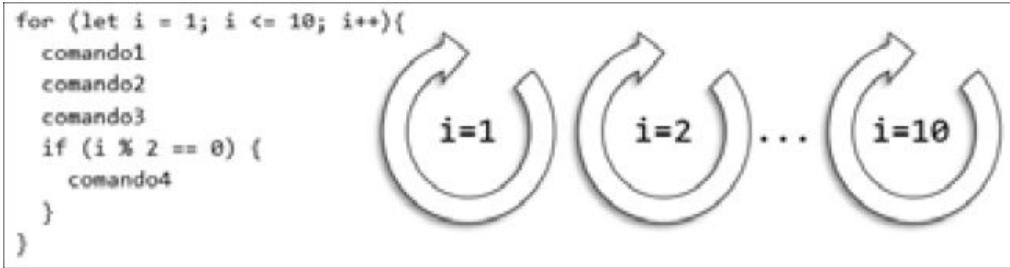


Figura 5.2 – A cada loop, a variável de controle “i” é incrementada em um.

Vamos continuar nas ilustrações sobre o funcionamento do comando `for`, pois entender os passos executados por esse comando é fundamental para a criação das estruturas de repetição. Observe a numeração inserida nas instruções que compõem o `for` exibidas na Figura 5.3.

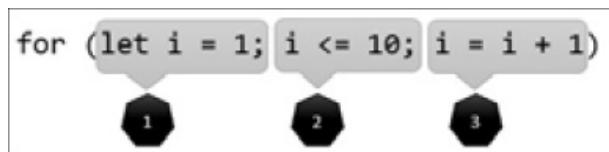


Figura 5.3 – Ao entender a sequência de execução das partes que compõem o comando `for`, você poderá montar diversas variações de valores para a variável de controle.

A sequência de execução das instruções é a seguinte: 1 e 2 (executa), 3 e 2 (executa), 3 e 2 (executa), (segue 3 e 2 até a condição ficar falsa).

Para facilitar a compreensão do que ocorre passo a passo na execução desse comando em um programa, vamos analisar o trecho de código a seguir.

```

let numeros = ""
for (let i = 1; i < 4; i = i + 1) {
    numeros = numeros + i
}
resp.innerHTML = numeros

```

Após a variável `numeros` ser declarada e inicializada, as seguintes operações são realizadas pelo laço `for`:

1. A variável `i` é declarada e recebe o valor 1.
2. O teste condicional é realizado (`i < 4`) e retorna verdadeiro.
3. Portanto, o comando do laço é executado: o valor de `i` (1) é atribuído à variável `numeros`, que recebe ela mesma + `i`, ou seja, `numeros = "1"`.

4. Volta-se ao comando `for` e a terceira instrução é executada: $i = i + 1$. Logo, $i = 2$.
5. O teste condicional é novamente realizado ($i < 4$) e continua verdadeiro.
6. Assim o comando do laço é executado, o valor de i (2) é atribuído à variável `numeros`, que recebe ela mesma $+ i$: `numeros = "12"`.
7. A terceira instrução do comando `for` é novamente executada, $i = i + 1$. Logo, $i = 3$.
8. O teste condicional é realizado ($i < 4$) e prossegue verdadeiro.
9. Mais uma vez, i (3) é atribuído a `numeros`, junto ao conteúdo anterior dessa variável. Agora, `numeros = "123"`.
10. Volta-se à execução da terceira instrução do `for`: $i = i + 1$. Logo, $i = 4$.
11. O teste condicional é realizado ($i < 4$) e retorna falso. O laço é finalizado e se executa o comando após o `for`.

Vamos iniciar a construção dos exemplos deste capítulo. Crie a pasta `cap05` e, dentro dela, crie as pastas `css`, `img` e `js` (da mesma forma como no Capítulo 4). Abra o Visual Studio Code e inicie um novo arquivo. Insira as linhas a seguir nesse arquivo e salve-o com o nome `estilos.css`, na pasta `css`.

```
img.normal { float: left; height: 300px; width: 300px; }
img.alta { float: left; height: 420px; width: 300px; }
h1 { border-bottom-style: inset; }
pre { font-size: 1.2em; }
```

Observe que criamos novas regras para a estilização dos elementos das páginas deste capítulo. Como alguns programas exibem uma resposta em várias linhas, definimos duas regras para as imagens: `alta` e `normal`. Para indicar que uma imagem deve ser estilizada segundo uma dessas regras, deve-se utilizar `class="regra"` na tag correspondente. Caso fôssemos hospedar a página em um provedor de conteúdo, o recomendado seria trabalhar o tamanho da imagem para ela possuir os mesmos valores indicados no estilo. Mesmo assim, designar o tamanho da imagem no CSS é importante a fim de que o navegador reserve o espaço adequado para a imagem enquanto ela é carregada. Foi acrescentada também a definição de um tamanho de fonte para a tag `pre`, que será utilizada com frequência para exibir a resposta nos

programas deste capítulo.

Nosso primeiro programa sobre repetições deve ler um número e apresentar a tabuada desse número – um exemplo geralmente utilizado para demonstrar o funcionamento do comando `for`. O código HTML deve ficar conforme o Exemplo 5.1, para gerar uma página de acordo com a ilustração da execução do programa da Figura 5.4.

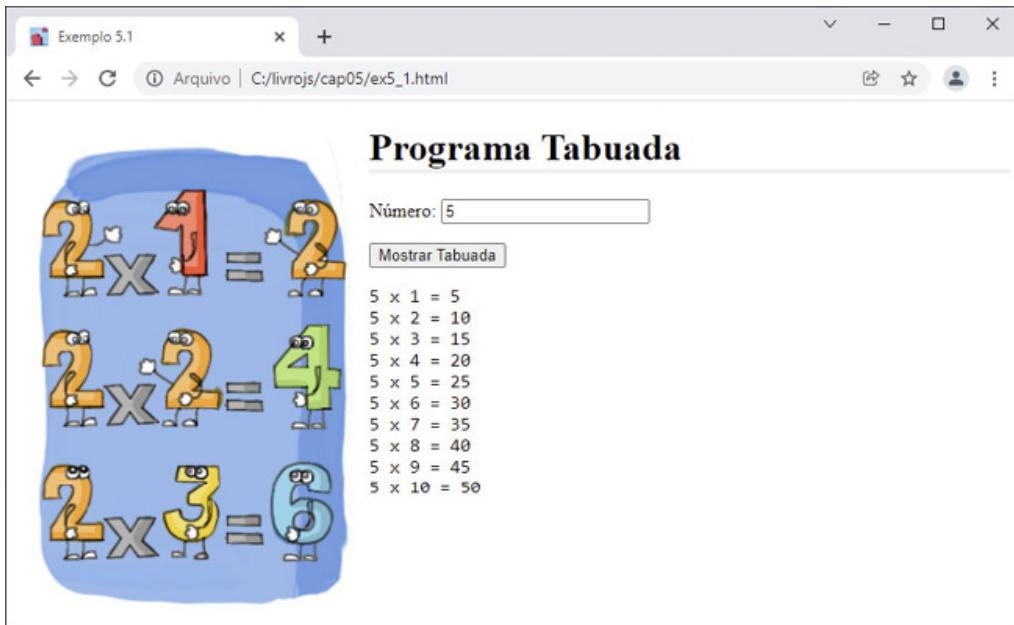


Figura 5.4 – Programa Tabuada: aplicação do comando `for`.

Exemplo 5.1 – Código HTML do Programa Tabuada (ex5_1.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1> Programa Tabuada </h1>
<form>
  <p>Número:
    <input type="number" id="inNumero" required>
  </p>
  <input type="submit" value="Mostrar Tabuada">
</form>
<pre></pre>
<script src="js/ex5_1.js"></script>
<!-- /body e /html -->
```

Observe que, para indicar que a imagem deve ser estilizada pela regra css

`img.alta`, foi acrescentado na tag `img` o atributo `class="alta"`. Além disso, uma nova tag foi utilizada para identificar o local em que a resposta do programa JavaScript será apresentada. Trata-se da tag `<pre>`, que possui como característica exibir um texto pré-formatado que mantém espaços e quebras de linha inseridas no código HTML, algo que não ocorre com as tags `<h3>` ou `<h4>` utilizadas nos exemplos anteriores.

Uma dica do editor Visual Studio Code: para criar o arquivo JavaScript indicado na tag `<script src="js/ex5_1.js">`, basta pressionar **Ctrl (Cmd, no Mac)** e clicar com o botão esquerdo do mouse sobre o nome do arquivo. Uma caixa indicando que o arquivo não foi encontrado é exibida com a opção **Criar Arquivo**. Clicar nessa opção faz com que o arquivo seja criado já no formato indicado pela extensão do nome do arquivo.

Insira nesse arquivo o programa JavaScript descrito a seguir:

Código JavaScript do Programa Tabuada (js/ex5_1.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("pre")

frm.addEventListener("submit", (e) => {    // "escuta" evento submit do form
  e.preventDefault()                  // evita envio do form
  const numero = Number(frm.inNumero.value) // obtém número informado
  let resposta = ""      // variável do tipo String, para concatenar a resposta
  // cria um laço de repetição (i começa em 1 e é incrementado até 10)
  for (let i = 1; i <= 10; i++) {
    // a variável resposta vai acumulando os novos conteúdos (nos 2 formatos)
    resposta = resposta + numero + " x " + i + " = " + (numero * i) + "\n"
    // resposta = `${resposta}${numero} x ${i} = ${numero * i}\n`
  }
  // o conteúdo da tag pre é alterado para exibir a tabuada do número
  resp.innerText = resposta
})
```

A estrutura de repetição implementada para acumular em uma string (`resposta`) a tabuada do número é semelhante ao formato discutido no passo a passo anterior. A repetição inicia em 1 e prossegue verdadeira enquanto o valor da variável `i` é menor ou igual a 10. Para gerar uma quebra de linha, é utilizado o caractere especial "`\n`". Depois de acumular os números para a

montagem da tabuada, o conteúdo da string resposta é exibido na página. Para montar o conteúdo da variável resposta pode ser feita uma atribuição concatenando strings e variáveis ou utilizando as template strings. Você pode optar por um dos formatos exibidos no código do exemplo ex5_1.js.

As instruções que compõem o comando `for` podem conter variáveis, ou seja, em vez de repetir até 10, a repetição poderia ir até um valor informado pelo usuário. No Exemplo 5.1, foi apresentada a forma tradicional, com a variável de controle iniciando em 1, repetindo enquanto ela for menor ou igual a 10, e sendo incrementada de 1 a cada repetição. No entanto, a variável de controle poderia incrementar de 2 em 2, por exemplo. Ou conter outra variável para indicar qual é o incremento, ou iniciar em um valor definido pelo usuário e repetir até outro valor também definido pelo usuário. Ou, ainda, diminuir o valor da variável de controle a cada repetição.

O nosso segundo exemplo ilustra a montagem de uma estrutura de repetição decrescente, com o valor inicial informado pelo usuário. O código HTML é semelhante ao Exemplo 5.1, contudo, nesse exemplo vamos exibir os números em uma única linha. Portanto, podemos utilizar novamente a tag `h3` para destacar a lista de números. A Figura 5.5 apresenta uma página com a demonstração da resposta do programa.



Figura 5.5 – Lista de números em ordem decrescente com o valor inicial informado pelo usuário.

Exemplo 5.2 – Código HTML do Programa Números Decrescentes (ex5_2.html)

O código do programa JavaScript 5.2 vai obter o valor digitado pelo usuário e apresentar todos os números inteiros existentes entre o número informado e 1, de forma decrescente.

Código JavaScript do programa Números Decrescentes (js/ex5_2.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  const numero = Number(frm.inNumero.value) // obtém número informado
  let resposta = `Entre ${numero} e 1: ` // String para montar a resposta
  for (let i = numero; i > 0; i = i - 1) { // cria um for decrescente
    resposta = resposta + i + ", " // resposta acumula números (e vírgulas)
  }
  resp.innerText = resposta // exibe a resposta
})
```

Observe que a variável de controle *i* é inicializada com o valor da variável *numero*. A condição agora é válida enquanto o *i* for maior que 0. E a variável *i* é decrementada em 1 (poderia ser abreviada por *i--*).

Pois é... uma vírgula ficou “sobrando” no final da listagem, o que faz com que a resposta do programa não fique muito bonita, como pode ser verificado na Figura 5.5. O melhor é omitir essa última vírgula ou substituí-la por um ponto final. Para resolver isso, temos 3 formas principais de solução (a partir do que vimos até agora). São elas: a) criar uma condição para verificar se o número da repetição é o último a ser exibido; b) isolar o último número; c) isolar o primeiro número. Observe como poderia ficar o comando *for* com o primeiro exemplo de solução.

```
for (let i = numero; i > 0; i--) {
  if (i == 1) {
    resposta = resposta + i + ". " // ou resposta = `${resposta}${i}.`
  } else {
    resposta = resposta + i + ", " // ou resposta = `${resposta}${i}, `
```

```
    }  
}
```

Para esse exercício, tal solução não é a recomendada, pois o número de comparações realizadas pelo programa é muito alto, e a solução, pouco eficiente. Imagine se o usuário digitar o número 100: o programa vai realizar 100 comparações, sendo que, em apenas uma delas, o `if` é verdadeiro.

Isolar o último número é uma alternativa mais adequada para esse programa. Ou seja, o comando `for` repete até o número 2 e fora da repetição é acrescentado o último número: “1.”. A mudança que deve ocorrer no programa é a seguinte:

```
for (let i = numero; i > 1; i--) {  
    resposta = resposta + i + ","  
}  
resposta = resposta + "1."
```

Em JavaScript, a variável `i` declarada no `for` a partir do comando `let` é uma variável de bloco. Ela não existe após o término do laço. Cuidado com esse detalhe.

A outra forma citada, isolar o primeiro número, segue a mesma lógica desse último exemplo, com uma pequena mudança. O primeiro valor conhecido é inicialmente atribuído à variável `resposta`. E, na repetição, o valor inicial da variável de controle exclui esse valor. As atribuições também devem ser modificadas. Observe as mudanças.

```
let resposta = `Entre ${numero} e 1: ${numero}` // já contém o 1º número  
for (let i = numero - 1; i > 0; i--) {  
    resposta = resposta + ", " + i  
}
```

Nessa última forma, dentro da repetição, `resposta` recebe `resposta`, a vírgula e o valor de `i`. Ou seja, o primeiro número é exibido sem a vírgula (antes do laço), e os próximos números (na repetição) são precedidos pela vírgula. Logo, o último número fica sem a vírgula.

O comando `for` é particularmente interessante de ser utilizado quando soubermos o número de repetições que devem ocorrer no programa. O exemplo da tabuada ilustra bem essa situação. Independentemente do

número informado pelo usuário, a exibição da tabuada vai de 1 até 10. No segundo exemplo, não sabemos previamente qual será o número informado pelo usuário, ou seja, quantas vezes a repetição vai ocorrer. Mas essa informação é obtida pelo programa antes da montagem da repetição, portanto o programa dispõe da informação para montar o for.

5.2 Repetição com teste no início: laços while

Um laço de repetição também pode ser criado com o comando `while`, que realiza um teste condicional logo no seu início, para verificar se os comandos do laço serão ou não executados. A tradução da palavra `while`, que em português significa enquanto, define bem o seu funcionamento: “enquanto a condição for verdadeira, execute”. A sintaxe do comando `while` é a seguinte:

```
while (condição) {  
    comandos  
}
```

As estruturas de repetição com teste no início, representadas pelo comando `while`, são utilizadas principalmente em programas que manipulam arquivos, para repetir a leitura de uma linha enquanto não atingir o final do arquivo. Elas também podem ser utilizadas para realizar as operações desenvolvidas com o comando `for`. Por exemplo, no Programa Números Decrescentes, destacado anteriormente, poderíamos substituir o comando `for` pelo comando `while`, da seguinte forma:

```
let i = numero          // declara e inicializa a variável i  
while (i > 0) {         // enquanto i maior que 0  
    resposta = resposta + i + ", " // acumula valores de i  
    i--                      // decrementa o i (idem a i = i - 1)  
}
```

Como o teste é realizado no início, é possível que os comandos do `while` não sejam executados. No exemplo acima, caso o usuário digite o número 0, a condição já é falsa na primeira verificação e o programa não entra no laço de repetição.

5.3 Repetição com teste no final: laços do.. while

Outra forma de criar laços de repetição em um programa é com a utilização do comando do.. while, cuja sintaxe é representada a seguir:

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

Uma sutil, porém importante, diferença entre as estruturas de repetição while e do.. while é a seguinte: com o comando while, a condição é verificada no início; enquanto, com o comando do.. while, a condição é verificada no final. Ou seja, com o do.. while, fica garantido que uma vez, no mínimo, os comandos que pertencem ao laço serão executados.

Geralmente optamos por utilizar os laços while e do.. while quando não soubermos previamente quantas vezes a repetição vai ocorrer. Pense em algo como o processo do recebimento de contas realizado por um terminal de caixa eletrônico; o sistema efetua o recebimento de uma conta e, no final, pergunta ao cliente se ele deseja pagar outra conta. Como os comandos utilizados para receber cada conta são os mesmos, eles ficam dentro de uma estrutura de repetição. E o programa pode ser utilizado tanto pelos clientes que desejam pagar apenas uma conta, quanto por aqueles que necessitam pagar várias contas.

No Exemplo 5.3, destacado a seguir, é utilizado o laço do.. while para validar uma entrada de dados do cliente. Caso o cliente não informe um número, o programa exibe um alerta e realiza novamente a leitura. O programa poderá executar o método prompt() uma única vez, ou inúmeras, dependendo do que for informado pelo usuário.

O programa utiliza os métodos prompt() e o alert() para ilustrar o funcionamento do laço criado com o comando do.. while na entrada de dados. Após a validação, o programa exibe todos os números pares entre 1 e o número informado pelo usuário. A Figura 5.6 apresenta um exemplo de saída de dados desse script.

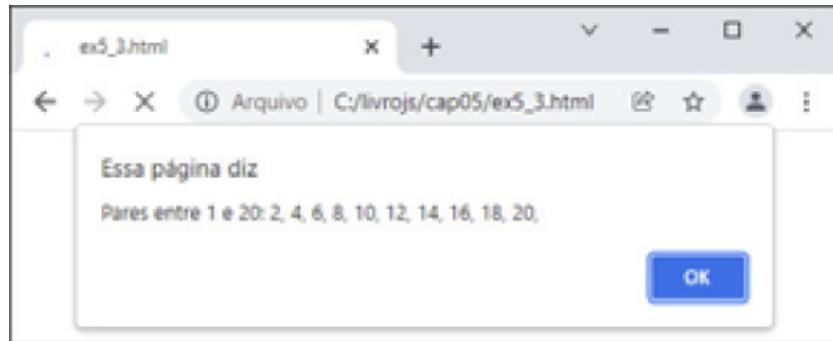


Figura 5.6 – Lista dos pares entre 1 e 20 (número informado pelo usuário).

Exemplo 5.3 – Programa JavaScript para ilustrar o funcionamento do laço do.. while (ex5_3.html)

```
<script>
let num // declara variável num com let, pois ela pode ser alterada
// e será acessada fora do bloco do.. while
do { // cria laço de repetição (faça...)
    num = Number(prompt("Número: ")) // lê um número
    if (num == 0 || isNaN(num)) { // se num=0 ou é inválido
        alert("Digite um número válido...")
    }
} while (num == 0 || isNaN(num)) // ... enquanto num=0 ou inválido
let pares = `Pares entre 1 e ${num}: ` // string que irá conter a resposta
for (let i = 2; i <= num; i = i + 2) {
    pares = pares + i + ","
}
alert(pares) // exibe lista dos números pares
</script>
```

Os comandos dentro do laço do.. while serão repetidos enquanto o número informado for 0 ou inválido. Um detalhe nessa condição é que o método `Number()`, se aplicado a um conteúdo vazio, retorna o valor 0. Caso o usuário informe um número inválido (um texto ou um número com vírgula, por exemplo), a conversão realizada pelo método `Number()` resultará em "NaN" (Not-a-Number) e a mensagem será exibida.

5.4 Interrupções nos laços (break e continue)

As linguagens de programação dispõem de dois comandos especiais para serem utilizados nas estruturas de repetição. São eles: `break` e `continue`. O `break`

sai do laço de repetição, enquanto o continue retorna ao início do laço.

Estes comandos nos auxiliam no controle de execução dos comandos do loop. Observe a partir da ilustração da Figura 5.7, o que ocorre no laço de repetição no momento em que os comandos continue e break são executados.

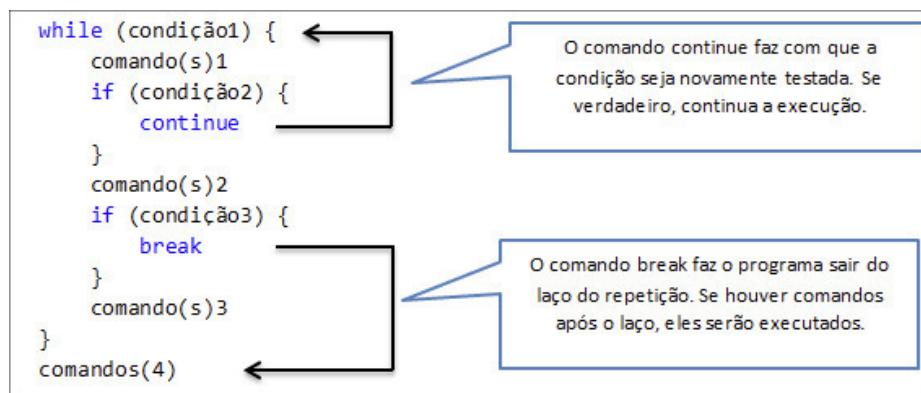


Figura 5.7 – Comandos continue e break modificam o fluxo dos comandos da repetição.

Os comandos break e continue podem ser utilizados nas três estruturas de repetição disponíveis: for, while ou do... while. Caso o comando continue seja executado em um laço for, o incremento ou decrecimento da variável de controle ocorre normalmente, como se o laço tivesse sido executado até o seu final.

O Exemplo 5.4 também utiliza os métodos prompt() e alert(), agora para demonstrar o funcionamento dos comandos break e continue. O programa realiza a leitura de um número e, caso o número for par ele exibe o dobro do número e se for ímpar, o triplo. A leitura continua até que o usuário informe 0 (ou algum valor inválido). Inicialmente, uma mensagem alertando sobre a execução do programa é apresentada.

Exemplo 5.4 – Uso dos comandos break e continue em um laço de repetição (ex5_4.html)

```
<script>
alert("Digite 0 para sair")
do {
    const num = Number(prompt("Número: ")) // lê o número
    if (num == 0 || isNaN(num)) {           // se num=0 ou inválido
        const sair = confirm("Confirma saída?") // solicita confirmação do usuário
```

```

if (sair) {
    break // sai da repetição
} else {
    continue // volta ao início do laço
}
}

if (num % 2 == 0) { // se par,
    alert(`O dobro de ${num} é: ${num * 2}`) // mostra o dobro
} else { // senão,
    alert(`O triplo de ${num} é: ${num * 3}`) // mostra o triplo
}

} while (true) // enquanto verdade (só sai do laço, pelo break)
alert("Bye, bye...")
</script>

```

Repare que um novo método foi utilizado para solicitar a confirmação de saída do programa. Trata-se do método `confirm()`, que exibe uma caixa de diálogo com os botões **Ok** e **Cancelar**. Ele retorna `true` (`ok`) ou `false` (`cancelar`) de acordo com a escolha do usuário. A Figura 5.8 ilustra a saída do método `confirm()`.

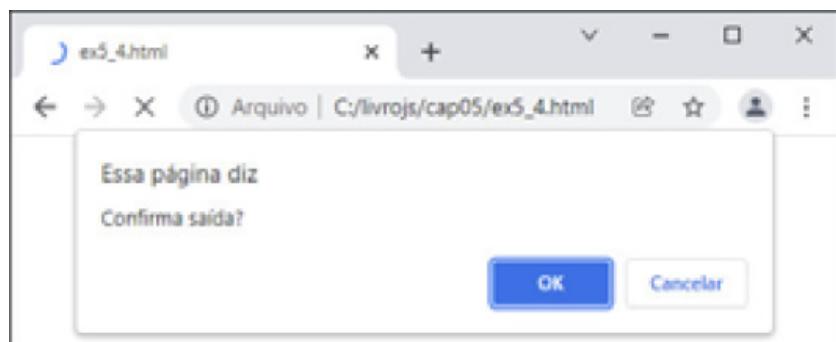


Figura 5.8 – Retorno do método `confirm()` vai definir a execução do `break` ou `continue`.

Caso o usuário confirme a saída do programa, o comando `break` é executado e o método `alert("Bye, bye...")`, após o laço de repetição, é chamado. Contudo, caso o usuário não confirme a saída, o comando `continue` retorna ao início do laço e uma nova leitura é realizada pelo método `prompt()`.

A condição inserida no comando `while()` foi substituída pelo valor `true`. Isso significa que a repetição não sairá pela análise da condição. A única forma de saída desse laço, portanto, é pela execução do comando `break`.

Outro detalhe neste programa: as variáveis `num` e `sair` foram declaradas com `const`. Mas, se elas podem receber valores diferentes a cada repetição, elas não deveriam ser declaradas com `let`? O que ocorre é que as variáveis criadas dentro de um bloco de comandos (no caso, o `while`) deixam de existir ao final do bloco. Assim, a cada repetição, essas variáveis são novamente criadas – sem nenhuma “lembrança” da declaração anterior.

5.5 Contadores e acumuladores

O uso de contadores e acumuladores em um programa permite a exibição de contagens e totalizações. Essas operações são realizadas sobre os dados manipulados pelo programa. Os contadores ou acumuladores possuem duas características principais:

- A variável contadora ou acumuladora deve receber uma atribuição inicial (geralmente zero).
- A variável contadora ou acumuladora deve receber ela mesma mais algum valor.

A diferença entre os contadores e os acumuladores é que o contador recebe ele mesmo mais 1 (ou algum valor constante), enquanto o acumulador recebe ele mesmo mais uma variável. Para fazer uma variável receber ela mesma mais algum valor podemos repetir o nome da variável na atribuição ou utilizar o operador “`+ =`”. Observe os exemplos:

```
let soma = 0      // deve ser declarada com let
soma = soma + preco // em uma repetição, soma irá acumular o preco
soma += preco     // forma simplificada para soma = soma + preco
```

O Exemplo 5.5 apresenta um exemplo de uso dos contadores e acumuladores. O programa faz a leitura de contas que devem ser pagas por um usuário. As contas são exibidas e no final da listagem o número de contas (contador) e a soma dos valores (acumulador) são destacados. A Figura 5.9 exibe uma tela com algumas contas digitadas na execução do programa.

Exemplo 5.5 – Código HTML do programa Contas do Mês (ex5_5.html)

```

<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa Contas do Mês</h1>
<form>
  <p>Descrição da Conta:<br/>
    <input type="text" id="inDescricao" required>
  </p>
  <p>Valor a Pagar R$:<br/>
    <input type="number" min="0" step="0.01" id="inValor" required>
  </p>
  <input type="submit" value="Registrar Conta">
</form>
<pre id="outResp1"></pre>
<pre id="outResp2"></pre>
<script src="js/ex5_5.js"></script>
<!-- /body e /html -->

```



Figura 5.9 – Exemplo de contadores e acumuladores.

Código JavaScript do programa Contas do Mês (js/ex5_5.js)

```

const frm = document.querySelector("form") // obtém elementos da página
const resp1 = document.querySelector("#outResp1")
const resp2 = document.querySelector("#outResp2")

let resposta = "" // string com a resposta a ser exibida
let numContas = 0 // inicializa contador...

```

```

let valTotal = 0 // e acumulador (variáveis globais)

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form

  const descricao = frm.inDescricao.value // obtém dados da conta
  const valor = Number(frm.inValor.value)

  numContas++ // adiciona valores ao contador e acumulador
  valTotal = valTotal + valor // ou: valTotal += valor
  resposta = resposta + descricao + " - R$: " + valor.toFixed(2) + "\n"
  resp1.innerText = `${resposta} -----`
  resp2.innerText = `${numContas} Conta(s) - Total R$: ${valTotal.toFixed(2)}`

  frm.inDescricao.value = "" // limpa campos do form
  frm.inValor.value = ""
  frm.inDescricao.focus() // posiciona no campo inDescricao do form
})

```

Observe que nesse programa fizemos uso de variáveis globais. Em JavaScript, uma variável global continua disponível em memória enquanto a página está ativa. Para acumular os valores das contas e listá-los sempre que o usuário adicionar uma nova conta, é necessário o uso de variáveis com esse escopo. A variável `numContas` atua como um contador, para apresentar a cada acréscimo de conta o número de contas inseridas pelo usuário. Já a variável `valTotal` é um acumulador, para somar o valor das contas inseridas no programa.

Podemos também utilizar os contadores e acumuladores para nos auxiliar na exibição de uma resposta. O Exemplo 5.6 recebe um número e informa se ele é ou não primo. Apenas para relembrar, um número primo é aquele que possui apenas 2 divisores: 1 e ele mesmo. Nesse programa, faremos uso de uma variável contadora para obter a quantidade de divisores do número informado pelo usuário. A Figura 5.10 exibe a tela do programa números primos.



Figura 5.10 – Programa Números Primos.

Exemplo 5.6 – Código HTML para exibir o layout do programa Números Primos (ex5_6.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa Números Primos</h1>
<form>
  <p>Número:<br/>
    <input type="number" id="inNumero" required>
  </p>
  <input type="submit" value="Verificar se é Primo">
</form>
<h3></h3>
<script src="js/ex5_6.js"></script>
<!-- /body e /html -->
```

Observação: segundo novos conceitos matemáticos, o número 1 não deve mais ser considerado número primo. Em razão disso, o teste para verificar se o número é primo, no Programa ex5_6.js a seguir, contém apenas a comparação if (numDivisores == 2).

Código JavaScript do programa Números Primos (js/ex5_6.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form
```

```

const num = Number(frm.inNumero.value) // obtém número informado
let numDivisores = 0 // declara e inicializa contador
for (let i = 1; i <= num; i++) { // percorre todos os possíveis divisores de num
  if (num % i == 0) { // verifica se i (1, 2, 3...) é divisor do num
    numDivisores++ // se é, incrementa contador
  }
}
if (numDivisores == 2) { // se possui apenas 2 divisores, é primo
  resp.innerText = `${num} É primo`
} else {
  resp.innerText = `${num} Não é primo`
}
)

```

Como pode ser observado nesse exemplo, o contador serve de apoio para exibir a resposta do programa. Embora funcione corretamente, esse programa pode conter várias melhorias para aprimorar o seu desempenho. Imagine se um usuário digitar o número 1000. O programa vai repetir o laço criado pelo comando `for` 1000 vezes e realizar igual número de testes. Porém, como o número 2 já é um divisor do número 1000, as demais 998 repetições e os testes do programa são desnecessários.

Para melhorar o desempenho desse algoritmo, podemos utilizar uma variável com um comportamento semelhante ao de um contador. A variável de controle recebe o valor inicial 0. E, caso uma condição no laço seja verdadeira, trocamos o valor dessa variável para 1 e forçamos a saída do loop (com o comando `break`). Essas variáveis agem como uma flag (sinalizadora ou bandeira) e indicam a presença ou ausência de algo no conjunto de dados em análise. A variável sinalizadora (flag) também poderia receber os valores lógicos `true` ou `false`.

Observe as mudanças no trecho final do programa, a fim de otimizar a sua performance.

```

let temDivisor = 0 // declara e inicializa a variável tipo flag

for (let i = 2; i <= num / 2; i++) { // percorre os possíveis divisores do num
  if (num % i == 0) { // se tem um divisor
    temDivisor = 1 // muda o flag
    break // sai da repetição
  }
}

```

```

    }

if (num > 1 && !temDivisor) { // se num > 1 e não possui divisor
  resp.innerHTML = `${num} É primo`
} else {
  resp.innerHTML = `${num} Não é primo`
}

```

No comando `if` final, utilizamos uma particularidade da linguagem JavaScript (e de outras linguagens): o valor 0 (zero) equivale a um valor lógico `false` e 1 (ou outro valor diferente de 0) equivale a um valor lógico `true`. Assim, testar: `temDivisor == 0`, `temDivisor == false` ou, ainda, `!temDivisor` produz o mesmo resultado. Usar `!temDivisor` (não `temDivisor`) facilita a compreensão.

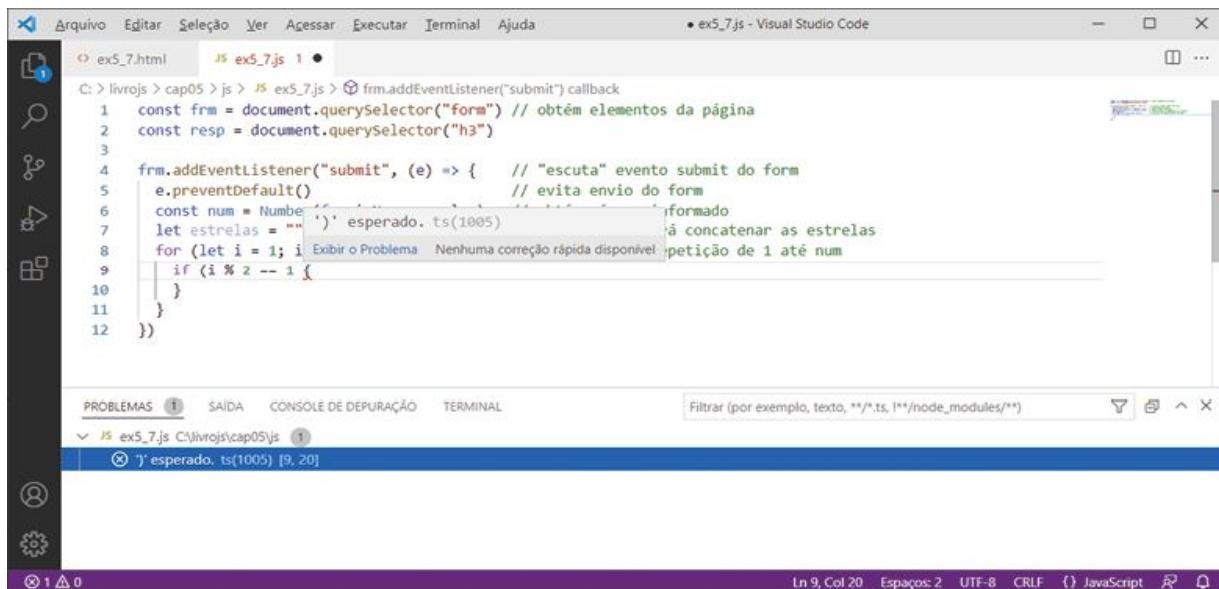
Com esse ajuste no loop `for` do programa, uma boa economia de processamento é realizada, pois, ao encontrar o primeiro divisor do número, a variável tipo flag tem o seu valor alterado e a repetição é interrompida. Para o número 1000 anteriormente citado, a repetição vai executar apenas 1 vez (já que o número 2 é um divisor do 1000). Outro detalhe: como o maior divisor inteiro possível de um número é a sua metade, indicamos que a repetição deve ir até `num / 2`. Talvez existam formas ainda mais eficientes para verificar se um número é primo. O objetivo neste momento não é aprofundar na matemática, mas destacar que, com pequenos ajustes, podemos otimizar o desempenho de nossos programas.

5.6 Depurar programas (detectar erros)

A palavra depurar, segundo o dicionário Aurélio online, significa tornar puro, limpar, retirar as impurezas. Em programação, depuração de programas é o nome dado ao processo de detectar e remover erros no código. Existem dois tipos principais de erros em um programa: erros de sintaxe e erros de lógica. Os erros de sintaxe impedem o programa de ser executado e referem-se à digitação incorreta de algum comando ou nome de variável. Em alguns casos, o próprio editor dá indicativos de que existe algum erro no código, como pode ser observado na Figura 5.11 – onde faltou um “)” no comando `if`. Os erros causam um sublinhado vermelho no código e o incremento do número exibido ao lado do “x” na barra inferior

do Visual Studio Code. Ao clicar nesse “x”, o editor exibe os problemas encontrados.

Existem plug-ins (complementos de programa) que podem ser adicionados ao Visual Studio Code para auxiliar no processo. Contudo, neste livro, vamos apresentar os recursos de depuração disponíveis no navegador Google Chrome, sem a necessidade de realizar novas instalações. Recursos semelhantes também existem nos demais navegadores web.



```
C:\livrojs>cap05>js> JS ex5_7.js > frm.addEventListener("submit") callback
1 const frm = document.querySelector("form") // obtém elementos da página
2 const resp = document.querySelector("h3")
3
4 frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
5   e.preventDefault() // evita envio do form
6   const num = Number(prompt("Digite um número"))
7   let estrelas = "" // variável para armazenar as estrelas
8   for (let i = 1; i <= num; i++) { // loop que irá concatenar as estrelas
9     if (i % 2 == 1) {
10       }
11     }
12   })

```

PROBLEMAS 1 SAÍDA CONSOLE DE DEPURAÇÃO TÉRMICO Filtrar (por exemplo, texto, **/*.ts, !**/node_modules/**)

JS ex5_7.js C:\livrojs\cap05\js 1

J' esperado. ts[1005] [9, 20]

Figura 5.11 – Alguns erros de sintaxe são detectados pelo próprio editor.

Os recursos de depuração dos browsers permitem identificar tanto erros de sintaxe quanto erros de lógica. Para analisar os recursos de depuração de programas JavaScript, vamos construir um exemplo – que também explora os conteúdos abordados neste capítulo. O programa deve ler um número que corresponde à quantidade de símbolos que devem ser preenchidos (em um cheque ou boleto bancário, por exemplo). O preenchimento deve intercalar os caracteres “*” e “-”. A Figura 5.12 ilustra uma execução desse programa.



Figura 5.12 – Programa Fábrica de Estrelas que será depurado no navegador.

Exemplo 5.7 – Código HTML do programa Fábrica de Estrelas (ex5_7.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Fábrica de Estrelas</h1>
<form>
<p>Número de Símbolos:
    <input type="number" id="inNumero" min="1" required>
</p>
    <input type="submit" value="Preencher Espaço">
</form>
<h3 id="outEspacos"></h3>
<script src="js/ex5_7.js"></script>
<!-- /body e /html -->
```

Os símbolos serão exibidos pelo programa JavaScript na linha da tag <h3>.

Código JavaScript do programa Fábrica de Estrelas (js/ex5_7.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
    e.preventDefault() // evita envio do form
    const num = Number(frm.inNumero.value) // obtém número informado
    let estrelas = "" // variável que irá concatenar as estrelas/traços
```

```

for (let i = 1; i <= num; i++) { // cria laço de repetição de 1 até num
  if (i % 2 == 1) {
    estrelas = estrelas + "*" // na posição ímpar do i: *
  } else {
    estrelas = estrelas + "-" // na posição par: -
  }
}
resp.innerHTML = estrelas // exibe as estrelas
})

```

Vamos criar alguns erros no código para verificar o funcionamento do depurador do Google Chrome. Inicialmente, modifique o nome da variável estrelas para estrela na declaração da variável. Para utilizar o depurador, carregue a página do Exemplo 5.7 no browser e, no menu superior direito, selecione **Mais Ferramentas > Ferramentas do Desenvolvedor**. Em seguida, selecione **Sources**. Caso o programa JavaScript não esteja na lista dos arquivos, pressione **F5** para atualizá-la. Para verificar o erro, selecione o arquivo ex5_7.js, preencha um valor no campo de formulário da página HTML e clique no botão **Preencher Espaço**. O depurador acusa o erro e exibe a mensagem indicando que a variável estrelas não foi definida, conforme ilustra a Figura 5.13.

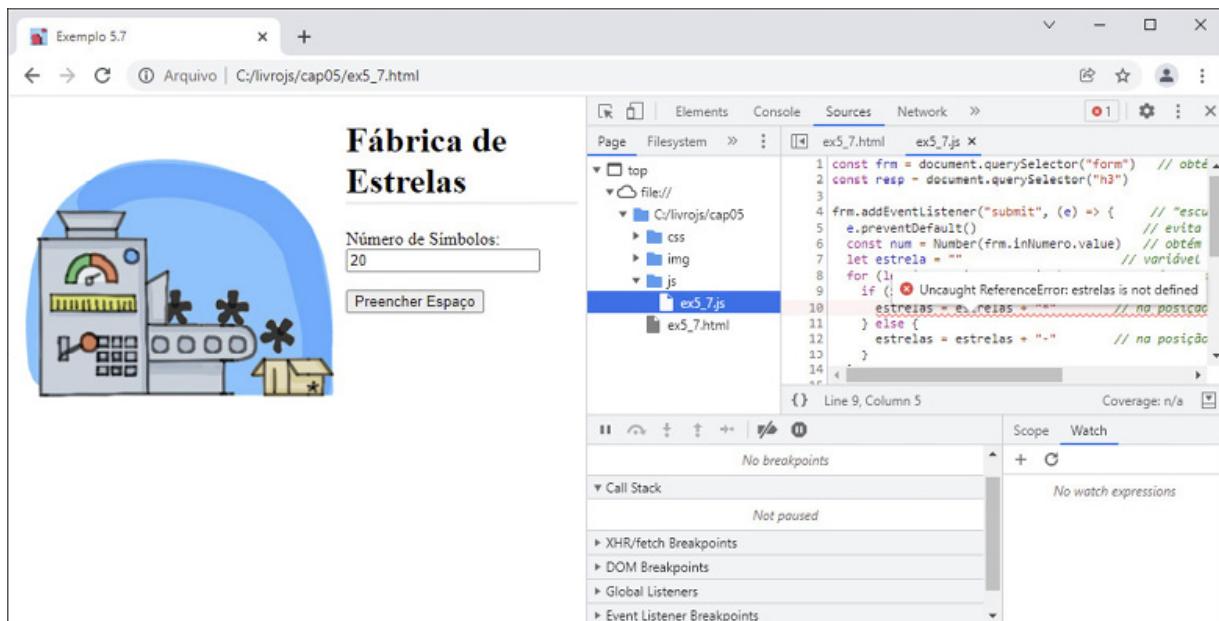


Figura 5.13 – Depurador posiciona na linha com erro e exibe descrição do problema verificado.

Os erros de lógica, por sua vez, são mais difíceis de serem detectados. Nesse caso, o programa é executado normalmente, porém não apresenta os resultados esperados. Para isso, o depurador dispõe de alguns recursos para nos auxiliar a identificar os problemas. Vamos demonstrar para que servem dois importantes recursos de depuração: os Breakpoints (pontos de parada) e a janela Watch (observador).

Um breakpoint define um ponto de parada em uma linha do programa. O programa é executado até aquela linha e, então, pode-se verificar o valor das variáveis naquele ponto de execução. Caso necessário, é possível adicionar outros pontos de parada, para que o programa seja executado passo a passo e, assim, analisar detalhadamente o seu funcionamento. Um pequeno erro de cálculo pode fazer, por exemplo, com que uma condição seja verdadeira. Com os breakpoints e a janela Watch, é possível verificar esses problemas.

Vamos realizar um teste para praticar esses recursos. Altere novamente o nome da variável que causou o erro anterior (estrelas) e atualize a página. Para criar um breakpoint, é necessário apenas clicar na margem esquerda do código, sobre o número da linha onde se deseja definir um ponto de parada. Podem ser adicionados vários breakpoints no mesmo programa. Informe um número no campo de formulário da página HTML e clique no botão para executar o programa JavaScript novamente. Observe que a execução vai até a linha onde você adicionou um breakpoint.

Para prosseguir com a execução do programa, pressione **F8** ou clique na seta da janela de depuração exibida. Embora o navegador exiba o conteúdo das variáveis nas linhas em que elas recebem alguma atribuição, também é possível utilizar a janela Watch para indicar as variáveis que se deseja observar na execução do programa. Para inserir uma variável para análise, clique no “+” abaixo de Watch. A Figura 5.14 ilustra o uso do depurador do Google Chrome com esses recursos em uso. Observe que os breakpoints foram definidos nas linhas dentro do laço de repetição. Assim, a cada repetição, é possível acompanhar a mudança que ocorre no conteúdo das variáveis *i* e estrelas, inseridas na janela Watch.

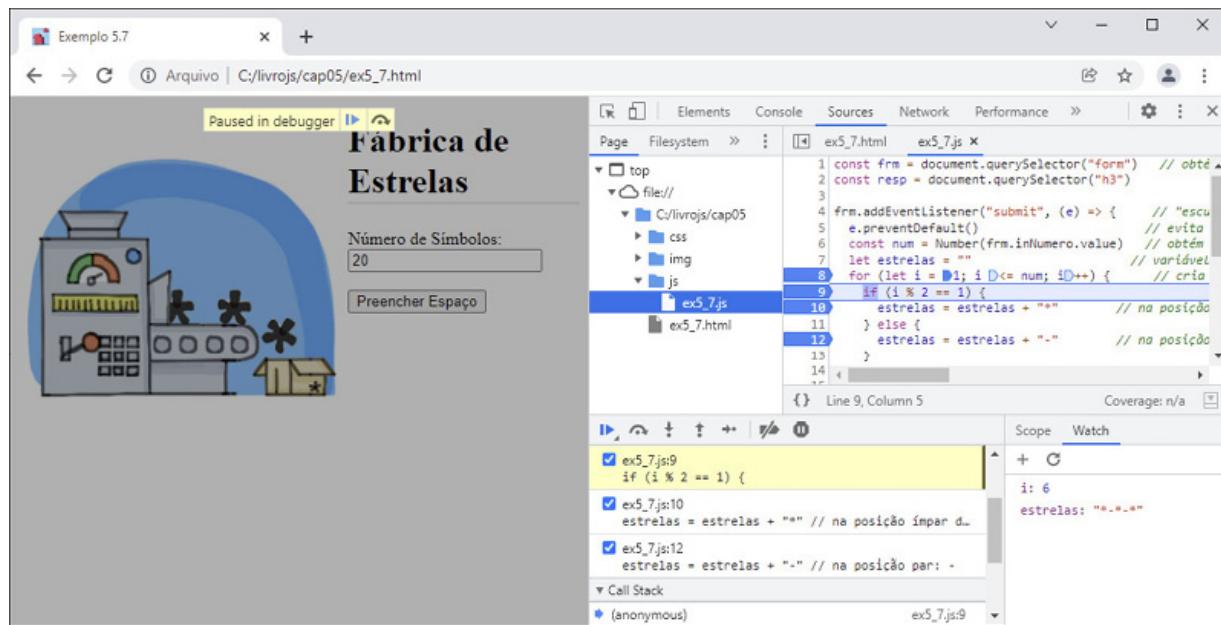


Figura 5.14 – Depurador do Chrome em ação.

Existem vários outros recursos que podem ser investigados no depurador de programas, como o uso do `console.log()`. Contudo, a maioria dos problemas pode ser detectada a partir do uso dos breakpoints e pela observação do conteúdo que as variáveis vão assumindo no decorrer do programa. Caso o seu código não funcione, peça auxílio ao depurador. Ele também é útil para você entender melhor o que está acontecendo no programa, servindo como fonte de aprendizado. Ou seja, o depurador é um importante aliado dos programadores, portanto, explore-o com frequência.

5.7 Exemplos de Algoritmos de Repetição com Node.js

Vamos construir novos exemplos de algoritmos de repetição, agora utilizando o Node.js. Como no capítulo anterior, crie a pasta `nodejs` (dentro de `cap05`) para manter a organização dos nossos arquivos.

a) A *Copa do Mundo* ocorre de 4 em 4 anos, desde 1930, exceto nos anos de 1942 e 1946 (Segunda Guerra Mundial). Construir um programa que repita a leitura de números (anos) até ser digitado 0. Informe para cada ano se ele é ou não ano de *Copa do Mundo*.

Código JavaScript do programa Anos de Copa do Mundo (nodejs/ex5_8.js)

```
const prompt = require("prompt-sync")()
console.log("Programa Anos de Copa do Mundo. Digite 0 para sair")
console.log("-----")
do {
    const ano = Number(prompt("Ano: "))
    if (ano == 0) {
        break // sai da repetição
    } else if (ano == 1942 || ano == 1946) {
        console.log(`Não houve Copa em ${ano} (Segunda Guerra Mundial)`)
    } else if (ano >= 1930 && ano % 4 == 2) {
        console.log(`Sim! ${ano} é ano de Copa do Mundo!`)
    } else {
        console.log(`Não... ${ano} não é ano de Copa do Mundo.`)
    }
} while(true)
```

Como não sabemos a quantidade de vezes que o usuário pretende repetir a digitação de números, utilizamos o do.. while. Os anos de 1942 e 1946 são tratados como exceção. Para verificar se o ano é de Copa do Mundo aplicamos o operador módulo, no caso % 4, pois ela ocorre de 4 em 4 anos. Podemos utilizar como exemplo o ano de 2002, que %4 resulta em 2. Essa é a regra que vale para todos os demais números. A seguir, o programa em execução.

```
C:\livrojs\cap05\nodejs>node ex5_8
Programa Anos de Copa do Mundo. Digite 0 para sair
-----
Ano: 2000
Não... 2000 não é ano de Copa do Mundo.
Ano: 2022
Sim! 2022 é ano de Copa do Mundo!
Ano: 0
```

b) *Elaborar um programa que leia o nome de um produto e o número de etiquetas a serem impressas desse produto. Exiba as etiquetas com o nome do produto, com no máximo 2 etiquetas por linha, conforme exemplo de execução do programa, demonstrado a seguir.*

```
C:\livrojs\cap05\nodejs>node ex5_9
```

Produto: Suco Natural de Uva

Nº de Etiquetas: 7

Suco Natural de Uva	Suco Natural de Uva
Suco Natural de Uva	Suco Natural de Uva
Suco Natural de Uva	Suco Natural de Uva
Suco Natural de Uva	Suco Natural de Uva

Código JavaScript do programa Etiquetas (nodejs/ex5_9.js)

```
const prompt = require("prompt-sync")()
const produto = prompt("Produto: ") // Lê nome do produto e ...
const num = Number(prompt("Nº de Etiquetas: ")) // quantidade de etiquetas
// Cria um laço de repetição até num/2 (pois imprime 2 etiquetas por linha)
for (let i = 1; i <= num / 2; i++) {
  console.log(` ${produto.padEnd(30)} ${produto.padEnd(30)} `)
}
if (num % 2 == 1) { // se quantidade solicitada de etiquetas for ímpar...
  console.log(produto) // imprime mais uma etiqueta
}
```

O método `.padEnd(30)` aplicado à variável `produto` nesse exemplo serve para preencher com espaços em branco o nome do produto até o tamanho total atingir 30 posições. Pode ser acrescentado outro parâmetro, após o número, para indicar o caractere a ser utilizado no preenchimento. Por exemplo: `produto.padEnd(30, ".")`. Existe também o método `padStart()` que acrescenta espaços (ou algum outro caractere, se indicado) no início do texto.

c) *Elaborar um programa para uma loja que leia o valor de uma conta e o número de vezes que um cliente deseja parcelar esse valor (em boletos ou carnê). Para facilitar o troco, o lojista deseja que as parcelas iniciais não tenham centavos, ou seja, centavos apenas na última parcela. Informe como resposta o valor de cada parcela, considerando essa situação.*

Código JavaScript do programa Cálculo das Parcelas (nodejs/ex5_10.js)

```
const prompt = require("prompt-sync")()
const valor = Number(prompt("Valor R$: ")) // Lê valor do carnê ...
const num = Number(prompt("Nº de Parcelas: ")) // e número de parcelas
const valorParcelas = Math.floor(valor / num) // calcula valor sem decimais
const valorFinal = valorParcelas + (valor % num) // calcula parcela final
```

```
for (let i = 1; i < num; i++) {           // enquanto i < num
    console.log(`#${i}ª parcela: R$ ${valorParcelas.toFixed(2)}`)
}
console.log(`#${num}ª parcela: R$ ${valorFinal.toFixed(2)}) // última parcela
```

Observe, no código do programa de Exemplo 5.10, que inicialmente calculamos o valor das parcelas. Após, com o comando `for`, são exibidas todas as parcelas, exceto a última – que é apresentada no final do algoritmo. Na linha do prompt, a execução deste programa.

```
C:\livrojs\cap05\nodejs>node ex5_10
Valor R$: 91.50
Nº de Parcelas: 3
1ª parcela: R$ 30.00
2ª parcela: R$ 30.00
3ª parcela: R$ 31.50
```

Pensou em alguma lógica diferente para resolver os exemplos dessa seção? Não deixe de experimentá-la! O bacana da área de programação de computadores é que os resultados dos nossos testes saem “na hora”. E, caso alguma ideia ou comando não tenha resultado no esperado, esteja certo de que essa experiência vai ser útil para você nos próximos Algoritmos.

5.8 Exercícios

Construa os seguintes algoritmos utilizando as rotinas de repetição apresentadas neste capítulo. As ilustrações servem para facilitar a compreensão do enunciado e estão no modelo HTML e JavaScript, mas você pode optar por realizar os exercícios com o Node, visto que tanto a Lógica quanto as estruturas de programação JavaScript utilizadas são as mesmas.

a) Elaborar um programa que leia o nome de uma fruta e um número. O programa deve repetir a exibição do nome da fruta, de acordo com o número informado. Utilize o “” para separar os nomes. A Figura 5.15 ilustra a execução do programa.*

Programa Repete Fruta

Fruta: Abacate

Número: 5

Repetir Fruta

Abacate * Abacate * Abacate * Abacate * Abacate

Figura 5.15 – O nome da fruta deve ser repetido de acordo com o número informado.

b) Digamos que o número de chinchilas de uma fazenda triplica a cada ano, após o primeiro ano. Elaborar um programa que leia o número inicial de chinchilas e anos e informe ano a ano o número médio previsto de chinchilas da fazenda. O número inicial de chinchilas deve ser maior ou igual a 2 (um casal). A Figura 5.16 exibe a página com um exemplo de saída do programa.

Programa Criação de Chinchilas

Nº Chinchilas: 4

Nº Anos: 6

Mostrar Previsão de Chinchilas

1º Ano: 4 Chinchilas
2º Ano: 12 Chinchilas
3º Ano: 36 Chinchilas
4º Ano: 108 Chinchilas
5º Ano: 324 Chinchilas
6º Ano: 972 Chinchilas

Figura 5.16 – Após o 1º ano, o número de chinchilas deve triplicar a cada ano.

c) Elaborar um programa que leia um número e verifique se ele é ou não perfeito. Um número dito perfeito é aquele que é igual à soma dos seus divisores inteiros (exceto o próprio número). O programa deve exibir os divisores do número e a soma deles. A Figura 5.17 exibe a página do programa com um exemplo de número perfeito.

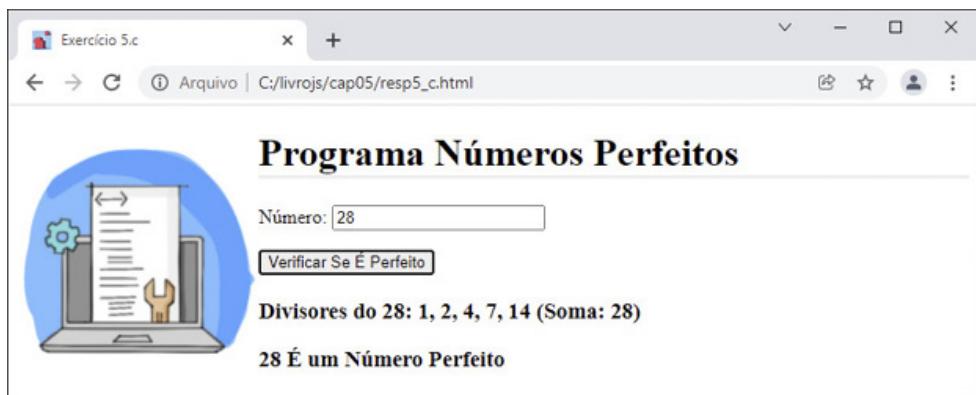


Figura 5.17 – Programa Números Perfeitos deve utilizar estruturas de repetição e acumuladores.

5.9 Considerações finais do capítulo

Os laços de repetição, criados com os comandos `for`, `while` e `do..while`, permitem fazer com que alguns comandos sejam executados várias vezes em um programa. Eles são muito úteis para permitir, por exemplo, que um sistema receba o pagamento de várias contas de um cliente ou que um layout com as poltronas disponíveis e ocupadas de um teatro seja montado na página. Nesses casos, os comandos que realizam essas tarefas ficam dentro de uma estrutura de repetição. Esses laços precisam prever um ponto de saída, ou seja, devem ser finitos.

O comando `for` contém uma variável de controle e sua sintaxe inclui três instruções: o valor inicial dessa variável de controle, a condição a ser verificada para que a repetição continue e o incremento ou decrecimento da variável a cada volta. Os valores em cada uma dessas instruções podem ser definidos com o uso de outras variáveis. O comando `for` é geralmente utilizado quando se sabe no início de sua composição o número de vezes em que esse laço será executado. O exemplo da montagem do layout pode ser construído com o comando `for`, já que o número de poltronas do teatro deve ser inicialmente conhecido pelo programador. Esse número de repetições também pode ser solicitado pelo sistema antes do uso do `for`, como solicitar o número de parcelas de um financiamento para, então, exibir datas e valores de cada parcela.

Os comandos `while` e `do.. while` também permitem criar estruturas de repetição. A diferença entre eles é que no `while` a condição é testada no início do laço, enquanto no `do.. while` a condição é testada no final. Isso significa que os comandos pertencentes ao `while` podem não ser executados – se a condição retornar falso na sua primeira verificação. Já com o `do.. while`, temos a garantia de que, no mínimo, uma vez os comandos do laço serão executados – pois o teste condicional é realizado ao final do laço. Essas estruturas, por sua vez, são geralmente utilizadas quando não sabemos previamente quantas vezes os comandos da estrutura serão executados, como no exemplo do pagamento de contas.

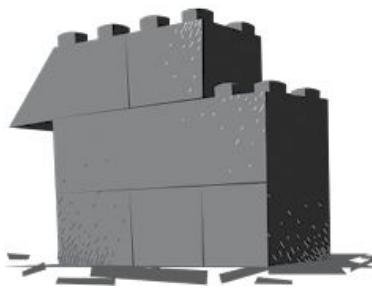
Existem dois comandos especiais que podem ser utilizados nos laços de repetição: `break` e `continue`. O `break` abandona o laço (executando o comando após o laço, se houver). Já o `continue` retorna ao início do laço (gerando um incremento ou decremento da variável de controle no comando `for`). Esses comandos conferem ao programador maior controle do fluxo de execução dos comandos na repetição.

Em alguns programas, é necessário realizar a contagem ou a soma de valores manipulados na estrutura de repetição. Para isso, devemos utilizar os contadores e acumuladores, que possuem duas características: devem ser inicializados (geralmente com 0) e receber na atribuição (dentro do laço) eles mesmos mais algum valor. Nos contadores, o incremento é uma constante e, nos acumuladores, uma variável.

Um importante auxílio aos programadores é o uso do debug. Ele permite identificar erros de sintaxe e de lógica em um programa. Os erros de sintaxe ocorrem quando inserimos um comando ou nome de variável incorreto no código, já os erros de lógica são mais difíceis de serem identificados, pois o programa funciona, porém não exibe o resultado esperado. Para isso, o processo de depuração dispõe de recursos como o uso dos breakpoints (pontos de parada) e da janela watch (observador) para análise e identificação dos problemas.

CAPÍTULO 6

Vetores



Os vetores ou arrays são estruturas que permitem armazenar uma lista de dados na memória principal do computador. Eles são úteis para inserir ou remover itens de uma lista de compras ou de alunos de uma turma, por exemplo. Com os vetores é possível recuperar todos os itens inseridos na lista. Um índice numérico (que começa em 0) identifica cada elemento da lista. A representação ilustrada na Tabela 6.1 contém uma lista de itens de um supermercado armazenada no vetor produtos.

Tabela 6.1 – Representação dos itens/elementos de um vetor

produtos	
0	Arroz
1	Feijão
2	Iogurte
3	Leite
4	Suco
5	Pão

Para referenciar um item do vetor, devemos indicar seu nome, seguido por um número entre colchetes que aponta para o seu índice. É importante reforçar que o vetor inicia pelo índice 0. Portanto, para obter o primeiro produto inserido no vetor, devemos utilizar: `produtos[0]`. Também poderíamos

alterar um produto da lista, com uma nova atribuição de conteúdo a um elemento do vetor, como:

```
produtos[2] = "Queijo"
```

Na linguagem JavaScript, não é necessário indicar o número total de elementos do vetor na sua declaração. Para declarar um vetor em JavaScript, devemos utilizar uma das seguintes formas:

```
const produtos = []
const produtos = new array()
```

Prefira utilizar a primeira forma, que é a recomendada. Também é possível declarar um vetor com algum conteúdo inicial (e, mesmo assim, adicionar ou remover itens no vetor no decorrer do programa). A instrução a seguir declara e insere três produtos no vetor:

```
const produtos = ["Arroz", "Feijão", "Iogurte"]
```

Vetores podem ser declarados com `const` e mesmo assim ter o valor dos seus elementos alterados. O que não pode ser feito com o `const` é uma reatribuição de valor a uma variável. Isso evita possíveis erros que poderiam ocorrer com o `var`, principalmente em programas maiores.

Talvez você esteja se perguntando: no Exemplo 5.5, nós inserimos uma lista de contas do mês: Como fizemos isso sem o uso dos vetores? Sim, é possível exibir os dados informados pelo usuário em um programa concatenando os conteúdos a serem exibidos em uma string. Contudo, não é possível (ou é muito mais complicado) gerenciar esses dados. A diferença entre o uso de variáveis e vetores é indicada a seguir:

- Uma variável armazena apenas um valor por vez; quando uma nova atribuição a essa variável é realizada, o seu valor anterior é perdido. Após as duas atribuições a seguir, a variável `idade` permanece apenas com o último valor que lhe foi atribuído.

```
let idade  
idade = 18  
idade = 15
```

- Já os vetores permitem armazenar um conjunto de dados e acessar todos os seus elementos pela referência ao índice que identifica cada um deles. Assim, após as duas atribuições a seguir, os dois valores atribuídos ao vetor `idade` podem ser acessados.

```

const idade = []
idade[0] = 18
idade[1] = 15

```

No Exemplo 6.1, destacado a seguir, vamos gerenciar uma lista de pacientes de um consultório odontológico. Nesse exemplo, evidenciam-se as vantagens do uso dos vetores. No entanto, antes de codificarmos esse programa, vamos ver os métodos JavaScript disponíveis para inclusão e exclusão de dados em vetores e também formas de exibir os elementos do vetor.

6.1 Inclusão e exclusão de itens

Depois de realizarmos a declaração do vetor, podemos gerenciar a lista com a inclusão e a exclusão de itens a esse vetor. Os principais métodos JavaScript que executam essas tarefas estão indicados na Tabela 6.2.

Tabela 6.2 – Métodos de inclusão e exclusão de itens em vetores

push()	Adiciona um elemento ao final do vetor.
unshift()	Adiciona um elemento ao início do vetor e desloca os elementos existentes uma posição abaixo.
pop()	Remove o último elemento do vetor.
shift()	Remove o primeiro elemento do vetor e desloca os elementos existentes uma posição acima.

Observe a partir das linhas de comentário o que acontece com o conteúdo do vetor cidades no script a seguir, após a execução dos métodos da Tabela 6.2.

```

<script>
  const cidades = ["Pelotas"] // declara e define conteúdo inicial do vetor

  cidades.push("São Lourenço") // adiciona cidade ao final do vetor
  console.log(cidades)      // ['Pelotas', 'São Lourenço']

  cidades.unshift("Porto Alegre") // adiciona ao início e desloca as demais
  console.log(cidades) // ['Porto Alegre', 'Pelotas', 'São Lourenço']

  const ultima = cidades.pop() // remove a última cidade do vetor
  console.log(cidades)      // ['Porto Alegre', 'Pelotas']

```

```
const primeira = cidades.shift() // remove a primeira e "sobe" as demais
console.log(cidades)          // ['Pelotas']
</script>
```

Note, no código anterior, que as operações de exclusão de itens são atribuídas às variáveis `ultima` e `primeira`. Elas recebem o elemento removido do vetor.

Existem outros métodos que podem ser utilizados para manipular os elementos de um vetor JavaScript. O método `splice` (na ideia de emendar) pode possuir diversos parâmetros e ser utilizado para alterar, inserir ou remover elementos do array. Já o método `slice` (na ideia de fatiar) obtém uma “fazia” de um vetor. Ele contém dois parâmetros que são posição inicial e final (que não é obrigatória) do array. Se posição inicial for um número negativo, ela indica a quantidade de elementos do final para o início que serão obtidos. Se posição final for um número negativo, ela indica a quantidade de elementos do fim para o início que devem ser descartados. Observe os exemplos:

```
<script>
const letras = ["A", "B", "C", "D"] // declara e define conteúdo inicial do vetor
const letras2 = letras.slice(-2)    // obtém 2 últimas letras
const letras3 = letras.slice(0, -1) // obtém do início até final, exceto a última
console.log(letras)              // ['A', 'B', 'C', 'D']
console.log(letras2)             // ['C', 'D']
console.log(letras3)             // ['A', 'B', 'C']

const retira = letras.splice(2, 1) // remove a partir da posição 2, 1 elemento
console.log(letras)              // ['A', 'B', 'D']
console.log(retira)              // ['C']
</script>
```

Detalhes: `slice()` não modifica o conteúdo do vetor original, enquanto `splice()` modifica. A variável `retira` é sempre um array, mesmo com um ou zero elementos.

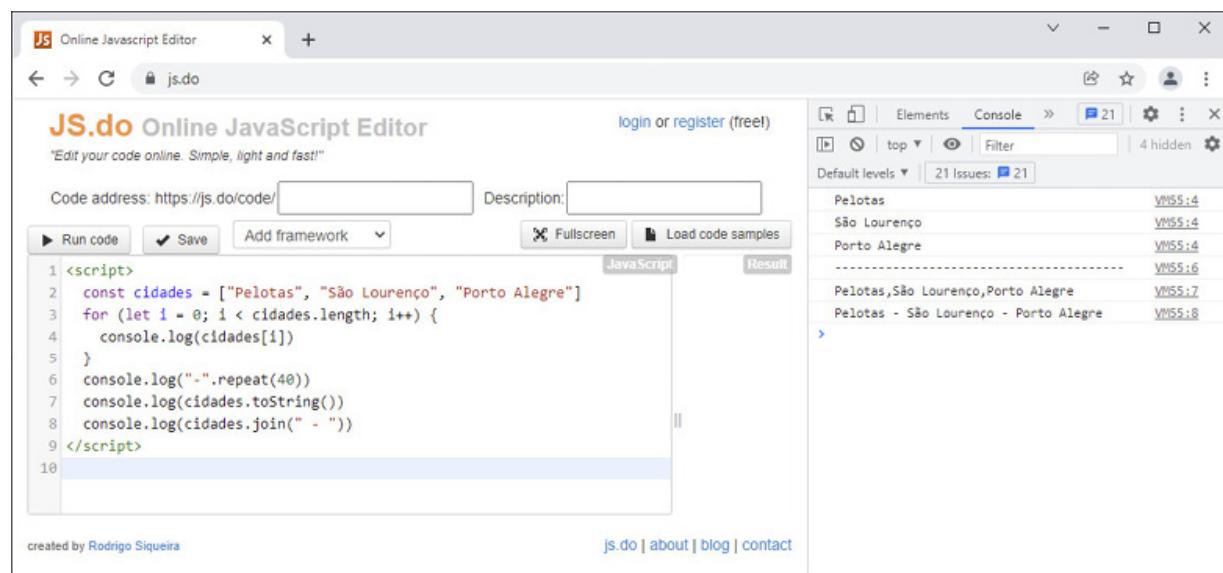
6.2 Tamanho do vetor e exibição dos itens

Uma propriedade importante utilizada na manipulação de vetores é a

propriedade `length`, que retorna o número de elementos do vetor. Vamos consultá-la quando quisermos percorrer a lista, realizar exclusões (para verificar antes da exclusão, se a lista está vazia) ou, então, para exibir o número total de itens do vetor. Para percorrer e exibir os elementos do vetor `cidades`, podemos utilizar o comando `for`, indicando que a variável de controle `i` começa em 0, e repetir o laço enquanto `i` for menor que `cidades.length`.

Outra forma de exibir o conteúdo do vetor é pelo uso dos métodos `toString()` e `join()`. Eles convertem o conteúdo do vetor em uma string, sendo que no método `toString()` uma vírgula é inserida entre os elementos e no `join()` podemos indicar qual caractere vai separar os itens. Vamos utilizar o site `js.do` para exemplificar o funcionamento desses métodos. Após acessar o site, você pode clicar em **Create new code**, digitar o código JavaScript e clicar em **Run**. Na Figura 6.1, esses métodos são exemplificados, precedidos pela instrução `for`, que percorre e exibe todos os elementos do vetor individualmente.

No código da Figura 6.1 foi utilizado também o método `"-".repeat(num)`, que faz com que um ou mais caracteres sejam repetidos `num` vezes. Já utilizamos os métodos `.toFixed(num)`, `.padEnd(num)` e agora `.repeat(num)`. Esses métodos simplificam operações realizadas sobre os dados do programa.



The screenshot shows the JS.do Online JavaScript Editor interface. On the left, the code editor contains the following JavaScript:

6.3 For..of e forEach()

Para percorrer os elementos de um vetor, a linguagem JavaScript dispõe também do loop `for.. of` e do método `forEach()`. Eles são equivalentes ao `for` tradicional, com uma sintaxe mais “enxuta”. Observe, nos exemplos a seguir, a forma de utilizar esses comandos para apresentar o conteúdo do array `cidades`. Começamos pelo `for.. of`.

```
for (const cidade of cidades) {  
    console.log(cidade)  
}
```

A cada interação, a variável `cidade` recebe um elemento do vetor `cidades`. Condições ou cálculos podem ser realizados dentro do loop com essa variável. E, como `cidade` é uma variável de bloco (deixa de existir após cada interação), podemos declará-la com `const`. Assim, a cada interação a variável não é modificada, mas, sim, deixa de existir e é novamente declarada.

Já o método `forEach()` é mais amplo e pode chamar uma função para manipular cada elemento do vetor. Uma forma simples de utilizá-lo para percorrer os elementos de um vetor é exemplificada a seguir:

```
cidades.forEach((cidade, i) => {  
    console.log(`#${i+1}ª Cidade: ${cidade}`)  
})
```

Observe que no `forEach()` podemos obter o conteúdo (`cidade`) e o índice (`i`) de cada elemento do vetor que estamos percorrendo – sendo que o índice (`i`) é opcional. A saída desse trecho de código (considerando o vetor `cidades` declarado na Figura 6.1) é a seguinte:

1ª Cidade: Pelotas
2ª Cidade: São Lourenço
3ª Cidade: Porto Alegre

Algumas operações sobre vetores ficam mais simples se executadas a partir do `forEach()`, como a soma dos elementos que compõem o vetor. Observe o código a seguir:

```
const numeros = [5, 10, 15, 20]  
let soma = 0  
numeros.forEach(num => soma += num)  
console.log(`Soma dos Números: ${soma}`)
```

Cada elemento do vetor `numeros` é recebido como `num` e então o acumulador `soma` é incrementado. Um cuidado com o método `forEach()` é que ele não permite interrupções no laço, ou seja, não podemos utilizar os comandos `break` e `continue` dentro dele. Dadas as considerações sobre cada uma das formas de exibir o conteúdo de um vetor, vamos criar um exemplo para ilustrar a aplicação dos métodos de inclusão, exclusão e listagem de dados de um vetor. Nosso programa deve controlar a lista de atendimentos dos pacientes de um consultório odontológico – como se fosse um painel em exposição em uma tv do consultório. A Figura 6.2 ilustra a página exibida pelo programa em execução.



Figura 6.2 – Um vetor gerencia a ordem dos atendimentos.

Como nos capítulos anteriores, crie uma nova pasta `cap06`, e nela as pastas `css`, `img` e `js`. O arquivo contendo os estilos das tags dos exemplos deste capítulo (`estilos.css`, da pasta `css`) deve ser criado com o seguinte conteúdo:

```
img.normal { float: left; height: 300px; width: 300px; }
img.alta { float: left; height: 420px; width: 300px; }
h1 { border-bottom-style: inset; }
pre { font-size: 1.2em; }
.fonte-azul { color: blue; }
.oculta { display: none; }
.exibe { display: inline; }
.italic { font-style: italic; }
```

A classe `fonte-azul` será utilizada nesse primeiro exemplo para destacar os pacientes em atendimento. Vamos, então, criar a página HTML do Exemplo

6.1, descrita a seguir:

Exemplo 6.1 – Código HTML do programa Consultório Odontológico (ex6_1.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Consultório Odontológico</h1>
<form>
  <p>Paciente:
    <input type="text" id="inPaciente" required autofocus>
    <input type="submit" value="Adicionar">
    <input type="button" value="Urgência" id="btUrgencia">
    <input type="button" value="Atender" id="btAtender">
  </p>
</form>
<h3>Em Atendimento:
  <span class="fonte-azul"></span>
</h3>
<pre></pre>
<script src="js/ex6_1.js"></script>
<!-- /body e /html -->
```

Nosso código HTML apresenta algumas “novidades” em relação aos exemplos dos capítulos anteriores. O campo de entrada de dados contém o atributo autofocus. Com esse atributo, define-se o campo do formulário em que o cursor ficará inicialmente posicionado. Também foi adicionada a tag span, dentro do h3. Com a tag span, pode-se identificar um local da página, que terá o seu conteúdo alterado pelo programa JavaScript, sem que ocorra uma quebra de linha. Portanto, o nome do paciente será exibido na tag h3 ao lado do texto "Em Atendimento: ". Além disso, foram acrescentados 2 botões no formulário, em acréscimo ao submit do form. O evento click será o responsável por disparar a programação associada a eles.

Crie o arquivo ex6_1.js dentro da pasta js e adicione as linhas de código a seguir. Você pode testar o funcionamento de cada botão em partes. É melhor, pois a quantidade de código a ser analisada (e passível de algum erro de sintaxe) é, consequentemente, menor. Essa é uma boa prática de ser adotada quando estivermos iniciando o aprendizado de uma nova linguagem de programação.

Código JavaScript do programa Consultório Odontológico (js/ex6_1.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const respNome = document.querySelector("span")
const respLista = document.querySelector("pre")

const pacientes = [] // declara vetor global

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  const nome = frm.inPaciente.value // obtém nome do paciente
  pacientes.push(nome) // adiciona o nome no final do vetor
  let lista = "" // string para concatenar pacientes
  // for "tradicional" (inicia em 0, enquanto menor que tamanho do array)
  for (let i = 0; i < pacientes.length; i++) {
    lista += `${i + 1}. ${pacientes[i]}\n`
  }
  respLista.innerText = lista // exibe a lista de pacientes na página
  frm.inPaciente.value = "" // limpa conteúdo do campo de formulário
  frm.inPaciente.focus() // posiciona o cursor no campo
})

// Adiciona um "ouvinte" para o evento click no btUrgencia que está no form
frm.btUrgencia.addEventListener("click", () => {
  // verifica se as validações do form estão ok (no caso, paciente is required)
  if (!frm.checkValidity()) {
    alert("Informe o nome do paciente a ser atendido em caráter de urgência")
    frm.inPaciente.focus() // posiciona o cursor no campo inPaciente
    return // retorna ao form
  }
  const nome = frm.inPaciente.value // obtém nome do paciente
  pacientes.unshift(nome) // adiciona paciente no início do vetor
  let lista = "" // string para concatenar pacientes
  // forEach aplicado sobre o array pacientes
  pacientes.forEach((paciente, i) => (lista += `${i + 1}. ${paciente}\n`))
  respLista.innerText = lista // exibe a lista de pacientes na página
  frm.inPaciente.value = "" // limpa conteúdo do campo de formulário
  frm.inPaciente.focus() // posiciona o cursor no campo
})

frm.btAtender.addEventListener("click", () => {
  // se o tamanho do vetor = 0
```

```

if (pacientes.length == 0) {
    alert("Não há pacientes na lista de espera")
    frm.inPaciente.focus()
    return
}
const atender = pacientes.shift() // remove do início da fila (e obtém nome)
respNome.innerText = atender // exibe o nome do paciente em atendimento
let lista = "" // string para concatenar pacientes
pacientes.forEach((paciente, i) => (lista += `${i + 1}. ${paciente}\n`))
respLista.innerText = lista // exibe a lista de pacientes na página
})

```

No início do programa JavaScript, criamos referência aos elementos a serem manipulados pelo programa e declaramos um vetor de escopo global, que será utilizado pelas três funções do programa.

Na função associada ao botão submit do form, é necessário obter o nome do paciente e utilizar o método `push()` – que insere o nome no final do vetor `pacientes`. Em seguida, o comando `for` percorre todos os itens do vetor, exibindo a lista dos pacientes na fila de espera. Observe que, dentro do laço `for`, o valor da variável `i` é adicionado a 1, pois ficaria estranho numerar a lista considerando a posição real dos elementos do vetor – que começa por 0.

Já a programação associada ao clique no botão **Urgência** inicia por uma condição: `if (!frm.checkValidity())`, ou então `if (frm.checkValidity() == false)`. Lembrem que a exclamação (!) significa `not`. As regras de validação adicionadas aos campos do form, como `required`, `min` e `max`, são avaliadas apenas quando o botão `submit` é clicado. Então, quando utilizarmos um `button` e quisermos validar o `form`, devemos acrescentar essa condição. Após o teste condicional, deve-se inserir o nome do paciente no início da fila. Portanto, o método `unshift(nome)` é utilizado.

Para apresentar a lista dos pacientes em espera, no botão **Urgência** e **Atender**, recorremos ao método `forEach()` – que é uma forma mais “elegante” de percorrer um vetor quando necessitamos obter o conteúdo e o índice do vetor. Quando necessitamos apenas acessar o conteúdo do vetor, o laço `for..of` é uma boa alternativa. Mas fique a vontade para utilizar a forma com a qual você se sentir mais confortável.

Observe que podemos definir um ouvinte de evento para os botões que pertencem ao form a partir do id de cada um deles, de forma semelhante ao que é feito para referenciar um campo do form.

Falta comentarmos sobre a programação associada ao evento de clique no botão **Atender**. Observe que ela começa pela verificação do número de elementos do vetor, pois, se o vetor estiver vazio, não há atendimentos para realizar. Em seguida, executa-se o método `shift()` que retira o primeiro paciente da lista de espera e o armazena na variável `atender`, exibida na sequência. No final, novamente é necessário apresentar a lista dos pacientes, agora sem o paciente removido.

6.4 Localizar conteúdo

Como o número de elementos de um vetor pode ser grande, as linguagens de programação dispõem de alguns métodos para nos auxiliar no controle de seu conteúdo. Um desses controles refere-se à verificação da existência ou não de um conteúdo do vetor. Os métodos `indexOf()`, `lastIndexOf()` e `includes()` cumprem esse papel.

No `indexOf()`, a busca ocorre a partir do início do vetor. Já no `lastIndexOf()` a busca é do final até o seu início. Caso o conteúdo exista no vetor, o número do índice da primeira ocorrência desse conteúdo é retornado. Caso o conteúdo pesquisado não exista no vetor, o valor -1 é devolvido. O método `includes()`, por sua vez, retorna verdadeiro ou falso, de acordo com a existência ou não do conteúdo no vetor. Observe o exemplo a seguir, contendo um vetor com a idade de crianças, que ilustra o funcionamento desses métodos.

```
<script>
const idades = [5, 6, 8, 3, 6, 9]
console.log(idades.indexOf(6))    // retorna 1
console.log(idades.lastIndexOf(6)) // retorna 4
console.log(idades.indexOf(7))    // retorna -1
console.log(idades.includes(3))   // retorna true
</script>
```

Lembre-se de que o vetor começa na posição 0. Então, o “desenho” dos

seus elementos é a representação ilustrada na Tabela 6.3.

O método `indexOf()` exibido no primeiro `alert()` desse script verifica a existência do número 6 no vetor. Ele retorna a posição da primeira ocorrência desse conteúdo (e não o número de vezes em que ele ocorre no vetor). Portanto, a posição 1 é retornada. Já o método `lastIndexOf()` age da mesma forma, contudo a procura ocorre do final em direção ao início do vetor. Dessa forma, a busca pelo valor 6, no `lastIndexOf()`, retorna a posição 4. E, caso um valor inexistente seja passado, o retorno será -1. Já o `includes()` retorna `true` para um valor existente.

Tabela 6.3 – Posição dos elementos no vetor idades

Idades	
0	5
1	6
2	8
3	3
4	6
5	9

Vamos construir um segundo exemplo, em que o método `includes()` é utilizado. Trata-se do “Jogo Descubra o Número” ilustrado na Figura 6.3.



Figura 6.3 – Programa Descubra o Número gera um número aleatório a cada novo jogo.

O programa utiliza a função matemática `Math.random()` discutida na Seção 2.8, para gerar um número aleatório entre 1 e 100 que deve ser descoberto pelo

usuário. Para evitar que o jogador aposte um número 2x (e perca uma chance), faz-se uso do método `includes()`. O código HTML desse jogo exibe um campo de entrada e três linhas para a exibição de mensagens: o número de erros e o vetor erros, o número de chances ainda disponíveis para o jogador e as dicas para auxiliar na descoberta do número.

Exemplo 6.2 – Código HTML do Programa Descubra o Número (ex6_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Jogo Descubra o Número</h1>
<form>
  <p>Número:</p>
  <input type="number" min="1" max="100" id="inNumero" required autofocus>
  <input type="submit" value="Apostar" id="btSubmit">
  <input type="button" value="Jogar Novamente" id="btNovo" class="oculta">
</p>
</form>
<h3>Erros: <span id="outErros">0</span></h3>
<h3>Chances: <span id="outChances">6</span></h3>
<h3 id="outDica" class="italico">Dica: É um número entre 1 e 100</h3>
<script src="js/ex6_2.js"></script>
<!-- /body e /html -->
```

Código JavaScript do Programa Descubra o Número (js/ex6_2.js)

```
const frm = document.querySelector("form")      // obtém elementos da página
const respErros = document.querySelector("#outErros")
const respChances = document.querySelector("#outChances")
const respDica = document.querySelector("#outDica")

const erros = [] // vetor de escopo global com números já apostados
const sorteado = Math.floor(Math.random() * 100) + 1 // num aleatório entre 1 e 100
const CHANCES = 6 // constante com o número máximo de chances

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form
  const numero = Number(frm.inNumero.value) // obtém número digitado
  if (numero == sorteado) { // se acertou
    respDica.innerText = `Parabéns!! Número sorteado: ${sorteado}`
```

```

frm.btSubmit.disabled = true // troca status dos botões
frm.btNovo.className = "exibe"
} else {
  if (erros.includes(numero)) { // se número existe no vetor erros (já apostou)
    alert(`Você já apostou o número ${numero}. Tente outro...`)
  } else {
    erros.push(numero) // adiciona número ao vetor
    const numErros = erros.length // obtém tamanho do vetor
    const numChances = CHANCES - numErros // calcula nº de chances
    // exibe nº de erros, conteúdo do vetor e nº de chances
    respErros.innerText = `${numErros} (${erros.join(", ")})`
    respChances.innerText = numChances
    if (numChances == 0) {
      alert("Suas chances acabaram...")
      frm.btSubmit.disabled = true
      frm.btNovo.className = "exibe"
      respDica.innerText = `Game Over!! Número Sorteado: ${sorteado}`
    } else {
      // usa operador ternário para mensagem da dica
      const dica = numero < sorteado ? "maior" : "menor"
      respDica.innerText = `Dica: Tente um número ${dica} que ${numero}`
    }
  }
}
frm.inNumero.value = "" // limpa campo de entrada
frm.inNumero.focus() // posiciona cursor neste campo
})

```

Vamos às explicações sobre os comandos utilizados nesse jogo. No início, são declarados um vetor para armazenar os números apostados, uma variável contendo o número aleatório a ser descoberto e uma constante com o número de chances do jogador. Mas por que os declarar com escopo global e não dentro da função (escopo local)? Relembrando... porque as variáveis de escopo global permanecem na memória principal do computador enquanto a página está carregada, e as variáveis locais, somente enquanto a função está em execução. Assim, se a variável sorteado fosse declarada dentro de uma função, um novo sorteio ocorreria a cada aposta de número feita pelo jogador.

Lembre-se, contudo, de que as variáveis globais devem ser utilizadas apenas quando realmente são necessárias, para evitar o consumo

desnecessário de memória do sistema.

Na programação associada ao evento submit do form são realizados vários testes. O primeiro verifica se o jogador acertou o número. Caso ok, uma mensagem de parabéns é exibida e os botões trocam de status. Senão, deve-se utilizar o método includes() para verificar se o número apostado já consta no vetor erros. Caso verdade, a mensagem de alerta é exibida – conforme ilustra a Figura 6.4.

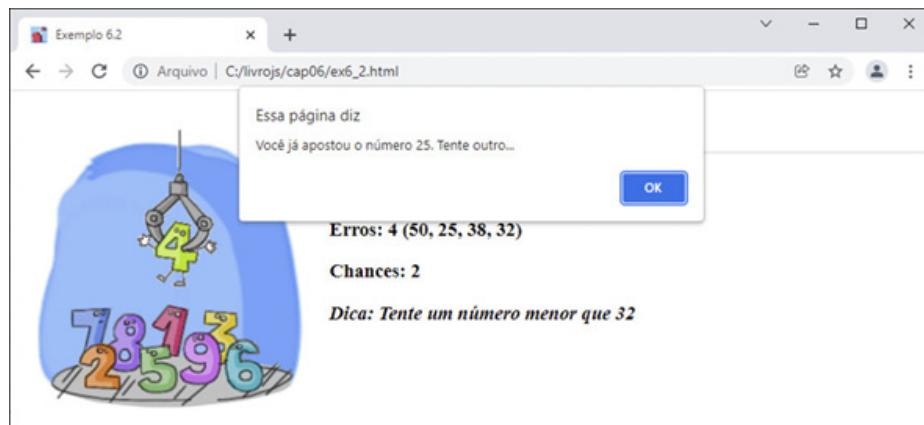


Figura 6.4 – O método includes() verifica se um número já consta no vetor erros.

As ações seguintes servem para adicionar o número ao vetor erros, exibir as mensagens na página e verificar o número de chances que o jogador ainda possui. As trocas de status dos botões impedem que o jogador continue apostando outros números, mesmo com as chances zeradas. Apenas o botão “Jogar Novamente” permanece disponível para o usuário – conforme ilustra a Figura 6.5.



Figura 6.5 – Troca de status dos botões impedem novas apostas.

Por fim, o click no botão **Jogar Novamente** contém uma chamada ao método location.reload() que recarrega a página. Dessa forma, um novo número é sorteado, o vetor é zerado e a dica inicial é exibida.

```
frm.btNovo.addEventListener("click", () => {
    location.reload() // recarrega a página
})
```

Observação: você acha que é muito difícil acertar um número aleatório entre 1 e 100, com apenas 6 chances? Da forma como a dica do programa é exibida, essa tarefa se torna possível se você utilizar a técnica da pesquisa binária. Descrevendo de forma resumida, a pesquisa binária consiste em referenciar sempre o elemento central de uma lista ordenada e descartar a parte que não contém o elemento pesquisado. Assim, sua primeira aposta deve ser o número 50. Se a dica sugerir um número maior, você tenta 75, e assim sucessivamente. Você vai acertar várias vezes!

6.5 Vetores de objetos

Um vetor pode conter uma lista de nomes, como no Exemplo 6.1, ou de números, como no Exemplo 6.2. Além disso, também é possível definir um vetor que contenha uma lista de objetos, com alguns atributos desse objeto. Poderíamos, por exemplo, ter o objeto produto, com os atributos nome, marca e preço. Ou o objeto filme, com os atributos título, gênero e duração.

Definir um vetor de objetos nos permite realizar operações sobre esse vetor, como classificar os seus elementos por um dos seus atributos.

Um vetor de objetos é declarado da mesma forma que um vetor simples. Na inserção de itens no vetor, contudo, devem-se indicar os atributos que o compõem. Observe o script a seguir, que manipula o vetor de objetos carros, com os atributos modelo e preço.

```
<script>
const carros = []
carros.push({modelo: "Sandero", preco: 46500})
carros.push({modelo: "Palio", preco: 37800})
for (const carro of carros) {
```

```
        console.log(`#${carro.modelo} - R$: ${carro.preco}`)
    }
</script>
```

Nesse script, o vetor é inicialmente declarado. Em seguida, são realizadas duas inclusões de veículos. Atente para a sintaxe que identifica um vetor de objetos: deve-se utilizar as chaves {} para delimitar os atributos e cada atributo deve ser seguido pelos “:” e pelo conteúdo que será atribuído a ele. Na sequência do script, utilizamos o comando for.. of para percorrer os elementos do vetor e apresentar o conteúdo de cada um dos seus atributos.

Um detalhe na atribuição de dados de um objeto em JavaScript é que, se o nome da variável for igual ao do atributo, pode-se omitir a atribuição. Observe o exemplo:

```
const carros = []
const modelo = "Fiesta"
const preco = 46800
carros.push({modelo, preco}) // ou carros.push({modelo: modelo, preco: preco})
```

6.6 Desestruturação e operador Rest/Spread

Um dos acréscimos das novas versões do JavaScript é a possibilidade de atribuir valores às variáveis via desestruturação dos elementos de vetores ou objetos. Podemos começar a demonstração do uso desse recurso no exemplo da seção anterior, onde foi utilizado o vetor de objetos carros, com os atributos modelo e preco. Na rotina de apresentação dos dados, pode-se desestruturar o objeto, conforme o exemplo a seguir:

```
for (const carro of carros) {
    const {modelo, preco} = carro // "desestrutura" objeto carro em modelo e preco
    console.log(`#${modelo} - R$: ${preco}`)
}
```

Os atributos modelo e preco, do objeto carro, são obtidos a partir de uma única atribuição. Como se refere a um objeto, os nomes dos atributos devem estar delimitados por {}. Nos exemplos de obtenção de dados de um Web Service, como os demonstrados nos capítulos finais do livro, em que um objeto contém vários atributos, é uma boa prática explorar essa técnica.

A desestruturação também pode ocorrer para obter os elementos de um

array, como no exemplo a seguir:

```
const pacientes = ["Ana", "Carlos", "Sofia"]
const [a, b, c] = pacientes
console.log(a) // Ana
console.log(b) // Carlos
console.log(c) // Sofia
```

Caso o array pacientes contenha apenas dois nomes, const c ficaria como undefined. Caso tenha mais nomes, apenas os três primeiros seriam obtidos. Também se pode desestruturar os elementos de um vetor com uma parte dele sendo atribuída a variáveis e outra parte a um outro vetor. Para isso, deve-se utilizar o operador Rest (...) que cria um novo vetor com os elementos “restantes”. Observe o exemplo a seguir:

```
const pacientes = ["Ana", "Carlos", "João", "Sofia"]
const [atender, proximo, ...outros] = pacientes
console.log(atender) // Ana
console.log(proximo) // Carlos
console.log(outros) // ["João", "Sofia"]
```

Caso o array pacientes contenha apenas um nome, ele é atribuído à variável atender, proximo fica como undefined e o array outros fica vazio. E, caso pacientes esteja vazio, as variáveis ficam undefined e outros = []. Na desestruturação dos elementos de um array, o operator Rest deve ser o último da lista de variáveis, justamente pelo fato de ele receber todos os demais elementos não referenciados pelas variáveis.

Os “...” também podem ser utilizados com uma ideia de “espalhar” os elementos de um array ou objeto – neste caso, recebendo a denominação de operador Spread. Observe a sua aplicação sobre um objeto:

```
const carro = { modelo: "Corsa", preco: 59500 }
const carro2 = { ...carro, ano: 2020 }
console.log(carro2) // {modelo: "Corsa", preco: 59500, ano: 2020}
```

E em aplicações sobre vetores, oferecendo uma forma alternativa para realizar inclusões de elementos no array.

```
let pacientes = ["João", "Sofia"]
pacientes = ["Ana", ...pacientes] // acrescenta "Ana" no início do vetor
pacientes = [...pacientes, "Maria"] // acrescenta "Maria" no final
```

Para esse último exemplo, tenha o cuidado de declarar pacientes com let, pois

ao declarar um array com `const` é possível realizar alterações em seus elementos a partir de métodos como `push()` ou `pop()`, mas não fazer uma reatribuição de valor a ele.

Portanto, os “...” podem servir para “espalhar” os elementos de um array ou objeto (Spread), ou então “juntar” elementos criando um novo array (Rest). No Capítulo 8, voltaremos a discutir sobre o operador Rest na passagem de parâmetros para uma função. O operador Spread também pode ser utilizado para criar uma cópia com os elementos de um vetor e, dessa forma, tem um comportamento semelhante ao método `slice()` – sem parâmetros, discutido anteriormente.

```
const pacientes2 = [...pacientes] // ou const pacientes2 = pacientes.slice()
```

6.7 Pesquisar e filtrar dados

Depois de possuirmos um conjunto de dados armazenados em uma lista, podemos exercitar algumas operações frequentemente realizadas sobre as listas, como pesquisa ou filtro dos dados. Vamos explorar nesta seção como percorrer os elementos do vetor para extrair algumas informações sobre o seu conteúdo.

São exemplos de filtros em um conjunto de dados a obtenção do nome e da nota dos alunos aprovados em uma prova, dos clientes com saldo negativo em uma agência bancária ou, então, das contas em atraso de uma empresa.

Nos programas que realizam essas operações um cuidado extra é necessário: o de informar ao usuário quando uma pesquisa não encontrou dados. Imagine que você está consultando uma página de uma livraria e há um campo para pesquisar pelo título do livro. Você escreve uma palavra-chave e fica aguardando a lista dos livros. Caso o sistema não localize livros com a palavra informada, é necessário retornar essa informação para o usuário. Caso contrário, ele ficará na espera, sem saber se a pesquisa foi concluída ou não.

Observe o trecho do script a seguir. Ele apresenta as idades que possuem valor maior ou igual a 18 armazenadas no vetor.

```
<script>
```

```

const idades = [12, 20, 15, 17, 14]
for (const idade of idades) {
  if (idade >= 18) {
    console.log(idade)
  }
}
</script>

```

Como fazer esse script apresentar uma mensagem indicando que não há idades maiores que 18 na lista? Olhando rapidamente o código, poderíamos pensar em colocar um `else` dentro do laço e exibir a mensagem. Contudo, isso faria com que a mensagem fosse exibida várias vezes, pois o comando `for` repete os comandos que estão dentro do laço várias vezes. Esse é um problema que percebo com frequência nas aulas, quando o tema é abordado.

A solução para esse cenário é utilizar uma variável de controle, como discutido no capítulo anterior (exemplo dos números primos). Essa variável, também denominada `flag` ou `sinalizador`, recebe um valor inicial antes da repetição. Caso a condição dentro do laço seja verdadeira, modifica-se o valor da variável. Após o laço, deve-se verificar, então, se a variável mantém o valor inicial. Isso significa que a condição testada no laço não ocorreu e que, portanto, a mensagem indicativa deve ser exibida. Observe o trecho de código a seguir, com o acréscimo dessa verificação.

```

<script>
  const idades = [12, 16, 15, 17, 14]
  let maiores = false
  for (const idade of idades) {
    if (idade >= 18) {
      console.log(idade)
      maiores = true
    }
  }
  if (!maiores) {
    console.log("Não há idades maiores que 18 na lista")
  }
</script>

```

Também poderíamos atribuir 0 (valor inicial) e 1 (dentro do laço) para a variável `maiores`. Entendida a forma de resolver algoritmos de pesquisas de dados em vetores a partir de repetições e condições? Vamos então ver como

esse processo pode ser simplificado com recursos adicionados nas novas versões do JavaScript, discutidos na seção a seguir.

6.8 Map, Filter e Reduce

Map, filter e reduce são métodos que permitem que operações sobre vetores sejam realizadas de um modo mais eficiente. Vamos apresentar alguns exemplos, para facilitar o entendimento.

Observação: Será empregada a notação das Arrow Functions (funções de seta), que, como veremos no capítulo sobre funções, possuem uma sintaxe mais curta, sem o uso do return em casos de funções com uma única linha.

Começamos com um exemplo sobre o método `map()`:

```
<script>
  const numeros = [10, 13, 20, 8, 15]      // vetor inicial
  constdobros = numeros.map(num => num * 2) // cada número é obtido e multiplicado
                                              // por 2, criando um novo array
  console.log(dobros.join(", "))           // 20, 26, 40, 16, 30
</script>
```

Na mesma ideia do `for..of` ou do `forEach()` – no sentido de percorrer cada elemento do vetor, o método `map()` cria um novo vetor com o resultado do processamento realizado sobre cada um dos elementos do vetor original. As operações podem ser realizadas também sobre arrays de objetos. Observe o script destacado na Figura 6.6.

The screenshot shows the JS.do Online JavaScript Editor interface. In the code editor area, there is a script block containing code to map an array of objects. The code defines two arrays: 'amigos' and 'amigos2'. 'amigos' contains three objects with names and ages. 'amigos2' is created by mapping 'amigos' to a new array where each object's age is subtracted from 2022. A 'console.log' statement is used to log the names and birth years of the friends. The output panel on the right shows the resulting log entries: Ana - Nasceu em: 2002, Bruno - Nasceu em: 2005, and Cátia - Nasceu em: 1997.

```

1 <script>
2 const amigos = [{nome: "Ana", idade: 20},
3                 {nome: "Bruno", idade: 17},
4                 {nome: "Cátia", idade: 25}]
5
6 const amigos2 = amigos.map(aux => ({nome: aux.nome, nasc: 2022 - aux.idade}))
7
8 for (const amigo of amigos2) {
9   console.log(`${amigo.nome} - Nasceu em: ${amigo.nasc}`)
10 }
11 </script>

```

Figura 6.6 – Uso do map() sobre um array de objetos.

No capítulo seguinte, veremos como obter o ano atual a partir de um método JavaScript (a ser inserido no lugar de 2022). Vamos nos concentrar, nesta seção, no entendimento da função `map()`. No exemplo da Figura 6.6, um novo array de objetos é criado, contendo o nome do amigo e o seu ano de nascimento. Observe que `aux` refere-se a cada elemento do vetor `amigos`. Portanto, o objetivo do `map()` é alterar os elementos de um vetor e, ao final, criar um novo, com a mesma quantidade de itens do vetor original.

O método `filter()` também cria um novo array. Agora cada elemento do vetor de origem é submetido a uma condição, que, se verdadeira, adiciona um elemento ao novo array. Observe o exemplo:

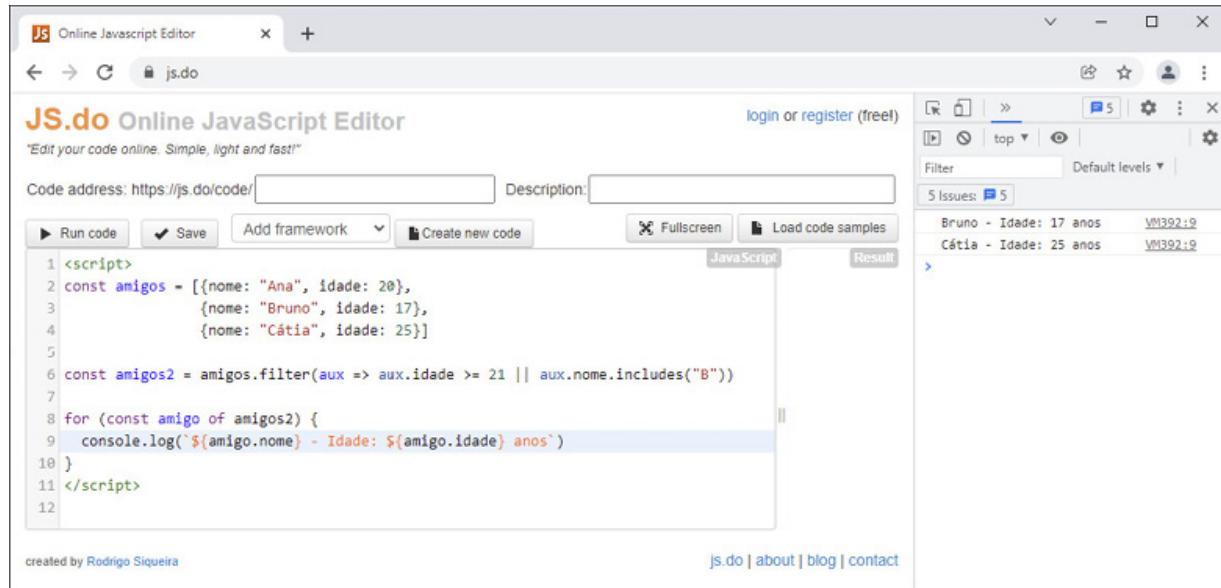
```

<script>
const numeros = [10, 13, 20, 8, 15]           // vetor inicial
const pares = numeros.filter(num => num % 2 == 0) // condição para o filtro
console.log(pares.join(', '))                  // 10, 20, 8
</script>

```

A mesma ideia vale para um vetor de objetos. O filtro é definido a partir de um ou mais atributos do objeto. E as condições podem conter os operadores lógicos `&&`, `||` ou `!` (and, or ou not), bem como os métodos já discutidos nos capítulos anteriores. No exemplo da Figura 6.7, os amigos com idade superior ou igual a 21 anos ou que contenham a letra B no nome são

selecionados.



The screenshot shows a web-based JavaScript editor interface. The main area contains the following code:

```
<script>
const amigos = [{nome: "Ana", idade: 20},
                {nome: "Bruno", idade: 17},
                {nome: "Cátia", idade: 25}]

const amigos2 = amigos.filter(aux => aux.idade >= 21 || aux.nome.includes("B"))

for (const amigo of amigos2) {
  console.log(`Amigo: ${amigo.nome} - Idade: ${amigo.idade} anos`)
}
</script>
```

The code defines an array of friends and filters it to include only those aged 21 or older or whose name starts with 'B'. It then logs each friend's name and age to the console. The right panel shows the results of the execution:

Amigo	Idade	Issue
Bruno	17 anos	VM392:9
Cátia	25 anos	VM392:9

Figura 6.7 – Filter cria um novo array de objetos.

Exibir mensagem indicando que o filtro não obteve registros que atendam ao critério de pesquisa também se torna mais simples com filter(), se comparado com o que foi discutido na seção anterior. Como um novo array é gerado, basta verificar se ele está vazio, com o seguinte código:

```
if (amigos2.length == 0) {
  console.log("Não há amigos com essas condições...")
}
```

Por fim, vamos analisar o método reduce(), útil para obter valores cumulativos (ou concatenados) a partir dos dados de um array. Observe os exemplos:

```
<script>
const numeros = [10, 13, 20, 8, 15]
const soma = numeros.reduce((acumulador, num) => acumulador + num, 0)
console.log(`Soma: ${soma}`) // Soma: 66
</script>
```

Entre parênteses (acumulador, num) representam, respectivamente, a variável que irá acumular os valores e cada elemento do array. Já acumulador + num é o que será atribuído à variável acumulador em cada passagem pelo laço de repetição que percorre o vetor. O último parâmetro, 0, é o valor inicial da variável acumulador. O método reduce(), neste exemplo, equivale aos

seguintes comandos:

```
let acumulador = 0
for (const num of numeros) {
  acumulador = acumulador + num
}
const soma = acumulador
```

Caso o último parâmetro (0) não seja fornecido ao `reduce()`, a variável `acumulador` é inicializada com o conteúdo do primeiro elemento do vetor, e a repetição inicia a partir do segundo elemento. Sem o parâmetro final, os comandos do `reduce()` equivalem a:

```
let acumulador = numeros[0]
for (let i = 1; i < numeros.length; i++) {
  acumulador = acumulador + números[i]
}
const soma = acumulador
```

A partir dos exemplos, é possível perceber que o valor resultante será o mesmo, com ou sem o parâmetro final no `reduce()`. Mas atenção para o seguinte detalhe: caso o vetor esteja vazio, não inicializar o `acumulador` irá causar um erro, pois o elemento 0 não existe e, portanto, não pode ser acessado. Por segurança, prefira inicializar o `acumulador`.

A sintaxe, ou seja, a regra de escrita do método `reduce()` possui variações, assim como `map()` e `filter()`. De acordo com os seus avanços sobre as técnicas de programação JavaScript, você naturalmente irá deparar com essas outras sintaxes. Ao utilizar o `reduce()` para manipular um array de objetos, utilizando a sintaxe curta, tenha o cuidado de sempre inicializar o `acumulador`. Na Figura 6.8, `reduce()` obtém a soma das idades e a concatenação dos nomes de uma lista de amigos.

The screenshot shows a web-based JavaScript editor interface. The title bar says "JS Online JavaScript Editor" and the address bar shows "js.do". The main area contains the following code:

```

1 <script>
2 const amigos = [{nome: "Ana", idade: 20},
3                 {nome: "Bruno", idade: 17},
4                 {nome: "Cátia", idade: 25}]
5
6 const soma = amigos.reduce((acumulador, amigo) => acumulador + amigo.idade, 0)
7 const todos = amigos.reduce((acumulador, amigo) => acumulador + amigo.nome + " ", "")
8
9 console.log(`Soma: ${soma}`)
10 console.log(`Todos: ${todos}`)
11 </script>

```

The right side of the interface shows the results of the execution: "Soma: 62" and "Todos: Ana Bruno Cátia". There are also some status messages at the bottom: "created by Rodrigo Siqueira", "js.do | about | blog | contact", and a sidebar with "Default levels" and "26 Issues".

Figura 6.8 – Método reduce() aplicado sobre um array de objetos.

Além de totais, outros valores podem ser obtidos a partir do método `reduce()`, já que ele percorre os elementos de um vetor, como se estivéssemos usando um laço `for`. O trecho de código a seguir recupera o maior valor contido no array.

```

const numeros = [10, 13, 20, 8, 15]
const maior = numeros.reduce((a, b) => Math.max(a, b), 0)
console.log(`Maior: ${maior}`) // Maior: 20

```

Nesse script, a variável `a` (acumulador) começa com 0 (valor após a última vírgula). O elemento inicial do array é atribuído a `b` (10). Logo, a variável `a` recebe 10, que é retorno de `Math.max(0, 10)`. No próximo passo, `b` recebe 13 e a comparação se dá entre 10 (`a`) e 13 (`b`). Portanto, `a` agora vale 13. O processo continua até passar por todos os elementos, resultando no maior valor do array, que é atribuído à variável `maior`.

Vamos construir um exemplo para explorar alguns dos recursos destacados até este ponto do capítulo. Nossa programa é para a “Revenda Herbie”, que vai armazenar em um vetor de objetos o modelo e o preço de carros disponíveis em uma revenda. **Map()**, **filter()** e **reduce()** são utilizados no exemplo. A página do programa, já com alguns veículos listados, é exibida na Figura 6.9.



Figura 6.9 – Um vetor de objetos armazena modelo e preço dos veículos.

Exemplo 6.3 – Código HTML do programa Revenda Herbie (ex6_3.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Revenda Herbie</h1>
<form>
  <p>Modelo: <input type="text" id="inModelo" required autofocus></p>
  <p>Preço R$:
    <input type="number" id="inPreco" required>
    <input type="submit" value="Adicionar">
  </p>
  <input type="button" value="Listar Todos" id="btListar">
  <input type="button" value="Filtrar por Preço" id="btFiltrar">
  <input type="button" value="Simular Promoção" id="btSimular">
</form>
<pre></pre>
<script src="js/ex6_3.js"></script>
<!-- /body /html -->
```

O programa JavaScript apresentado a seguir contém a programação associada a cada um dos botões exibidos na página. Vamos discutir sobre ela em etapas.

Código JavaScript do programa Revenda Herbie (js/ex6_3.js)

```
const frm = document.querySelector("form") // obtém elementos da página
```

```

const resp = document.querySelector("pre")
const carros = [] // declara vetor global

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form
  const modelo = frm.inModelo.value // obtém dados do form
  const preco = Number(frm.inPreco.value)
  carros.push({ modelo, preco }) // adiciona dados ao vetor de objetos
  frm.inModelo.value = "" // limpa campos do form
  frm.inPreco.value = ""
  inModelo.focus() // posiciona cursor em inModelo
  // dispara um evento de click em btListar (equivale a um click no botão na página)
  frm.btListar.dispatchEvent(new Event("click"))
})

```

A programação associada ao evento submit do form obtém os dados informados pelo usuário e os adiciona ao array de objetos a partir da sintaxe curta: carros.push({ modelo, preco }), que equivale a carros.push({ modelo: modelo, preco: preco }). No final da função há um método ainda não utilizado: dispatchEvent(). Ele dispara um novo evento, como se tivéssemos clicado no botão listar da página. Dessa forma, após incluir o veículo no vetor, a listagem dos dados é atualizada, o que confirma para o usuário, que a inclusão foi realizada com sucesso.

A seguir a programação associada ao click do botão **Listar**. Como o usuário pode clicar no botão **Listar** antes de fazer qualquer inclusão, realizamos o teste para verificar se o array está vazio. Para montar a lista de veículos a ser exibida utilizamos o método reduce().

```

frm.btListar.addEventListener("click", () => { // "escuta" clique em btListar
  if (carros.length == 0) { // se tamanho do vetor é igual a 0
    alert("Não há carros na lista")
    return
  }
  // método reduce() concatena uma string, obtendo modelo e preço de cada veículo
  const lista = carros.reduce((acumulador, carro) =>
    acumulador + carro.modelo + " - R$: " + carro.preco.toFixed(2) + "\n", "")
  resp.innerHTML = `Lista dos Carros Cadastrados\n${"-".repeat(40)}\n${lista}`
})

```

Já a programação associada ao clique no botão **Filtrar** solicita inicialmente o valor máximo disponível pelo cliente - conforme ilustra a Figura 6.10. A

partir do método filter(), apenas os registros com preço inferior ou igual ao informado são inseridos no array carrosFilter. Para exibir os registros você pode repetir o uso do reduce(), como no código anterior, ou utilizar um for..of. Ou seja, escolha a forma que mais lhe agrada. A Figura 6.11 apresenta os registros filtrados pelo programa.

```
frm.btFiltrar.addEventListener("click", () => {
  const maximo = Number(prompt("Qual o valor máximo que o cliente deseja pagar?"))
  if (maximo == 0 || isNaN(maximo)) {    // se não informou ou valor inválido
    return                      // ... retorna
  }
  // cria um novo vetor com os objetos que atendem a condição de filtro
  const carrosFilter = carros.filter(carro => carro.preco <= maximo)
  if (carrosFilter.length == 0) {  // se tamanho do vetor filtrado é 0
    alert("Não há carros com preço inferior ou igual ao solicitado")
    return
  }
  let lista = ""
  for (const carro of carrosFilter) { // percorre cada elemento do array
    lista += `${carro.modelo} - R$: ${carro.preco.toFixed(2)}\n`
  }
  resp.innerHTML = `Carros Até R$: ${maximo.toFixed(2)}\n${"-".repeat(40)}\n${lista}`
})
```



Figura 6.10 – O programa solicita o valor limite, para então realizar o filtro.

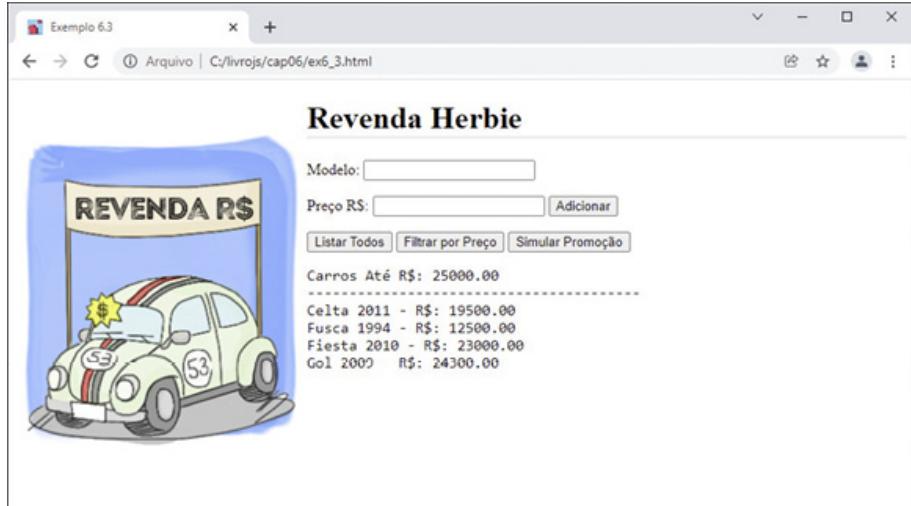


Figura 6.11 – Apenas os veículos que obedecem ao critério de pesquisa são exibidos.

Para concluir, vamos apresentar o código associado ao clique no botão **Simular Promoção**. Nele exemplificamos o uso do método `map()`. Inicialmente, solicita-se a taxa de desconto que o proprietário da revenda deseja aplicar ao preço de cada veículo. Após, um novo vetor é criado e exibido pelo programa.

```
frm.btSimular.addEventListener("click", () => {
  const desconto = Number(prompt("Qual o percentual de desconto: "))
  if (desconto == 0 || isNaN(desconto)) { // se não informou ou valor inválido
    return // ... retorna
  }
  const carrosDesc = carros.map(aux => ({
    modelo: aux.modelo,
    preco: aux.preco - (aux.preco * desconto / 100)
  }))
  let lista = ""
  for (const carro of carrosDesc) { // percorre cada elemento do array
    lista += `${carro.modelo} - R$: ${carro.preco.toFixed(2)}\n`
  }
  resp.innerHTML = `Carros com desconto: ${desconto}%\n${"-".repeat(40)}\n${lista}`
})
```

Observação: Neste exemplo, no lugar de utilizar o método `map()` também poderíamos realizar o cálculo com base no vetor original, utilizando um `for`. Talvez fosse inclusive mais simples... A vantagem do `map()` é que ele retorna um novo vetor – o que o torna interessante em alguns casos.

6.9 Classificar os itens do vetor

JavaScript dispõe do método `sort()` para classificar os itens de um vetor em ordem alfabética crescente. Esse processo é importante para apresentar os nomes de alunos de uma turma ou de contatos telefônicos, por exemplo, em ordem alfabética. Ao exibir os dados de uma lista em ordem, conseguimos facilmente localizar algum de seus elementos.

Ao executar o método `sort()`, o vetor passa a ficar ordenado. Caso seja necessário manter a lista na ordem original e apenas apresentar em alguma função do programa os dados ordenados, é possível criar uma cópia do vetor original a partir do método `slice()` ou do operador “...”.

Caso seja necessário classificar os dados em ordem decrescente, podemos utilizar em conjunto os métodos `sort()` e `reverse()`. O método `reverse()`, como o nome sugere, inverte a ordem dos elementos de um vetor. Observe o trecho de código a seguir e as saídas exibidas pelo `alert()` no comentário ao lado de cada linha.

```
<script>
  const nomes = ["Pedro", "Ana", "João"]
  nomes.sort()
  console.log(nomes.join(", ")) // Ana, João, Pedro
  nomes.reverse()
  console.log(nomes.join(", ")) // Pedro, João, Ana
</script>
```

Um detalhe importante sobre o processo de ordenação de listas em JavaScript é que os dados do vetor são classificados como strings, mesmo que o seu conteúdo seja formado apenas por números. Uma classificação de números como strings faz com que o número “2” seja considerado maior que “100”, por exemplo. Isso porque a comparação é realizada da esquerda para a direita, caractere por caractere. Para contornar essa situação, é possível definir uma função que vai subtrair os dados, de dois a dois, em cada comparação. Como já destacado, a criação de funções com parâmetros será abordada no Capítulo 8. Por ora, é importante entender a sintaxe utilizada para que a ordenação de uma lista de números funcione como o esperado. Observe o script a seguir:

```

<script>
const numeros = [50, 100, 2]
numeros.sort()
console.log(numeros.join(", ")) // 100, 2, 50
numeros.sort((a, b) => a - b)
console.log(numeros.join(", ")) // 2, 50, 100
</script>

```

Ordenar os dados de uma lista, além de ser útil para exibir os dados de uma forma organizada, também é importante para nos auxiliar no processo de obtenção de algumas respostas em um programa. Vamos praticar isso em um novo exemplo. Considere a seguinte situação: O síndico de um determinado condomínio deseja criar uma brinquedoteca no salão do condomínio. Para tanto, necessita de um programa que leia nome e idade de crianças e exiba o número e o percentual de crianças em cada idade, a fim de que os brinquedos sejam comprados de acordo com a faixa etária delas. O programa deve armazenar os dados em um vetor de registros e apresentar o resumo conforme solicitado. A Figura 6.12 exibe a página com uma lista de crianças inseridas no vetor.



Figura 6.12 – Os dados inseridos serão ordenados por idade para obter-se o resumo da lista.

Exemplo 6.4 – Código HTML do programa Brinquedoteca (ex6_4.html)

```

<!-- doctype, html, head e body (conf. exemplo 4.2) -->


```

```

<h1>Programa Brinquedoteca</h1>
<form>
  <p>Nome da Criança:<br/>
    <input type="text" id="inNome" required autofocus>
  </p>
  <p>Idade:<br/>
    <input type="number" id="inIdade" min="0" required>
    <input type="submit" value="Adicionar">
  </p>
  <input type="button" value="Listar Todos" id="btListar">
  <input type="button" value="Resumir por Idade" id="btResumir">
</form>
<pre></pre>
<script src="js/ex6_4.js"></script>
<!-- /body e /html -->

```

O código HTML segue o padrão utilizado nos exemplos anteriores. O programa JavaScript, nas suas funções de inserir e listar as crianças armazenadas no vetor de objetos, também é semelhante ao programa da “Revenda Herbie”. As novidades estão na programação associada ao botão **Resumir por Idade**. Vamos, inicialmente, ver o código associado à Inclusão e Listagem dos dados.

Código JavaScript do programa Brinquedoteca (js/ex6_4.js)

```

const frm = document.querySelector("form") // obtém elementos a serem manipulados
const resp = document.querySelector("pre")

const criancas = [] // declara vetor global

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  const nome = frm.inNome.value // obtém conteúdo dos campos
  const idade = Number(frm.inIdade.value)
  criancas.push({ nome, idade }) // adiciona dados ao vetor de objetos
  frm.reset() // limpa campos do form
  frm.inNome.focus() // posiciona no campo de formulário
  frm.btListar.dispatchEvent(new Event("click")) // dispara click em btListar
})

frm.btListar.addEventListener("click", () => {

```

```

if (criancas.length == 0) { // se vazio, exibe alerta
    alert("Não há crianças na lista")
    return
}
let lista = "" // para concatenar lista de crianças
for (const crianca of criancas) {
    const { nome, idade } = crianca // desestrutura o objeto
    lista += nome + " - " + idade + " anos\n"
}
resp.innerHTML = lista // exibe a lista
})

```

Já a programação do “Resumir por Idade” percorre o vetor para produzir uma saída conforme a ilustração da Figura 6.13. Analise com atenção o código necessário para obter esses dados.

```

frm.btResumir.addEventListener("click", () => {
    if (criancas.length == 0) {
        alert("Não há crianças na lista")
        return
    }
    const copia = [...criancas] // cria cópia do vetor criança
    copia.sort((a, b) => a.idade - b.idade) // ordena pela idade
    let resumo = "" // para concatenar saída
    let aux = copia[0].idade // menor idade do vetor ordenado
    let nomes = [] // para inserir nomes de cada idade
    for (const crianca of copia) {
        const { nome, idade } = crianca
        if (idade == aux) { // se é da mesma idade auxiliar...
            nomes.push(nome) // adiciona ao vetor nomes
        }
        else { // senão, monta o resumo para cada idade
            resumo += aux + " ano(s): " + nomes.length + " criança(s) - "
            resumo += ((nomes.length / copia.length) * 100).toFixed(2) + "%\n"
            resumo += "(" + nomes.join(", ") + ")\n\n"
            aux = idade // obtém a nova idade na ordem
            nomes = [] // limpa o vetor dos nomes
            nomes.push(nome) // adiciona o primeiro da nova idade
        }
    }
    // adiciona a última criança
    resumo += aux + " ano(s): " + nomes.length + " criança(s) - "
    resumo += ((nomes.length / copia.length) * 100).toFixed(2) + "%\n"

```

```

    resumo += "(" + nomes.join(", ") + ")\n\n"
    resp.innerHTML = resumo // exibe a resposta
})

```

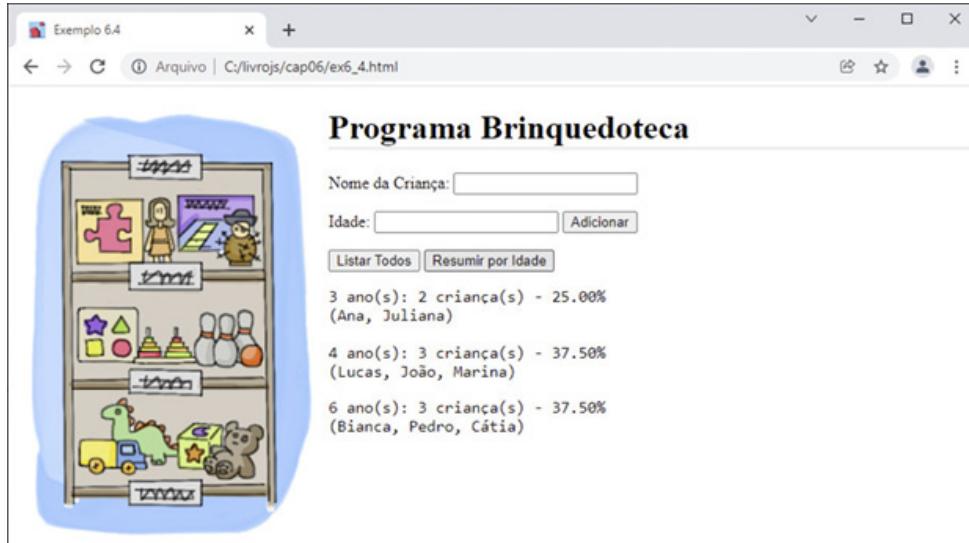


Figura 6.13 – Os alunos são agrupados por idade.

Para resolver um problema diferente, como o do Exemplo 6.4, podemos fazer uso de lápis e papel para “desenhar” a representação dos dados a serem manipulados. Após a ordenação dos elementos do vetor, para o conjunto de dados de entrada exemplificado na Figura 6.12, nossa lista apresenta o desenho conforme ilustra a Tabela 6.4:

Tabela 6.4 – Conteúdo do vetor de objetos após a ordenação

	Nome	Idade
0	Ana	3
1	Juliana	3
2	Lucas	4
3	João	4
4	Marina	4
5	Bianca	6
6	Pedro	6
7	Cátia	6

Ao visualizar a representação dos dados, melhora a percepção de como podemos montar o algoritmo para extrair as informações desejadas.

Observe os passos executados para listar o número, o percentual e o nome das crianças em cada idade.

Após as validações, criamos uma cópia do vetor original. Dessa forma, se o usuário clicar no botão **Listar Todos**, a lista com as crianças permanece exibida na ordem informada ao sistema. O próximo passo é atribuir para uma variável auxiliar a idade da primeira criança e criar um vetor que vai conter os nomes das crianças em cada idade. Agora, podemos percorrer a lista (com o comando `for`) e comparar as idades com essa variável auxiliar. Se a idade da criança for igual ao valor da variável auxiliar, essa criança deve ser inserida na lista de nomes. Caso contrário, significa que houve uma troca de idades na lista. Portanto, devemos concluir a idade anterior e iniciar uma nova lista para a idade em questão.

Após a repetição, é preciso ter o cuidado de adicionar à resposta os dados das crianças da última idade da lista ordenada, pois os seus dados foram adicionados ao vetor nomes, porém ainda não foram concatenados na resposta.

Certo!? Nos exercícios deste capítulo, vamos praticar a programação que envolve a manipulação de vetores. Assim, você poderá se divertir e descobrir como é legal pensar em estratégias para resolver um problema envolvendo listas de dados. Reserve um tempo para isso. Você vai gostar!

Antes, vamos ver alguns exemplos com Node.js que também podem ser utilizados como exercícios. Veja os códigos em Node.js dos capítulos anteriores e os recursos discutidos aqui. Unindo esses conhecimentos você pode tentar resolver os algoritmos e depois comparar com o exemplo de resolução apresentado.

6.10 Manipulação de vetores com Node.js

Como já mencionado nos capítulos iniciais, os métodos JavaScript são os mesmos, independentemente se executados no prompt com o Node.js ou nas páginas web com o browser. Vamos resolver alguns exemplos de programação com vetores utilizando o Node.js, então?

a) Elaborar um programa que leia nome e nota de ‘n’ alunos até o usuário digitar ‘Fim’ no nome. Após, verifique e exiba a maior nota da turma. Se a maior nota for superior ou igual a 7, exiba os alunos que a obtiveram. Caso contrário, exiba a mensagem “Não há alunos em destaque na turma”.

```
const prompt = require("prompt-sync")()
console.log("Informe os alunos. Após, digite 'Fim' no nome para sair")
const alunos = [] // declara vetor
do {
    const nome = prompt("Nome: ") // lê o nome
    if (nome == "Fim") { // antes de ler a nota, verifica
        break // sai da repetição
    }
    const nota = Number(prompt("Nota: ")) // lê a nota
    alunos.push({nome, nota}) // adiciona dados ao vetor de objetos
    console.log(`Ok! Aluno(a) cadastrado(a)...`)
} while(true)
console.log("-".repeat(40)) // exibe 40 -
const maior = alunos.reduce((a, b) => Math.max(a, b.nota), 0) // obtém a maior nota
console.log(`Maior Nota: ${maior}`) // exibe a maior nota
if (maior >= 7) { // para verificar se tem destaque na turma
    const destaque = alunos.filter(aluno => aluno.nota == maior) // filtro
    for (const destaque of destaque) { // percorre os alunos em destaque
        console.log(`- ${destaque.nome}`) // mostra os nomes (precedidos por -)
    }
} else { // senão, vai exibir a mensagem
    console.log("Não há alunos em destaque na turma")
}
```

Observe que o programa solicita o nome e, antes de fazer a leitura da nota, verifica se o usuário digitou ‘Fim’ no nome. Isso evita a digitação de uma nota quando se deseja sair da repetição.

Nesse exemplo, utilizamos o método `reduce()` para obter a maior nota do aluno. Existem outras formas para fazer isso, como classificar o vetor por ordem de nota, fazer um `reverse()` e obter a nota do 1º aluno (na posição 0). Na sequência do programa, é exibida a maior nota e, caso ela seja superior ou igual a 7, são exibidos os alunos em destaque utilizando um filtro (já que vários alunos podem ter essa maior nota). A seguir, o programa em execução.

C:\livrojs\cap06\nodejs>**node ex6_5**

Informe os alunos. Após, digite 'Fim' no nome para sair

Nome: Carla

Nota: 7.5

Ok! Aluno(a) cadastrado(a)...

Nome: Michele

Nota: 7.2

Ok! Aluno(a) cadastrado(a)...

Nome: Paulo

Nota: 7.5

Ok! Aluno(a) cadastrado(a)...

Nome: Fim

Maior Nota: 7.5

- Carla

- Paulo

b) Elaborar um programa que leia nome e idade de ‘n’ clientes de um banco (até ser digitado ‘Fim’ no nome). Após, classifique e exiba os clientes em 2 grupos: preferencial (a partir de 60 anos) e Fila normal (até 59 anos). Informe a ordem de atendimento em cada grupo de acordo com a chegada dos clientes.

```
const prompt = require("prompt-sync")()
console.log("Informe os clientes em ordem de chegada ou 'Fim' no nome para sair")
const clientes = [] // declara vetor
do {
    const nome = prompt("Nome: ") // lê o nome
    if (nome == "Fim") { // antes de ler a idade, verifica
        break // sai da repetição
    }
    const idade = Number(prompt("Idade: "))
    clientes.push({ nome, idade }) // adiciona ao vetor de objetos
    console.log("Ok! Cliente inserido na fila...")
} while (true)
console.log("\nFila Preferencial") // \n no início gera uma nova linha
console.log("-".repeat(40)) // exibe 40 -
const filaPref = clientes.filter(cliente => cliente.idade >= 60)
filaPref.forEach((fila, i) => {
    console.log(` ${i + 1}. ${fila.nome}`)
})
console.log("\nFila Normal")
console.log("-".repeat(40))
const filaNormal = clientes.filter(cliente => cliente.idade < 60)
```

```
filaNormal.forEach((fila, i) => {
  console.log(`#${i + 1}. ${fila.nome}`)
})
```

Com o uso do método filter, criamos dois novos vetores. Após, a partir do método forEach() percorremos cada vetor para exibir os clientes de cada grupo. Um exemplo de execução desse programa é exibido a seguir:

```
C:\livrojs\cap06\nodejs>node ex6_6
Informe os clientes em ordem de chegada ou 'Fim' no nome para sair
Nome: Marcos
Idade: 32
Ok! Cliente inserido na fila...
Nome: Teresa
Idade: 67
Ok! Cliente inserido na fila...
Nome: William
Idade: 27
Ok! Cliente inserido na fila...
Nome: Fim
```

Fila Preferencial

1. Teresa

Fila Normal

1. Marcos
2. William

c) *Elaborar um programa que simule saques em um caixa eletrônico de um banco. Ler o valor solicitado por clientes até ser digitado 0. Sabendo que o caixa dispõe apenas de notas de 10, exiba após cada leitura se o saque é válido ou inválido. Ao final, listar os saques válidos e a soma dos saques. Exiba também o número de saques inválidos.*

```
const prompt = require("prompt-sync")()
console.log("Informe o valor dos saques ou 0 para sair")
const saques = [] // declara vetor
do {
  const valor = Number(prompt("Saque R$: ")) // lê valor
  if (valor == 0) {
    break // sai da repetição
```

```

}

saques.push(valor) // adiciona ao vetor saques
if (valor % 10 == 0) {
  console.log("Saque Realizado com Sucesso")
} else {
  console.log("Erro... Valor Inválido (deve ser múltiplo de 10)")
}
} while (true)
console.log("\nSaques Válidos") // \n no início gera uma nova linha
console.log("-".repeat(40)) // exibe 40 -
const saquesValidos = saques.filter(saque => saque % 10 == 0)
for (const saque of saquesValidos) {
  console.log(saque.toFixed(2))
}
console.log("-".repeat(40))
const totalSacado = saquesValidos.reduce((total, saque) => total + saque, 0)
console.log(`Total dos Saques: R$ ${totalSacado.toFixed(2)}`)

const saquesInvalidos = saques.length - saquesValidos.length
console.log(`\nNº de Tentativas de Saques (saques inválidos): ${saquesInvalidos}`)

```

Observe que utilizamos o filter para obter a lista dos saques válidos e reduce para calcular a soma desses saques. Já a quantidade de saques inválidos foi obtida pela subtração do tamanho dos dois vetores manipulados pelo programa. A seguir, um exemplo de execução desse programa.

```

C:\livrojs\cap06\nodejs>node ex6_7
Informe o valor dos saques ou 0 para sair
Saque R$: 75
Erro... Valor Inválido (deve ser múltiplo de 10)
Saque R$: 80
Saque Realizado com Sucesso
Saque R$: 100
Saque Realizado com Sucesso
Saque R$: 0

```

Saques Válidos

```

80.00
100.00

```

```

Total dos Saques: R$ 180.00
Nº de Tentativas de Saques (saques inválidos): 1

```

6.11 Exercícios

Diversas são as pesquisas que destacam que o aprendizado ocorre de forma muito mais significativa quando a teoria é acompanhada pela prática daquilo que se está estudando. Portanto, não deixe de realizar os exercícios de fixação sobre algoritmos que manipulam listas de dados, sugeridos a seguir.

a) *Elaborar um programa para gerar uma tabela com os jogos de uma fase eliminatória de um campeonato. O programa deve conter três funções (a serem executadas no evento click de cada botão) para: 1) validar o preenchimento, adicionar um clube ao vetor e listar os clubes; 2) listar os clubes (se houver); 3) montar a tabela de jogos, no formato primeiro x último, segundo x penúltimo e assim por diante. Exibir mensagem e não listar a tabela de jogos, caso o número de clubes informados seja ímpar. A Figura 6.14 ilustra a página gerada com a tabela de jogos.*



Figura 6.14 – Tabela dos jogos eliminatórios exibida ao clicar no botão “Montar Tabela de Jogos”.

b) *Elaborar um programa que adicione números a um vetor. O programa deve impedir a inclusão de números repetidos. Exibir a lista de números a cada inclusão. Ao clicar no botão Verificar Ordem, o programa deve analisar o conteúdo do vetor e informar se os números estão ou não em ordem crescente. A Figura 6.15 demonstra um exemplo de execução do programa.*



Figura 6.15 – Programa deve ler números e verificar se eles estão em ordem crescente.

c) Elaborar um programa que leia nome e número de acertos de candidatos inscritos em um concurso. Listar os dados a cada inclusão. Ao clicar no botão **Aprovados 2^a Fase**, ler o número de acertos para aprovação dos candidatos para a 2^a fase do concurso, conforme ilustra a Figura 6.16. O programa deve, então, exibir os candidatos aprovados, ou seja, apenas os que obtiveram nota maior ou igual à nota informada. Exibir os candidatos aprovados em ordem decrescente de número de acertos (Figura 6.17). Caso nenhum candidato tenha sido aprovado, exibir mensagem.

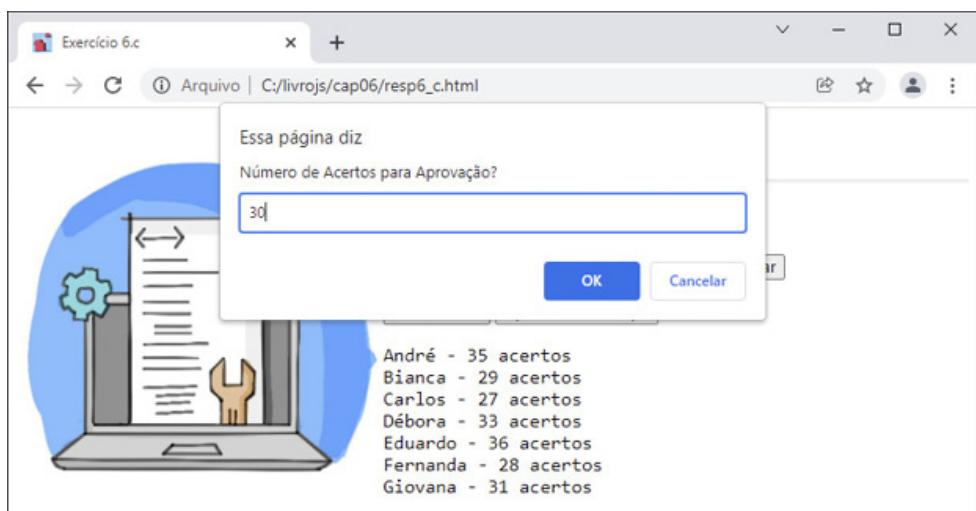


Figura 6.16 – Ao clicar no botão “Aprovados 2^a Etapa”, ler nº de acertos para aprovação.



Figura 6.17 – Listar candidatos aprovados em ordem decrescente de nº de acertos.

6.12 Considerações finais do capítulo

Os vetores permitem armazenar listas de dados em um programa. Cada vetor contém um nome e um índice, cujo primeiro elemento ocupa a posição zero do vetor. Assim, um vetor de frutas poderia conter os elementos: frutas[0] = "Banana", frutas[1] = "Maçã" e frutas[2] = "Mamão". Em JavaScript, os vetores podem aumentar e diminuir de tamanho dinamicamente, não sendo necessário especificar o número de elementos da lista na sua declaração. As linguagens de programação dispõem de diversos métodos para trabalhar com os vetores. Dentre as principais ações realizadas pelos métodos de manipulação de vetores, destacam-se:

- **Inserir e remover elementos no início e no final do vetor** – Para a inserção, utilize os métodos `unshift()` e `push()`, que inserem, respectivamente, um novo elemento no início e no final do vetor. Já a exclusão pode ser realizada com os métodos `shift()` e `pop()`, para remover o primeiro e o último elemento da lista.
- **Exibir o conteúdo do vetor** – A partir do uso da propriedade `length`, que retorna o tamanho do vetor, e de um laço de repetição criado com o comando `for`, é possível acessar cada um dos elementos do vetor. Nesse caso, utiliza-se a variável de controle do laço (`i`) para referenciar cada

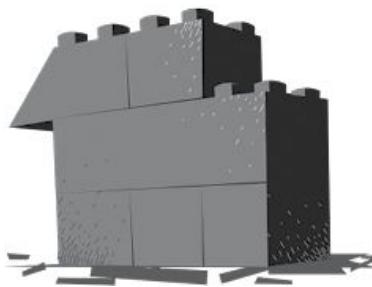
índice. Além do `for` “tradicional” pode-se ainda utilizar o loop `for...of` ou o método `forEach()`. O método `join()` também é frequentemente utilizado para exibir uma lista, pois ele converte o conteúdo do vetor em uma única string, separando os elementos por um ou mais caracteres.

- **Localizar um item no vetor** – Para essa tarefa, a linguagem JavaScript dispõe dos métodos `indexOf()`, `lastIndexOf()` e `includes()`. Os dois primeiros retornam a posição da primeira ocorrência do conteúdo pesquisado no vetor da esquerda para a direita ou da direita para a esquerda, respectivamente. Caso o conteúdo não exista no vetor, o valor `-1` é retornado em ambos os métodos. Já o método `includes()` retorna verdadeiro ou falso, conforme a existência ou não de um conteúdo no vetor.
- **Manipular um vetor de objetos, com diferentes atributos** – Com um vetor de objetos, podemos armazenar, por exemplo, os atributos modelo, marca, ano e preço dos veículos de uma revenda. As operações de inserção, exclusão e classificação dos elementos do vetor ocorrem para todos os seus atributos.
- **Filtrar os dados da lista** – A filtragem consiste em recuperar os elementos da lista que obedecem a algum critério, como os veículos da revenda com preço inferior a R\$ 20.000,00. Para realizar essa tarefa, é possível percorrer a lista e, a partir de uma condição, selecionar os itens em que a comparação retorna verdadeiro.
- **Classificar os elementos do vetor** – A classificação de dados, além de organizar o conteúdo da lista para uma exibição em ordem alfabética dos seus elementos, serve muitas vezes, também, como apoio para o processo de obtenção de informações relevantes sobre os dados da lista. Por exemplo, calcular e listar as diferentes marcas armazenadas em um vetor de objetos, exibindo a quantidade de veículos de cada marca. Esse processo é simplificado se o vetor estiver classificado pelo campo marca. O método `sort()` classifica os itens de um vetor em ordem crescente. Utilizar o método `sort()`, seguido pelo método `reverse()`, faz com que a lista fique em ordem decrescente.

As tarefas de percorrer, filtrar e totalizar os dados de um vetor em JavaScript contam ainda com os métodos `map()`, `filter()` e `reduce()`. Para utilizá-los, tenha cuidado com a sintaxe de cada método, bem como de seus parâmetros.

CAPÍTULO 7

Strings e datas



Os exemplos e exercícios desenvolvidos nos capítulos anteriores, no geral, realizaram operações sobre números e listas de dados. Neste capítulo, abordaremos rotinas de programação que trabalham com cadeias de caracteres (strings) e datas. Para realizar operações sobre strings e datas, as linguagens de programação dispõem de métodos próprios para esse fim. Para as strings, por exemplo, há métodos para obter cada uma das letras que compõem uma palavra, converter uma palavra para letras maiúsculas ou minúsculas ou, então, extrair partes de uma palavra. Para as datas, por sua vez, há métodos para criar objetos do tipo Date e realizar cálculos sobre as datas – como adicionar ou subtrair dias, meses ou anos a uma data.

Os exemplos de rotinas de programação que realizam operações sobre cadeias de caracteres e datas são muitos. Para as strings, pode-se validar o preenchimento de um nome em um formulário de cadastro, criar uma sugestão de e-mail com as iniciais de nome e sobrenome de um aluno ou, então, criar um jogo em que o usuário deve acertar as letras que compõem uma palavra. Para as datas, é possível calcular a diferença de dias entre duas datas, determinar a data de vencimento das parcelas de uma compra ou, então, calcular a data limite para a obtenção de um determinado desconto.

As tarefas relacionadas ao processo de validação de senhas de usuários de um sistema também podem ser realizadas a partir dos métodos e

propriedades de manipulação de strings. Diante da relevância do tema Segurança de Informação para os sistemas atuais, criar regras de composição para os caracteres de uma senha assume um papel de destaque em qualquer sistema. Implementar uma política de senhas em um sistema pode impedir que senhas tradicionalmente utilizadas (e fracas), como “12345”, “admin” ou “admin123”, sejam inseridas livremente no sistema. Senhas como essas, tornam um sistema vulnerável a qualquer invasor e jogam por terra todos os esforços que visam garantir a manutenção dos atributos de segurança no sistema.

Com as propriedades e os métodos de manipulação de strings, é possível verificar se uma senha conta com um número mínimo de caracteres, se é formada por letras e números, se contém letras maiúsculas e minúsculas e, ainda, se utiliza algum caractere especial na sua composição. Uma seção deste capítulo abordará esse assunto, com a criação de um exemplo de programa para verificar se uma senha cumpre com algumas regras de composição para ser válida.

Deixamos para conversar sobre manipulação de strings depois de tratar do processo de criação de vetores, pois alguns dos métodos utilizados no tratamento de strings retornam um vetor de elementos. Além disso, alguns métodos, como o `indexOf()`, `includes()`, e a propriedade `length`, têm comportamentos comuns tanto nos vetores, quanto nas strings, o que facilita o processo de aprendizagem.

Nas próximas seções deste capítulo, vamos apresentar os métodos e propriedades utilizados para realizar essas operações e discutir sobre a aplicação desses métodos no desenvolvimento de programas JavaScript.

7.1 Percorrer os caracteres de uma string

Vamos começar nosso estudo sobre os métodos JavaScript disponíveis para a manipulação de cadeias de caracteres por um dos seus métodos mais simples, que é o `charAt()`. Esse método retorna o caractere de uma posição da palavra. Assim como nos vetores, a posição inicial da string é 0. Observe a representação da Figura 7.1.

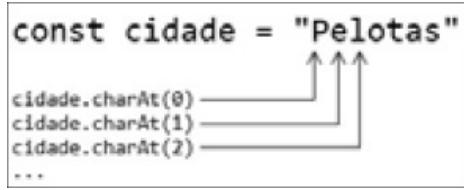


Figura 7.1 – Método charAt() obtém cada um dos caracteres de uma string.

A propriedade `length`, já utilizada para obter o tamanho de um vetor, é importante também na manipulação de strings. Ela retorna o tamanho (número de caracteres) da string. Com a propriedade `length` e o método `charAt()`, é possível, portanto, percorrer todos os caracteres de uma string. Se quisermos verificar, por exemplo, quantas palavras contém o texto de um anúncio, podemos implementar o script descrito a seguir. Não se esqueça de criar uma nova pasta para os exemplos deste capítulo (`cap07`) e, dentro dela, as pastas `css`, `img` e `js`. Em seguida, crie o arquivo `ex7_1.html`. Para alguns exemplos simples deste capítulo vamos trabalhar com os métodos `prompt()` e `alert()`.

Exemplo 7.1 – Programa que lê um anúncio e informa número de palavras (ex7_1.html)

```

<script>
  const anuncio = prompt("Anúncio: ") // lê o anúncio
  let numPalavras = 0 // inicializa contador
  const tam = anuncio.length // obtém o tamanho
  for (let i = 0; i < tam; i++) { // percorre os caracteres do anúncio
    if (anuncio.charAt(i) == " ") { // se encontrou um espaço
      numPalavras++ // incrementa contador
    }
  }
  // exibe anúncio e número de palavras (que é o nº de espaços + 1)
  alert(`Anúncio: ${anuncio}\nNº Palavras: ${numPalavras + 1}`)
</script>

```

O script lê o anúncio, descobre o tamanho do texto e percorre todos os caracteres que o compõem. Ao encontrar um espaço, acrescenta-se 1 para a variável `numPalavras`. Como um texto com 5 espaços contém 6 palavras, deve-se acrescentar 1 antes de exibir a resposta para o usuário. Esse cálculo também pode ser obtido com o uso do método `split()` visto mais adiante neste capítulo. A Figura 7.2 exibe o `alert()` desse script, com um exemplo de

anúncio informado pelo usuário.

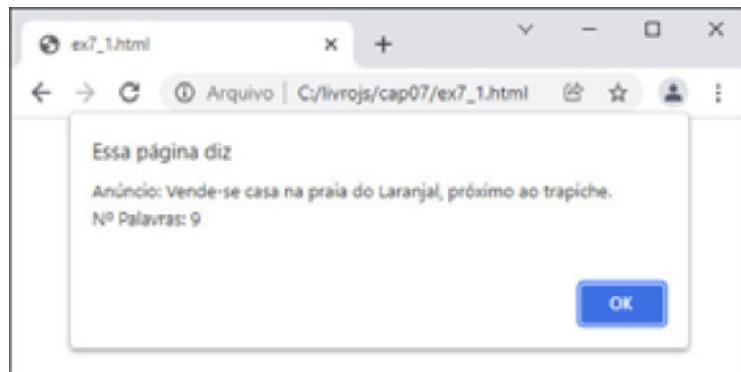


Figura 7.2 – Uso do método charAt() para percorrer os caracteres de uma string.

Como indicado anteriormente neste capítulo, uma string pode ser tratada como um vetor de caracteres. Logo, a forma alternativa de percorrer um vetor utilizando `for..of` também está disponível, simplificando nossos programas. Para o Exemplo 7.1, a rotina a seguir também obtém o número de palavras do anúncio:

```
let numPalavras = 0      // inicializa contador
for (const letra of anuncio) { // percorre os caracteres do anúncio
    if (letra == " ") {      // se encontrou um espaço...
        numPalavras++      // incrementa contador
    }
}
```

Vamos construir alguns exemplos de programas para manipular strings. Começamos pelo programa “Qual é a fruta?”, que deve ler uma palavra (sugere-se uma fruta) e exibir, após o clique no botão **Mostrar Dica**, a letra inicial da fruta e as demais ocorrências dessa letra na palavra. As outras letras não devem ser exibidas, apenas um sublinhado (underline) “_” para representar cada letra. O conteúdo do campo de entrada deve ser substituído por asteriscos, conforme ilustra a Figura 7.3. Esse jogo completo será desenvolvido no Capítulo 11. Por enquanto, vamos apenas entender como mostrar uma dica da palavra.



Figura 7.3 – No programa, percorre-se a palavra para exibir a letra inicial ou o sublinhado.

Para construir esse segundo exemplo, crie o arquivo estilos.css na pasta css. Insira nele as seguintes regras de estilização a serem aplicadas aos elementos HTML das páginas deste capítulo:

```
img { float: left; height: 300px; width: 300px; }
h1 { border-bottom-style: inset; }
.entre-letras { letter-spacing: 0.5em; }
.alinha-direita { text-align: right; }
```

Em seguida, crie os arquivos ex7_2.html e ex7_2.js (na pasta js) com os códigos descritos a seguir.

Exemplo 7.2 – Código HTML do Programa “Qual é a fruta?” (ex7_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1> Programa Qual é a Fruta? </h1>
<form>
  <p> Fruta:
    <input type="text" id="inFruta" autofocus required>
    <input type="submit" value="Jogar">
  </p>
</form>
<h3> Descubra: <span class="entre-letras"></span></h3>
<script src="js/ex7_2.js"></script>
<!-- /body e /html -->
```

Código JavaScript do programa Qual é a fruta?

(js/ex7_2.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("span")

frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
  e.preventDefault() // evita envio do form
  const fruta = frm.inFruta.value.toUpperCase() // conteúdo do campo em maiúsculas
  let resposta = "" // string que irá conter a resposta

  for (const letra of fruta) { // percorre todos os caracteres da fruta
    if (letra == fruta.charAt(0)) { // se letra igual a letra inicial da string...
      resposta += fruta.charAt(0) // adiciona a letra inicial
    } else { // senão, ...
      resposta += "_" // adiciona o sublinhado
    }
  }

  resp.innerText = resposta // exibe a resposta
  frm.inFruta.value = "*".repeat(fruta.length) // preenche com "*", conforme tamanho
})
```

Nosso programa segue o padrão dos exemplos dos capítulos anteriores. Após criar uma referência aos elementos que serão manipulados pelo programa, adicionamos um ouvinte para o evento submit do form, e nele obtemos a palavra informada pelo usuário. Para converter essa palavra para letras maiúsculas, utiliza-se o método `toUpperCase()`, discutido a seguir. Sobre essa palavra é que o programa vai agir, ou seja, percorrer os seus caracteres para exibir a saída conforme solicitado. Observe que na repetição realizamos uma comparação para verificar se cada letra obtida no `for..of` é igual ao caractere inicial da string. Se for, ele é concatenado. Caso contrário, adiciona-se o underline (“_”).

Para concatenar “*” no campo de formulário e ocultar a palavra informada, deve-se fazer uma atribuição ao campo de formulário, a partir de sua propriedade `value`. A quantidade de “*” é obtida a partir da propriedade `length`, que retorna o tamanho de uma string.

7.2 Converter para letras maiúsculas ou

minúsculas

Para converter para letras maiúsculas ou minúsculas os caracteres de uma palavra, a linguagem JavaScript dispõe dos métodos `toUpperCase()` e `toLowerCase()` que devem ser aplicados na palavra ou na letra que se deseja realizar a conversão.

O uso desses métodos, além de servir para apresentar uma palavra em caixa alta ou baixa, também é importante para auxiliar nas condições envolvendo palavras ou letras. Como sabemos, as linguagens diferenciam as letras maiúsculas de suas equivalentes minúsculas em uma comparação. Assim, utilizar esses métodos pode simplificar algumas condições criadas em um programa. Observe o trecho do script a seguir:

```
<script>
  while (true) { // cria repetição
    // comandos ...
    const continua = prompt("Continuar (S/N)?") // lê uma entrada
    if (continua.toUpperCase() == "N") { // converte em maiúscula
      break // sai da repetição
    }
  }
</script>
```

Ao aplicar o método `toUpperCase()` na variável `continua`, a comparação retorna verdadeiro mesmo quando o usuário digitar “n” (minúsculo) para a pergunta realizada pelo `prompt()` – pois antes da comparação a variável é convertida para maiúscula. No Exemplo 7.2, também foi importante adicionar o método `toUpperCase()`, pois, se o usuário informar a letra inicial maiúscula e as demais, minúsculas, o programa não funcionaria corretamente.

Nosso terceiro exemplo sobre manipulação de strings explora o uso desses métodos. O programa deve ler uma palavra e exibi-la de forma invertida. Um detalhe a ser observado nesse exemplo: na inversão, o primeiro caractere deve ficar em letra maiúscula e os demais, em minúsculas. A Figura 7.4 ilustra uma execução desse programa.

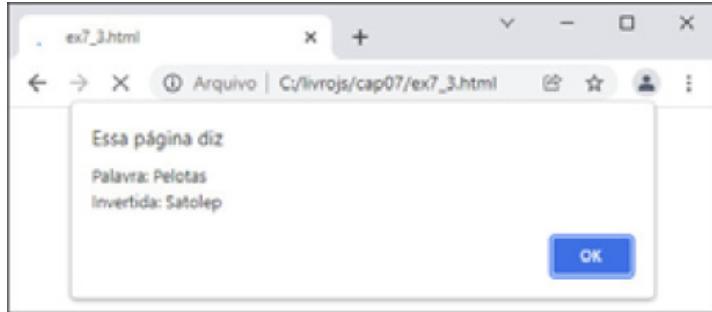


Figura 7.4 – Na inversão, apenas a letra inicial deve ficar em caixa alta.

Exemplo 7.3 – Programa Inverte Palavra (ex7_3.html)

```
<script>
const palavra = prompt("Palavra: ") // lê a palavra
const tam = palavra.length // obtém o tamanho

// inverso inicia com a última letra da palavra em caixa alta
let inverso = palavra.charAt(tam - 1).toUpperCase()

// for decrescente percorre as demais letras e ...
for (let i = tam - 2; i >= 0; i--) {
    inverso += palavra.charAt(i).toLowerCase() // converte-as em caixa baixa
}

// exibe palavra original e invertida
alert(`Palavra: ${palavra}\nInvertida: ${inverso}`)
</script>
```

Perceba, no Exemplo 7.3, que os métodos `charAt()` e `toUpperCase()` ou `toLowerCase()` são aplicados sobre palavra em uma mesma instrução. Isso significa que o comando faz a obtenção de um caractere da string e já o converte para uma letra maiúscula ou minúscula.

7.3 Cópia de caracteres e remoção de espaços da string

Além do método para obter um caractere de uma string, as linguagens de programação dispõem também de métodos para recuperar partes da string. Em JavaScript, um desses métodos é o `substr()`, que contém dois parâmetros: posição inicial da string e número de caracteres a serem copiados. Caso

apenas a posição inicial seja informada, todos os caracteres dessa posição até o final da string são copiados. Observe os exemplos no script a seguir:

```
<script>
const palavra = "saladas"
const copia1 = palavra.substr(2)           // obtém "ladas"
const copia2 = palavra.substr(2, 4)         // obtém "lada"
const copia3 = palavra.substr(0, palavra.length-1) // obtém "salada"
const copia4 = palavra.substr(-2)          // obtém "as"
</script>
```

Lembre-se de que a posição inicial da string é referenciada pelo índice 0. Assim, copia1 contém todos os caracteres da posição 2 até o seu final. Para a variável copia2, foi informado também o número de caracteres que se deseja obter. Já para copia3, indicamos a posição inicial e para o número de caracteres informamos a propriedade length-1, ou seja, serão copiados todos os caracteres exceto o último. Ah... temos, então, mais uma possibilidade para retirar aquela última vírgula de uma lista de números, discutida no Capítulo 5.

E, para a variável copia4, foi passado um valor negativo. Ele serve para indicar que a posição inicial é contada da direita para a esquerda. Assim, para obter o último caractere de uma string, podemos utilizar um dos seguintes métodos:

```
const ultima1 = palavra.substr(-1)
const ultima2 = palavra.charAt(palavra.length-1)
```

Os métodos substring() e slice() também podem ser utilizados para obter partes de uma string, com pequenas variações de parâmetros em relação ao substr().

Caso o usuário tenha digitado um espaço extra no final ou no início de uma palavra, esse espaço é considerado um caractere existente na string. Obter o último caractere, por exemplo, nos retornaria um espaço em branco. Para remover esses espaços no início e no final de uma string, a linguagem JavaScript dispõe do método trim(). Caso nosso objetivo seja remover os espaços extras apenas do início da string, existe o método trimStart(). Já trimEnd() remove todos os espaços do final da string. No Exemplo 7.4, apresentado a seguir, vamos utilizar esse recurso.

7.4 Localizar um ou mais caracteres na string

Estão lembrados dos métodos `indexOf()`, `lastIndexOf()` e `includes()` utilizados para pesquisar um conteúdo em uma lista de dados armazenada em um vetor? Eles também podem ser aplicados sobre uma string e possuem a mesma finalidade: localizar um caractere (ou mais caracteres) agora em uma string. No `indexOf()`, a pesquisa se dá a partir do início da string, enquanto, no `lastIndexOf()`, a pesquisa ocorre da direita para a esquerda. Caso o conteúdo não exista, o valor `-1` é retornado. Já o `includes()` retorna `true` ou `false` de acordo com a existência ou não dos caracteres. Confira os exemplos a seguir:

```
<script>
const palavra = "saladas"
const posicao1 = palavra.indexOf("a")      // retorna 1
const posicao2 = palavra.lastIndexOf("a")  // retorna 5
const posicao3 = palavra.indexOf("sal")    // retorna 0
const posicao4 = palavra.indexOf("e")      // retorna -1
const existe = palavra.includes("d")       // retorna true
</script>
```

Assim como nos vetores, os métodos `indexOf()` e `lastIndexOf()`, retornam a posição da primeira ocorrência do caractere pesquisado. Vamos construir um exemplo de aplicação desses métodos e também do método `substr()` visto na seção anterior? O programa “Nome no Crachá” deve ler o nome completo de um participante em um evento e exibir apenas o seu nome e sobrenome. A Figura 7.5 exemplifica uma entrada e saída desse programa.



Figura 7.5 – Programa “Nome no Crachá” utiliza os métodos discutidos nas seções 7.3 e 7.4.

Exemplo 7.4 – Código HTML do Programa Nome no Crachá (ex7_4.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa E-mail Institucional</h1>
<form>
  <p>Funcionário:
    <input type="text" id="inFuncionario" autofocus required>
  </p>
  <input type="submit" value="Gerar E-mail">
</form>
<h3></h3>
<script src="js/ex7_5.js"></script>
<!-- /body e /html -->
```

A página HTML contém as tags já utilizadas em outros exemplos. No programa JavaScript, descrito a seguir, é apresentada uma das formas de obter o primeiro e o último nome do participante para gerar a sua credencial no evento. Note que, ao obter o nome informado pelo usuário na página web, foram retirados os espaços em branco do início e final da string – a partir do método `trim()`. E que uma validação é adicionada para verificar se o nome não possui espaços (entre os nomes).

Programa JavaScript que exibe primeiro e último nome (js/ex7_4.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  // obtém o nome informado e retira espaços em branco do início e final da string
  const nome = frm.inNome.value.trim()

  if (!nome.includes(" ")) { // se o nome não (!) possuir espaço
    alert("Informe o nome completo... ")
    return
  }
```

```

const priEspaco = nome.indexOf(" ") // posição do primeiro espaço
const ultEspaco = nome.lastIndexOf(" ") // posição do último espaço
// copia nome e sobrenome usando os parâmetros do substr()
const cracha = nome.substr(0, priEspaco) + nome.substr(ultEspaco)

resp.innerHTML = `Crachá: ${cracha}` // exibe a resposta
})

```

Com o uso do método `indexOf()`, identificamos a posição do primeiro espaço no nome do participante. Já o `lastIndexOf()` é utilizado para obter a posição do último espaço. Em seguida, a variável `cracha` recebe duas substrings: a primeira da posição inicial do nome, com a quantidade de caracteres indicados em `priEspaco`, e a segunda da posição do último espaço em diante.

7.5 Dividir a string em elementos de vetor

Um método a que recorremos com frequência quando se faz necessário trabalhar com strings é o `split()`. Ele converte a string em elementos de vetor a cada ocorrência de um determinado caractere. Observe o conteúdo da variável `sabores` a seguir e um exemplo de uso do método `split()`.

```

const sabores = "calabresa, 4 queijos, atum, frango"
const partes = sabores.split(",")

```

Como uma vírgula (",") foi passada como parâmetro para o método `split()`, o vetor `partes` vai conter os seguintes elementos:

```

// partes[0] = "calabresa"
// partes[1] = " 4 queijos"
// partes[2] = " atum"
// partes[3] = " frango"

```

Isso torna esse método útil e aplicável a uma série de operações que podem ser realizadas sobre uma cadeia de caracteres. Vamos construir um exemplo para resolver o seguinte problema: uma empresa necessita de um programa que gere um e-mail institucional para todos os seus funcionários. O e-mail deve ser formado pelas letras iniciais do nome do funcionário e de seu sobrenome, seguido pelo "@empresa.com.br". A Figura 7.6 ilustra um exemplo de execução desse programa.



Figura 7.6 – Programa gera um e-mail com as iniciais do nome e o sobrenome do funcionário.

Exemplo 7.5 – Código HTML do Programa E-mail Institucional (ex7_5.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Programa E-mail Institucional</h1>
<form>
<p>Funcionário:<br/>
<input type="text" id="inFuncionario" autofocus required>
</p>
<input type="submit" value="Gerar E-mail">
</form>
<h3></h3>
<script src="js/ex7_5.js"></script>
<!-- /body e /html -->
```

Programa JavaScript que gera e-mail com iniciais e sobrenome (js/ex7_5.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  const funcionario = frm.inFuncionario.value // obtém nome do funcionário

  // divide o nome em itens de vetor, criados a cada ocorrência do espaço
  const partes = funcionario.split(" ")
  let email = "" // vai concatenar letras
  const tam = partes.length // obtém nº de itens do vetor partes
```

```

for (let i = 0; i < tam - 1; i++) { // percorre itens do vetor (exceto o último)
    email += partes[i].charAt(0) // e obtém a letra inicial de cada item
}

// concatena as letras iniciais com a última parte (sobrenome) e empresa
email += partes[tam - 1] + "@empresa.com.br"

resp.innerText = `E-mail: ${email.toLowerCase()}` // exibe e-mail em minúsculas
})

```

O programa `ex7_5.js` obtém o nome do funcionário e divide-o em elementos de vetor a partir da ocorrência do espaço. Dessa forma, o nome “Paulo Ricardo Costa”, do exemplo, passa a ocupar três posições em um vetor. Em seguida, a repetição criada com o comando `for` vai percorrer os elementos do vetor, exceto o último (por isso, o `for` repete enquanto `i < tam-1`). E, dentro da repetição, obtém-se a primeira letra de cada uma dessas partes. Ao final, concatena-se ao `email` a última parte do nome, ou seja, o sobrenome e o nome da empresa.

7.6 Validar senhas com o método `match()`

O método `match()` é ótimo para implementar uma política de senhas em um sistema. Com ele, é possível verificar a existência de letras maiúsculas, minúsculas, números e símbolos em uma string. Seu funcionamento utiliza o conceito de expressões regulares. Uma expressão regular contém um conjunto de caracteres que indicam um padrão a ser pesquisado. Esse assunto, expressões regulares, é amplo e existem diversos livros dedicados ao tema. Vamos utilizá-los de uma forma simples, porém adequada ao que pretendemos aqui: validar os caracteres que compõem uma senha.

A sintaxe básica do método `match()` a ser utilizada nesta seção está descrita a seguir:

```
const vetor = palavra.match(/[A-Z]/g)
```

`/[A-Z]/` é o padrão de expressão regular que se deseja encontrar na palavra. A opção `g` (global) indica que a pesquisa deve retornar todas as ocorrências dos caracteres na string. O retorno é um vetor contendo os elementos

encontrados ou null, caso o padrão não exista na string fornecida. Observe os seguintes exemplos:

```
<script>
const palavra = "#SenhA_123!"
const vetor1 = palavra.match(/[a-z]/g)    // e,n,h
const vetor2 = palavra.match(/[A-Z]/g)    // S,A
const vetor3 = palavra.match/[0-9]/g)     // 1,2,3
const vetor4 = palavra.match/\W/_/g)      // #,_!
const vetor5 = palavra.match/[T-Z]/g)     // null
</script>
```

Para vetor1, foram atribuídas as letras minúsculas da senha. Enquanto vetor2 ficou com as letras maiúsculas e vetor3, com os números. Já para vetor4, utilizou-se o metacaractere \w que possui um significado especial na expressão regular e que retorna os símbolos da string analisada. Como o underline “_” não é recuperado pelo \w, acrescentamos “_” (que significa ou “_”). Por último, vetor5 ficará com null, pois não há letras maiúsculas entre T e Z na senha avaliada.

Existem diversas outras opções para montar o padrão de pesquisa da expressão regular, como verificar se há palavras na string que iniciam ou terminam por determinados caracteres ou, então, quais os caracteres que estão fora de um determinado padrão. Porém, com os exemplos anteriormente apresentados, já é possível criar um programa que implemente uma política de senhas em relação as suas regras de composição.

Vamos, então, construir um exemplo de programa de validação de senhas? Suponha que, para ser válida, uma senha deva possuir as seguintes regras de composição: a) possuir entre 8 e 15 caracteres; b) possuir, no mínimo, 1 número; c) possuir, no mínimo, 1 letra minúscula; d) possuir, no mínimo, 2 letras maiúsculas; e) possuir, no mínimo, 1 símbolo. A Figura 7.7 ilustra o funcionamento desse programa.

Exemplo 7.6 – Código HTML do Programa Valida Senha (ex7_6.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->
```

```

```

```

<h1> Programa Valida Senha </h1>
<form>
  <p> Senha:
    <input type="text" id="inSenha">
  </p>
  <input type="submit" value="Verificar Validade">
</form>
<h3></h3>
<script src="js/ex7_6.js"></script>
<!-- /body e /html -->

```

A página HTML contém um campo `<input type="text" ...>` que permite visualizar o que é digitado. Você deve substituir `text` por `password` em campos de senhas para ofuscar os caracteres que são digitados. No exemplo, utilizamos `text` apenas para verificar o funcionamento do programa.

Programa JavaScript de validação da senha (js/ex7_6.js)

```

const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

frm.addEventListener("submit", (e) => {
  e.preventDefault()          // evita envio do form
  const senha = frm.inSenha.value // obtém senha informada pelo usuário
  const erros = []            // vetor com erros

  // verifica se o tamanho da senha é inválido
  if (senha.length < 8 || senha.length > 15) {
    erros.push("possuir entre 8 e 15 caracteres")
  }

  // verifica se não possui números
  if (senha.match(/\d/g) == null) {
    erros.push("possuir números (no mínimo, 1)")
  }

  // verifica se não possui letras minúsculas
  if (!senha.match(/[a-z]/g)) {
    erros.push("possuir letras minúsculas (no mínimo, 1)")
  }

  // verifica se não possui letras maiúsculas ou se possui apenas 1
  if (!senha.match(/[A-Z]/g) || senha.match(/[A-Z]/g).length == 1) {

```

```

    erros.push("possuir letras maiúsculas (no mínimo, 2)")
}

// verifica se não possui símbolos ou "_"
if (!senha.match(/[^W|_]/g)) {
    erros.push("possuir símbolos (no mínimo, 1)")
}

// se vetor está vazio (significa que não foram encontrados erros)
if (erros.length == 0) {
    resp.innerText = "Ok! Senha Válida"
} else {
    resp.innerText = `Erro... A senha deve ${erros.join(", ")}`
}
})

```

O programa JavaScript recupera a senha informada pelo usuário. Em seguida, cria um vetor que vai armazenar os erros encontrados. Caso um padrão de caracteres não seja localizado na senha, uma nova mensagem é inserida no vetor. No final, verifica-se o número de elementos do vetor. Caso seja 0, significa que a senha cumpre as regras de composição estabelecidas. Observe que podemos testar se o retorno do método é == null ou, então, se ele é false (!).



Figura 7.7 – Exemplo com mensagens de erro.

7.7 Substituição de caracteres

As expressões regulares também são utilizadas como parâmetro do método `replace()`, quando quisermos substituir um caractere (ou um conjunto de

caracteres) por outro em uma string. Por padrão, a substituição incide apenas sobre a primeira ocorrência do caractere na string. Com o uso de uma expressão regular, com a opção g (global) indicada, a troca ocorre em toda a string. Um detalhe importante é o de que a string mantém o seu conteúdo original. Apenas a variável que recebe o retorno do método ou o conteúdo que é apresentado pelo programa vai conter as substituições dos caracteres indicados. A sintaxe do método `replace()` é a seguinte:

```
const novaStr = str.replace(caracterePesquisado, novoCaractere)
```

Portanto, se a variável `senha` tiver o conteúdo indicado a seguir:

```
const senha = "ABACAD"
```

A execução do método `replace()` sobre essa variável vai produzir os seguintes novos conteúdos:

```
const senha1 = senha.replace("A", "X") // XBACAD
const senha2 = senha.replace(/A/g, "X") // XBXCXD
```

Nesses exemplos, com uma string sendo passada no caractere pesquisado, para `senha1`, a troca ocorre apenas para o primeiro “A”. Já para `senha2`, com a indicação de uma expressão regular com o g, a troca é de todas as letras “A” por “X”. Também podemos utilizar o método `replace()` para retirar um caractere de uma string. Acompanhe os exemplos:

```
const app = "App Controle Financeiro"
const app2 = app.replace(" ", "")           // AppControle Financeiro
const app3 = app.replace(/ /g, "")          // AppControleFinanceiro
const app4 = app.replace(/ /g, "").toLowerCase() // appcontrolefinanceiro
```

Novamente, na variável `app2`, quando uma string é passada como primeiro argumento do método `replace()`, a troca do espaço, ou melhor, a retirada do espaço, já que estamos trocando por um “”, ocorre apenas para o primeiro espaço na string. Para a variável `app3`, como passamos como argumento uma expressão regular com a opção g, a substituição ocorre em toda a string. Já para `app4`, a troca é seguida por uma conversão dos caracteres para letras minúsculas.

No Exercício 7.c, no final deste capítulo, que deve verificar se uma frase é um palíndromo, uma boa dica é utilizar o método `replace()` para retirar os espaços em branco de uma string em toda a frase antes de realizar a análise

dos caracteres.

7.8 Manipulação de datas

Para trabalhar com datas em JavaScript, podemos declarar uma variável que recebe a data atual. Ou melhor, declarar um objeto que recebe uma instância do objeto Date. Para realizar essa tarefa, utilize a instrução a seguir:

```
const hoje = new Date()
```

Em seguida, há diversos métodos específicos para a manipulação de datas. Alguns são utilizados para extrair partes da data, outros, para modificar as partes que compõem uma data. Também é possível realizar operações matemáticas envolvendo datas. Nesta seção, abordaremos esse assunto.

Os métodos `getDate()`, `getMonth()` e `getFullYear()` são utilizados para obter, respectivamente, o dia, o mês e o ano de uma data. Já os métodos `setDate()`, `setMonth()` e `setFullYear()` permitem alterar o dia, o mês e o ano da data. Vamos apresentar pequenos scripts para exemplificar a aplicação desses métodos. O script destacado a seguir exibe a data atual e o dia seguinte a essa data. Caso seja o último dia do mês, a linguagem se encarrega de calcular corretamente a nova data.

```
<script>
  const hoje = new Date()
  const amanha = new Date()
  const dia = amanha.getDate()
  amanha.setDate(dia + 1)
  console.log(`Hoje: ${hoje}\nAmanhã: ${amanha}`)
</script>
```

A saída no console desse script demonstra o formato padrão de uma data no JavaScript, que é a seguinte:

```
Hoje: Wed Jan 05 2022 20:37:51 GMT-0300 (Horário Padrão de Brasília)
Amanhã: Thu Jan 06 2022 20:37:51 GMT-0300 (Horário Padrão de Brasília)
```

Perceba que, para calcular a data de amanhã, primeiro criamos um objeto que recebe a data atual. Depois extraímos o dia dessa data. Então, modificamos o dia da data, atribuindo mais 1 ao valor da variável dia. É possível realizar um processo semelhante para obter o próximo mês e o

próximo ano.

Como o valor retornado pelos métodos `getDate()`, `getMonth()` e `getFullYear()` é um número, podemos naturalmente executar operações matemáticas sobre os dados extraídos de uma data. No script a seguir (`ex7_7.html`), extraímos o ano atual para calcular o ano de nascimento de uma pessoa.

Exemplo 7.7 – Programa que calcula ano de nascimento de uma pessoa (`ex7_7.html`)

```
<meta charset="utf-8">
<script>
const anoAtual = new Date().getFullYear()
const idade = prompt('Quantos anos você comemora em ${anoAtual}?')
const anoNasc = anoAtual - idade
alert('Ah... Então você nasceu em ${anoNasc}')
</script>
```

Já para calcular a diferença de dias entre duas datas, é necessário entender uma importante particularidade da linguagem JavaScript em relação aos objetos `Date`: as datas JavaScript são armazenadas internamente como um valor numérico. Assim, uma data também pode ser criada ou calculada a partir de um número expresso em milissegundos, a contar do dia 1 de janeiro de 1970.

Dessa forma, se quisermos obter a diferença de dias entre duas datas, podemos subtrair as datas e dividir o valor por 86400000, que é o número de milissegundos de um dia: 24 horas * 60 minutos * 60 segundos * 1000 milissegundos.

Vamos aplicar essa fórmula em um novo exemplo. Nosso programa deve ler a data de vencimento e o valor de uma conta. Caso a conta esteja em atraso, o programa deve calcular o valor da multa e dos juros a serem acrescentados ao valor total. O código HTML contém um novo tipo de campo HTML e outra forma de exibir uma resposta ao usuário. Analise o código da página `ex7_8.html` e, na sequência, vamos destacar esses detalhes.

Exemplo 7.8 – Código HTML do Programa Caixa da Loja (`ex7_8.html`)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->
```

```


<h1> Programa Caixa da Loja </h1>
<form>
  <p>Data de Vencimento:
    <input type="date" id="inDataVenc" autofocus required>
  </p>
  <p>Valor da Conta R$:
    <input type="number" id="inValor" min=0 step="0.01" class="alinha-direita" required>
    <input type="submit" value="Calcular Multa e Juros">
  </p>
  <p>Valor da Multa R$:
    <input type="number" id="outMultta" class="alinha-direita" readonly>
  </p>
  <p>Valor do Juros R$..:
    <input type="number" id="outJuros" class="alinha-direita" readonly>
  </p>
  <p>Total a Pagar R$...:
    <input type="number" id="outTotal" class="alinha-direita" readonly>
    <input type="reset" value="Nova Conta">
  </p>
</form>
<script src="js/ex7_8.js"></script>
<!-- /body e /html -->

```

Para receber a data no Programa 7.8, foi utilizada a tag `<input type="date">`, que foi acrescentada ao HTML 5 e permite a digitação ou seleção de uma data a partir de uma caixa que exibe um calendário. Há ainda outra mudança nesse código HTML em relação aos exemplos anteriores. Utilizamos campos `<input type="number">` para exibir a saída de dados. Assim, podemos aplicar a esses campos um estilo que exibe o conteúdo deles alinhado à direita. Repare que foi adicionado aos campos de saída o atributo `readonly`, que, como o nome sugere, torna esses campos espaços destinados apenas à leitura de dados.

Crie agora o programa `ex7_8.js` para verificar se uma conta está em atraso e simular o valor da multa e juros a serem aplicados sobre ela.

Programa JavaScript para calcular multa e juros de contas em atraso (js/ex7_8.js)

```

const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("h3")

```

```

const TAXA_MULTA = 2 / 100 // multa por atraso
const TAXA_JUROS = 0.33 / 100 // juros por dia de atraso

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  const dataVenc = frm.inDataVenc.value
  const valor = Number(frm.inValor.value)
  const hoje = new Date() // cria variáveis (instancia objetos)
  const vencto = new Date() // do tipo Date()

  const partes = dataVenc.split("-") // data vem no formato aaaa-mm-dd
  vencto.setDate(Number(partes[2]))
  vencto.setMonth(Number(partes[1]) - 1)
  vencto.setFullYear(Number(partes[0]))

  const atraso = hoje - vencto // calcula a diferença de dias entre datas (em ms)
  let multa = 0 // inicializa multa e juros com 0
  let juros = 0

  if (atraso > 0) { // se conta estiver em atraso ...
    // converte ms do atraso em dias (1 dia = 24h x 60min x 60seg x 1000ms: 86400000)
    const dias = atraso / 86400000
    multa = valor * TAXA_MULTA // calcula multa e juros
    juros = valor * TAXA_JUROS * dias
  }
  const total = valor + multa + juros // calcula o total

  frm.outMultia.value = multa.toFixed(2) // exibe os valores com 2 decimais
  frm.outJuros.value = juros.toFixed(2)
  frm.outTotal.value = total.toFixed(2)
})

```

Vamos aos comentários sobre esse exemplo. Após criar referência aos elementos manipulados pelo programa (form e h3), definimos a taxa de multa e juros a ser aplicada sobre as contas em atraso e adicionamos um ouvinte para o evento submit do form. Na sequência, após obter a data de vencimento e valor da conta, instanciamos dois objetos do tipo Date(), um para manter a data atual e outro para conter a data de vencimento da conta a ser paga.

Usamos o método split() para dividir a data em partes e, então, modificamos

cada um dos atributos da data de vencimento. Depois subtraímos as duas datas (hoje – vencto), o que gera um número em milissegundos, o qual, dividido por 86400000 (número de milissegundos de um dia), retorna o número de dias entre as duas datas. Com ele, calculamos a multa e os juros a serem adicionados ao total que deve ser pago pelo cliente. A Figura 7.8 ilustra o funcionamento do programa.



Figura 7.8 – Cálculo da multa e juros de uma conta em atraso.

Para inserir uma nova conta, neste exemplo, não é necessário adicionar programação JavaScript ao código. Como as respostas são também exibidas em campos de formulário, o próprio `<input type="reset" ...>` realiza a tarefa de limpar todos os espaços do form.

Um cuidado especial com a manipulação de datas em JavaScript é necessário quando quisermos exibir o mês de uma data no formato dd/mm/aaaa. A variação do mês na linguagem JavaScript inicia em 0 e vai até 11. Para o dia e o ano, não há esse problema. Observe o seguinte script:

```
<script>
  const hoje = new Date()
  const dia = hoje.getDate()
  const mes = hoje.getMonth()
  const ano = hoje.getFullYear()
  console.log(hoje)
  console.log(`Data: ${dia}/${mes}/${ano}`)
</script>
```

A saída no console da execução desse script é:

Wed Jan 05 2022 21:52:58 GMT-0300 (Horário Padrão de Brasília)

Data: 5/0/2022

Portanto, para exibir o valor numérico do mês corretamente nesse formato, deve-se adicionar 1 à variável relativa ao mês de uma data. Mas e se quisermos exibir dia e mês com 2 dígitos? Nesse caso, podemos utilizar o método `padStart()`, já abordado em capítulos anteriores. Há um segundo parâmetro que pode ser informado no `padStart()`, que é o caractere a ser inserido para preencher o espaço. Contudo, o `padStart()` é aplicado sobre variáveis do tipo `String`. Então, precisamos inicialmente converter as variáveis `dia` e `mes`, que são do tipo `number` para `String`. Caso você esteja em dúvida sobre o tipo de uma variável, utilize o comando `typeof`. Observe a sintaxe desses comandos:

```
console.log(typeof dia)      // mostra o tipo da variável dia: number
const dia2 = dia.toString()  // converte dia para String
const dia3 = dia2.padStart(2, "0") // dia com 2 espaços: adiciona 0 antes se dia < 10
```

É possível converter o número para `String` e utilizar o método `padStart()` em uma mesma linha. O script a seguir exibe a data atual no formato `dd/mm/yyyy` com os dois dígitos para dia e mês:

```
<script>
  const hoje = new Date()
  const dia = hoje.getDate()
  const mes = hoje.getMonth() + 1
  const ano = hoje.getFullYear()
  const dia2 = dia.toString().padStart(2, "0")
  const mes2 = mes.toString().padStart(2, "0")
  console.log(`Data: ${dia2}/${mes2}/${ano}`)
</script>
```

Com os métodos vistos nesta seção, é possível definir uma data, modificar partes da data e realizar operações que envolvem datas. Essas são as tarefas de que geralmente necessitamos em um programa. Contudo, existem diversos outros métodos JavaScript que trabalham com datas. Esses métodos realizam as operações vistas aqui de outras formas, e, ainda, obtêm outras informações de uma data, como o dia da semana. Além disso, é possível trabalhar com horas. Você pode aprofundar-se nesse assunto, como também nos demais temas abordados no livro. A área de programação é um campo fértil para a realização de pesquisas, portanto, explore-a conforme a

necessidade dos programas que você vai desenvolver.

7.9 Strings e datas com Node.js

Vamos construir mais alguns exemplos de programas que exploram a manipulação de strings e datas em JavaScript, agora utilizando o Node.js. Você pode tentar resolvê-lo antes de ver o exemplo de resolução. É um bom treino para você avançar no seu processo de aprendizagem de Lógica de Programação. Crie a pasta nodejs e nela construa os seguintes Algoritmos:

a) Elaborar um programa que leia uma fórmula matemática e analise se os parênteses utilizados na fórmula estão corretos. A análise deve considerar dois fatores: o número de “(“ deve ser igual ao número de “)” e, ao ler a fórmula da esquerda para a direita, em nenhum momento, o número de “)” pode ser maior que o número de “(”, ou seja, não pode fechar um parêntese sem antes ter aberto.

Programa JavaScript para analisar os parênteses de uma fórmula matemática (nodejs/ex7_9.js)

```
const prompt = require("prompt-sync")() // pacote para entrada de dados
const formula = prompt("Fórmula: ") // lê a fórmula
let abre = 0 // contadores
let fecha = 0
for (const simbolo of formula) { // percorre os caracteres da fórmula
  if (simbolo == "(") {
    abre++
  } else if (simbolo == ")") {
    fecha++
  }
  // se, em algum momento, o número de fecha for maior que abre...
  if (fecha > abre) {
    break // ... sai da repetição
  }
}
if (abre == fecha) {
  console.log("Ok! Fórmula correta (em relação aos parênteses)")
} else {
  console.log("Erro... Fórmula incorreta")
}
```

Exemplo de execução do programa:

C:\livrojs\cap07\nodejs>**node ex7_9**

Fórmula: $((2+5)*4)/3$

Ok! Fórmula correta (em relação aos parênteses)

b) Elaborar um programa que leia a altura de uma árvore (número de linhas) e após exiba a árvore iniciando com 2 estrelas (asteriscos) e aumentando em 2 a cada linha. Fazer com que a árvore tenha uma margem esquerda fixa de 30 espaços e fique centralizada, conforme ilustra a execução do programa a seguir:

C:\livrojs\cap07\nodejs>**node ex7_10**

Altura da Árvore: 8

```
**
 ****
 *****
 ******
 *****
 *****
```

Programa JavaScript para exibir uma árvore de estrelas (nodejs/ex7_10.js)

```
const prompt = require("prompt-sync")() // pacote para entrada de dados
const altura = Number(prompt("Altura da Árvore: ")) // lê o número de linhas (altura)
console.log() // deixa uma linha em branco
for (let i=1; i<=altura; i++) { // inicia repetição
    const espacos = 30 + (altura - i) // calcula espaços do início
    console.log(" ".repeat(espacos) + "*".repeat(i*2)) // exibe cada linha
}
```

Você pode incrementar a nossa árvore. Uma sugestão é colocar um tronco com 2 estrelas por linha e com um comprimento que tenha relação com a altura da árvore. Talvez com a metade das linhas... Faça uns testes!

c) *Elaborar um programa que solicite um número de parcelas que devem ser geradas e calcule a data de cada uma dessas parcelas, uma para cada mês, a partir do mês seguinte ao atual, mantendo o dia atual. Observe o exemplo de execução desse programa, considerando que a data atual seja 5*

de janeiro de 2022.

```
C:\livrojs\cap07\nodejs>node ex7_11
Quantas Parcelas? 6
1a Parcela: 05/02/2022
2a Parcela: 05/03/2022
3a Parcela: 05/04/2022
4a Parcela: 05/05/2022
5a Parcela: 05/06/2022
6a Parcela: 05/07/2022
```

Programa JavaScript para gerar as datas de vencimentos de parcelas (nodejs/ex7_11.js)

```
const prompt = require("prompt-sync")() // pacote para entrada de dados
const parcelas = Number(prompt("Quantas Parcelas? "))
const data = new Date() // data atual
for (let i = 1; i <= parcelas; i++) {
  data.setMonth(data.getMonth() + 1) // aumenta um mês na data
  const dia = data.getDate()
  const mes = data.getMonth() + 1 // mês varia de 0 a 11, por isso, +1
  const ano = data.getFullYear()
  const diaZero = dia < 10 ? "0" + dia : dia // acrescenta 0 se dia menor que 10
  const mesZero = mes < 10 ? "0" + mes : mes // acrescenta 0 se mês menor que 10
  console.log(`#${i}ª Parcela: ${diaZero}/${mesZero}/${ano}`)
}
```

Observe que utilizamos um operador ternário para acrescentar “0” antes de dias ou meses menores que 10. Em um exemplo anterior, recorremos ao `padStart()`. Duas alternativas para fazer esse ajuste no formato de datas em JavaScript.

7.10 Exercícios

E, como nos capítulos anteriores, vou disponibilizar mais alguns exercícios sobre os assuntos abordados neste capítulo. Você pode resolvê-los a partir da criação de uma página HTML e do programa JavaScript que irá interagir com ela (como ilustrado nas figuras a seguir), ou a partir do prompt de comandos, com o Node.js.

a) Você deve desenvolver um programa de criptografia para transmitir

mensagens entre amigos. O programa deve ler uma mensagem e, em seguida, exibi-la criptografada. A criptografia consiste em: a) exibir todas as letras das posições pares da mensagem; b) exibir todas as letras das posições ímpares da mensagem. A Figura 7.9 exibe a mensagem criptografada. O programa deve conter ainda um botão para decriptografar a mensagem, ou seja, retornar a mensagem original a partir do texto cifrado.



Figura 7.9 – Exemplo da mensagem criptografada.

b) Uma palavra ou frase é um palíndromo quando pode ser lida nos dois sentidos, como RADAR, MUSSUM, ABBA. Elaborar um programa que leia uma frase e informe se ela é ou não um palíndromo (converter a frase para caixa alta). A Figura 7.10 apresenta uma frase que é um palíndromo.

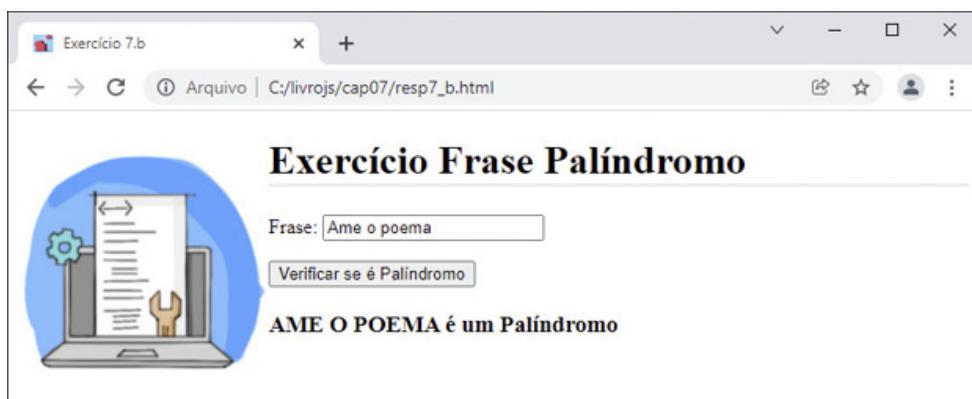


Figura 7.10 – Um palíndromo pode ser lido nos dois sentidos.

c) Suponha que o prazo para o pagamento de uma infração de trânsito com desconto seja de 90 dias. Elaborar um programa que leia a data de uma

infração e o valor da multa. Informe qual a data limite do pagamento com desconto (até 90 dias) e o valor a ser pago até essa data (com 20% de desconto). A Figura 7.11 ilustra o funcionamento do programa.



Figura 7.11 – Programa multa de trânsito deve utilizar os métodos de manipulação de datas.

7.11 Considerações finais do capítulo

A manipulação de strings e as operações envolvendo datas são tarefas importantes e frequentemente requeridas no desenvolvimento de sistemas. Gerar uma sugestão padrão de e-mail para os funcionários de uma empresa a partir de alguns caracteres do seu nome ou validar os caracteres que compõem uma senha, a fim de evitar o cadastro de senhas fracas, são tarefas relacionadas à manipulação de strings. Calcular a data de vencimento das parcelas de um financiamento ou, então, os dias de atraso de uma conta, a fim de determinar o valor da taxa de juros, são exemplos de operações relacionadas às variáveis do tipo data.

Neste capítulo, discutimos sobre os métodos que permitem realizar operações sobre strings e datas. Para as strings, as seguintes ações podem ser executadas:

- **Extraír cada um dos caracteres que compõem a string** – O método `charAt()` é responsável por retornar o caractere em uma posição específica da string. Portanto, se utilizado em conjunto com o comando

`for` e a propriedade `length` (tamanho do texto), o método `charAt()` permite percorrer todos os caracteres de uma palavra ou frase.

- **Converter os caracteres de uma palavra para letras maiúsculas ou minúsculas** – Os métodos `toUpperCase()` e `toLowerCase()` realizam, respectivamente, essas tarefas e podem ser aplicados sobre uma palavra ou um caractere da palavra.
- **Copiar e localizar caracteres na string** – A cópia, realizada pelo método `substr()`, deve conter a indicação da posição inicial e de quantos caracteres devem ser copiados. Caso a quantidade de caracteres não seja informada, todos os caracteres da string da posição indicada até o seu final são retornados. Já os métodos `indexOf()` e `lastIndexOf()` servem para retornar a posição de um ou mais caracteres pesquisados na string. A pesquisa pode ser do início em direção ao final do texto (`indexOf`) ou do final em direção ao início (`lastIndexOf`). Se os caracteres não forem encontrados na string, o valor -1 é retornado.
- **Dividir a string em elementos de vetor a partir da ocorrência de um caractere** – Esta ação é realizada pelo método `split()` e pode ser explicada a partir de um exemplo: considerando que a variável `nome = "Ana Maria Costa"`, executar `const partes = nome.split(" ")` fará com que o vetor `partes` contenha três elementos: `partes[0] = "Ana"; partes[1] = "Maria"; partes[2] = "Costa"`.
- **Pesquisar caracteres e substituir um ou mais caracteres na palavra** – Estas tarefas são realizadas, respectivamente, pelos métodos `match()` e `replace()`. Uma expressão regular, que define um padrão de pesquisa dos caracteres a serem localizados ou substituídos, pode ser utilizada nesses métodos. O método `match()` é especialmente útil para validar regras de composição de senhas, pois permite verificar a existência de letras maiúsculas, minúsculas, números e símbolos em uma string.

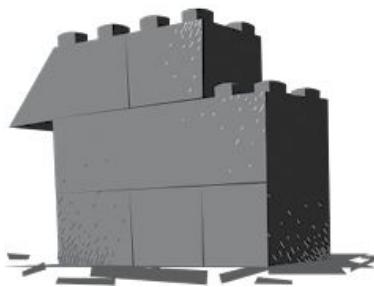
Já para as datas, existe outro conjunto particular de métodos. Nesta seção, exploramos os métodos `getDate()`, `getMonth()` e `getFullYear()` que recuperam, respectivamente, dia, mês e ano de uma data. Já os métodos `setDate()`, `setMonth()` e `setFullYear()`, por sua vez, servem para alterar cada uma das partes

que compõem uma data e permitem realizar cálculos como somar ou subtrair dias, meses e anos na data. Para criar um objeto do tipo Date, deve-se utilizar a instrução `const data = new Date()`.

Em tempo: Em recente release da ECMA, foi acrescentado ao JavaScript o método `at()`. Ele é semelhante ao `charAt()`, porém pode ser aplicado a strings e vetores. Sua vantagem é permitir o uso de índices negativos para acessar as últimas posições da string ou vetor. Assim, `palavra.at(-1)`, obtém a última letra da variável `palavra`.

CAPÍTULO 8

Funções e eventos



Neste capítulo, vamos avançar no processo de criação de funções em um programa e no uso dos eventos JavaScript. Lidamos com as funções e os eventos desde o Capítulo 2, porém na sua forma básica. Todas as programações foram associadas, no geral, ao evento submit do form a partir da declaração de uma arrow function (função de seta). É possível criar funções com passagem de parâmetros, criar funções anônimas, fazer uma função devolver um determinado valor ou um conjunto de valores e, ainda, utilizar uma função como um módulo que contém um trecho de código que se repete e que pode ser acionado em vários pontos do programa. Assim, cada vez que necessitamos executar esse trecho de código, chamamos o módulo, sem a necessidade de reescrever seus comandos.

Os eventos também podem ser mais bem explorados, pois há vários outros além do submit e click trabalhados até aqui nos exemplos do livro, como change, blur, focus, keypress... Eles ocorrem e podem ser programados quando, por exemplo, o usuário trocar um campo de seleção, sair de um campo de edição, posicionar o cursor em um elemento do formulário, ou, então, pressionar uma determinada tecla no preenchimento de um campo. Vamos verificar como podemos controlar melhor a execução de um programa ao adicionar ações associadas a esses eventos nos exemplos deste capítulo.

Sobre o processo de modularização de programas, é importante destacar um aspecto que vai facilitar a compreensão desse recurso: os módulos servem para melhor organizar nosso código, permitir a reutilização de trechos de programa e facilitar a sua leitura e manutenção. Ou seja, de certa forma, modifica o foco em relação ao que vimos até aqui, pois nossos programas foram evoluindo em complexidade, mas sempre com recursos essenciais para a resolução de um problema do cliente. Agora, nosso objetivo continua sendo resolver um problema a partir de um sistema computacional, porém com o foco em produzir códigos mais fáceis de manipular e passíveis de reaproveitamento.

Vamos continuar nessa discussão. Para exibir uma mensagem de “Aprovado” ou “Reprovado”, é essencial que o programa contenha uma rotina condicional. Para calcular e exibir o valor das parcelas de um financiamento, seja qual for o número delas, é fundamental criar uma rotina de repetição. Para manipular uma lista, é necessário armazená-la em um vetor. Entretanto, o acréscimo dos módulos em um programa não se apresenta com uma utilidade tão clara como no uso das condições, repetições e vetores. Principalmente para aqueles que estão ingressando no mundo da codificação de programas e inserindo esse recurso em pequenos algoritmos.

Contudo, o uso da modularização é igualmente essencial para a organização do programa. Um exemplo: você precisa desenvolver um sistema para uma empresa que deve cadastrar os clientes, funcionários e estagiários. Em todos esses cadastros, deve-se validar o CPF da pessoa. Se você desenvolver uma função para validar o CPF, ela poderá ser utilizada nos três cadastros desse sistema. Basta criar a função que receba como parâmetro um número de CPF e testá-la. Uma vez concluída, ela poderá ser utilizada nesses cadastros e em outros sistemas que você vai desenvolver.

As funções são também chamadas de métodos, procedures e módulos – com pequenas variações de definição dependendo da linguagem e do paradigma de programação utilizado.

8.1 Functions e Arrow Functions

A partir do momento em que nossos programas vão crescendo em tamanho e complexidade, dividi-los em blocos de código menores, ou seja, em funções, apresenta inúmeras vantagens. Pode-se destacar a possibilidade de reaproveitamento de código, a melhor organização e maior facilidade em entender um problema grande dividindo-o em blocos menores (dividir para conquistar), além dos benefícios relacionados ao trabalho em equipe.

As próprias linguagens de programação são organizadas sobre essa lógica: há funções (métodos) para, por exemplo, verificar se um determinado conteúdo consta em um vetor, converter os caracteres de uma string em maiúsculas e classificar em ordem crescente uma lista de nomes. Imagine se você, toda vez que necessitasse ordenar uma lista de produtos a serem exibidos em um programa, tivesse de fazer isso comparando item por item? Melhor chamar o método que realiza esse processo e que já foi testado inúmeras vezes.

Uma função em JavaScript pode ser construída com a palavra reservada `function` ou como uma declaração de constante – usando uma notação conhecida como arrow function (função de seta), em que a função é atribuída para uma variável. Observe a sintaxe das duas formas:

```
<script>
  function ola() {
    alert("Olá. Seja muito bem-vindo!")
  }
  ola()

  const ola2 = () => {
    alert("Olá. Seja muito bem-vindo, novamente!")
  }
  ola2()
</script>
```

As vantagens da segunda forma estão relacionadas à proteção dada às `const` em JavaScript e a uma sintaxe mais curta – que não necessita de um `return` – em funções que podem ser construídas com uma única atribuição. Se, por exemplo, executarmos um programa que contenha a declaração de duas

`functions` com o mesmo nome, o programa irá rodar, sem acusar erro. Caso isso aconteça a partir da declaração de `const`, a linguagem alerta para o erro no código.

Contudo, caso você se sinta mais confortável declarando os seus módulos de código com `function`, não há problema. Há várias discussões na web sobre o tema. Alguns preferem continuar com `function` – pois ela permite uma leitura mais intuitiva do código. Outros preferem `const` por permitir uma sintaxe mais curta e com a proteção citada no parágrafo anterior. Então, saiba que há justificativas para a sua escolha.

Como pode ser observado no exemplo, para chamar (executar) uma função, apenas escreva o nome da `function` no programa.

8.2 Funções com passagem de parâmetros

Nós já utilizamos funções com passagem de parâmetros desde o primeiro capítulo deste livro. Na função/método `alert()`, deve-se passar por parâmetro a mensagem a ser exibida. Ou seja, um parâmetro é uma informação enviada para uma função no momento em que necessitamos chamá-la. Os parâmetros permitem ampliar as funcionalidades da função. Imaginem se tivéssemos uma função para exibir “Muito Obrigado！”, e outra para exibir “Por favor, digite corretamente os dados”. Precisaríamos de inúmeras funções no programa. Em vez disso, passamos a mensagem a ser exibida pela função no momento em que a acionamos no programa. Chamar a função consiste em uma linha de código com o nome da função com os parâmetros inseridos dentro dos parênteses. Observe novamente a sintaxe do `alert()`:

```
// chama o método alert() passando o texto "Muito Obrigado!"  
alert("Muito Obrigado!")
```

Vamos construir um primeiro exemplo, para realizar uma tarefa simples, usando a passagem de parâmetros. Neste capítulo, também vamos desenvolver pequenos scripts para exemplificar os temas abordados e alguns programas maiores. Crie a estrutura de pastas como nos capítulos anteriores (`cap08` e, dentro dela, `css`, `img` e `js`).

Exemplo 8.1 – Exemplo de função com passagem de parâmetro (ex8_1.html)

```
<script>
const situacao = (nota, media) => {
  if (nota >= media) {
    alert("Aprovado")
  } else {
    alert("Reprovado")
  }
}
const prova1 = Number(prompt("Qual Nota: "))

situacao(prova1, 7) // chama a função situacao passando 2 parâmetros
</script>
```

Observe que a função situacao() recebe os parâmetros nota e media – que devem estar entre parênteses após o sinal de “=” . O programa inicia pela leitura da nota do aluno. Em seguida, é realizada uma chamada da função. O conteúdo da variável prova1 e o valor "7" são então passados para a função. Nela, esses valores são atribuídos para nota e media.

É comum criarmos funções com parâmetros contendo valores default (por falta), ou seja, se esse parâmetro não for informado, o valor padrão é atribuído para essa variável. A média, por exemplo, poderia conter o valor default 7. Assim, a função pode receber apenas nota para todas as avaliações em que a média é 7. Para avaliações com média diferente de 7, deve-se, então, informar os 2 parâmetros. A forma de indicar um valor padrão para um parâmetro é:

```
const situacao = (nota, media = 7) => { ... }
```

Os termos parâmetro e argumento são utilizados para denominar as variáveis passadas no momento da chamada da função. Há uma pequena diferença entre eles: os nomes das variáveis (nota e media) são chamados de parâmetros, já os valores reais desses parâmetros (o valor da nota1 e o valor 7) são chamados de argumentos da função. Contudo, no geral, os termos parâmetros e argumentos são utilizados indistintamente.

8.3 Funções com retorno de valor

No Exemplo 8.1, a função recebeu parâmetros e então apresentou dentro da função uma mensagem ao usuário. Contudo, nossas funções se tornam mais úteis se retornarem um valor, pois dessa forma o programa que chamou a função define o que deseja realizar com o conteúdo retornado. Para fazer uma função retornar um valor, utiliza-se o comando `return` seguido do conteúdo de retorno. Para que a função do Exemplo 8.1 retorne um valor indicativo da situação do aluno, poderíamos modificá-la da seguinte forma:

```
const situacao = (nota, media) => {
    const resultado = nota >= media ? "Aprovado" : "Reprovado"
    return resultado
}
```

Também poderíamos utilizar o operador condicional na função original do Exemplo 8.1. Observe, porém, que a função recebe os parâmetros e com base neles define o conteúdo da variável `resultado`. Então, ocorre o retorno do conteúdo dessa variável. Agora é o programa principal que define o que fará com o `resultado`. Ele pode atribuir esse retorno a uma variável e, em seguida, exibi-la na página, como nas linhas de código a seguir:

```
const aluno1 = situacao(prova1, 7)
resp.innerHTML = `Situação: ${aluno1}`
```

Ou, então, utilizar o próprio retorno da função como parte de um cálculo ou como parâmetro de outro método. Observe a linha de código a seguir, onde a função é chamada e seu retorno é utilizado para compor a string a ser exibida pelo `alert()`:

```
alert(`A situação do aluno é: ${situacao(prova1, 7)}`)
```

Como destacado anteriormente, `functions` com uma única atribuição – declaradas com `const` – podem omitir o `return`. O valor atribuído a `const` é retornado por ela. Nossa função pode, então, ser reduzida a:

```
const situacao = (nota, media) => (nota >= media ? "Aprovado" : "Reprovado")
```

Os parênteses depois da seta são opcionais. Caso a função contenha um único parâmetro, os parênteses envolvendo esse parâmetro também são opcionais.

Vamos agora construir um exemplo com uso de funções com retorno de valor. Nosso programa será para a Revenda Avenida. Ele deve ler modelo,

ano de fabricação e preço do veículo. Em seguida, o programa deve classificar o veículo como: “Novo” (do ano atual), “Seminovo” (até 2 anos de uso) ou “Usado”. Também deve apresentar o valor da entrada e o saldo em 10x (sem juros). A entrada deve ser de 50% para veículos novos ou de 30% para veículos classificados como seminovos ou usados. Para a classificação e o cálculo da entrada, serão utilizadas funções com retorno de valor. A Figura 8.1 ilustra a página com um veículo para exemplificar os dados de entrada e saída do programa.



Figura 8.1 – Programa usa funções com passagem e retorno de parâmetros.

Começamos pela criação do arquivo com os estilos que serão utilizados nos exemplos deste capítulo. Salve esse arquivo na pasta `css`, com o nome `estilos.css`.

```
h1 { border-bottom-style: inset; }
pre { font-size: 1.2em; }
img.normal { float: left; height: 300px; width: 300px; }
img.alta { float: left; height: 420px; width: 300px; }
span { margin-left: 70px; }
select { width: 150px; }
.detalhes { width: 380px; }
.oculta { display: none; }
.exibe { display: inline; }
```

O código HTML do programa mantém o padrão dos demais exemplos.

Exemplo 8.2 – Programa vai classificar e calcular valor

da entrada e parcelas (ex8_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Revenda Avenida - Promoção</h1>
<form>
  <p>Modelo do Veículo:
    <input type="text" id="inModelo" required autofocus>
  </p>
  <p>Ano de Fabricação:
    <input type="number" id="inAno" required>
  </p>
  <p>Preço R$:
    <input type="number" min="0" step="0.01" id="inPreco" required>
  </p>
  <input type="submit" value="Classificar - Calcular Entrada e Parcelas">
</form>
<h3 id="outResp1"></h3>
<h3 id="outResp2"></h3>
<h3 id="outResp3"></h3>
<script src="js/ex8_2.js"></script>
<!-- /body e /html -->
```

Já o código JavaScript contém as funções que recebem parâmetros e devolvem valores para quem as chamou. Vamos dividi-la em três partes para destacar melhor esse recurso de programação. Na primeira parte, o programa contém o código que captura os elementos da página e define a programação associada ao submit do form.

Código JavaScript que chama funções com retorno de valor (js/ex8_2.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp1 = document.querySelector("#outResp1")
const resp2 = document.querySelector("#outResp2")
const resp3 = document.querySelector("#outResp3")

frm.addEventListener("submit", (e) => {
  e.preventDefault()          // evita envio do form
  const modelo = frm.inModelo.value // obtém o conteúdo dos campos
  const ano = Number(frm.inAno.value)
  const preco = Number(frm.inPreco.value)
  const classificacao = classificarVeiculo(ano) // chama funções e atribui
```

```

const entrada = calcularEntrada(preco, classificacao) // ... retorno às variáveis
const parcela = (preco - entrada) / 10 // usa retorno da função para cálculo

resp1.innerText = modelo + " - " + classificacao // exibe as respostas
resp2.innerText = `Entrada R$: ${entrada.toFixed(2)}`
resp3.innerText = `+10x de R$: ${parcela.toFixed(2)}`
})

```

Observe que, após obter os dados do form, são feitas chamadas às funções `classificarVeiculo()` e `calcularEntrada()` com parâmetros sendo passados. O retorno dessas funções é atribuído para as variáveis `classificacao` e `entrada` que, na sequência, são exibidas na página. Ou seja, nós delegamos partes das tarefas desse programa para funções. Vamos agora conferir a programação de cada uma dessas funções.

```

// função recebe o ano do veículo como parâmetro
const classificarVeiculo = (ano) => {
  const anoAtual = new Date().getFullYear() // obtém o ano atual
  let classif
  if (ano == anoAtual) { // condições para definir classificação do veículo
    classif = "Novo"
  } else if (ano == anoAtual - 1 || ano == anoAtual - 2) {
    classif = "Seminovo"
  } else {
    classif = "Usado"
  }
  return classif // retorna a classificação
}

```

A função `classificarVeiculo()` recebe um ano como parâmetro. Esse valor deve ser passado na chamada da função. A partir dele, realizam-se as comparações para atribuir à variável `classif` a indicação de que o veículo é novo, seminovo ou usado (segundo uma classificação hipotética do exemplo). No final, é necessário retornar essa variável para o programa que chamou a função. Esse valor será então atribuído no programa principal para a variável `classificacao`.

Vamos agora destacar a programação da função `calcularEntrada()`, descrita a seguir:

```

// função recebe valor e status do veículo como parâmetro
const calcularEntrada = (valor, status) =>

```

```
status == "Novo" ? valor * 0.5 : valor * 0.3
```

Observe que agora são passados dois parâmetros: `valor` e `status`. Utilizei esses nomes para destacar que os nomes dos parâmetros não precisam ser iguais aos nomes das variáveis do programa principal. E, como o retorno pode ser indicado a partir de uma única atribuição, não é necessário utilizar a palavra `return`.

Você poderia perguntar: mas não seria mais fácil realizar esse cálculo ou a classificação do veículo no programa principal? Talvez, mas, ao dividir as tarefas de um programa em blocos menores, é possível obter as vantagens destacadas no início deste capítulo: facilidade de compreensão, manutenção e reutilização de código. Pensando na ideia de um sistema maior, se quisermos em algum outro ponto desse sistema exibir a classificação de um veículo novamente, não seria necessário refazer as condições. Basta chamar a função que realiza essa tarefa.

8.4 Funções com parâmetros Rest

Lembram do operador Rest (...) utilizado para unir um conjunto de elementos em um vetor? Ele também pode ser utilizado nas funções para receber um conjunto de parâmetros, que são convertidos para um vetor. Observe o exemplo a seguir:

Exemplo 8.3 – Função com uso dos parâmetros Rest (ex8_3.html)

```
<script>
const calcularMedia = (...notas) => {
  const num = notas.length // notas é um array
  if (num == 0) {
    console.log("Informe, no mínimo, uma nota")
    return
  }
  let soma = 0 // vai acumular a soma das notas
  for (const nota of notas) {
    soma += nota // soma o valor dos argumentos
  }
  const media = soma / num // calcula a media
  console.log(`Média: ${media.toFixed(1)}`)
}
```

```

}

// exemplos de chamada de calcularMedia() com nº de parâmetros diferentes
calcularMedia(6, 7, 8)      // Média: 7.0
calcularMedia(2, 10)        // Média: 6.0
calcularMedia(7.5, 10, 8, 9.5) // Média: 8.8
calcularMedia()            // Informe, no mínimo, uma nota
</script>

```

A função inicia com um teste que exibe uma mensagem caso o número de argumentos seja igual a zero. Como o operador Rest cria um vetor com os parâmetros passados para a função, pode-se utilizar a propriedade `length` para recuperar o tamanho do vetor. Em seguida, é possível utilizar uma estrutura de repetição com o comando `for..of` para obter o valor de cada elemento passado na chamada da função. Observe que realizamos no programa várias chamadas à função, todas com diferente número de argumentos. Nos comentários, o valor exibido em cada chamada.

Esse processo de enviar vários parâmetros para serem trabalhados em uma função também pode ser realizado a partir da palavra reservada `arguments`. Com `arguments` é possível ter acesso a cada um dos argumentos passados à função – igualmente manipulados como elementos de vetor.

8.5 Funções anônimas

As funções anônimas permitem definir a programação de um bloco de código sem atribuir um nome para a função. Nos programas desenvolvidos nos capítulos anteriores, essa sintaxe das arrow functions já estava presente, geralmente associadas ao evento `submit` do form. O Exemplo 2.1, por exemplo, contém o seguinte código:

```

frm.addEventListener("submit", (e) => {
  const nome = frm.inNome.value
  resp.innerText = `Olá ${nome}`
  e.preventDefault()
})

```

Ou seja, ao clicar no botão `submit` do form é executada uma função anônima que recebe “`e`” como parâmetro, seguida pela notação de uma arrow function. Vamos utilizar o método `SetInterval()` para construir um novo

exemplo de uso para as funções anônimas. O método `SetInterval()` faz uma chamada de função a cada intervalo de tempo, indicado em milissegundos.

```
const mostraHora = () => {
  const data = new Date()
  const hora = data.getHours()
  const min = data.getMinutes()
  const seg = data.getSeconds()
  console.log(`Atenção para o horário: ${hora}:${min}:${seg}`)
}
setInterval(mostraHora, 5000)
```

Então, se você executar o código anterior, vai receber no seu console uma nova mensagem a cada 5 segundos:

```
Atenção para o horário: 11:43:25
Atenção para o horário: 11:43:30
Atenção para o horário: 11:43:35
....
```

Podemos construir esse mesmo script, utilizando uma função anônima, da seguinte forma:

```
setInterval(() => {
  const data = new Date()
  const hora = data.getHours()
  const min = data.getMinutes()
  const seg = data.getSeconds()
  console.log(`Atenção para o horário: ${hora}:${min}:${seg}`)
}, 5000)
```

Outro método JavaScript que permite executar funções com um controle de tempo é o `setTimeout()`. Ele será exemplificado no Capítulo 13, e difere de `setInterval()`, por executar uma função após alguns milissegundos, porém apenas uma única vez.

Ainda há vários outros recursos sobre funções que podem ser explorados. É possível fazer uma função retornar um conjunto de valores a partir do uso de vetores, passar objetos por referência, ou seja, que alteram o conteúdo dos parâmetros em memória refletindo-os no restante do programa, criar uma função dentro de outra função ou, então, fazer uma função chamar a si mesma (recursividade), recursos um pouco mais avançados sobre esse tema. Pesquise sobre eles, à medida que a complexidade dos sistemas que você

for desenvolver também avançar.

8.6 Eventos JavaScript

Um evento é a ocorrência de uma ação, geralmente produzida por um usuário, em uma página. Clicar em um botão, selecionar um item, sair de um campo, pressionar uma tecla, passar o mouse sobre uma imagem, redimensionar a página são alguns dos eventos que podem ser controlados em um sistema. Adicionar programação nas páginas web associadas à ocorrência dos diversos eventos JavaScript permite criar maior interatividade com o usuário, dando maior dinamismo à página. A partir da programação dos eventos, é possível, por exemplo, trocar uma imagem quando o usuário modifica a seleção de um item em uma lista de botões (Exemplo 9.1), exibir mensagens de advertência quando o usuário sai de um campo de edição com um conteúdo inválido ou, então, executar uma ação vinculada ao pressionamento de uma determinada tecla, além de vários outros.

A lista de eventos JavaScript passíveis de programação é grande. Eles podem estar relacionados com eventos de interface do usuário (`load`, `unload`, `resize`), eventos de mouse (`click`, `dblclick`, `mouseover`), eventos de teclado (`keypress`, `keydown`, `keyup`) ou eventos de formulário (`change`, `focus`, `blur`). A sintaxe para definir a programação de um evento é semelhante à utilizada a partir do segundo capítulo deste livro, modificando apenas o nome do evento que ficará associado a uma função ou que define a execução de uma função anônima.

O Exemplo 8.4 explora a programação de novos eventos (além do `submit`) visando dar maior interatividade a um sistema de controle de pedidos de uma pizzaria. Imagine que a página deva substituir o bloco de pedidos de um garçom, que, então, vai utilizar um tablet ou smartphone para anotar o pedido de cada cliente. O acesso se dará a partir do navegador do aparelho. A página, portanto, deve conter recursos que facilitem o atendimento para o garçom. Um desses recursos é que a lista de itens deve conter apenas pizzas ou apenas bebidas, conforme a seleção inicial do tipo de item. Ou seja,

quando ocorrer a troca entre pizza e bebidas, o conteúdo da lista de itens do pedido deve ser modificado. A Figura 8.2 apresenta esse recurso, em que a troca para bebidas modificou os itens do campo select.



Figura 8.2 – Quando ocorre a troca entre Pizza e Bebidas, os itens do campo select são modificados.

Outro recurso importante para o sistema, que pode ser implementado a partir da programação dos eventos JavaScript, é o do exibir uma dica quando o usuário posicionar no campo **Detalhes do Item**. A dica deve conter o número máximo de sabores da pizza, de acordo com o tamanho desta, selecionado no campo “Item”. A mensagem deve ser exibida no próprio campo de edição a partir da propriedade placeholder, que é um texto apresentado no campo e que desaparece quando o usuário inicia a digitação. A Figura 8.3 ilustra a página no momento em que o campo **Detalhes do Item** recebe o foco. Ao sair do campo, deve-se limpar o conteúdo dessa propriedade.



Figura 8.3 – A dica é alterada ao posicionar no campo de edição, conforme o tamanho da pizza.

Exemplo 8.4 – Código HTML da página da Pizzaria Avenida (ex8_4.html)

```
<!-- doctype, html, head e body (conf. exemplo 4.2) -->

<h1>Pizzaria Avenida - Controle de Pedidos</h1>
<form>
  <p>Item do Pedido:<br/>
    <input type="radio" name="produto" id="rbPizza" checked autofocus> Pizza
    <input type="radio" name="produto" id="rbBebida"> Bebida
    <span>Item:</span>
    <select id="inPizza">
      <option value="media">Pizza Média</option>
      <option value="grande">Pizza Grande</option>
      <option value="familia">Pizza Família</option>
    </select>
    <select id="inBebida" class="oculta">
      <option value="refri">Refrigerante Litro</option>
      <option value="suco">Jarra de Suco</option>
      <option value="agua">Água Mineral</option>
    </select>
  </p>
  <p>Detalhes do Item:<br/>
    <input type="text" id="inDetalhes" class="detalhes">
    <input type="submit" value="Adicionar">
  </p>
</form>
<pre></pre>
```

```
<script src="js/ex8_4.js"></script>
<!-- /body e /html -->
```

Um detalhe importante nesse código HTML é que criamos dois campos do tipo select: um contendo opções de pizza e outro contendo opções de bebida. Contudo, observe que a tag `<select id="inBebida" ...>` contém o atributo `class="oculta"`. E, no arquivo `estilos.css`, está a estilização dessa classe, com a declaração `"display: none;"`. Dessa forma, essa lista não é exibida quando a página é carregada.

Vamos então aos comentários sobre o código JavaScript, que será dividido em partes, de acordo com o evento programado.

Código JavaScript para o Controle de Pedidos (`js/ex8_4.js`)

```
const frm = document.querySelector("form") // obtém elementos da página
const resp = document.querySelector("pre")

const itens = [] // vetor global para armazenar os itens do pedido

frm.rbPizza.addEventListener("click", () => { // quando radio button é clicado
  frm.inBebida.className = "oculta" // oculta select das bebidas
  frm.inPizza.className = "exibe" // exibe select das pizzas
})

frm.rbBebida.addEventListener("click", () => { // quando radio button é clicado
  frm.inPizza.className = "oculta" // oculta select das pizzas
  frm.inBebida.className = "exibe" // exibe select das bebidas
})
```

O programa começa pela criação de referências aos elementos da página e pela declaração de um vetor global que irá armazenar os itens do pedido. Na sequência, são adicionadas duas funções anônimas associadas ao evento click dos radio buttons. Ou seja, quando o garçom trocar de pizza para bebida, ou vice-versa, a função será executada. E o que cada uma delas faz? Modifica o estilo dos selects, tornando um deles oculto e o outro, visível. A propriedade `className` indica o estilo, definido no arquivo `estilos.css`, a ser aplicado sobre o elemento.

O próximo trecho de código do arquivo `ex8_4.js` está relacionado ao campo

“Detalhes do Item”. Adicione os seguintes comandos nesse programa.

```
frm.inDetalhes.addEventListener("focus", () => { // quando campo recebe o foco
  if (frm.rbPizza.checked) { // se radiobutton rbPizza estiver marcado
    const pizza = frm.inPizza.value // obtém value do item selecionado
    // uso do operador ternário, para indicar o número de sabores
    const num = pizza == "media" ? 2 : pizza == "grande" ? 3 : 4
    // atributo placeholder exibe uma dica de preenchimento do campo
    frm.inDetalhes.placeholder = `Até ${num} sabores`
  }
})

frm.inDetalhes.addEventListener("blur", () => { // quando campo perde o foco
  frm.inDetalhes.placeholder = "" // limpa a dica de preenchimento
})
```

Dois novos eventos foram programados nesse código: focus e blur. O evento blur ocorre toda vez que o campo `inDetalhes` recebe o foco. Ele verifica se o campo do tipo `radiobutton` das pizzas está marcado. Se estiver, vamos obter o `value` do item selecionado, que poderá ser médio, grande ou família. Com o operador condicional, que cria uma condição em uma linha, atribuímos para a variável `num` o número máximo de sabores da pizza. Esse valor é, então, exibido pela propriedade `placeholder` do campo.

Perceba que adicionamos dois “ouvintes” para o campo `inDetalhes`: um para o evento `focus` (que ocorre quando o campo recebe o foco) e outro para o evento `blur` (que ocorre quando o campo perde o foco). No `blur`, é retirada a dica de preenchimento (importante para quando o garçom sair do campo e voltar ao `radiobutton`, por exemplo).

Na sequência, destacamos a função principal do programa, associada ao evento `submit` do `form`. Nela, criamos um `if` que verifica se o `rbPizza` está marcado. Se não estiver, significa que `rbBebida` está (pois campos do tipo `radiobutton` têm essa característica de manter uma opção selecionada – uma vez que um deles foi inicialmente marcado). Acompanhe os comandos utilizados no restante dessa função (anônima):

```
frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita envio do form
  let produto
  if (frm.rbPizza.checked) {
```

```

const num = frm.inPizza.selectedIndex // obtém nº do item selecionado
produto = frm.inPizza.options[num].text // texto do item selecionado
} else {
  const num = frm.inBebida.selectedIndex
  produto = frm.inBebida.options[num].text
}
const detalhes = frm.inDetalhes.value      // conteúdo do inDetalhes
itens.push(produto + " (" + detalhes + ")") // adiciona ao vetor
resp.innerText = itens.join("\n")          // exibe os itens do pedido

frm.reset() // limpa o form
frm.rbPizza.dispatchEvent(new Event("click")) // dispara click em rbPizza
})

```

Para obter o conteúdo do campo `select`, utilizamos duas propriedades: uma retorna o número do item selecionado (`selectedIndex`) e outra o texto contido nesse item (`options[num].text`). O conteúdo é então adicionado ao vetor e exibido no elemento que está abaixo do form na página (`pre`).

Observe que, após exibir os itens do pedido, o conteúdo do form é limpo e é disparado um evento `click` para o radio button `pizza`. Isso faz com que a programação associada ao `click` desse botão seja executada.

Dessa forma, concluímos as functions que realizam a inserção de itens para o controle de pedidos de uma pizzaria. Esses dados não ficam salvos no navegador do aparelho utilizado pelo garçom. Ah... esse será então o assunto do nosso próximo capítulo!

Este livro, como indicado no capítulo inicial, não pretende ser uma referência da linguagem JavaScript, então não vamos apresentar uma listagem com todos os eventos da linguagem. O objetivo desta secção foi demonstrar os benefícios de programar diversos eventos em um programa, destacando exemplos de interação proporcionados pela execução de rotinas acionadas quando alguns desses eventos ocorrem. Nos demais exemplos do livro, em especial os do Capítulo 11, continuaremos a explorar os eventos e as funções com passagem e retorno de parâmetros.

Funções com passagem e retorno de valores são um tema muito importante no universo da programação de computadores. Elas são igualmente importantes na Programação Orientada a Objetos, um paradigma de

programação essencial para o desenvolvimento de sistemas, e que você deve ter como meta de estudo após o aprendizado das técnicas elementares de programação discutidas neste livro. Vamos agora ver um exemplo de funções com o Node.js e desenvolver alguns exercícios sobre os assuntos destacados neste capítulo.

8.7 Funções com Node.js

O programa destacado a seguir explora o uso de funções para gerenciar um vetor de objetos. Trata-se de um cadastro de vinhos. Vamos construir e explicar cada parte desse programa.

Código JavaScript do Programa Cadastro de Vinhos (nodejs/ex8_5.js)

```
const prompt = require("prompt-sync")()

const vinhos = []

function titulo(texto) { // recebe, por parâmetro, o texto a ser exibido
    console.log()
    console.log(texto)
    console.log("=".repeat(40))
}

// Programa Principal
do {
    titulo("====< Cadastro de Vinhos >====")
    console.log("1. Inclusão de Vinhos")
    console.log("2. Listagem de Vinhos")
    console.log("3. Pesquisa por Tipo")
    console.log("4. Média e Destaques")
    console.log("5. Finalizar")
    const opcao = Number(prompt("Opção: "))
    if (opcao == 1) {
        incluir()
    } else if (opcao == 2) {
        listar()
    } else if (opcao == 3) {
        pesquisar()
    } else if (opcao == 4) {
```

```
    calcularMedia()
} else {
    break
}
} while (true)
```

No exemplo em Node.js, vamos utilizar a sintaxe tradicional de declaração de funções (function). Inicialmente, importamos o pacote prompt-sync e declaramos o array vinhos. Após, é criada a função titulo, que será utilizada em todo o programa. Ela recebe um texto a ser exibido no início do programa principal e em cada função. Como destacado anteriormente, as funções servem para auxiliar o programador na elaboração do programa. Assim, em vez de escrever essas três linhas no programa principal e nas funções, basta chamar a função titulo, com o texto a ser exibido. Outra vantagem é que, se quisermos exibir essa mensagem inicial de modo diferente, com o texto em letras maiúsculas, por exemplo, é necessário alterar um único local do código, ou seja, na própria função e todos os cabeçalhos do programa passam a ser exibidos nesse padrão.

Após a função titulo, temos o programa principal, que exibe as opções disponíveis no menu e faz uma chamada para cada função correspondente. Observe que temos um laço de repetição que faz com que as opções sejam novamente exibidas após a execução de cada função. Para sair desse laço, o usuário deve informar a opção correspondente ao finalizar, ou melhor, qualquer valor fora do intervalo de 1 ao 4. Você pode acrescentar um else if (opção == 5), para executar o break e outro else com uma mensagem de “Opção Inválida”, caso prefira.

Vamos agora apresentar cada uma das funções previstas pelo programa. Elas geralmente são inseridas antes do programa principal. Começamos pela function incluir().

```
function incluir() {
    titulo("====< Inclusão de Vinhos >====")

    const marca = prompt("Marca...: ") // lê os dados do vinho
    const tipo = prompt("Tipo....: ")
    const preco = Number(prompt("Preço R$: "))
```

```
    vinhos.push({marca, tipo, preco}) // insere um objeto no vetor
    console.log("Ok! Vinho cadastrado com sucesso")
}
```

Compete à função incluir receber os dados do vinho a ser cadastrado, adicionar um novo objeto no vetor vinhos e exibir a mensagem confirmando a inserção. Observe inicialmente a chamada à função titulo, que serve para destacar essa rotina no programa.

A próxima função do programa é a listar, cujo código é apresentado a seguir:

```
function listar() {
  titulo("====< Lista de Vinhos Cadastrados >===")
  console.log("Marca..... Tipo..... Preço R$:")

  // percorre o vetor para exibir todos os vinhos
  for (const vinho of vinhos) {
    console.log(` ${vinho.marca.padEnd(20)} ${vinho.tipo.padEnd(20)} ` +
      `${vinho.preco.toFixed(2).padStart(9)})`)
  }
}
```

Com a inclusão e a listagem concluídas, você já pode testar essas rotinas no seu programa. A seguir a exemplificação do funcionamento da rotina de inclusão de vinhos:

```
C:\livrojs\cap08\nodejs>node ex8_5
```

```
====< Cadastro de Vinhos >===
=====
1. Inclusão de Vinhos
2. Listagem de Vinhos
3. Pesquisa por Tipo
4. Média e Destaques
5. Finalizar
Opção: 1
```

```
====< Inclusão de Vinhos >===
=====
Marca...: Garibaldi
Tipo....: Tinto Suave
Preço R$: 15.90
Ok! Vinho cadastrado com sucesso
```

Após realizar a inclusão, o programa exibe novamente o menu principal. Ao selecionar a opção 2, a listagem é exibida (com mais alguns vinhos cadastrados).

```
====< Lista de Vinhos Cadastrados >=====
=====
Marca..... Tipo..... Preço R$:
Garibaldi Tinto Suave 15.90
Country Wine Branco Suave 16.90
Jota Pe Tinto Seco 20.90
Sangue de Boi Tinto Suave 14.90
Chalise Branco Seco 19.90
```

Vamos destacar a seguir a função pesquisar, que deve receber um tipo de vinho e listar apenas os vinhos desse tipo. Caso não haja vinhos do tipo solicitado pelo usuário, o programa deve exibir mensagem.

```
function pesquisar() {
    titulo("====< Pesquisa por Tipo de Vinho >====")

    const pesq = prompt("Tipo: ") // lê o tipo do vinho a pesquisar

    let contador = 0 // contador para verificar se existe
    console.log("Marca..... Tipo..... Preço R$:")

    for (const vinho of vinhos) {
        if (vinho.tipo.toUpperCase().includes(pesq.toUpperCase())) {
            console.log(`${vinho.marca.padEnd(20)} ${vinho.tipo.padEnd(20)} ` +
                `${vinho.preco.toFixed(2).padStart(9)}`)
            contador++ // se entrou no if, incrementa o contador
        }
    }

    // se percorreu todos os vinhos e contador continua == 0, significa que não há
    if (contador == 0) {
        console.log(`Obs.: Não há vinhos cadastrados do tipo "${tipo}"`)
    }
}
```

A seguir, a execução da função pesquisar. Observe que, como foi utilizado o método includes(), a palavra pesquisada pode constar apenas em parte do conteúdo cadastrado. Além disso, como os dados são convertidos para maiúsculas na comparação, independe do formato digitado pelo usuário.

```
====< Pesquisa por Tipo de Vinho >====  
=====
```

Tipo: branco

Marca..... Tipo..... Preço R\$:

Country Wine Branco Suave 16.90

Chalise Branco Seco 19.90

Para concluir este programa vamos destacar a função que calcula a média e exibe o vinho de maior e de menor valor do cadastro. Para obter esses valores extremos, utilizamos o método `sort`, para classificar os vinhos por ordem de preço. Dessa forma, o primeiro vinho será o de menor valor e o último, o de maior.

```
function calcularMedia() {  
    titulo("====< Média e Destaques do Cadastro de Vinhos >====")  
  
    const num = vinhos.length // obtém número de elementos do vetor  
    if (num == 0) {  
        console.log(`Obs.: Não há vinhos cadastrados`)  
        return  
    }  
  
    let total = 0 // para acumular o total  
    for (const vinho of vinhos) {  
        total += vinho.preco  
    }  
  
    const media = total / num // calcula o preço médio  
  
    const vinhos2 = [...vinhos] // cria uma cópia do vetor original  
  
    vinhos2.sort((a, b) => a.preco - b.preco) // ordena por preço  
  
    const menor = vinhos2[0] // menor preço é o primeiro (posição 0)  
    const maior = vinhos2[num-1] // maior preço é o último (posição num-1)  
  
    console.log(`Preço Médio dos Vinhos R$: ${media.toFixed(2)}`)  
    console.log(`Menor Valor R$: ${menor.preco.toFixed(2)} - ${menor.marca}`)  
    console.log(`Maior Valor R$: ${maior.preco.toFixed(2)} - ${maior.marca}`)  
}
```

A seguir, a execução desta rotina (opção 4):

```
====< Média e Destaques do Cadastro de Vinhos >====
```

=====

Preço Médio dos Vinhos R\$: 17.70
Menor Valor R\$: 14.90 – Sangue de Boi
Maior Valor R\$: 20.90 – Jota Pe

Com este exemplo, concluímos os conteúdos geralmente abordados na disciplina de Algoritmos e Lógica de Programação. A forma de realizar esses Algoritmos em Node.js se assemelha às formas tradicionalmente utilizadas em outras Linguagens, como C, Java, Pascal ou, até mesmo, o Português Estruturado. Os próximos capítulos focam em recursos envolvendo a programação JavaScript para construção de páginas web interativas.

8.8 Exercícios

Os exercícios a seguir devem ler alguns dados e exibir uma resposta. Neles, proponho a criação de funções com passagem de parâmetros. Lembre-se de que os benefícios mais visíveis da modularização de programas serão percebidos em sistemas maiores. No entanto, para que possamos empregar essas técnicas em sistemas maiores, devemos treiná-las em pequenos programas. Portanto, mãos na massa... ou melhor no teclado!

a) Elaborar um programa que leia o nome e a idade de um atleta de um clube de natação. O programa deve exibir o nome com “-” abaixo das letras do nome e a categoria do atleta, que pode ser “Infantil” (até 12 anos), “Juvenil” (entre 13 e 18 anos) ou “Adulto” (acima de 18 anos). O programa deve conter as funções:

- retornarTracos() – que receba um nome como parâmetro e retorne uma linha com “-” para as letras do nome (nos espaços, manter os espaços).
- categorizarAluno() – que receba um número como parâmetro e retorne a categoria do aluno, conforme indicação no enunciado do exercício.

A Figura 8.4 ilustra os dados de entrada e saída do programa.

Figura 8.4 – Os traços devem ser exibidos apenas abaixo das letras do nome.

b) Elaborar um programa que leia o nome completo de um aluno. O programa deve validar o preenchimento de um nome completo e exibir a senha inicial do aluno, a qual será o sobrenome seguido pelo número de vogais do nome completo dele. O programa deve conter as funções:

- validarNome() – que receba um nome como parâmetro e retorne true (nome completo) ou false (nome incompleto).
- obterSobrenome() – que receba um nome como parâmetro e retorne o último nome do aluno em letras minúsculas.
- contarVogais() – que receba um nome e retorne o número de vogais deste, com dois dígitos.

A Figura 8.5 destaca um exemplo de execução desse exercício.

Figura 8.5 – Funções retornam os dados para exibir a resposta do programa.

c) Elaborar um programa para uma veterinária, o qual leia o preço de uma vacina e se o cliente possui ou não convênio. Caso possua algum convênio, exibir uma caixa de seleção com os convênios “Amigo dos Animais” e “Saúde Animal”. O programa deve exibir o valor do desconto (10% sem convênio; 20% para “Amigo dos Animais”; 50% para “Saúde Animal”) e o valor a ser pago. Criar a função:

- calcularDesconto() – que receba os parâmetros valor e taxa de desconto. Retornar o valor do desconto.

A Figura 8.6 exibe a página para um cliente que possui convênio. Observe que a caixa de seleção não deve ser exibida no início do programa.

The screenshot shows a web browser window titled "Exercício 8.c". The address bar indicates the file is located at "C:/livrojs/cap08/resp8_c.html". The main content is a form for a veterinary clinic named "Veterinária Avenida". On the left, there is a graphic of a laptop displaying a document with a gear icon. The form fields include:

- "Valor da Vacina:" input field containing "140.00"
- "Possui Convênio:" radio button group where "Sim" is selected (indicated by a blue border)
- "Convênio:" dropdown menu showing "Saúde Animal"
- "Calcular Desconto" button
- "Desconto RS: 70.00" displayed below the button
- "A Pagar RS: 70.00" displayed below the discount amount

Figura 8.6 – Campo select com os convênios deve ser exibido apenas ao marcar “Sim” no radio button.

8.9 Considerações finais do capítulo

Dominar o processo de construção de funções com passagem de parâmetros e retorno de valor é um importante avanço para quem deseja tornar-se um profissional na área da programação de computadores. Desenvolver programas com o uso de funções organiza melhor o código, facilita o entendimento e, consequentemente, a manutenção do programa, além de

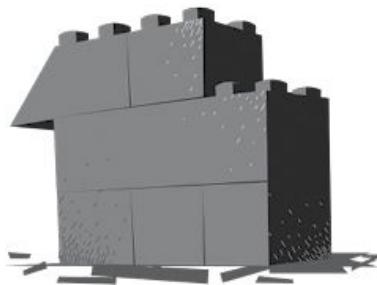
permitir a reutilização de módulos já desenvolvidos. Dessa forma, uma função criada e testada uma vez pode ser novamente utilizada em outra parte do programa. Isso também economiza tempo, produz sistemas mais confiáveis e permite gerenciar melhor o trabalho em equipe. Esses benefícios, contudo, evidenciam-se de uma forma mais clara em sistemas maiores. Em pequenos exemplos, é necessário sermos criativos para pensar quais funções poderiam ser criadas para melhor organizar o código.

Os parâmetros consistem em informações que devem ser fornecidas no momento em que realizamos a chamada da função. Utilizá-los torna a função mais ampla. Imagine uma função para o cálculo de um desconto em um sistema de folha de pagamento. Essa função poderia calcular e retornar o valor do desconto de 10% do salário base da empresa. Nesse caso, seu uso se limitaria aos funcionários que recebem o salário base e têm o desconto de 10%. Porém, se a função realizar o cálculo do desconto recebendo o valor do salário e a taxa de desconto como parâmetros, ela se torna muito mais útil, podendo ser aplicada no cálculo do desconto de todos os funcionários da empresa.

Já os eventos, além de serem a forma usual de acionar uma função em JavaScript, constituem um recurso capaz de oportunizar diversas formas de interação com os usuários de uma página web. Antes que um usuário clique no botão **Confirmar Compra** de um site de comércio eletrônico, por exemplo, várias outras ações foram por ele realizadas. É possível acrescentar “ouvintes” para essas ações nos elementos da página e chamar funções no momento em que cada uma dessas ações ocorrer. Pode-se exibir o produto com a cor selecionada pelo cliente quando ele trocar o item no campo cor ou exibir uma imagem ampliada do produto quando o usuário mover o mouse sobre uma figura. Isso faz com que o cliente se sinta mais bem atendido e pode ser um importante diferencial nas páginas que você vai desenvolver profissionalmente.

CAPÍTULO 9

Persistência de dados com localStorage



Nos exemplos e exercícios dos capítulos anteriores, principalmente naqueles que manipulavam listas de dados, foi possível perceber que as informações digitadas eram perdidas a cada atualização da página. Neste capítulo, veremos como fazer com que informações se tornem persistentes, ou seja, permaneçam salvas no navegador do usuário. Desse modo, mesmo quando o usuário fechar e abrir o navegador, ou até mesmo reiniciar o computador, sua lista de pacientes, veículos ou itens de um pedido (exemplos já utilizados no livro) poderá ser recuperada, evitando que todos os dados tenham de ser novamente digitados.

Uma das formas de fazer isso no HTML 5 é utilizando o localStorage. Com ele é possível, por exemplo, salvar os dados de um cliente de uma loja, como nome, idade ou clube pelo qual o cliente torce. Como as informações ficam salvas no navegador do cliente, quando ele retornar ao site da loja, é possível obter essas informações e personalizar a página de acordo com as escolhas anteriormente feitas por ele. Isso permite, por exemplo, exibir uma saudação de boas-vindas com o nome do cliente, apresentar na página principal a lista dos produtos em destaque de acordo com a faixa etária do cliente ou, ainda, personalizar a página com as cores do seu clube favorito.

Também poderíamos utilizar o localStorage para salvar a lista com os itens de um carrinho de compras do cliente, permitindo que ele rapidamente conclua o processo em outro momento, detalhe que pode se tornar um diferencial interessante para o site de uma empresa. No desenvolvimento de sites para empresas ou de sistemas para os mais variados fins, é de fundamental importância que vejamos cuidadosos com os detalhes. Tentar perceber uma possível necessidade do cliente pode ser um fator decisivo para o seu projeto de software ser bem-sucedido.

As linguagens de programação permitem salvar dados em arquivos texto e banco de dados. Em arquivos texto, no geral, as informações salvas são pequenos conjuntos de dados ou dados relacionados às configurações do sistema. Há também os arquivos nos formatos JSON e XML, frequentemente utilizados para troca de informações entre aplicativos. Já os bancos de dados são utilizados para armazenar as informações principais de um sistema (cadastro de clientes, produtos, vendas, entre outros) com diversos recursos que visam desde à manutenção da integridade das informações até controles avançados relacionados a questões de segurança. Com JavaScript, vamos simular neste capítulo como persistir dados utilizando o localStorage, o que, de certo modo, pode estar relacionado ao processo tradicional de salvar dados em arquivos texto.

Além do localStorage, há outras formas de salvar dados no navegador do usuário em JavaScript, como a partir do uso de cookies, sessionStorage, indexedDB e Web SQL. Optamos pelo localStorage por ser ele uma nova implementação do HTML 5, pela capacidade de armazenar uma quantidade maior de dados (até 5 MB, no geral) e também pela facilidade de uso.

Os métodos do localStorage vistos a seguir têm as mesmas funcionalidades se aplicados ao sessionStorage. Eles pertencem a Web Storage API (*Application Programming Interface*) do HTML 5, sendo que no localStorage os dados se mantêm persistentes até serem excluídos pelo usuário, enquanto no sessionStorage os dados se mantêm apenas por uma sessão (basicamente, enquanto o navegador estiver aberto).

9.1 Salvar e recuperar dados

Para salvar uma informação no navegador do usuário com o localStorage, devemos utilizar o método `setItem()`. Esse método contém dois parâmetros: chave (nome da variável) e valor (conteúdo da variável). Vamos utilizar um editor online JavaScript, como o site js.do, para os nossos primeiros exemplos deste capítulo. Acesse o endereço do site e limpe os códigos do programa de exemplo. Para salvar de forma persistente um conteúdo para a variável `idade`, digite as seguintes linhas:

```
<script>
  localStorage.setItem("idade", 17)
</script>
```

Também se pode utilizar a notação de objeto com o ponto para separar `localStorage` e o nome da variável (`localStorage.idade = 17`) ou ainda o formato de elementos de vetor (`localStorage['idade'] = 17`). Contudo, o uso do método `setItem()` é a forma recomendada.

Depois de digitar os comandos do script no editor online, clique em **Run code**. Os dados serão salvos no navegador; eles ficam armazenados até que uma ação do usuário ocorra no sentido de excluir ou alterar o seu conteúdo. Algo semelhante ao que ocorre com um arquivo de textos que você salvou no seu computador com o bloco de notas, por exemplo. Uma das formas de visualizar, alterar ou excluir esse dado é acessando o menu **Ferramentas do Desenvolvedor** no browser. No Google Chrome, selecione a guia **Application**, clique em **Local Storage** e, então, no endereço do site, conforme ilustra a Figura 9.1.

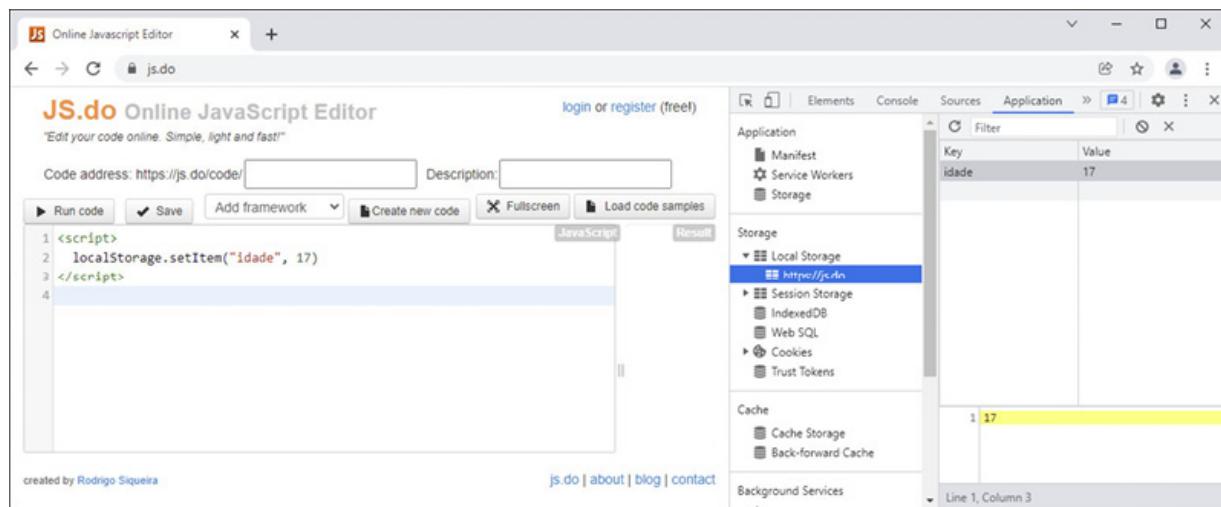


Figura 9.1 – Dados salvos no localStorage podem ser manipulados no navegador.

Um detalhe importante sobre esse processo é que os dados salvos pertencem ao navegador e ao domínio que armazenou os dados. Portanto, se você salvar informações utilizando um navegador e depois utilizar qualquer outro para acessar a mesma página, essas informações não serão recuperadas. O mesmo vale para endereços distintos. Se a loja X salvou um conjunto de dados de uma visita de um cliente, a loja Y não terá acesso a essas informações, mesmo que elas utilizem nomes idênticos de variáveis.

Outro detalhe é que o usuário pode remover “acidentalmente” os dados armazenados na Local Storage se limpar o histórico de navegação de seu browser. Portanto, não desenvolva programas para salvar dados relevantes dos usuários com esse recurso.

Para recuperar, bem como verificar a existência de um dado armazenado no navegador do usuário, devemos utilizar o método `getItem()` com o nome da chave utilizada.

```
const idade = localStorage.getItem("idade")
```

Os dados são salvos como strings. Portanto, cuidado com a execução de cálculos, principalmente de adição, sobre os dados que você armazenou no localStorage. Como vimos no Capítulo 1, se um conteúdo for do tipo string, a linguagem JavaScript concatena os dados para compor o resultado em uma adição. Faça o teste. Modifique o script anterior para recuperar o valor

armazenado no navegador e adicionar 1.

```
<script>
  const idade = localStorage.getItem("idade")
  const soma = idade + 1
  alert(soma)
</script>
```

O resultado exibido pelo método `alert()` é 171. A fim de resolver esse problema, devemos utilizar a `function Number()` para realizar a conversão. Nesse exemplo, podemos aplicar `Number()` na linha em que é feita a atribuição à variável `idade`, como a seguir:

```
const idade = Number(localStorage.getItem("idade"))
```

Vamos construir um programa exemplo para a aplicação dos métodos `setItem()` e `getItem()` em uma página de uma loja de esportes. Para isso, crie a pasta `cap09` e, dentro dela, as pastas `css`, `img` e `js`. Em seguida, crie o arquivo `estilos.css` na pasta `css` e informe as seguintes regras de estilização:

```
pre { font-size: 1.2em; }
.cores-Brasil { color: black; background-color: red; }
.cores-Pelotas { color: blue; background-color: yellow; }
.cores-Farroupilha { color: green; background-color: white; }
```

Porém, antes de digitar o código do primeiro programa do Capítulo 9, vamos conversar sobre um aliado importante que podemos utilizar na construção do layout de nossas páginas: o Bootstrap.

9.2 Uma “pitada” de Bootstrap

Como salientamos nos capítulos iniciais, não temos a intenção neste livro de destacar questões relacionadas ao design da página. Nossa foco é trabalhar os assuntos ligados à lógica de programação. Contudo, podemos melhorar a aparência de nossos exemplos de modo bastante simples com o uso de uma biblioteca ou framework de estilos CSS. Uma das bibliotecas de componentes front-end de maior destaque na atualidade é o Bootstrap, que é open source e pode ser utilizado gratuitamente em seus projetos.

Mas o que é uma biblioteca de estilos CSS? Desde o Capítulo 4, referenciamos um arquivo CSS contendo os estilos utilizados em cada

exemplo. Para as imagens, aplicamos um estilo de alinhamento flutuante à esquerda, para a linha de cabeçalho acrescentamos uma borda inferior no formato inset, já as fontes exibidas no elemento `pre` foram, geralmente, formatadas para um tamanho maior etc. Uma biblioteca de estilos CSS é basicamente um arquivo que contém um conjunto de estilos CSS prontos para serem aplicados em nossas páginas. Para utilizá-la, basta criar uma referência ao arquivo de estilos da biblioteca e indicar o nome da classe de cada elemento HTML a ser estilizado. Da mesma forma como em nossos exemplos, mas, agora, com referência aos nomes de classes definidas pelo framework.

Utilizar uma biblioteca de componentes front-end profissional traz inúmeras vantagens, como padronização das páginas, maior rapidez no desenvolvimento do código, maior facilidade de elaboração do layout do site, além da segurança de que estamos utilizando um modelo amplamente testado e que produz páginas responsivas no moderno conceito de *mobile first*.

Para utilizar o Bootstrap é necessário baixar os arquivos do site da biblioteca ou então referenciar um CDN (*Content Delivery Network* – Rede de Distribuição de Conteúdo) – um site onde os arquivos do framework estão disponíveis. Nos exemplos do livro, vamos utilizar a versão corrente do Bootstrap, que é a 5.1.3 e referenciar diretamente seu arquivo de estilos de um CDN – conforme o endereço indicado no site www.getbootstrap.com, o site oficial da biblioteca (Figura 9.2).

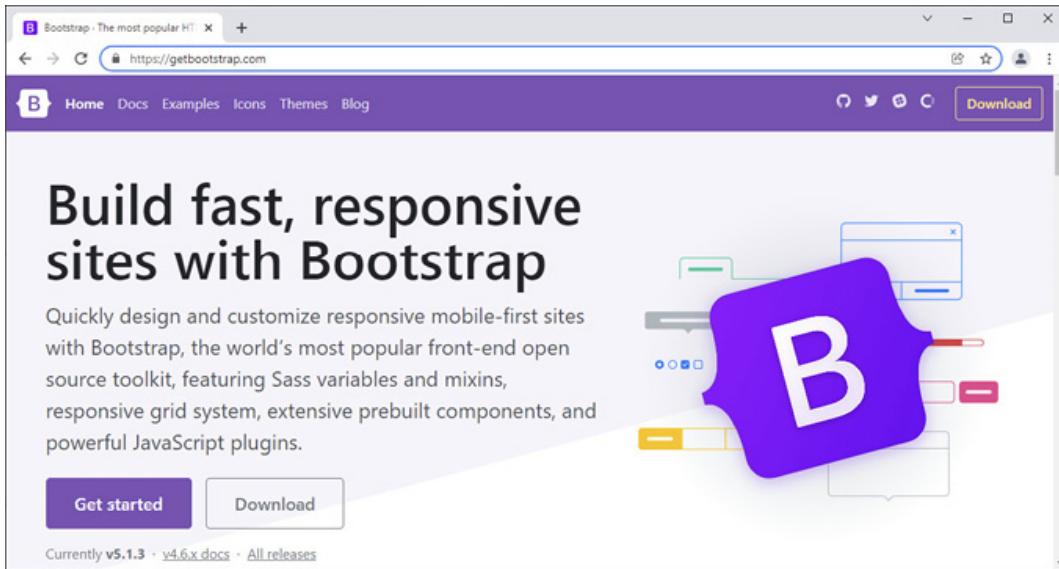


Figura 9.2 – Site oficial do Bootstrap.

O Bootstrap permite organizar o layout de uma página em *grids*. Assim, uma seção da página pode ser dividida em até 12 grids ou colunas. A soma dos tamanhos das colunas deve totalizar 12. Cada grid é criada com um elemento `div`, uma espécie de container, ou seja, uma “caixa” na qual diversos elementos da página podem ser inseridos. Essas “caixas” devem ficar dentro de um container maior e podem ainda conter novas subdivisões.

Vamos construir um primeiro exemplo e, à medida que esses elementos forem sendo utilizados, bem como, estilos de botões, tabelas e campos de formulário aplicados, vamos destacar a função de cada um deles.

Nosso exemplo é para a criação de uma página de uma loja de esportes. O cliente pode selecionar o clube pelo qual ele torce e essa seleção deve ficar salva no navegador. Conforme o clube, as cores da página (texto e cor de fundo) e o símbolo do clube são alterados. Observe o exemplo ilustrado na Figura 9.3. Naturalmente, você pode modificar esse programa inserindo os clubes de sua cidade ou estado.

O código HTML do Exemplo 9.1 destacado a seguir exibe a página da loja de esportes sem apresentar, inicialmente, a imagem do clube do cliente. Para ocultar a imagem, vamos utilizar a classe `d-none` do Bootstrap.

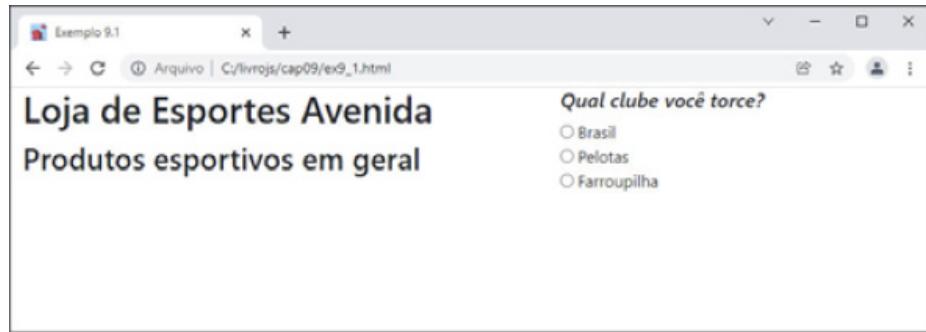


Figura 9.3 – Página inicial da loja de esportes com os estilos do Bootstrap inseridos.

Exemplo 9.1 – Código HTML do programa Loja de Esportes (ex9_1.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link rel="icon" href="img/livrojs.png">
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css">
<link rel="stylesheet" href="css/estilos.css">
<title>Exemplo 9.1</title>
</head>
<body>
<div class="container-fluid">
<div class="row" id="divTitulo">
<div class="col-sm-7">
<h1>Loja de Esportes Avenida</h1>
<h2>Produtos esportivos em geral</h2>
</div>
<div class="col-sm-3">
<h5>Qual clube você torce?</h5>
<form>
<input type="radio" name="clube" id="rbBrasil"> Brasil <br>
<input type="radio" name="clube" id="rbPelotas"> Pelotas <br>
<input type="radio" name="clube" id="rbFarroupilha"> Farroupilha
</form>
</div>
<div class="col-sm-2">
<img id="imgClube" class="d-none">
```

```

    </div>
</div>
</div>
<script src="js/ex9_1.js"></script>
</body>
</html>

```

Observe que no cabeçalho da página ex9_1.html acrescentamos a referência ao CDN do Bootstrap 5. Além dele, há também referência ao arquivo estilos.css, criado anteriormente, e que contém a estilização dos elementos utilizados nos exemplos deste capítulo.

No corpo da página do Exemplo 9.1, há uma div da classe container-fluid, que é a div onde todos os elementos dessa página são inseridos. Em seguida, é definida uma div que vai organizar uma seção da página que é identificada pelo id divTitulo. Dentro dela, são inseridas três colunas. A primeira coluna (col-sm-7) exibe o nome da loja, a segunda (col-sm-3) exibe uma lista com os campos de formulário do tipo radio contendo o nome dos clubes (da cidade da loja) e a última (col-sm-2) é reservada para exibir o símbolo do clube a ser selecionado pelo cliente. Perceba que a soma dos tamanhos das divs é 12 (col-sm-7, col-sm-3, col-sm-2).

Mas por que apenas a div class="row" é identificada por um nome? Porque ao trocar de clube vamos modificar o estilo dos elementos dessa div, que ocupa a parte superior da página. A Figura 9.4 apresenta a página com um dos clubes selecionados.

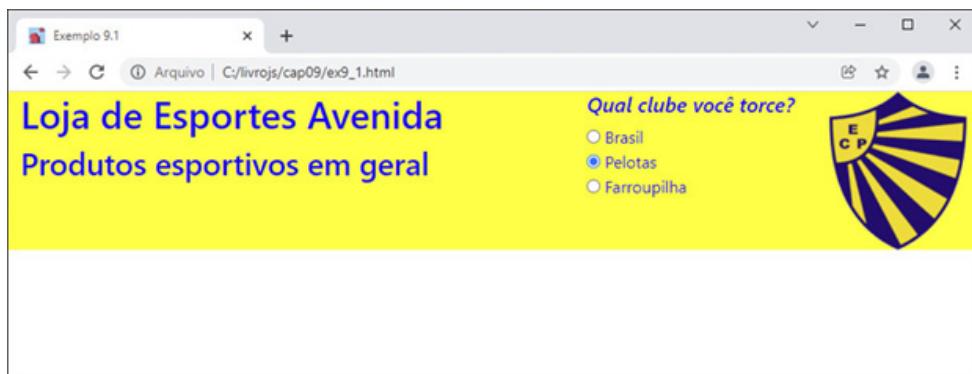


Figura 9.4 – Ao selecionar um clube, as cores de fundo, da fonte e o símbolo são alterados.

Para implementar as trocas de imagem e cores, bem como para salvar a

seleção do clube pelo qual o cliente torce, deve-se criar o código JavaScript contendo a função trocarClube descrita a seguir.

Código JavaScript do programa Loja de Esportes (js/ex9_1.js)

```
const frm = document.querySelector("form")
const imClube = document.querySelector("#imgClube")
const dvTitulo = document.querySelector("#divTitulo")

const trocarClube = () => {
    let clube // variável que irá receber o nome do clube

    if (frm.rbBrasil.checked) { // verifica qual radiobutton está selecionado
        clube = "Brasil"
    } else if (frm.rbPelotas.checked) {
        clube = "Pelotas"
    } else {
        clube = "Farroupilha"
    }

    // define as classes de dvTitulo: row e cores do clube
    dvTitulo.className = `row cores-${clube}`

    // modifica a imagem de acordo com a seleção do cliente
    imClube.src = `img/${clube.toLowerCase()}.png`
    imClube.className = "img-fluid" // muda o estilo para exibir a imagem
    imClube.alt = `Símbolo do ${clube}` // modifica atributo alt

    localStorage.setItem("clube", clube) // salva no navegador a escolha do cliente
}

// associa ao evento change de cada botão do form a função trocarClube
frm.rbBrasil.addEventListener("change", trocarClube)
frm.rbPelotas.addEventListener("change", trocarClube)
frm.rbFarroupilha.addEventListener("change", trocarClube)
```

Já para verificar se o cliente selecionou um clube em sua visita anterior ao site e exibir as cores e o símbolo do clube de acordo com o conteúdo salvo pelo método `setItem` anterior, deve-se adicionar o seguinte código (no mesmo arquivo `ex9_1.js`):

```
const verificarClube = () => {
```

```

if (localStorage.getItem("clube")) { // se já estiver salvo algum clube
    const clube = localStorage.getItem("clube") // obtém o nome do clube

    if (clube == "Brasil") { // conforme o clube, marca checked
        frm.rbBrasil.checked = true
    } else if (clube == "Pelotas") {
        frm.rbPelotas.checked = true
    } else {
        frm.rbFarroupilha.checked = true
    }
    trocarClube() // chama function que troca imagem e cores
}
}

// ao carregar a página, verifica se cliente já selecionou clube anteriormente
window.addEventListener("load", verificarClube)

```

Observe que utilizamos o método `getItem()` para inicialmente verificar a existência de uma variável salva no navegador com o nome `clube`. Caso exista, recuperamos o conteúdo dessa variável, também utilizando o `getItem()`. Conforme o conteúdo da variável, realizamos via programação a troca do clube selecionado nos campos de formulário do tipo `radio`. Após essa troca, a função `trocarClube()` é chamada. Ela vai, então, alterar a classe de estilos da `divTitulo` e do símbolo do clube na página.

9.3 Remover dados do `localStorage`

É importante adicionar em nossos programas uma opção que permita ao usuário remover os dados salvos no `localStorage`. Para realizar essa tarefa, a linguagem JavaScript dispõe dos métodos `removeItem()` e `clear()`. O método `removeItem()` é utilizado para remover o conteúdo de uma variável salva no domínio da página que o criou. Já o método `clear()`, por sua vez, remove todas as variáveis pertencentes a um domínio e armazenadas em seu navegador.

Por exemplo, se quisermos remover a idade salva no script inicial deste capítulo, devemos retornar ao site [w3schools.com](https://www.w3schools.com) e digitar o seguinte trecho de código:

```

<script>
    localStorage.removeItem("idade")

```

```
</script>
```

Você pode rodar o script depois de acessar as ferramentas do desenvolvedor no Google Chrome, clicar na guia **Application** e no endereço do site (w3schools.com) em **Local Storage**. Observe que a variável desaparece após a execução do script.

Vamos acrescentar no Exemplo 9.1 a opção **Nenhum** para caso o cliente queira retornar às cores iniciais exibidas pela página. Sugiro que você crie um novo arquivo HTML, copie o código do arquivo ex9_1.html e acrescente mais essa opção nas linhas do `input type="radio"`. Remova também as tags `<form>` e `</form>` do código HTML. Salve o arquivo com o nome ex9_2.html. As opções disponíveis devem então ficar conforme o trecho de código a seguir:

```
...
<input type="radio" name="clube" id="rbBrasil"> Brasil <br>
<input type="radio" name="clube" id="rbPelotas"> Pelotas <br>
<input type="radio" name="clube" id="rbFarroupilha"> Farroupilha <br>
<input type="radio" name="clube" id="rbNenhum" checked> Nenhum
...

```

Observe que o novo item adicionado contém o atributo `checked`. Ele indica que essa opção deve começar selecionada (para o caso da primeira visita ao site ou quando o cliente não selecionou algum clube).

Se na página HTML as alterações foram de apenas duas tags, no nosso programa JavaScript, as modificações serão várias. Como discutimos nos capítulos iniciais deste livro, há diversas maneiras de resolver um problema em termos de programação. Vamos demonstrar isso aqui. O programa ex9_2.js vai explorar um novo método para recuperar os elementos `input type="radio"`, sem a tag `form`, e utilizar comandos de repetição para selecionar os inputs do código HTML. Começamos discutindo sobre o método `querySelectorAll()`.

9.4 Uso do `querySelectorAll()`

Em todos os nossos programas, utilizamos o método `querySelector()` para recuperar e manipular os elementos da página HTML. Essa é a forma mais rápida de acessar um elemento, visto que o `id` deve ser único para cada

elemento da página. Porém, existem outros métodos que também podem ser utilizados e que, em alguns casos, se tornam mais práticos em termos de organização do código. Um deles é o método `querySelectorAll()`.

Como o nome sugere, esse método seleciona todos os elementos da página de uma determinada tag ou classe. Ele permite, por exemplo, acessar os parágrafos, as imagens ou, então, as linhas de cabeçalho do documento associados a um estilo. Os elementos são obtidos como itens de um vetor. Logo, é necessário indicar o índice do elemento a ser manipulado. Retorne ao site w3schools.com (ou outro editor JavaScript online) e digite os comandos a seguir, para exemplificar esse método.

```
<p> Exemplo </p>
<p> Capítulo 9 </p>
<script>
const p = document.querySelectorAll("p")
p[0].style.color = "blue"
p[1].style.color = "red"
</script>
```

Esse documento HTML contém dois parágrafos. No script, o método `querySelectorAll()` os recupera e, a partir de então, é possível acessá-los como itens de um vetor. Esses itens podem ser contados (a partir da propriedade `length`), manipulados em uma estrutura de repetição ou acessados individualmente com a indicação do índice do elemento que se quer referenciar. No exemplo anterior, a cor de cada parágrafo é alterada.

Também é possível limitar a busca pela referência ao container “pai” dos elementos que se pretende obter. Esse assunto será discutido no próximo capítulo, mas observe o exemplo para verificar a sintaxe a ser empregada a fim de capturar apenas alguns dos parágrafos do documento:

```
const dvResposta = document.querySelector("divResposta")
const p = dvResposta.querySelectorAll("p")
```

Ou seja, inicialmente se cria uma referência ao elemento “pai” (`divResposta`). Em seguida, obtêm-se os parágrafos “filhos” desse elemento. Portanto, apenas os parágrafos que estão dentro de `divResposta` serão recuperados.

Vamos então aos códigos do programa `ex9_2.js`. A captura dos elementos, bem como a associação do evento `change` à função `trocarClube`, pode ser feita da

seguinte forma:

```
const inRadios = document.querySelectorAll("input") // captura tags input da página
// percorre os elementos para associar function ao evento change
for (const inRadio of inRadios) {
    inRadio.addEventListener("change", trocarClube)
}
```

Observe que utilizamos o `querySelectorAll()` para obter os elementos `input` da página. Dessa forma, o método `addEventListener()` pode ser utilizado dentro de uma estrutura de repetição, na qual se define que cada item do `input type="radio"` vai chamar a função `trocarClube()`, quando houver uma mudança de clube por parte do usuário.

Mas vamos modificar também o conteúdo da função `trocarClube`. Agora para demonstrar outra forma de realizar a troca do nome dos clubes e do estilo da `divTitulo` e para exemplificar o método `removeItem()` da Local Storage.

Código JavaScript com opção para excluir dados do localStorage (js/ex9_2.js)

```
const trocarClube = () => {
    const clubes = ["Brasil", "Pelotas", "Farroupilha"] // vetor com a lista de clubes

    let selecao
    // percorre os inRadios para verificar qual está selecionado
    for (let i = 0; i < inRadios.length; i++) {
        if (inRadios[i].checked) {
            selecao = i // se selecionado, armazena índice do radio selecionado
            break      // e sai da repetição
        }
    }

    if (selecao <= 2) { // se selecao <= 2, então torce por algum clube
        dvTitulo.className = `row cores-${clubes[selecao]}` // modifica a cor
        // muda a propriedade src com a imagem do clube selecionado
        imClube.src = `img/${clubes[selecao].toLowerCase()}.png`
        imClube.className = "img-fluid" // muda estilo para exibir imagem
        imClube.alt = `Símbolo do ${clubes[selecao]}` // texto alternativo
        localStorage.setItem("clube", clubes[selecao]) // salva nome do clube
    } else {           // else (selecionou "Nenhum")
        dvTitulo.className = "row" // tira a classe de cor de divTitulo
        imClube.className = "d-none" // oculta a imagem
```

```

imClube.alt = ""           // limpa texto alternativo
localStorage.removeItem("clube") // remove variável do localStorage
}
}

```

Observe que criamos um vetor para conter a lista de clubes exibidos na página. Para verificar qual radiobutton está selecionado, criamos um for. A posição do item selecionado é atribuída a uma variável e, na sequência, ela é utilizada para obter o nome do clube. Caso a opção “**Nenhum**” seja selecionada, deve-se redefinir o estilo da divTitulo apenas para row (sem a classe das cores do clube), ocultar a imagem e remover o nome do clube da localStorage.

Já na function verificarClube, as condições devem ser ajustadas para alterar o item selecionado de acordo com o nome dado ao vetor que contém os elementos input type="radio" da página.

```

const verificarClube = () => {
  if (localStorage.getItem("clube")) { // se já estiver salvo algum clube
    const clube = localStorage.getItem("clube") // obtém o nome do clube

    // conforme o clube, marca um dos elementos do input type radio
    if (clube == "Brasil") {
      inRadios[0].checked = true
    } else if (clube == "Pelotas") {
      inRadios[1].checked = true
    } else {
      inRadios[2].checked = true
    }
    trocarClube() // chama a function que troca a imagem e cores
  }
}

// ao carregar a página, verifica se cliente já selecionou clube anteriormente
window.addEventListener("load", verificarClube)

```

Rode o programa e verifique o conteúdo da variável clube na guia **Application**, nas ferramentas do desenvolvedor no Google Chrome. Observe que os exemplos locais ficam em **Local Storage** no item **file:///**. A Figura 9.5 exibe a página com essa guia selecionada.

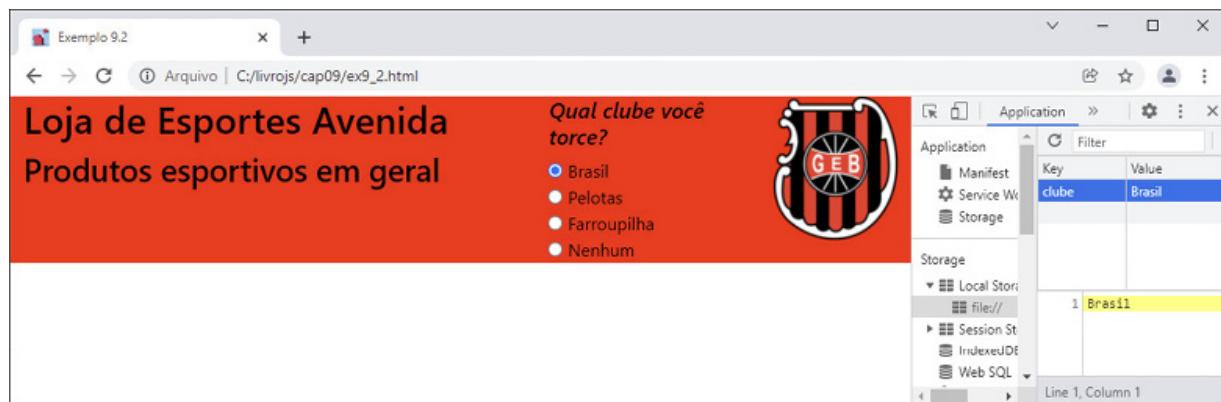


Figura 9.5 – Ao trocar de clube, o conteúdo da localSotage é alterado, e em “Nenhum”, removido.

9.5 Manipular listas no localStorage

Ok. Vimos como salvar o nome de um clube ou uma idade no navegador do usuário. Mas como poderíamos, por exemplo, armazenar uma lista de contatos ou uma lista de compras em um supermercado? Ah... com as técnicas de programação vistas anteriormente!

No capítulo sobre manipulação de strings, vimos como separar partes de um texto a partir da ocorrência de um determinado caractere. Essa será a abordagem utilizada. O caractere ";" pode servir como delimitador dos itens inseridos. Assim, o conteúdo inicial atribuído à variável frutas, por exemplo, na localStorage pode ser:

```
frutas = "Banana"
```

Novas inclusões devem inserir o caractere delimitador. Assim:

```
frutas = "Banana;Maçã"
```

Vamos aplicar essa ideia em um novo exemplo. Você já foi a algum hotel, nas férias, e fizeram a brincadeira “Qual é o peso da melancia?”?. As respostas dos hóspedes são, geralmente, anotadas em uma folha, sendo que dois hóspedes não podem apostar um mesmo número. Em um determinado horário, a melancia é então pesada. O ganhador é a pessoa que acertou o peso ou quem chegou mais perto do número correto. Vamos programar esse jogo usando o localStorage? Assim, a pessoa que controla a brincadeira pode utilizar um tablet ou um celular para cadastrar as apostas a partir do

navegador do dispositivo, sem perder os dados ao fechar accidentalmente o navegador.

Exemplo 9.3 – Código HTML da página “Qual é o peso da Melancia?” (ex9_3.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container-fluid">
  <div class="row">
    <div class="col-sm-4">
      
    </div>
    <div class="col-sm-8">
      <h1>Qual é o peso da Melancia?</h1>
      <form>
        <p><label for="inNome">Nome do Apostador:</label>
           <input type="text" id="inNome" class="form-control" required autofocus></p>
        <p><label for="inPeso">Peso (em gramas):</label>
           <input type="text" id="inPeso" class="form-control" required></p>
        <p><input type="submit" value="Apostar" class="btn btn-primary">
           <input type="button" value="Vencedor" id="btVencedor"
                 class="btn btn-warning">
           <input type="button" value="Limpar Apostas" id="btLimpar"
                 class="btn btn-danger"></p>
        </form>
        <pre></pre>
      </div>
    </div>
    </div>
    <script src="js/ex9_3.js"></script>
  <!-- /body e /html -->
```

A Figura 9.6 ilustra a tela inicial da página. Foram utilizados novos estilos do Bootstrap e uma nova tag HTML. Trata-se da tag `label` que é uma etiqueta para o campo de formulário a ser preenchido. Ao clicar sobre a etiqueta, o cursor se move para o campo correspondente, sendo um recurso muito importante para questões de acessibilidade. Os estilos do Bootstrap foram agora aplicados aos campos de formulário (`class="form-control"`) e aos botões (`class="btn ..."`). O complemento do estilo de cada botão define a sua cor. As demais tags já foram utilizadas em exemplos anteriores.

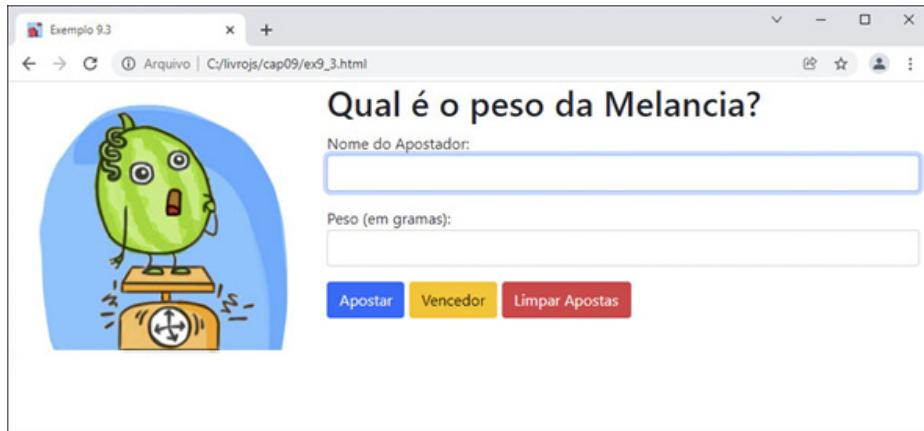


Figura 9.6 – Página inicial do jogo “Qual é o peso da Melancia?”.

Vamos discutir a programação de cada botão separadamente. Começamos pela programação vinculada ao evento `submit` do form, que ocorre quando o usuário clicar no botão **Apostar**.

Programa JavaScript para cadastrar as apostas em localStorage (js/ex9_3.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const respLista = document.querySelector("pre")

frm.addEventListener("submit", (e) => {
  e.preventDefault() // evita o envio do form

  const nome = frm.inNome.value // conteúdo do campo nome
  const peso = Number(frm.inPeso.value) // conteúdo do campo peso (em número)

  // chama function que verifica se peso já foi apostado
  if (verApostaExiste(peso)) {
    alert("Alguém já apostou este peso, informe outro...")
    frm.inPeso.focus()
    return
  }

  if (localStorage.getItem("melanciaNome")) { // se houver dados em localStorage
    // obtém o conteúdo já salvo e acrescenta "," + dados da aposta
    const melanciaNome = localStorage.getItem("melanciaNome") + ";" + nome
    const melanciaPeso = localStorage.getItem("melanciaPeso") + ";" + peso
    localStorage.setItem("melanciaNome", melanciaNome) // salva os dados
    localStorage.setItem("melanciaPeso", melanciaPeso)
  } else { // senão, é a primeira aposta
    localStorage.setItem("melanciaNome", nome)
    localStorage.setItem("melanciaPeso", peso)
  }
})
```

```

localStorage.setItem("melanciaNome", nome) // salva os dados (sem ";")
localStorage.setItem("melanciaPeso", peso)
}

mostrarApostas() // chama function que mostra as apostas já salvas
frm.reset() // limpa o form
frm.inNome.focus() // joga o foco (cursor) no campo inNome
})

```

O programa inicia pela captura do form e do local onde a lista das apostas deve ser exibida. Na programação associada ao evento submit do form, obtém-se os conteúdos informados no formulário. Como o programa não deve aceitar duas apostas com o mesmo peso, o passo seguinte é verificar se já existe uma aposta salva em localStorage com o valor informado pelo usuário. Para isso, é realizada uma chamada a função verApostaExiste(), passando o peso como argumento. A programação dessa arrow function é listada a seguir:

```

const verApostaExiste = (peso) => {
  if (localStorage.getItem("melanciaPeso")) { // se existir dados em localStorage
    // obtém seu conteúdo e a string é dividida em itens de vetor a cada ";"
    const pesos = localStorage.getItem("melanciaPeso").split(";")

    // O peso deve ser convertido em string, pois o vetor contém strings
    return pesos.includes(peso.toString())
  } else {
    return false
  }
}

```

A função verifica inicialmente se existem dados salvos no localStorage. Caso existam, o conteúdo da variável melanciaPeso é atribuído para um vetor (utiliza-se o split() para converter a string em elementos de vetor a cada ocorrência do “;”). Em seguida, recorre-se ao método includes(), que retorna verdadeiro caso o peso exista no vetor e falso, caso contrário. Como o conteúdo do vetor é formado por strings (já que no localStorage os dados são sempre armazenados como strings), é necessário converter o peso para string, a partir do método toString().

Para finalizar a discussão sobre o processo de inclusão de dados nesse programa, observe que é necessário tratar de forma diferente a primeira

inclusão de dados no localStorage. Como discutido no início desta seção, é acrescentado um delimitador (“;”) para separar os itens inseridos após a inclusão inicial.

A exibição das apostas já realizadas é feita a partir de outra arrow function. Como esse programa ficou um pouco maior, é possível explorar as técnicas de modularização de programas, discutidas no capítulo anterior. Como a apresentação das apostas deve ocorrer a cada inserção, no início do programa (para exibir as apostas anteriormente salvas no navegador) e também depois de limpar as apostas, separar essa rotina em uma função é importante por dois aspectos principais: facilidade de compreensão e reúso de código. O código do mostrarApostas() é apresentada na sequência.

```
const mostrarApostas = () => {
    // se não há apostas armazenadas em localStorage
    if (!localStorage.getItem("melanciaNome")) {
        // limpa o espaço de exibição das apostas (para quando "Limpar Apostas")
        respLista.innerHTML = ""
        return // retorna (não executa os comandos abaixo)
    }

    // obtém o conteúdo das variáveis salvas no localStorage, separando-as
    // em elementos de vetor a cada ocorrência do ";"
    const nomes = localStorage.getItem("melanciaNome").split(";");
    const pesos = localStorage.getItem("melanciaPeso").split(";")

    let linhas = "" // irá acumular as linhas a serem exibidas

    // repetição para percorrer todos os elementos do vetor
    for (let i = 0; i < nomes.length; i++) {
        // concatena em linhas os nomes dos apostadores e suas apostas
        linhas += nomes[i] + " - " + pesos[i] + "\r\n"
    }

    // exibe as linhas (altera o conteúdo do elemento respLista)
    respLista.innerHTML = linhas
}

// chama a function quando a página é carregada, para mostrar apostas salvas
window.addEventListener("load", mostrarApostas)
```

Essa função contém alguns comandos já discutidos anteriormente, como o

uso do `split()` para separar o conteúdo do `localStorage` em elementos de vetor. Para apresentar o conteúdo do vetor, utilizamos uma técnica trabalhada nos capítulos 5 e 6, que é a de concatenar os dados a serem apresentados em uma variável com quebras de linhas (`\n`), a partir de uma estrutura de repetição. Uma chamada à função deve ser inserida no código do programa `ex9_3.js` para que os dados salvos no `localStorage` sejam exibidos sempre que a página for carregada. Assim, as apostas dos hóspedes do hotel não serão perdidas caso o usuário feche o navegador de forma involuntária, ou, até mesmo, se o computador for desligado.

Falta, portanto, apresentar a programação que deve exibir o ganhador da melancia: aquele que acertar o peso ou chegar mais próximo do correto. Nesse último caso, poderia ocorrer um empate entre dois hóspedes, se, por exemplo, o hóspede A apostar no peso 8250gr, o hóspede B apostar em 8350gr e o peso correto da melancia for 8300gr. Caso essa situação ocorra, ganha quem der o palpite correto (ou melhor, mais próximo do correto) primeiro. Definidos os critérios do jogo, vamos ao código da arrow function.

```
frm.btVencedor.addEventListener("click", () => {
    // se não há apostas armazenadas em localStorage
    if (!localStorage.getItem("melanciaNome")) {
        alert("Não há apostas cadastradas")
        return // retorna (não executa os comandos abaixo)
    }

    // solicita o peso correto da melancia
    const pesoCorreto = Number(prompt("Qual o peso correto da melancia?"))

    // se não informou, retorna
    if (pesoCorreto == 0 || isNaN(pesoCorreto)) {
        return
    }

    // obtém os dados armazenados, separando-os em elementos de vetor
    const nomes = localStorage.getItem("melanciaNome").split(";");
    const pesos = localStorage.getItem("melanciaPeso").split(";")

    // valor inicial para vencedor é o da primeira aposta
    let vencedorNome = nomes[0]
    let vencedorPeso = Number(pesos[0])
```

```

// percorre as apostas
for (let i = 1; i < nomes.length; i++) {
    // calcula a diferença de peso do "vencedor" e da aposta atual
    const difVencedor = Math.abs(vencedorPeso - pesoCorreto)
    const difAposta = Math.abs(Number(pesos[i]) - pesoCorreto)

    // se a diferença da aposta atual (no for) for menor que a do "vencedor"
    if (difAposta < difVencedor) {
        vencedorNome = nomes[i]      // troca o "vencedor"
        vencedorPeso = Number(pesos[i]) // para este elemento
    }
}

// monta mensagem com dados do vencedor
let mensagem = "Resultado - Peso Correto: " + pesoCorreto + "gr"
mensagem += "\n-----"
mensagem += "\nVencedor: " + vencedorNome
mensagem += "\nAposta: " + vencedorPeso + "gr"
alert(mensagem)
})

```

A função começa com uma condição. Se não houver apostas cadastradas, deve-se retornar ao programa. Observe que utilizamos uma exclamação (!) no if antes do localStorage. Ela indica a negação (se não ...). O passo seguinte é solicitar o peso da melancia e, na sequência, recuperar o conteúdo das apostas armazenadas no navegador do usuário.

Para descobrir qual aposta é a vencedora, utilizamos uma rotina comumente empregada nesse tipo de processo. Atribuímos um conteúdo inicial para variáveis de controle, e dentro da repetição esse conteúdo é trocado sempre que um valor mais próximo for localizado. Recorremos ao método Math.abs(), que retorna o valor absoluto do cálculo (sempre positivo), a fim de que a diferença seja calculada igualmente para quem apostou um valor maior ou menor do que o peso correto. No final, os dados do vencedor são exibidos, como observado na Figura 9.7.



Figura 9.7 – Vencedor é quem acertou ou chegou mais perto do peso correto.

E, para finalizar, a programação da função limparApostas() remove o conteúdo do localStorage e atualiza a listagem das apostas.

```
frm.btLimpar.addEventListener("click", () => {
    // solicita confirmação para excluir as apostas
    if (confirm("Confirma exclusão de todas as apostas?")) {
        localStorage.removeItem("melanciaNome") // remove as variáveis salvas
        localStorage.removeItem("melanciaPeso") // em localStorage
        mostrarApostas() // exibe a listagem vazia
    }
})
```

Muito legal ficou o nosso programa! Vamos praticar esse processo de armazenamento local de dados com JavaScript? A seguir, sugiro alguns exercícios para você praticar o tema deste capítulo.

9.6 Exercícios

a) Acrescentar no site da Loja de Esportes um contador de visitas do cliente ao site. Assim, na primeira visita do cliente à loja, exibir em um parágrafo do site a mensagem:

Muito Bem-Vindo! Esta é a sua primeira visita ao nosso site.

Nas próximas visitas, exibir:

Que bom que você voltou! Esta é a sua visita de número x ao nosso site.

b) Elaborar um programa para cadastrar produtos na lista de compras da semana. Salvar e exibir a listagem dos produtos em ordem alfabética. A Figura 9.8 exibe a ilustração da página com alguns dados inseridos.



Figura 9.8 – Os produtos devem ser salvos e exibidos em ordem alfabética.

c) Elaborar um programa para cadastrar os serviços a serem realizados por um auto center, que organize a ordem de execução dos serviços. Armazenar os serviços no navegador do usuário e a cada clique no botão **Executar Serviço** remover o primeiro serviço e exibi-lo na página. O programa deve atualizar o número de serviços pendentes quando: a) a página for aberta; b) um serviço for incluído; c) um serviço for removido. A Figura 9.9 ilustra a página do sistema.



Figura 9.9 – Os serviços pendentes devem ser salvos no navegador e removidos a cada execução.

9.7 Considerações finais do capítulo

Armazenar dados é um recurso fundamental para a programação de computadores. Nas disciplinas introdutórias de Algoritmos e Lógica de Programação, realizamos esse processo, no geral, com arquivos texto, com os quais é possível manipular pequenos volumes de dados. Para aplicações maiores, com diversos cadastros (clientes, produtos, fornecedores, vendas, entre outros), o recomendado é trabalhar com banco de dados, que contêm mecanismos avançados para o gerenciamento dos dados de uma aplicação.

Neste capítulo, simulamos o processo de armazenamento de dados em arquivos texto com a linguagem JavaScript a partir do uso do `localStorage`, uma API do HTML 5 que fornece mecanismos seguros para cadastrar dados no navegador do usuário. Os dados são armazenados em pares de nome de variável (chave)/conteúdo (valor). Dados salvos em um domínio web só podem ser manipulados por esse domínio.

A linguagem JavaScript dispõe dos métodos `localStorage.setItem("chave", valor)` para gravar um conteúdo no Local Storage e `localStorage.getItem("chave")` para recuperar esse conteúdo. Também é possível excluir os dados salvos pelo cliente em seu navegador, a partir dos métodos `localStorage.removeItem("chave")` e `localStorage.clear()`, que excluem uma variável ou todas as variáveis salvas em um domínio, respectivamente. Pode-se acompanhar o conteúdo dessas variáveis nas opções do menu **Ferramentas do desenvolvedor** disponível no seu navegador.

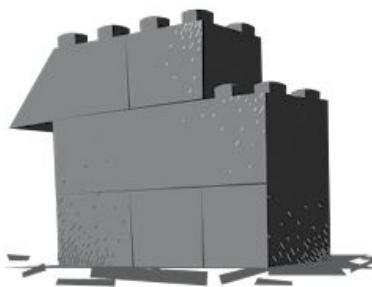
A capacidade de armazenamento dos dados salvos no Local Storage por domínio é, no geral, de 5 MB. Com isso, é possível manipular pequenas listas de dados explorando as técnicas de programação discutidas nos capítulos que abordaram o tratamento de strings e vetores. Dessa forma, pequenos cadastros podem ser realizados de forma persistente, tornando muito mais atrativos os nossos programas.

No entanto, além dos aspectos relacionados à prática das técnicas de programação com persistência de dados, importantes para o processo de aprendizado de algoritmos, salvar dados no navegador do usuário é um recurso igualmente importante na construção de sistemas web. É, no

mínimo, desejável fazer com que o site “se lembre” de escolhas realizadas pelos clientes, como da lista de produtos inseridos em um carrinho de compras, cujo processo pode ter sido interrompido por alguma situação atípica enfrentada pelo cliente. Ele certamente ficará satisfeito se não precisar selecionar novamente os produtos, caso queira concluir a compra em outro momento.

CAPÍTULO 10

Inserir elementos HTML via JavaScript



As técnicas de programação vistas até aqui são tradicionalmente trabalhadas em disciplinas de Lógica de Programação e Algoritmos. Condições, repetições, vetores, strings, funções e persistência de dados são conteúdos essenciais para o aprendizado de qualquer linguagem de programação. Neste capítulo, vamos dar alguns exemplos de como utilizar essas técnicas agora voltadas para dois dos muitos objetivos que a linguagem JavaScript pode assumir no processo de construção de páginas web: ampliar a interação com o usuário e auxiliar na composição do layout da página.

Para ampliar a interação com o usuário, veremos como inserir elementos HTML na página, modificar suas propriedades, estilos, bem como remover esses elementos. Depois de o usuário interagir com os dados da página, é possível enviar esses dados para um Web Services ou uma API, um programa desenvolvido no servidor web e que pode ter muitas funções, como cadastrar novos contatos ou retornar detalhes dos produtos de uma loja. Esse assunto de construção de Web Services será abordado no Capítulo 12. Nos exemplos do capítulo atual, vamos utilizar a API Local Storage do HTML 5.

Já o processo de utilizar um código JavaScript para auxiliar na composição

do layout de uma página HTML pode conter outros inúmeros exemplos. No Capítulo 11, montaremos o layout das poltronas disponíveis e ocupadas de um teatro. Cada poltrona é uma figura e há figuras diferentes para representar uma poltrona disponível, ocupada ou reservada pelo cliente. A linguagem JavaScript pode ser utilizada para executar a tarefa de inserir as imagens na página e trocar seus atributos para representar os diferentes status das poltronas.

Para “mergulhar” nesse contexto, é necessário entender a forma utilizada pela linguagem HTML para organizar a estrutura dos elementos (tags) que compõem uma página. Dessa forma, será possível determinar com precisão o local da página onde um parágrafo, imagem ou qualquer outro elemento será inserido. Observe o código HTML descrito a seguir:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Lógica de Programação e Algoritmos com JavaScript</title>
</head>
<body>
  <h1>Capítulo 10</h1>
  <p>Inserir Elementos HTML via JavaScript</p>

  <div id="quadro1" style="display: inline-block">
    
    <p>Detalhes da Figura 1</p>
  </div>

  <div id="quadro2" style="display: inline-block">
    
    <p>Detalhes da Figura 2</p>
  </div>
</body>
</html>
```

Os elementos HTML que compõem uma página são organizados pelo navegador na memória principal da máquina como uma estrutura hierárquica semelhante a uma árvore genealógica de uma família. A convenção HTML utilizada para representar os componentes de uma página

é chamada de DOM (*Document Object Model*) ou modelo de objeto do documento. Por ser semelhante à estrutura hierárquica de uma árvore genealógica, ela é também denominada de árvore DOM ou árvore do documento. A Figura 10.1 ilustra a estruturação hierárquica da árvore DOM das tags da página HTML anterior.

Observe, a partir da Figura 10.1, que a tag `html` se comporta como “pai” das tags `head` e `body`. A tag `body`, por sua vez, tem como “filhas” as tags `h1`, `p` e as `divs`. Cada `div` tem duas tags filhas, `img` e `p`. A codificação CSS também utiliza com frequência essa estrutura para identificar os elementos a serem estilizados na página. Em JavaScript, para inserir uma nova tag HTML na página, deve-se criar uma referência ao elemento pai, e adicionar a ele um filho. Por exemplo, se quisermos acrescentar um novo parágrafo para exibir outras informações sobre a imagem `fig1.jpg`, deve-se utilizar o método `querySelector()` para criar uma referência ao elemento `quadro1`, e acrescentar a ele uma tag `p`, que será então filha desse elemento e “irmã” das tag `img` e `p` já existentes. Perceba também que tags de texto, como `p` e `h1`, possuem um filho que é o próprio texto.

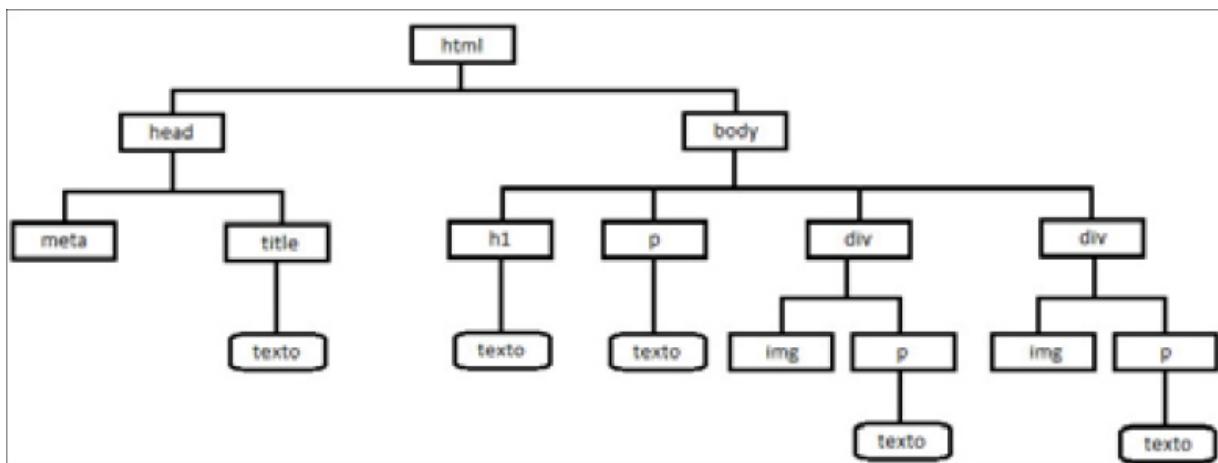


Figura 10.1 – Estrutura da árvore DOM do documento HTML anterior.

10.1 Inserir e manipular elementos de texto

Para inserir novos elementos de texto em uma página web via programação JavaScript, podem ser utilizados os métodos `createElement()`, `createTextNode()` e `appendChild()`. O método `createElement()` é responsável por criar um novo

elemento a ser adicionado na página. Ele permite criar parágrafos, linhas de cabeçalhos, quebras de linha, imagens e outros elementos. No caso dos elementos de texto, é necessário utilizar o método `createTextNode()` com o texto a ser inserido. Já o método `appendChild()` tem a tarefa de indicar a relação “pai” e “filho” entre os elementos que compõem a árvore DOM do documento HTML. A partir dele, é possível definir o local da página onde o elemento será posicionado.

Vamos construir um exemplo em que esses métodos são utilizados. Como de costume, inicie pela criação da pasta `cap10` e, dentro dela, `css`, `img` e `js`. Nosso primeiro programa tem como objetivo adicionar itens em uma lista, processo semelhante ao já desenvolvido em outras páginas deste livro. Contudo, os itens da lista são agora inseridos a partir de tags HTML criadas pelo programa JavaScript. Também será possível selecionar linhas, recuperar seu conteúdo e remover uma tag no documento. A Figura 10.2 ilustra a página web do programa “Tarefas do Dia”, já com algumas tarefas adicionadas.



Figura 10.2 – As tarefas são tags h5 inseridas no documento pelo programa JavaScript.

Depois de criar a estrutura de pastas, crie o arquivo com as regras para a estilização dos elementos utilizados nos exemplos deste capítulo. Salve o arquivo, contendo as linhas a seguir, na pasta `css` com o nome `estilos.css`.

```
h1 { border-bottom-style: inset; }  
.tarefa-selecionada { color: red; font-style: italic; }
```

```

.tarefa-normal { color: black; }
img.moeda1-00 { height: 84px; width: 84px; margin-bottom: 5px; }
img.moeda0-50 { height: 66px; width: 66px; margin-bottom: 5px; }
img.moeda0-25 { height: 76px; width: 76px; margin-bottom: 5px; }
img.moeda0-10 { height: 60px; width: 60px; margin-bottom: 5px; }
.alinha-direita { text-align: right; }

```

O código HTML utilizado para exibir a página inicial do primeiro programa deste capítulo está descrito a seguir e deve ser salvo com o nome ex10_1.html.

Exemplo 10.1 – Página HTML do programa Tarefas do Dia (ex10_1.html)

```

<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container-fluid">
    <div class="row">
        <div class="col-sm-4">
            
        </div>
        <div class="col-sm-8" id="divQuadro">
            <h1>Tarefas do Dia (a realizar)</h1>
            <form>
                <p><label for="inTarefa">Tarefa:</label>
                    <input type="text" class="form-control" id="inTarefa" required autofocus>
                </p>
                <p><input type="submit" class="btn btn-primary" value="Adicionar &#10003;">
                    <input type="button" class="btn btn-info"
                        value="Selecionar &#8645;" id="btSelecionar">
                    <input type="button" class="btn btn-danger"
                        value="Retirar Selecionada &#10007;" id="btRetirar">
                    <input type="button" class="btn btn-success"
                        value="Gravar &#9673;" id="btGravar">
                </p>
            </form>
        </div>
    </div>
</div>
<script src="js/ex10_1.js"></script>
<!-- /body e /html -->

```

Observe que, em cada botão, após o texto, há um código precedido por &#. Esse código insere um símbolo na página web e, nesse caso, é útil para ilustrar a ação de cada botão. Caso tenha se interessado pelo assunto,

pesquise na web por “códigos de símbolos HTML” para ver outros símbolos.

Vamos agora apresentar a programação associada ao evento submit do form, que ocorre quando o usuário clicar sobre o botão **Adicionar**, e que contém os métodos destacados nesta seção. Crie o arquivo ex10_1.js (na pasta js) e informe os comandos a seguir, necessários para inserir uma tag h5 com o texto de cada tarefa digitada pelo usuário.

Programa JavaScript que insere elementos na página (js/ex10_1.js)

```
const frm = document.querySelector("form") // obtém elementos da página
const dvQuadro = document.querySelector("#divQuadro")

frm.addEventListener("submit", (e) => {
    e.preventDefault() // evita envio do form

    const tarefa = frm.inTarefa.value // obtém o conteúdo digitado

    const h5 = document.createElement("h5") // cria o elemento HTML h5
    const texto = document.createTextNode(tarefa) // cria um texto
    h5.appendChild(texto) // define que texto será filho de h5
    dvQuadro.appendChild(h5) // e que h5 será filho de divQuadro

    frm.inTarefa.value = "" // limpa o campo de edição
    frm.inTarefa.focus() // joga o cursor neste campo
})
```

Observe os comandos necessários para inserir a nova tag dentro do container divQuadro. Vamos recuperar esses comandos a seguir, a fim de destacar a função de cada um deles no programa. No início, além da captura do form, realizamos também a captura do elemento da página em que as novas tags serão inseridas.

```
const dvQuadro = document.querySelector("#divQuadro")
```

Em seguida, dentro da arrow function, cria-se a tag h5 e o texto a ser exibido na página.

```
const h5 = document.createElement("h5") // cria o elemento HTML h5
const texto = document.createTextNode(tarefa) // cria um texto
```

E, então, indica-se que o texto é filho da tag h5 e que a tag h5, por sua vez, é filha de divQuadro. Isso é feito com o uso do método appendChild().

```
h5.appendChild(texto) // define que texto será filho de h5  
dvQuadro.appendChild(h5) // e que h5 será filho de divQuadro
```

Dessa forma, a cada clique no botão **Adicionar**, uma nova tag h5 é inserida no documento e cada uma delas é armazenada pelo navegador no modelo da página na memória (árvore DOM) como filha de divQuadro. O resultado é uma lista de tarefas, como as exibidas pela Figura 10.2.

Também é possível acessar cada um dos elementos inseridos na página, modificar seus estilos e removê-los. Vamos, primeiro, selecionar uma tarefa. Para isso, faremos uso do método querySelectorAll(), discutido no capítulo anterior. Um clique no botão **Selecionar** vai destacar uma tarefa, alterando a classe da tag h5 para tarefa-selecionada. A seleção começa pelo h5 de índice 0, que é o primeiro da lista. Um novo clique no botão **Selecionar** faz com que a próxima linha seja selecionada. Se o clique ocorrer quando a última tarefa estiver em destaque, o programa retorna para a primeira linha. A Figura 10.3 exemplifica a execução da function associada ao clique nesse botão.



Figura 10.3 – Um clique no botão **Selecionar** destaca uma tarefa, que pode, então, ser removida.

Acrescente ao arquivo ex10_1.js a programação a seguir, que ficará vinculada ao click no botão btSelecionar.

```
frm.btSelecionar.addEventListener("click", () => {
```

```

const tarefas = document.querySelectorAll("h5") // obtém tags h5 da página

if (tarefas.length == 0) {
    alert("Não há tarefas para selecionar") // se não há tarefas, exibe alerta
    return // e retorna
}

let aux = -1 // variável auxiliar para indicar linha selecionada

// percorre a lista de elementos h5 inseridos na página, ou seja, tarefas
for (let i = 0; i < tarefas.length; i++) {
    // se tag é da class tarefa-selecionada (está selecionada)
    if (tarefas[i].className == "tarefa-selecionada") {
        tarefas[i].className = "tarefa-normal" // troca para normal
        aux = i // muda valor da variável auxiliar
        break // sai da repetição
    }
}

// se a linha que está selecionada é a última, irá voltar para a primeira
if (aux == tarefas.length - 1) {
    aux = -1
}

tarefas[aux + 1].className = "tarefa-selecionada" // muda estilo da próxima linha
})

```

Observe que a arrow function inicia pela captura das tags h5 inseridas na página, atribuídas a const tarefas. Como o método querySelectorAll() retorna um vetor, pode-se utilizar a propriedade length para obter o número de elementos do vetor, ou seja, de tags h5 existentes no documento. Se esse número for 0, não há tags para selecionar.

Na sequência do programa, deve-se verificar qual das linhas está selecionada a fim de fazer com que a seleção avance para a linha seguinte. Para isso, utiliza-se uma variável auxiliar, com o valor inicial de -1, e percorrem-se os elementos do vetor a fim de verificar a propriedade className de cada um deles. Se alguma das tags h5 for do estilo tarefa-selecionada, deve-se fazer com que ela volte ao estilo padrão, além de alterar o valor da variável aux. Antes de realizar a troca do estilo, verifique se a linha selecionada é a última, para, nesse caso, fazer com que o programa destaque

novamente a primeira linha. Como a seleção deve sempre avançar quando o usuário clicar no botão selecionar, modifica-se a classe da tarefa na posição aux + 1.

O terceiro botão desse programa é o **Retirar Selecionada**, cuja programação é listada a seguir.

```
frm.btRetirar.addEventListener("click", () => {
    const tarefas = document.querySelectorAll("h5") // obtém tags h5 da página

    let aux = -1 // variável auxiliar para indicar linha selecionada

    // percorre a lista das tarefas inseridas na página (elementos h5)
    tarefas.forEach((tarefa, i) => {
        if (tarefa.className == "tarefa-selecionada") { // se é da classe tarefa-selecionada
            aux = i // muda valor da variável aux
        }
    })

    if (aux == -1) { // se não há tarefa selecionada (ou se lista vazia...)
        alert("Selecione uma tarefa para removê-la...")
        return
    }

    // solicita confirmação (exibindo o conteúdo da tag h5 selecionada)
    if (confirm(`Confirma Exclusão de "${tarefas[aux].innerText}"?`)) {
        dvQuadro.removeChild(tarefas[aux]) // remove um dos filhos de divQuadro
    }
})
```

Nessa função, recorremos ao método `forEach()` para percorrer as tags h5 da página e verificar qual delas está selecionada, ou seja, a propriedade `className` igual a `tarefa-selecionada`. Se houver alguma tag h5 selecionada, solicita-se a confirmação do usuário. Observe que é possível recuperar o conteúdo de uma tag, utilizando a propriedade `innerText`. A Figura 10.4 ilustra essa parte do programa.

Caso o usuário confirme a exclusão, a tag h5 é excluída. Perceba que, para excluir um elemento, é necessário referenciar o elemento pai, que, no caso, é a `divQuadro`.

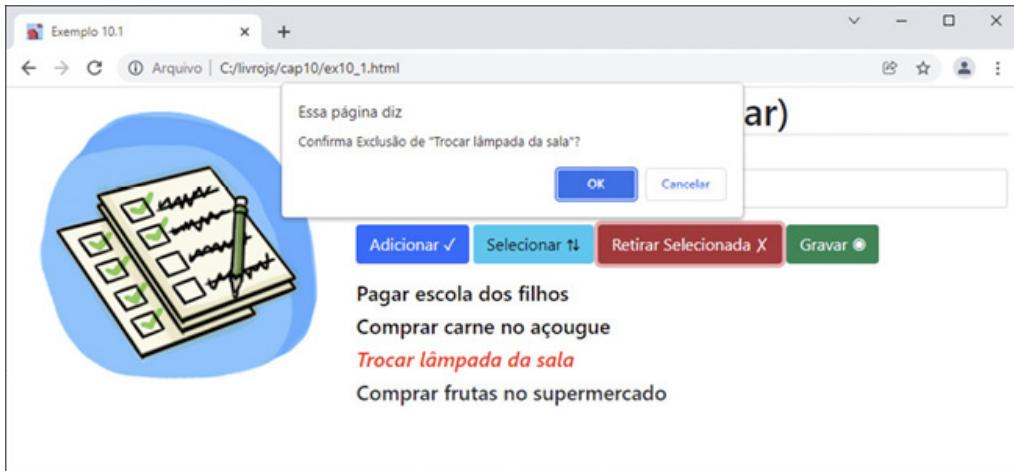


Figura 10.4 – O conteúdo da tag selecionada é recuperado para confirmar a exclusão.

A programação associada ao quarto e último botão desse programa deve recuperar o conteúdo das tags h5 exibidas na página e as armazenar em localStorage. Acrescenta-se entre as tarefas o delimitador “;”. Adicione as seguintes linhas ao arquivo ex10_1.js a fim de que o programa execute também essa ação.

```

frm.btGravar.addEventListener("click", () => {
    const tarefas = document.querySelectorAll("h5") // obtém tags h5 da página

    if (tarefas.length == 0) {
        alert("Não há tarefas para serem salvas") // se não há tarefas, exibe alerta
        return // retorna
    }

    let dados = "" // irá "acumular" os dados a serem salvos
    tarefas.forEach(tarefa => {
        dados += tarefa.innerText + ";" // acumula conteúdo de cada h5
    })

    // grava os dados em localStorage, removendo o último ";"
    localStorage.setItem("tarefasDia", dados.slice(0, -1))

    // confere se dados foram armazenados em localStorage
    if (localStorage.getItem("tarefasDia")) {
        alert("Ok! Tarefas Salvas")
    }
})

```

Para o nosso programa ficar completo, falta acrescentar no arquivo ex10_1.js a ação de recuperar as tarefas anteriormente salvas quando a página for carregada. Ela deve estar associada ao evento `load` da página e inicia verificando se existe algum dado salvo em `localStorage`. Se existir, cria as tags `h5` com a lista de tarefas salvas. O processo de inserção das tags na página é semelhante ao explicado anteriormente no evento `submit` do form.

```
window.addEventListener("load", () => {
    // verifica se há tarefas salvas no navegador do usuário
    if (localStorage.getItem("tarefasDia")) {
        // cria um vetor com a lista de tarefas (separadas pelo split(";"))
        const dados = localStorage.getItem("tarefasDia").split(";")

        // percorre os dados armazenados em localStorage
        dados.forEach(dado => {
            const h5 = document.createElement("h5") // cria o elemento HTML h5
            const texto = document.createTextNode(dado) // cria um texto
            h5.appendChild(texto) // define que texto será filho de h5
            dvQuadro.appendChild(h5) // e que h5 será filho de divQuadro
        })
    }
})
```

10.2 Inserir imagens

Para inserir uma imagem, bem como qualquer outro elemento na página web a partir de um programa JavaScript, é necessário seguir os mesmos passos já discutidos na seção anterior: a) identificar na árvore DOM o local onde a imagem será inserida; b) criar uma referência ao elemento pai dessa imagem; c) criar a imagem; d) modificar seus atributos; e) indicar a relação de pai e filho entre os objetos.

Caso o pai do elemento a ser inserido na página seja o próprio documento HTML, pode-se criar uma referência ao elemento `body` da seguinte forma:

```
const body = document.querySelector("body")
```

Nesse caso, ao utilizar o método `appendChild()`, o elemento inserido será renderizado após as tags já existentes no corpo da página.

Vamos construir um exemplo em que um programa JavaScript faz a

inserção de imagens na página. Pensei em uma brincadeira que pode ser feita com crianças para que elas calculem a soma de moedas exibidas de forma aleatória na página. A cada execução, o programa deve gerar um novo número de moedas e a criança deve informar a soma e conferir se o cálculo que fez está correto. A Figura 10.5 exibe a página inicial do programa.

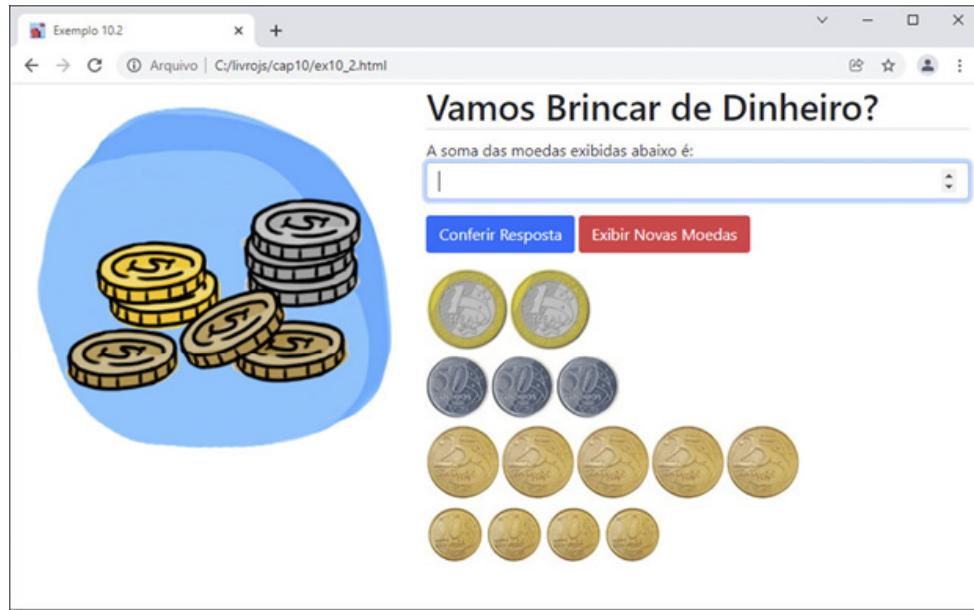


Figura 10.5 – As imagens das moedas são inseridas em número aleatório pelo programa JavaScript.

O código HTML descrito a seguir cria o layout inicial da página sem as imagens das moedas. Elas são inseridas pelo programa JavaScript. Crie um novo documento para digitar esse código e salve-o com o nome ex10_2.html.

Exemplo 10.2 – Página HTML “Vamos Brincar de Dinheiro?” (ex10_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container-fluid">
  <div class="row">
    <div class="col-sm-5">
      
    </div>
    <div class="col-sm-7" id="divMoedas">
      <form>
        <h1>Vamos Brincar de Dinheiro?</h1>
```

```

<p><label for="inSoma">A soma das moedas exibidas abaixo é:</label>
  <input type="number" min="0" step="0.05" class="form-control"
    id="inSoma" required autofocus>
</p>
<p><input type="submit" class="btn btn-primary" value="Conferir Resposta">
  <input type="reset" class="btn btn-danger" value="Exibir Novas Moedas">
</p>
</form>
</div>
</div>
</div>
<script src="js/ex10_2.js"></script>
<!-- /body /html -->

```

Para inserir as moedas na página, crie o arquivo ex10_2.js. Ele contém a programação associada a alguns eventos e uma arrow function. Vamos apresentar, inicialmente, a programação que deve ocorrer quando a página for carregada (load), além da captura do form e da divMoedas.

Programa JavaScript que insere imagens na página (js/ex10_2.js)

```

const frm = document.querySelector("form") // obtém elementos da página
const dvMoedas = document.querySelector("#divMoedas")

window.addEventListener("load", () => {
  // gera números aleatórios, entre 1 e 5, para cada moeda
  const num1_00 = Math.ceil(Math.random() * 5)
  const num0_50 = Math.ceil(Math.random() * 5)
  const num0_25 = Math.ceil(Math.random() * 5)
  const num0_10 = Math.ceil(Math.random() * 5)

  // define texto alternativo das imagens (acessibilidade)
  const alt1_00 = "Moedas de um real"
  const alt0_50 = "Moedas de Cinquenta Centavos"
  const alt0_25 = "Moedas de Vinte e Cinco Centavos"
  const alt0_10 = "Moedas de Dez Centavos"

  // chama o método criarMoedas passando os argumentos
  criarMoedas(num1_00, "1_00.jpg", alt1_00, "moeda1-00")
  criarMoedas(num0_50, "0_50.jpg", alt0_50, "moeda0-50")
  criarMoedas(num0_25, "0_25.jpg", alt0_25, "moeda0-25")
  criarMoedas(num0_10, "0_10.jpg", alt0_10, "moeda0-10")

```

)

Quando a página for carregada, realiza-se a geração aleatória de números entre 1 e 5 atribuídos para as variáveis que indicam a quantidade de moedas de um real, cinquenta centavos, vinte e cinco centavos e dez centavos. Para isso, acionam-se os métodos Math.ceil(), que arredonda um valor para cima, e Math.random(), que gera um valor aleatório entre 0 e 0.99 (com diversas decimais). Esses métodos foram discutidos no Capítulo 2.

A seguir, faz-se a atribuição dos textos alternativos a serem associadas a cada imagem. Como destacado em outros pontos do livro, esse é um recurso de acessibilidade. O texto alternativo é apresentado por navegadores não gráficos e leitores de tela para pessoas com necessidades especiais. Os textos são atribuídos para variáveis com o intuito de facilitar a leitura no livro (evitar linhas longas).

Na sequência, realizam-se chamadas à função criarMoedas() passando os argumentos relativos a cada moeda que deve ser criada. Essa arrow function é destacada a seguir:

```
const criarMoedas = (num, moeda, textoAlt, classe) => {
    // cria laço de repetição para inserir várias imagens de moedas na página
    for (let i = 1; i <= num; i++) {
        const novaMoeda = document.createElement("img") // cria elemento img
        novaMoeda.src = "img/" + moeda // atributo src
        novaMoeda.textAlt = textoAlt // texto alternativo
        novaMoeda.className = classe // classe da moeda(css)
        dvMoedas.appendChild(novaMoeda) // hierarquia DOM
    }
    const br = document.createElement("br") // cria uma quebra de linha (br)
    dvMoedas.appendChild(br)
}
```

Observe que a função possui quatro variáveis sendo recebidas por parâmetro. Dessa forma, ela pode ser utilizada para inserir na página o número aleatório de todas as moedas do programa. Após a inserção de um tipo de moeda, deve ser criada na página uma quebra de linha. Isso faz com que as moedas sejam exibidas em linhas diferentes, conforme pode ser percebido na Figura 10.5.

Você já pode rodar essa primeira parte do programa. Note que a cada

atualização da página um número diferente de moedas é exibido.

Vamos então à segunda etapa do programa destinada a conferir a resposta dada pela criança. Acrescente as seguintes linhas ao arquivo ex10_2.js.

```
frm.addEventListener("submit", (e) => {
  e.preventDefault(); // evita envio do form

  const soma = Number(frm.inSoma.value) // valor informado pelo usuário
  const moedas = dvMoedas.querySelectorAll("img") // obtém img filhas de dvMoedas
  let totalMoedas = 0 // declara e inicializa acumulador

  // percorre as tags img (em moedas) e verifica propriedade className
  for (const moeda of moedas) {
    if (moeda.className == "moeda1-00") {
      totalMoedas += 1 // acumula 1 (para moedas de 1)
    } else if (moeda.className == "moeda0-50") {
      totalMoedas += 0.5 // acumula 0.50 (para moedas de 0.50)
    } else if (moeda.className == "moeda0-25") {
      totalMoedas += 0.25 // acumula 0.25 (para moedas de 0.25)
    } else {
      totalMoedas += 0.1 // acumula 0.10 (para moedas de 0.10)
    }
  }

  const div = document.createElement("div") // cria elemento div
  const h3 = document.createElement("h3") // cria elemento h3

  let mensagem
  // verifica se o valor informado é igual ao total de Moedas exibido
  if (soma == totalMoedas.toFixed(2)) {
    div.className = "alert alert-success" // define a classe da div
    mensagem = "Parabéns!! Você acertou!" // e mensagem a ser exibida
  } else {
    div.className = "alert alert-danger"
    mensagem = `Ops... A resposta correta é ${totalMoedas.toFixed(2)}`
  }
  const texto = document.createTextNode(mensagem) // cria elemento de texto
  h3.appendChild(texto) // texto é filho de h3
  div.appendChild(h3) // h3 é filho da div criada na function
  dvMoedas.appendChild(div) // e a div com alerta é filha de dvMoedas

  frm.submit.disabled = true // desabilita botão (resposta já foi exibida)
```

})

Normalmente, há várias maneiras de se resolver um problema computacional. Para calcular o total das moedas exibidas na página, a forma mais simples seria criar uma variável global, atribuir um valor de acordo com as variáveis geradas quando a página é carregada e comparar esse valor quando o usuário clicar em **Conferir Resposta**. Como o objetivo deste capítulo é manipular os elementos da página, optamos por fazer de uma forma diferente, sem utilizar a variável global.

Na programação exibida anteriormente, obtemos todas as imagens filhas de divMoedas, com o uso do método `querySelectorAll()`. A partir de uma estrutura de repetição, percorrem-se as imagens obtidas a fim de comparar um de seus atributos (no caso, `className`) e calcular o valor total.

A exibição da mensagem de resposta correta ou incorreta é feita nesse programa de um modo diferente. No lugar do `alert()`, utilizamos uma caixa estilizada pelo Bootstrap e criada no programa com o método `createElement()`. A Figura 10.6 exibe a caixa de mensagem, com um exemplo de anúncio de resposta incorreta e exibição do valor total das moedas.

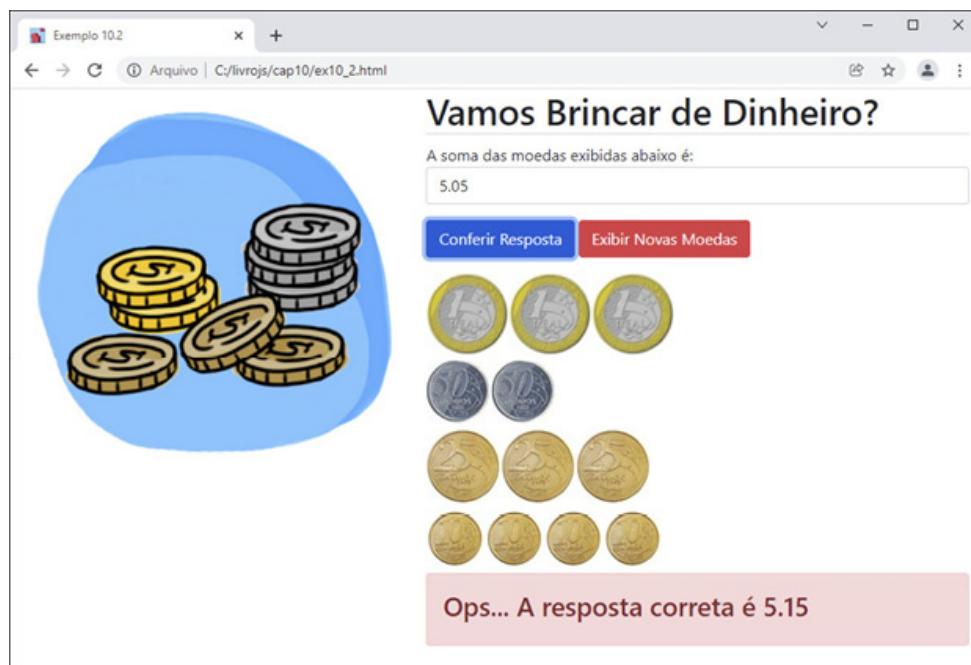


Figura 10.6 – Caixa de alerta do Bootstrap é criada na página para exibir mensagem. A cor da caixa muda de acordo com a resposta (correta ou

incorreta).

Para concluir esse exemplo, falta digitar a programação associada ao evento reset do formulário e que contém o texto “Exibir Novas Moedas”. Ele pode apenas fazer um reload na página.

```
frm.addEventListener("reset", () => {
    window.location.reload()
})
```

10.3 Manipular tabelas HTML

As tabelas são uma importante forma de exibir dados nas páginas HTML. As tags básicas para criar uma tabela são: `table` para definir a tabela, `tr` para criar uma nova linha e `td` para criar uma nova coluna em uma linha. Também há a tag `th` que define uma célula de cabeçalho na tabela.

Para definir uma nova tabela via programação JavaScript, é possível utilizar os mesmos comandos das seções anteriores e indicar que um texto deve ser filho de `td`, `td` filho de `tr` e `tr` filho de `table`. Contudo, para facilitar a manipulação das tabelas em um programa, existem os métodos `insertRow(num)` e `insertCell(num)`, que inserem, respectivamente, uma linha e uma coluna na tabela. Então, pode-se criar uma tabela na página ou uma referência a uma tabela existente, e manipular sua estrutura adicionando novas linhas e colunas. Ambos os métodos recebem como parâmetro a posição da linha ou coluna a ser inserida na tabela. 0 (zero) identifica a linha ou coluna inicial. Para inserir uma linha no final da tabela, deve-se passar como argumento o valor -1.

Também é possível remover uma linha da tabela a partir do método `deleteRow(num)`. O parâmetro indica o número da linha a ser removida. Já a propriedade `length`, sempre relacionada ao tamanho de um objeto, pode igualmente ser utilizada para recuperar o número de linhas da tabela a partir da instrução `table.rows.length`.

Outro importante recurso para a manipulação de tabelas via programação JavaScript é ter acesso ao conteúdo de uma célula da tabela, o que pode ser feito com a criação de referências aos índices que apontam para uma linha e

coluna da tabela. Utilizar `table.rows[0].cells[0]` recupera o conteúdo da primeira célula da tabela. Ou seja, colocar essa instrução dentro de um laço de repetição vai permitir a obtenção de todo o conteúdo da tabela.

Vamos construir um exemplo que explora esses métodos e propriedades na manipulação de tabelas via JavaScript e que demonstra outro exemplo de uso para o `event` que pode ser recebido por uma função: identificar o elemento alvo de um clique. Nossa terceiro programa deste capítulo deve cadastrar filmes na `localStorage` e exibir uma tabela contendo título, gênero e uma coluna que permite ao usuário excluir um filme. A Figura 10.7 exibe a página desse programa com alguns filmes cadastrados.

Crie um novo arquivo HTML e salve-o com o nome de `ex10_3.html`. O código a ser inserido nesse arquivo é descrito no Exemplo 10.3.

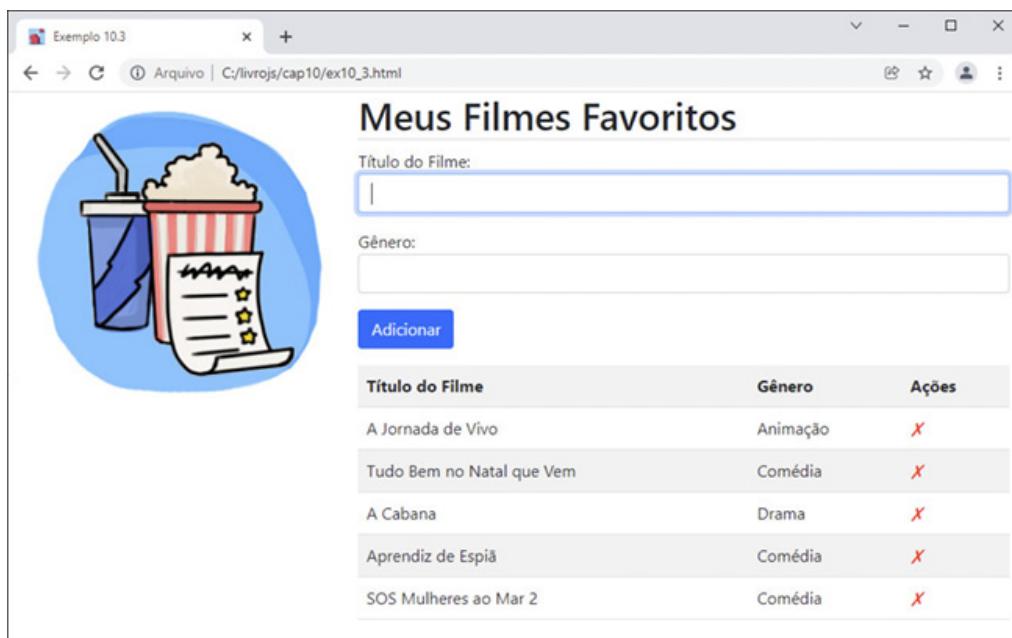


Figura 10.7 – Os filmes salvos em `localStorage` são exibidos em uma tabela na página.

Exemplo 10.3 – Página HTML que vai exibir uma tabela de filmes (ex10_3.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container-fluid">
  <div class="row">
    <div class="col-sm-4">
```

```


</div>
<div class="col-sm-8">
    <h1>Meus Filmes Favoritos</h1>
    <form>
        <p><label for="inTitulo">Título do Filme:</label>
            <input type="text" class="form-control" id="inTitulo" required autofocus>
        </p>
        <p><label for="inGenero">Gênero:</label>
            <input type="text" class="form-control" id="inGenero" required>
        </p>
        <input type="submit" class="btn btn-primary float-right mb-3"
            value="Adicionar">
    </form>
    <table class="table table-striped">
        <tr>
            <th>Título do Filme</th>
            <th>Gênero</th>
            <th>Ações</th>
        </tr>
    </table>
</div>
</div>
</div>
<script src="js/ex10_3.js"></script>
<!-- /body /html -->

```

Observe que a tabela é inicialmente criada no código HTML, e sua linha de cabeçalho, inserida. As demais linhas com os filmes informados pelo usuário serão acrescentadas pelo programa JavaScript. Vamos a ele... crie um novo arquivo e salve-o na pasta js com o nome ex10_3.js. Vamos apresentar a explicação do programa em partes. Começamos pela captura dos elementos e pela arrow function associada ao evento submit do form.

Programa JavaScript que insere linhas e colunas em uma tabela (js/ex10_3.js)

```

const frm = document.querySelector("form") // obtém elementos da página
const tbFilmes = document.querySelector("table")

frm.addEventListener("submit", (e) => {
    e.preventDefault() // evita envio do form

```

```

const titulo = frm.inTitulo.value // obtém conteúdo dos campos
const genero = frm.inGenero.value

inserirLinha(titulo, genero) // chama function que insere filmes na tabela
gravarFilme(titulo, genero) // chama function que grava dados em localStorage

frm.reset()      // limpa campos do form
frm.inTitulo.focus() // posiciona o cursor em inTitulo
})

```

Essa função obtém o conteúdo dos campos de formulário e delega as tarefas de inserir uma nova linha na tabela e um novo filme na localStorage às funções inserirLinha() e gravarFilme(). A função inserirLinha() contém as principais novidades dessa seção. Acrescente esse código no arquivo ex10_3.js.

```

const inserirLinha = (titulo, genero) => {
  const linha = tbFilmes.insertRow(-1) // adiciona uma linha na tabela

  const col1 = linha.insertCell(0) // cria colunas na linha inserida
  const col2 = linha.insertCell(1)
  const col3 = linha.insertCell(2)

  col1.innerText = titulo // joga um conteúdo em cada célula
  col2.innerText = genero
  col3.innerHTML = "<i class='exclui' title='Excluir'>&#10008</i>"
}

```

Observe que a arrow function recebe os parâmetros titulo e genero para, então, acrescentar uma linha à tabela a partir do método insertRow(). O valor -1 indica que a linha será inserida no final da tabela. O próximo passo é inserir as colunas nessa linha. Para isso, recorremos ao método insertCell(). E, por último, é feita a atribuição do conteúdo de cada célula.

Para que um símbolo seja renderizado na terceira coluna da tabela, utilizamos a propriedade innerHTML discutida na Seção 2.6. Essa é uma forma segura de utilizar essa propriedade, visto que o conteúdo a ser renderizado não vem de um campo preenchido pelo usuário, e sim de um dado fixo inserido pelo programa.

Caso você tenha ficado com alguma dúvida quanto à diferença entre as propriedades innerText e innerHTML, faça o seguinte teste: substitua a innerHTML

por innerText e rode o programa. Você vai observar que com innerText o navegador exibe o texto <i class...>, e não o símbolo exibido na Figura 10.7. Vamos então a function gravarFilme(), que também recebe os parâmetros titulo e genero. Eles devem ser gravados em localStorage. Para separar os filmes, é utilizado o delimitador “;”.

```
const gravarFilme = (titulo, genero) => {
    // se houver dados salvos em localStorage
    if (localStorage.getItem("filmesTitulo")) {
        // ... obtém os dados e acrescenta ";" e o título/gênero informado
        const filmesTitulo = localStorage.getItem("filmesTitulo") + ";" + titulo
        const filmesGenero = localStorage.getItem("filmesGenero") + ";" + genero
        localStorage.setItem("filmesTitulo", filmesTitulo) // grava dados
        localStorage.setItem("filmesGenero", filmesGenero) // em localStorage
    } else {
        // senão, é a primeira inclusão (salva sem delimitador)
        localStorage.setItem("filmesTitulo", titulo)
        localStorage.setItem("filmesGenero", genero)
    }
};
```

A partir das funções apresentadas, o programa lê os dados informados pelo usuário, exibe-os nas tabelas a partir dos métodos insertRow() e insertCell() e os salva em localStorage. A função destacada a seguir visa recuperar os dados dos filmes salvos em localStorage e os exibir quando a página for carregada. Assim, o usuário não perde a sua lista de filmes favoritos salvos na execução anterior do programa.

```
window.addEventListener("load", () => { // ao carregar a página
    // se houver dados salvos em localStorage
    if (localStorage.getItem("filmesTitulo")) {
        // obtém conteúdo e converte em elementos de vetor (na ocorrência ";")
        const titulos = localStorage.getItem("filmesTitulo").split(";");
        const generos = localStorage.getItem("filmesGenero").split(":");

        // percorre os elementos do vetor e os insere na tabela
        for (let i = 0; i < titulos.length; i++) {
            inserirLinha(titulos[i], generos[i]);
        }
    }
});
```

A programação associada ao evento `load` da página verifica se há dados salvos em `localStorage` e, se houver, joga o conteúdo de cada campo em elementos de vetor. Em seguida, percorre os elementos dos vetores e os insere na tabela, a partir da chamada ao `inserirLinha()`. Você pode testar o funcionamento do programa com as funções criadas até este ponto. Elas são responsáveis pelo processo de inclusão e exibição dos dados em uma tabela da página.

A função apresentada a seguir permite remover um filme da tabela e também da `localStorage`. O entendimento de que um elemento está vinculado ao seu “pai” também é importante aqui, pois recorremos a ele para remover a linha do símbolo sobre o qual o usuário clicou. Observe o complemento de código destacado a seguir e, em seguida, retomamos essa discussão.

```
tbFilmes.addEventListener("click", (e) => {
    // se a classe do elemento alvo clicado contém exclui
    if (e.target.classList.contains("exclui")) {
        // acessa o "pai do pai" do elemento alvo, e obtém o texto do 1º filho
        const titulo = e.target.parentElement.parentElement.children[0].innerText

        if (confirm(`Confirma Exclusão do Filme "${titulo}"?`)) {
            // remove a linha da tabela, correspondente ao símbolo de excluir clicado
            e.target.parentElement.parentElement.remove()

            localStorage.removeItem("filmesTitulo") // exclui filmes salvos em...
            localStorage.removeItem("filmesGenero") // localStorage

            // salva novamente (se existir), acessando o conteúdo da tabela
            for (let i = 1; i < tbFilmes.rows.length; i++) {
                // obtém o conteúdo da tabela (coluna 0: título; coluna 1: gênero)
                const auxTitulo = tbFilmes.rows[i].cells[0].innerText
                const auxGenero = tbFilmes.rows[i].cells[1].innerText
                gravarFilme(auxTitulo, auxGenero) // chama gravarFilme com dados da tabela
            }
        }
    }
});
```

Vamos destacar cada parte da arrow function anterior, pois ela contém alguns elementos novos. O primeiro ponto a destacar é que a programação está associada ao clique do usuário sobre a tabela. Em seguida, temos um `if` para

testar se a classe do elemento alvo contém “exclui” – algo atribuído anteriormente na função incluirLinha. Também poderíamos fazer o teste da seguinte forma:

```
if (e.target.className.includes("exclui")) {...}
```

Ou seja, colocamos o nosso programa na “escuta” de um clique sobre a tabela, mas nossa programação só irá ocorrer se o clique for sobre um símbolo de excluir pertencente a table. Dentro do if há o seguinte comando:

```
const titulo = e.target.parentElement.parentElement.children[0].innerText
```

Como discutido neste capítulo, os elementos HTML da página estão organizados em uma estrutura hierárquica. Então, para obter o texto correspondente à linha do símbolo de excluir clicado, utilizamos duas vezes o parentElement. Ou seja, o “pai do pai” do símbolo – onde o “pai” do símbolo é o td e o “pai” do td é o tr. Já children[0].innerText recupera o texto da primeira célula desta linha, que é o título do filme. Esse título é então exibido na caixa de confirmação de exclusão, conforme ilustra a Figura 10.8.

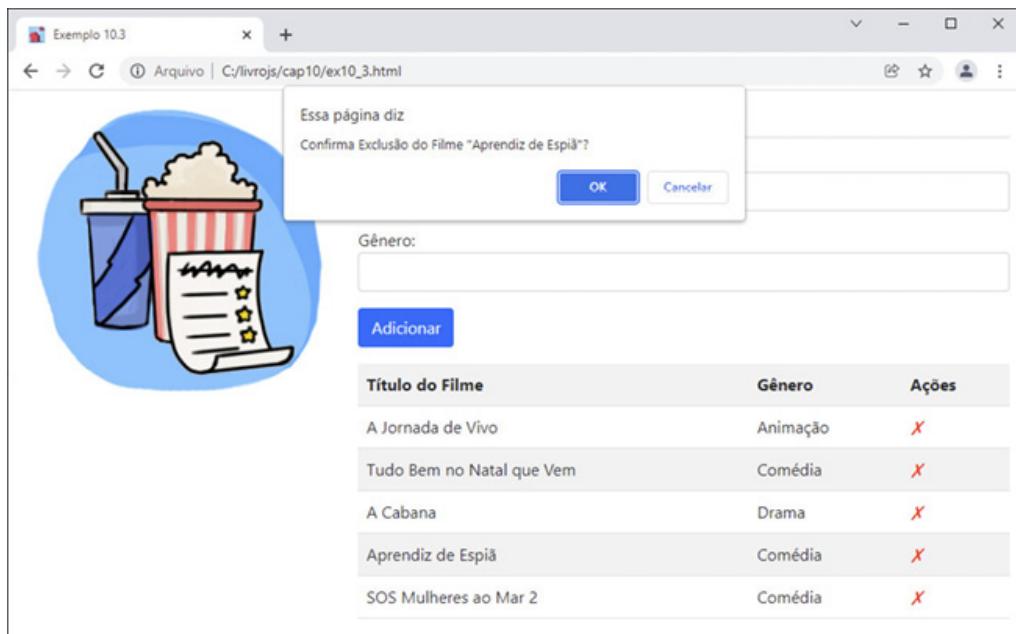


Figura 10.8 – O título do filme é exibido para que o usuário confirme a exclusão.

Caso o usuário confirme a exclusão, remove-se a linha da tabela a partir do comando `e.target.parentElement.parentElement.remove()`. Em seguida, removem-se as variáveis salvas em localStorage. Na sequência, obtém-se o conteúdo das

células da tabela e chama-se a function que grava esses dados em localStorage. Remover tudo primeiro para depois incluir novamente parece um pouco estranho... Mas, para essa estrutura de armazenamento, é o modo mais simples de executar essa tarefa.

Como destacado no capítulo anterior, os dados armazenados na Local Storage podem ser excluídos se o usuário limpar o histórico de navegação em seu browser. Portanto, cuidado com as informações salvas nesse local pelos programas desenvolvidos em nossos exemplos. Eles objetivam simular o processo de persistência de dados e devem ser substituídos por aplicações que enviam dados para Web Services a fim de evitar possíveis perdas de dados. Esse assunto é abordado nos capítulos 12 e 13.

O próximo capítulo contém mais alguns exemplos com programas maiores como esse último de cadastro de filmes. Selecionei três exercícios de fixação dos assuntos abordados neste capítulo para você praticar o que foi visto aqui. Um ótimo trabalho!! Ah... não se esqueça de que um exemplo de resposta com o programa de cada enunciado está disponível no site da editora.

10.4 Exercícios

a) Criar dez imagens de números (de 0 a 9) como se fossem velas de aniversário e salvá-las na pasta img. Em seguida, elaborar um programa que leia uma idade e insira as imagens correspondentes na página conforme o número informado. O programa deve permitir idades entre 1 e 120 anos. A Figura 10.9 exibe a página com um exemplo de velas inseridas pelo programa.



Figura 10.9 – As imagens das velas devem ser inseridas na página pelo programa JavaScript.

b) Elaborar um programa que leia um nome e, ao clicar no botão **Exibir Partes do Nome**, insira linhas de cabeçalho h3 na página com as partes do nome em cores aleatórias. Ao clicar no botão, o programa deve verificar a existência de linhas de cabeçalho h3 na página, excluindo-as se houver. A Figura 10.10 contém um exemplo de execução desse programa.

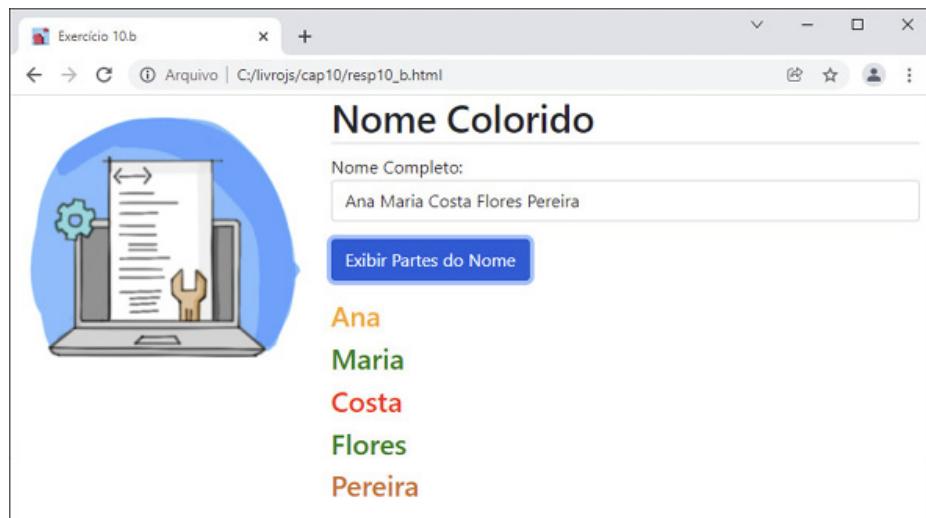


Figura 10.10 – As partes do nome são tags h3 inseridas na página com cores aleatórias.

c) Elaborar um programa que leia o nome de um clube e, ao clicar em **Adicionar**, insira-o na página a partir de uma tag h5 (alinhada à direita e em itálico). Ao clicar em **Montar Tabela de Jogos**, o programa deve

verificar se o número de tags h5 existentes na página é par. Se for, exiba os jogos (na ordem de inserção) em uma tabela, também inserida pelo programa na página. Os clubes devem ser recuperados das tags h5 existentes na página. Se o número de tags h5 for ímpar, exiba mensagem de advertência. Depois de montar a tabela, o programa deve desabilitar os botões **Adicionar** e **Montar Tabela de Jogos**. A Figura 10.11 ilustra uma execução desse programa.

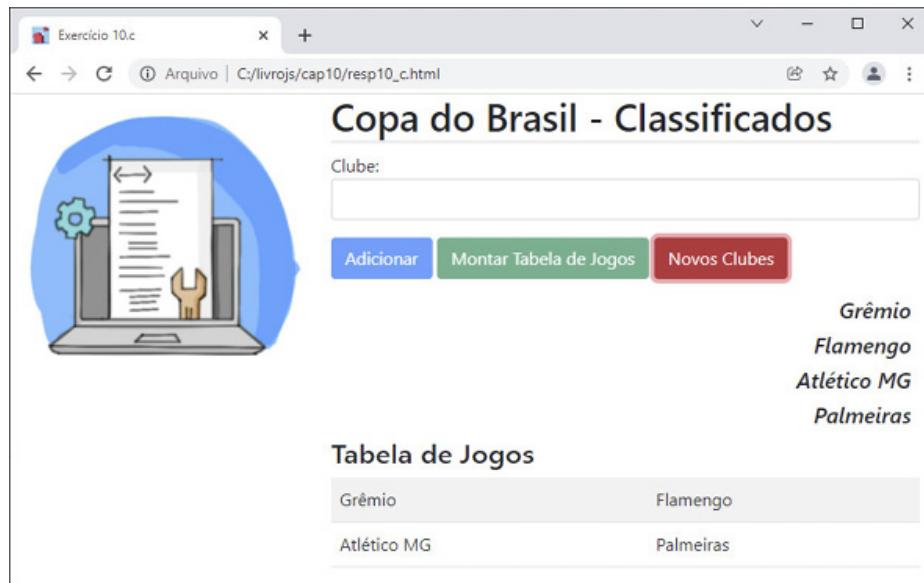


Figura 10.11 – O programa deve inserir os clubes em tags h5 e, em seguida, exibir os jogos em uma tabela.

10.5 Considerações finais do capítulo

Entre as muitas funções que podem ser atribuídas aos programas JavaScript no processo de construção de sites web, interagir com o usuário e auxiliar na montagem do layout da página são tarefas que merecem destaque. Neste capítulo, procuramos explorar a inserção de elementos HTML na página via JavaScript a fim de tornar mais ricas as interações com os usuários. Naturalmente, essa riqueza de elementos e organização de layout vai expandir de acordo com a nossa criatividade e também com os conhecimentos adquiridos em áreas como *User Experience*, entre outras. No geral, construir um site profissional é um trabalho desenvolvido por uma equipe de pessoas, com tarefas definidas de acordo com a

especialidade de cada um. Para participar de uma dessas equipes, identifique a área em que se sente mais confortável e aprofunde seus estudos nela.

Para realizar a tarefa de inserir novos elementos na página, é necessário compreender o modelo utilizado pelos navegadores para organizar a relação entre os elementos que compõem a página. A árvore DOM organiza em memória os elementos como se eles fossem parte de uma árvore genealógica de uma família. Dessa forma, ao inserir um elemento, deve-se determinar quem é o seu “pai”, para assim indicar com precisão o local da página em que esse elemento será renderizado.

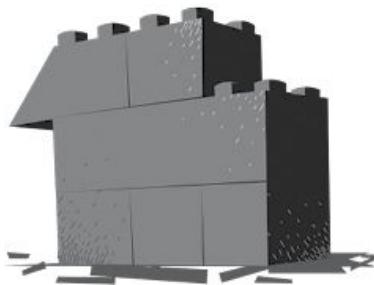
Os métodos JavaScript `createElement()` e `appendChild()` são os responsáveis para criar uma nova tag no documento e acrescentar um “filho” a um determinado elemento HTML da página, respectivamente. Para tags de texto, como parágrafos ou linhas de cabeçalho, é preciso utilizar o método `createTextNode()` com o conteúdo a ser inserido no documento. Em seguida, deve-se igualmente utilizar o método `appendChild()` para indicar a qual elemento esse texto ficará vinculado, ou seja, quem é o seu pai.

Já as tabelas HTML contêm métodos especiais que simplificam o processo. São eles: `insertRow()`, `insertCell()` e `deleteRow()`. Organizar os dados de uma lista em tabela facilita a sua leitura, sendo que diversos estilos CSS podem ser aplicados sobre as linhas e colunas da tabela a fim de destacar seus elementos.

Além da possibilidade de inserir novos elementos na página, entender a organização do modelo da árvore DOM também nos facilita na tarefa de percorrer os elementos da página a fim de recuperar ou alterar o seu conteúdo. Assim, podemos obter as imagens filhas de uma `div` na página, os campos de formulário exibidos em uma tabela, o conteúdo e as tags `h5` do documento e, então, alterar suas propriedades, como cores e estilos de fonte. Ou seja, com JavaScript, temos acesso a toda a página e podemos encantar o usuário com programas que podem auxiliá-lo em diversas rotinas do seu dia a dia.

CAPÍTULO 11

É muito bom programar... Programe!



Chegamos ao Capítulo 11, no qual novos exemplos são construídos. Este capítulo tem dois objetivos principais: revisar os recursos discutidos ao longo do livro e estimular o leitor a prosseguir nos seus estudos sobre programação, a partir da percepção de que programar pode ser uma tarefa divertida e desafiadora. Três programas são apresentados. O interessante em Algoritmos é que podemos ter diversas ideias de controles que podem ser implementados e criar pequenos programas para gerenciá-los. Vários exemplos ao longo do livro foram assim construídos, com programas para: controlar os veículos de uma revenda, o cálculo da multa e juros de uma conta se ela estiver em atraso, os pedidos de uma pizzaria, o percentual de crianças em cada faixa etária de um condomínio etc.

Ao imaginar problemas que podem ser transformados em um algoritmo, estaremos treinando diversos recursos de programação, pois esses problemas, com certeza, vão conter rotinas condicionais, repetições, uso de vetores, módulos, manipulação de strings ou datas. Talvez fiquem mais interessantes se os dados do programa forem persistidos em Local Storage. E o legal na área de programação é que você não precisa levar meses para ver se uma rotina funciona. A resposta é obtida na hora, se o programa roda

e produz os resultados corretos para todos os possíveis dados de entrada, podemos ir em frente e acrescentar novas funcionalidades ao sistema até entender que ele está concluído.

Caso o programa que você construiu não rode nos seus primeiros testes ou apresente erros para alguns dados de entrada, é possível recorrer ao debug. No Capítulo 5, vimos como acionar o depurador de programas no navegador. Com ele, é possível verificar erros de sintaxe, acompanhar passo a passo a execução do programa e conferir o valor que as variáveis assumem após a execução de cada linha. O debug é uma ferramenta fundamental tanto para novos programadores quanto para programadores experientes. Habitue-se a utilizá-lo.

Além disso, temos a vantagem de que a área de programação possui uma comunidade ativa de pessoas dispostas a colaborar na internet. Assim, caso você possua alguma dúvida, provavelmente outros já tiveram essa mesma dúvida antes de você, colocaram a questão em algum fórum e obtiveram uma resposta. Então ao pesquisar sobre a sua dúvida na rede, você vai encontrar comentários e soluções possíveis para o seu problema. Um site de destaque nessa área é o *stack overflow*, que possui um sistema de votação para as respostas consideradas mais relevantes a cada pergunta registrada pelos seus participantes. Ou seja, você consegue verificar qual resposta sobre alguma dúvida foi mais bem avaliada por outros programadores.

Portanto, não se sinta intimidado a construir os próprios Algoritmos. Eles são uma importante forma de avançar nos estudos e fazer com que você seja o gestor do seu processo de aprendizagem.

Nossos exemplos são programas para: a) controlar as apostas de um Jockey Clube; b) controlar as reservas de poltronas de um teatro em um determinado espetáculo; c) construir um jogo de descubra a palavra para pais brincarem com os filhos. Vamos ao primeiro exemplo.

11.1 Programa Jockey Club

Você já foi a algum Jockey Club e realizou uma aposta? Ao final do páreo, as apostas são totalizadas, e o prêmio total, reduzido da comissão do Jockey

Club, é dividido entre os apostadores que acertaram o número do cavalo vencedor. Vamos construir um programa para simular o controle de apostas de um páreo do Jockey Club e, ao final, ler o número do cavalo vencedor e exibir os valores totais de apostas e do cavalo vencedor.

O objetivo desse programa é revisar os recursos de criação de módulos em um programa, passagem e retorno de parâmetros, que são questões fundamentais para você avançar nos estudos. Esses recursos são relevantes também na programação orientada a objetos – que, naturalmente, é o próximo assunto a ser estudado após o aprendizado de Algoritmos e Lógica de Programação.

A Figura 11.1 apresenta a página inicial do programa. Como nos demais capítulos, crie a pasta `cap11` e dentro dela adicione as pastas `css`, `img` e `js`. Na pasta `css`, crie o arquivo `estilos.css` com o conteúdo descrito a seguir:

```
h1 { border-bottom-style: inset; }
h5 { font-style: italic; }
pre { font-style: italic; font-size: 1.1em; }
figure { display: inline-block; font-size: 0.7em; margin: 5px; }
img.poltrona { width: 24; height: 16px; }
.entre-letras { letter-spacing: 0.5em; }
```



Figura 11.1 – Página inicial do programa Jockey Club.

O código HTML para exibir a página ilustrada na Figura 11.1 contém apenas uma novidade em relação aos exemplos anteriores. Crie o arquivo `ex11_1.html`, insira esse código HTML no arquivo e, em seguida, vamos discutir sobre as tags dessa página.

Exemplo 11.1 – Página para controle de apostas de um Jockey Club (ex11_1.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container-fluid">
  <div class="row">
    <div class="col-sm-4">
      
    </div>
    <div class="col-sm-8">
      <h1>Jockey Club - Controle de Apostas</h1>
      <form>
        <div class="row">
          <div class="col-sm-6">
            <label for="inCavalo">Nº Cavalo:</label>
            <input type="number" min="1" class="form-control" id="inCavalo"
                   required autofocus>
          </div>
          <div class="col-sm-6">
            <h5 id="outCavalo" class="mt-4"></h5>
          </div>
        </div>
        <div class="row">
          <div class="col-sm-6">
            <label for="inAposta">Aposta R$:</label>
            <input type="number" class="form-control" min="1" step="0.5"
                   id="inValor" required>
          </div>
          <div class="col-sm-6">
            <input type="submit" class="btn btn-primary mt-4" value="Apostar"
                   id="btApostar">
            <input type="button" class="btn btn-warning mt-4" value="Resumo"
                   id="btResumo">
            <input type="button" class="btn btn-success mt-4" value="Ganhador"
                   id="btGanhador">
            <input type="button" class="btn btn-danger mt-4" value="Novo Páreo"
                   id="btNovo">
          </div>
        </div>
      </form>
      <pre class="mt-3"></pre>
    </div>
  </div>
</div>
```

```
</div>
<script src="js/ex11_1.js"></script>
<!-- /body e /html -->
```

A novidade é que dentro da segunda coluna (`<div class="col-sm-8">`) foram criadas duas novas `div` com `class="row"`. Elas são responsáveis por criar as duas linhas com campos de formulário e botões da parte superior da página. Essas linhas são divididas em duas colunas de mesmo tamanho (`<div class="col-sm-6">`). Você pode aplicar esse recurso em outras páginas já desenvolvidas para melhor dividir os elementos que compõem a página.

Como destacado, esse programa objetiva revisar os conceitos de funções com passagem de parâmetros. A relação dos cavalos participantes do páreo é inserida no programa a partir de uma constante. Naturalmente, você pode adicionar novos recursos a esse programa, como criar uma rotina para cadastrar os cavalos de cada páreo. O programa inicia pela captura dos elementos da página, pela declaração da constante com os nomes dos cavalos e o vetor global que vai armazenar as apostas. Em seguida, adicione a programação associada ao evento `submit` do `form` e salve-os no arquivo `ex11_1.js`, dentro da pasta `js`.

Um detalhe nos exemplos deste capítulo é que vamos passar a utilizar o “`;`” no final das linhas. Como indicado no primeiro capítulo, o “`;`” é opcional em JavaScript. Entendo ser interessante vermos alguns programas com “`;`”, para identificarmos em quais linhas eles podem ser inseridos.

Programa JavaScript para controle de apostas do Jockey Club (js/ex11_1.js)

```
const frm = document.querySelector("form"); // captura elementos da página
const respLista = document.querySelector("pre");
const respCavalo = document.querySelector("#outCavalo");

// nome dos cavalos participantes do páreo
const CAVALOS = ["Marujo", "Tordilho", "Belga", "Twister", "Jade", "Lucky"];

// vetor que irá armazenar um objeto aposta (com nº cavalo e valor da aposta)
const apostas = [];

frm.addEventListener("submit", (e) => {
```

```

e.preventDefault(); // evita envio do form

const cavalo = Number(frm.inCavalo.value); // dados do form
const valor = Number(frm.inValor.value);

// adiciona ao vetor de objetos (atributos cavalo e valor)
apostas.push({ cavalo, valor });
// variável para exibir a lista das apostas realizadas
let lista = `Apostas Realizadas\n${"-".repeat(25)}\n`;

// percorre o vetor e concatena em lista as apostas realizadas
for (const apostas of apostas) {
  lista += `Nº ${apostas.cavalo} ${obterCavalo(apostas.cavalo)}`;
  lista += ` - R$: ${apostas.valor.toFixed(2)}\n`;
}
respLista.innerText = lista; // exibe a lista das apostas

frm.reset();
frm.inCavalo.focus(); // posiciona o cursor em inCavalo
);

```

A aposta é inserida como um array de objetos em apostas. Lembre-se de que em JavaScript as seguintes instruções são equivalentes:

```
apostas.push({ cavalo, valor }); // apostas.push({ cavalo: cavalo, valor: valor });
```

Na sequência, percorre-se o array de objetos para exibir as apostas realizadas. Observe que há uma chamada a arrow function obterCavalo, descrita a seguir:

```

const obterCavalo = (num) => {
  const posicao = num - 1; // posição no vetor (subtrai 1, pois inicia em 0)
  return CAVALOS[posicao]; // nome do cavalo (da const CAVALOS)
};

```

Ao digitar um número de cavalo e sair do campo de entrada de dados inCavalo, o programa exibe o nome do cavalo (se válido) e mostra o número de apostas realizadas nesse cavalo, bem como a soma dessas apostas. A Figura 11.2 ilustra o funcionamento dessa rotina no programa.



Figura 11.2 – Ao sair do campo de edição, o programa exibe dados do cavalo e de apostas.

Para isso, uma função associada ao evento blur do elemento frm.inCavalo foi criada. Esse evento ocorre quando um campo perde o foco. Acrescente essa rotina em seu programa.

```
frm.inCavalo.addEventListener("blur", () => {
    // se não preencheu o campo, limpa respCavalo e retorna
    // (não exibe mensagem de alerta, pois pode sair por um clique em Ganhador)
    if (frm.inCavalo.value == "") {
        respCavalo.innerText = "";
        return;
    }

    const numCavalo = Number(frm.inCavalo.value); // nº do cavalo convertido em Number

    if (!validarCavalo(numCavalo)) {
        alert("Nº do cavalo inválido");
        frm.inCavalo.focus();
        return;
    }
    const nome = obterCavalo(numCavalo); // atribui retorno das funções à variáveis
    const contaNum = contarApostas(numCavalo);
    const total = totalizarApostas(numCavalo);

    // exibe nome, nº de apostas e total apostado no cavalo
    respCavalo.innerText = `${nome} (Apostas: ${contaNum} - R$: ${total.toFixed(2)})`;
});
```

Na programação associada ao evento blur de um campo de formulário, é

necessário ter o cuidado de que o usuário pode não preencher esse campo. Ele realiza algumas apostas, o programa posiciona nesse campo de edição e, em determinado momento, o usuário pode clicar no botão **Ganhador**, por exemplo. Nesse caso, o usuário executou uma ação válida no sistema. Como saiu do campo, o programa não pode exigir que ele digite um dado válido. Essa situação é verificada no primeiro if da função. Caso ele informe um número inválido, verificado pela função `validarCavalo()`, a arrow function exibe uma mensagem de alerta e joga novamente o foco nesse campo. Ainda nesse caso, o programa retorna o controle ao usuário, que deverá executar outra operação válida. Na sequência, ocorrem chamadas às funções `obterCavalo()`, `contarApostas()` e `totalizarApostas()`. Todas elas recebem como parâmetro o número do cavalo. Acrescente essas funções ao programa Jockey Club.

```
const obterCavalo = (num) => {
    const posicao = num - 1; // posição no vetor (subtraí 1, pois inicia em 0)
    return CAVALOS[posicao]; // nome do cavalo (da const CAVALOS)
};

const validarCavalo = (num) => {
    // retorna o valor resultante da condição (true ou false)
    return num >= 1 && num <= CAVALOS.length;
};

const contarApostas = (num) => {
    let contador = 0;
    // percorre o vetor apostas
    for (const apostas of apostas) {
        // verifica se a aposta é no cavalo passado como parâmetro
        if (apostas.cavalo == num) {
            contador++; // conta +1 quando a aposta for no cavalo do parâmetro
        }
    }
    return contador; // retorna o nº de apostas no cavalo numCavalo
};

const totalizarApostas = (num) => {
    let total = 0;
    for (const apostas of apostas) {
        if (apostas.cavalo == num) {
            total += apostas.valor; // soma o valor das apostas
        }
    }
}
```

```

        }
    }

    return total; // retorna a soma dos valores apostados em numCavalo
};

```

Como destacado anteriormente, quando o campo `inCavalo` perde o foco, o programa exibe ao lado desse campo o nome do cavalo e os dados das apostas realizadas no cavalo. Um detalhe importante nesse programa é que, quando esse campo receber novamente o foco, os dados exibidos ao lado do campo devem ser limpos, para não mostrar as informações da aposta anteriormente realizada. Isso pode ser feito com o trecho de programa descrito a seguir.

```

// quando o campo recebe o foco, limpa o conteúdo e dados do cavalo
frm.inCavalo.addEventListener("focus", () => {
    frm.inCavalo.value = "";
    respCavalo.innerText = "";
});

```

A partir da inclusão dessas functions ao arquivo `ex11_1.js`, você já pode realizar os testes de inclusão de apostas. A página deve exibir os dados do cavalo e apresentar as apostas em uma lista, semelhante à Figura 11.2.

Falta-nos a programação a ser associada aos botões **Resumo**, **Ganhador** e **Novo Páreo**. Começamos pela apresentação do código associado ao clique no botão **Resumo**.

```

frm.btResumo.addEventListener("click", () => {
    // vetor com valores zerados para cada cavalo
    const somaApostas = [0, 0, 0, 0, 0, 0];

    // percorre apostas e acumula na posição do cavalo apostado (-1, pois inicia em 0)
    for (const aposta of apostas) {
        somaApostas[aposta.cavalo - 1] += aposta.valor;
    }

    // exibe o resultado
    let resposta = `Nº Cavalo..... R$ Apostado\n${"-".repeat(35)}\n`;
    CAVALOS.forEach((cavalo, i) => {
        resposta += ` ${i + 1} ${cavalo.padEnd(20)}`;
        resposta += ` ${somaApostas[i].toFixed(2).padStart(11)}\n`;
    })
    respLista.innerText = resposta;
}

```

});

Observe que a função inicia pela criação de um vetor com 6 posições preenchidas com o valor inicial 0. Cada uma dessas posições irá armazenar o total apostado no respectivo número do cavalo. Na sequência, percorre-se o vetor de apostas, modificando esse vetor. Por exemplo, se uma aposta foi de R\$ 5,00 no cavalo 3, o vetor somaApostas, na posição 2 (3 - 1), irá acumular esse valor. Ao final, o resumo de apostas é exibido, conforme ilustra a Figura 11.3.



Figura 11.3 – Resumo das apostas com totais apostados em cada cavalo.

Ao clicar sobre o botão **Ganhador**, o programa inicialmente solicita e valida o número do cavalo ganhador. Para calcular o total apostado, recorremos ao “elegante” método `reduce()`. Na sequência, cria-se a variável `resumo`, que irá concatenar as respostas a serem exibidas. Observe que as funções `totalizarApostas()` e `contarApostas()` são reutilizadas, pois os dados a serem apresentados são os mesmos anteriormente exibidos ao lado do nome do cavalo na inclusão da aposta. Esse trecho do código, bem como a Figura 11.4 que ilustra a sua execução, estão apresentados a seguir.

```
frm.btGanhador.addEventListener("click", () => {
    // solicita o número do cavalo ganhador (já converte para número)
    const ganhador = Number(prompt("Nº Cavalo Ganhador:"));

    // para validar o preenchimento do prompt anterior
    if (isNaN(ganhador) || !validarCavalo(ganhador)) {
        alert("Cavalo Inválido");
```

```

        return;
    }

// uso do método reduce para somar o valor das apostas
const total = apostas.reduce((acumulador, aposta) => acumulador + aposta.valor, 0);

// concatena em resumo o resultado a ser exibido na página
let resumo = `Resultado Final do Páreo\n${'-' . repeat(30)}\n`

resumo += `Nº Total de Apostas: ${apostas.length}\n`;
resumo += `Total Geral R$: ${total.toFixed(2)}\n\n`;
resumo += `Ganhador Nº ${ganhador} - ${obterCavalo(ganhador)}\n\n`;
resumo += `Nº de Apostas: ${contarApostas(ganhador)}\n`;
resumo += `Total Apostado R$: ${totalizarApostas(ganhador).toFixed(2)}`;

respLista.innerText = resumo; // exibe o resultado

frm.btApostar.disabled = true; // desabilita os botões apostar e ganhador
frm.btGanhador.disabled = true;
frm.btNovo.focus(); // joga o foco no botão "Novo Páreo"
})

```



Figura 11.4 – Ao clicar em Ganhador, os valores totais e do cavalo vencedor são exibidos.

Para finalizar esse programa, inclua a programação associada ao evento click do botão btNovo. Observe que recorremos, novamente, ao método window.location.reload() que recarrega a página para uma nova execução do programa.

```
// recarrega a página (para funções com apenas 1 linha, não é necessário {})
frm.btNovo.addEventListener("click", () => window.location.reload());
```

11.2 Programa Reserva de Poltronas em Teatro

O segundo programa a ser exemplificado neste capítulo já foi citado algumas vezes no livro. Trata-se de um programa que monta o layout com as poltronas disponíveis e ocupadas para um determinado show em um teatro. Acredito que ele ilustra muito bem uma das funções que podem ser realizadas por um programa JavaScript no processo de construção de páginas web, que é o de auxiliar na montagem do layout da página. A Figura 11.5 apresenta a página desse programa, já com algumas poltronas ocupadas.

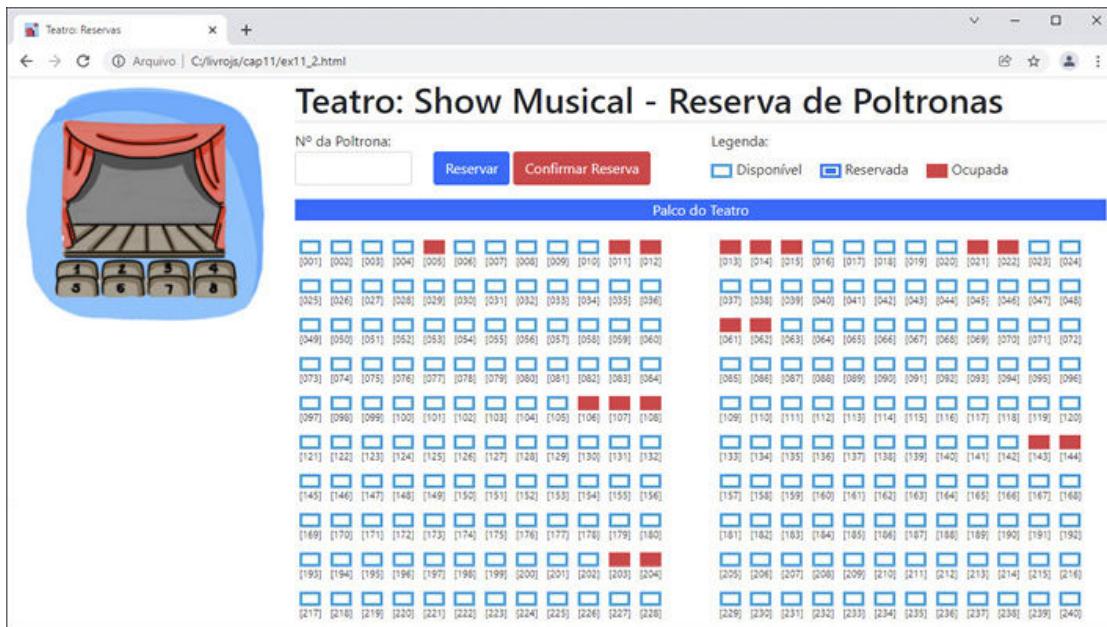


Figura 11.5 – A montagem das poltronas do Teatro é realizada pelo programa JavaScript.

Crie a página ex11_2.html e insira nela as seguintes tags HTML.

Exemplo 11.2 – Página para a reserva de poltronas de um Teatro (ex11_2.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
```

```

<div class="container-fluid">
  <div class="row">
    <div class="col-sm-3">
      
    </div>
    <div class="col-sm-9">
      <h1>Teatro: Show Musical - Reserva de Poltronas</h1>
      <form>
        <div class="row">
          <div class="col-sm-2">
            <p><label for="inPoltrona">Nº da Poltrona:</label>
            <input type="number" class="form-control" id="inPoltrona"
                   min="1" required autofocus>
            </p>
          </div>
          <div class="col-sm-4">
            <input type="submit" class="btn btn-primary mt-4" value="Reservar">
            <input type="button" class="btn btn-danger mt-4"
                   value="Confirmar Reserva" id="btConfirmar">
          </div>
        <div class="col-sm-6">
          Legenda:
          <p class="mt-2"> Disponível
             Reservada
             Ocupada
          </p>
        </div>
      </form>
      <div class="row">
        <div class="col-sm-12" id="divPalco">
          <p class="bg-primary text-white text-center"> Palco do Teatro </p>
        </div>
      </div>
    </div>
  </div>
  <script src="js/ex11_2.js"></script>
<!-- /body e /html -->

```

A exibição do campo de formulário, dos botões e da legenda na parte superior da página é dividida em colunas, da mesma forma como no exemplo do Jockey Club. Observe que nenhuma poltrona do Teatro é

exibida pelo código HTML. Essa tarefa fica a cargo do programa JavaScript. Crie, na pasta js, o arquivo ex11_2.js. Insira inicialmente a declaração das variáveis e a programação associada ao evento load que realiza a montagem do palco.

Programa JavaScript para exibição e controle de reservas de poltronas (js/ex11_2.js)

```
const frm = document.querySelector("form"); // captura elementos da página
const dvPalco = document.querySelector("#divPalco");

const POLTRONAS = 240; // constante com o número de poltronas do teatro

const reservadas = []; // vetor com as poltronas reservadas pelo cliente

window.addEventListener("load", () => {
    // operador ternário: se houver dados salvos em localStorage, faz um split(";") e
    // atribui esses dados ao array, caso contrário, o array é inicializado vazio
    const ocupadas = localStorage.getItem("teatroOcupadas")
        ? localStorage.getItem("teatroOcupadas").split(";")
        : [];

    // repetição para montar o nº total de poltronas (definida na constante)
    for (let i = 1; i <= POLTRONAS; i++) {
        const figure = document.createElement("figure"); // cria tag figure
        const imgStatus = document.createElement("img"); // cria tag img

        // se a posição consta em ocupadas, exibe a imagem ocupada, senão, disponível
        imgStatus.src = ocupadas.includes(i.toString())
            ? "img/ocupada.jpg"
            : "img/disponivel.jpg";
        imgStatus.className = "poltrona"; // classe com dimensão da img
        const figureCap = document.createElement("figcaption"); // cria figcaption

        // quantidade de zeros antes do número da poltrona
        const zeros = i < 10 ? "00" : i < 100 ? "0" : "";

        const num = document.createTextNode(`[${zeros}${i}]`); // cria texto

        figureCap.appendChild(num); // define os pais de cada tag criada
        figure.appendChild(imgStatus);
        figure.appendChild(figureCap);
```

```

// se i módulo 24 == 12 (é o corredor: define margem direita 60px)
if (i % 24 == 12) figure.style.marginRight = "60px";

dvPalco.appendChild(figure); // indica que figure é filha de divPalco

// se i módulo 24 == 0: o comando apóis && será executado (insere quebra de linha)
(i % 24 == 0) && dvPalco.appendChild(document.createElement("br"));
}

});

```

Para a exibição da imagem e do texto com o número da poltrona exibido abaixo da imagem, utilizamos a tag `figure`, uma novidade do HTML 5. Sua estilização pode ser conferida no arquivo `estilos.css` anteriormente descrito. Uma tag `figure` é uma espécie de container com uma figura (`img`) e uma legenda (`figcaption`). Para inserir as poltronas na página, é utilizado um laço de repetição com o comando `for` e dentro dele são criados os elementos HTML que montam o layout das poltronas. Caso o número já esteja salvo em `localStorage`, a imagem `ocupada.jpg` é exibida. Caso contrário, o atributo `src` vai apresentar a imagem `disponivel.jpg`.

Observe que nas posições múltiplas de 12 a margem direita do elemento é modificada, a fim de destacar um espaço de corredor no teatro, como pode ser observado na Figura 11.5. Nas posições múltiplas de 24, uma quebra de linha é inserida. Esses números poderiam também ser atribuídos a constantes, como feito com o número total de poltronas. Teste a execução dessa primeira parte do programa. Inicialmente, todas as 240 poltronas do teatro serão exibidas como disponíveis.

Nessa primeira parte do programa, exploramos algumas das sintaxes para o comando `if` disponíveis no JavaScript, as quais são: a) o uso do operador ternário; b) o `if` sem o `else` inserido em uma única linha; c) uma forma alternativa para definir uma condição a partir do operador `&&`, onde o teste é definido antes dos `&&`. Nessa forma, se o teste retornar verdadeiro, a instrução apóis os `&&` é executada.

Para realizar a reserva de uma poltrona, o programa primeiro destaca as poltronas escolhidas pelo cliente com uma imagem diferente, como exibido na Figura 11.6. A programação associada ao clique no botão **Reservar**

(evento submit do form) é descrita na sequência.

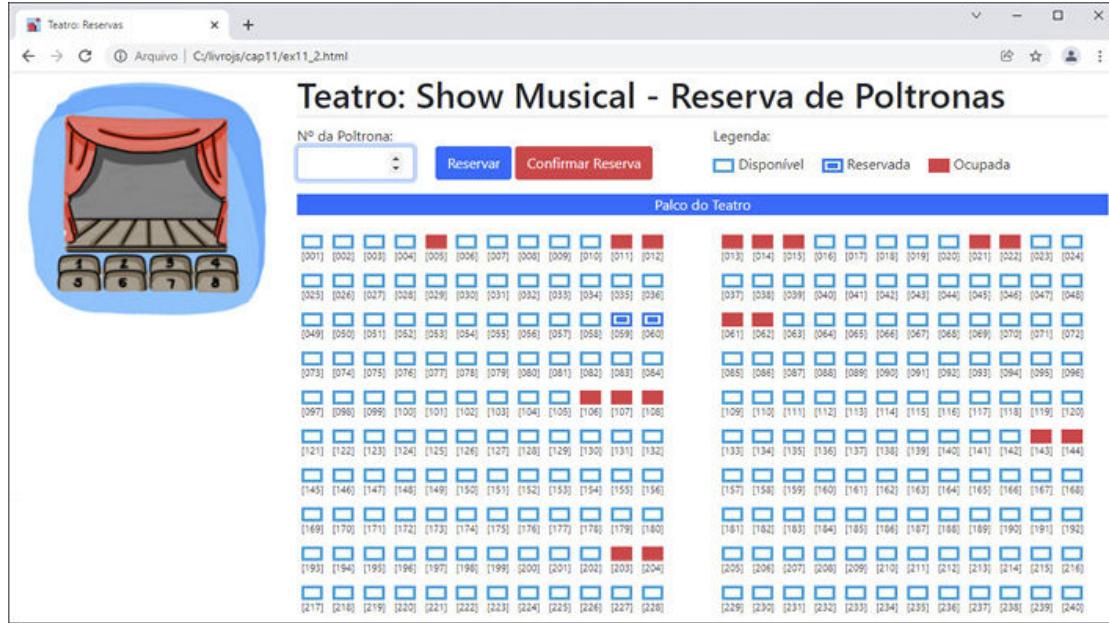


Figura 11.6 – A reserva da poltrona é realizada em duas etapas, primeiro as poltronas são destacadas com uma figura diferente.

```
frm.addEventListener("submit", (e) => {
  e.preventDefault(); // evita envio do form
```

```
const poltrona = Number(frm.inPoltrona.value); // obtém conteúdo de inPoltrona
```

```
// valida o preenchimento do campo de entrada... não pode ser maior que a const
if (poltrona > POLTRONAS) {
  alert("Informe um número de poltrona válido");
  frm.inPoltrona.focus();
  return;
}
```

```
const ocupadas = localStorage.getItem("teatroOcupadas")
? localStorage.getItem("teatroOcupadas").split(";")
: [];
```

```
// se poltrona escolhida já está ocupada (existe em localStorage)
if (ocupadas.includes(poltrona.toString())) {
  alert(`Poltrona ${poltrona} já está ocupada...`);
  frm.inPoltrona.value = "";
  frm.inPoltrona.focus();
  return;
```

```

}

// captura imagem da poltrona, filha de divPalco. É -1 pois começa em 0
const imgPoltrona = dvPalco.querySelectorAll("img")[poltrona - 1];

imgPoltrona.src = "img/reservada.jpg"; // modifica atributo da imagem

reservadas.push(poltrona); // adiciona poltrona ao vetor reservadas

frm.inPoltrona.value = ""; // limpa campo
frm.inPoltrona.focus(); // jogo o foco em inPoltrona
});

```

Observe que o programa recupera as poltronas salvas em localStorage e impede a reserva de uma poltrona já ocupada. Para trocar a imagem da poltrona e indicar que ela foi reservada, é utilizado o método `querySelectorAll()` indicando o índice do elemento a ser referenciado. Na sequência do código, o atributo `src` da imagem é alterado.

Para finalizar esse programa, acrescente o código associado ao clique no botão **Confirmar Reserva** descrito a seguir. Essa função obtém as poltronas armazenadas no vetor global `reservadas` e as armazena em localStorage. Após a execução dessa function, as poltronas reservadas passam a ser exibidas como ocupadas (como na Figura 11.5 destacada anteriormente).

```

frm.btConfirmar.addEventListener("click", () => {
  if (reservadas.length == 0) {
    alert("Não há poltronas reservadas");
    frm.inPoltrona.focus();
    return;
  }

  const ocupadas = localStorage.getItem("teatroOcupadas")
    ? localStorage.getItem("teatroOcupadas").split(";")
    : [];

  // for decrescente, pois as reservas vão sendo removidas a cada alteração da imagem
  for (let i = reservadas.length - 1; i >= 0; i--) {
    ocupadas.push(reservadas[i]);

    // captura imagem da poltrona, filha de divPalco. É -1 pois começa em 0
    const imgPoltrona = dvPalco.querySelectorAll("img")[reservadas[i] - 1];
  }
}

```

```
imgPoltrona.src = "img/ocupada.jpg"; // modifica atributo da imagem  
  
reservadas.pop(); // remove do vetor a reserva já alterada  
}  
  
localStorage.setItem("teatroOcupadas", ocupadas.join(";"));  
});
```

11.3 Jogo “Descubra a Palavra”

Que tal construir um jogo para brincar com crianças ou amigos, a partir dos recursos de programação vistos no livro e utilizando apenas HTML, CSS e JavaScript? Esse é o objetivo desse exemplo. Trata-se do jogo “Descubra a Palavra”, no qual palavras com dicas devem ser cadastradas e outro jogador precisa acertar as letras que compõem a palavra até completá-la. Vamos dividi-lo em três partes: cadastro de palavras, listagem das palavras e o jogo em si.

11.3.1 Cadastro de palavras

O programa deve conter uma página que realize o cadastro de palavras e de suas respectivas dicas. Várias palavras podem ser cadastradas. Ao iniciar o jogo, o programa realiza o sorteio de uma dessas palavras, a partir do método `Math.random()` já utilizado em outros programas do livro.

A Figura 11.6 exibe a página que realiza o cadastro de palavras do jogo. O código HTML descrito na sequência deve ser inserido no arquivo `ex11_3.html` e contém uma novidade, a inserção de links na página. Para melhor organizar o código das funções que compõem o programa, vamos dividi-lo em três partes. Para isso, acrescentamos links, isto é palavras ou botões inseridos em uma página que, ao serem clicados, carregam outras páginas. A tag HTML que cria um link é ` texto de link `. Observe, no código a seguir, os exemplos de uso dessas tags.

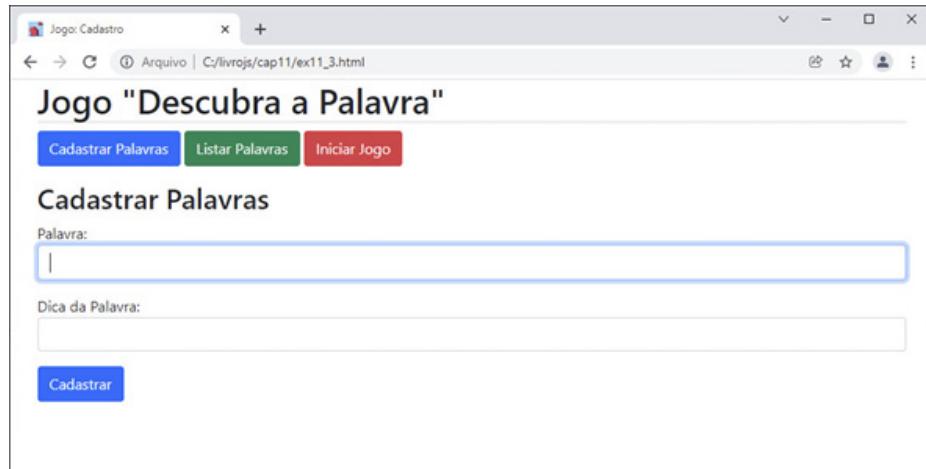


Figura 11.7 – O jogador pode cadastrar várias palavras. Uma delas é sorteada ao iniciar o jogo.

Exemplo 11.3 – Página de cadastro de palavras do jogo (ex11_3.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container">
  <h1>Jogo "Descubra a Palavra"</h1>
  <a href="ex11_3.html" class="btn btn-primary">Cadastrar Palavras</a>
  <a href="ex11_3lista.html" class="btn btn-success">Listar Palavras</a>
  <a href="ex11_3jogo.html" class="btn btn-danger">Iniciar Jogo</a>
  <h2 class="mt-3">Cadastrar Palavras</h2>
  <form>
    <p><label for="inPalavra">Palavra:</label>
      <input type="text" class="form-control" id="inPalavra" required autofocus>
    </p>
    <p><label for="inDica">Dica da Palavra:</label>
      <input type="text" class="form-control" id="inDica" required>
    </p>
    <input type="submit" class="btn btn-primary" value="Cadastrar">
  </form>
</div>
<script src="js/ex11_3.js"></script>
<!-- /body e /html -->
```

Já o programa JavaScript que realiza o cadastro de palavras deve validar o preenchimento de uma palavra válida (sem espaços) e salvar os dados em Local Storage. Essas ações já foram realizadas em outros programas. Crie o arquivo ex11_3.js (na pasta js) e insira o código descrito a seguir.

Programa JavaScript que realiza o cadastro das palavras (js/ex11_3.js)

```
const frm = document.querySelector("form"); // cria referência ao form

frm.addEventListener("submit", (e) => {
  e.preventDefault(); // evita envio do form

  // obtém conteúdo dos campos (.trim() remove espaços na palavra no início e fim)
  const palavra = frm.inPalavra.value.trim();
  const dica = frm.inDica.value;

  // valida preenchimento (palavra não deve possuir espaço em branco no meio)
  if (palavra.includes(" ")) {
    alert("Informe uma palavra válida (sem espaços)");
    frm.inPalavra.focus();
    return;
  }

  // se já existem dados em localStorage, grava conteúdo anterior+";"+palavra / dica
  if (localStorage.getItem("jogoPalavra")) {
    localStorage.setItem("jogoPalavra",
      localStorage.getItem("jogoPalavra") + ";" + palavra);
    localStorage.setItem("jogoDica", localStorage.getItem("jogoDica") + ";" + dica );
  } else {
    // senão, é a primeira inclusão: grava apenas a palavra / dica
    localStorage.setItem("jogoPalavra", palavra);
    localStorage.setItem("jogoDica", dica);
  }

  // verifica se salvou
  if (localStorage.getItem("jogoPalavra")) {
    alert(`Ok! Palavra ${palavra} Cadastrada com Sucesso`);
  }

  frm.reset(); // limpa o form
  frm.inPalavra.focus(); // joga foco em inPalavra
});


```

Como pode ser observado no código ex11_3.js, os dados são armazenados em duas variáveis de localStorage: jogoPalavra e jogoDica. Ao acrescentar um nova palavra ou dica, o delimitador “;” é acrescentado na variável.

11.3.2 Listagem de palavras

O programa que exibe a listagem das palavras e permite a exclusão de alguma palavra é bastante semelhante ao código do programa de cadastro de filmes, visto no capítulo anterior. O que acrescentamos nesse programa foi um checkbox que inicialmente aparece desmarcado (para que o jogador não veja facilmente a lista das palavras cadastradas). A Figura 11.8 ilustra a tela exibida pelo programa quando se clica no link **Listar Palavras**.



Figura 11.8 – Para evitar a exibição direta das palavras cadastradas, o checkbox “Mostrar Palavras” inicia sempre desmarcado.

No entanto, caso seja necessário dar manutenção nas palavras cadastradas no jogo, pode-se clicar no botão **Mostrar Palavras** e a lista é exibida. Os dados são apresentados em uma tabela, e cada linha contém um botão que permite excluir a palavra/dica. A Figura 11.9 apresenta uma lista de palavras cadastradas no programa.

The screenshot shows a web application window titled "Jogo: Listagem". The address bar indicates the file is located at C:/livrojs/cap11/ex11_3lista.html. The main title is "Jogo 'Descubra a Palavra'". Below it are three buttons: "Cadastrar Palavras" (blue), "Listar Palavras" (green), and "Iniciar Jogo" (red). The section "Listagem das Palavras" contains a table with the following data:

Palavra	Dica	Ações
Sorvete	Consumido principalmente no verão	X
Caderno	É utilizado na escola	X
Recreio	Hora de brincar e se alimentar	X
Cenoura	É bom para a saúde e os coelhos gostam	X
Cobertor	Utilizado no inverno	X
Papelaria	Tem material escolar	X

A checkbox labeled "Mostrar Palavras" is checked. The table has a striped background.

Figura 11.9 – As palavras cadastradas no jogo podem ser excluídas.

O código HTML com as tags que exibem a estrutura inicial dessa página é descrito a seguir. Crie um novo arquivo, chamado ex11_3lista.html, para acrescentar os comandos. Observe que a parte inicial, até os links, é semelhante ao código anterior.

Exemplo 11.3lista – Página da listagem de palavras cadastradas (ex11_3lista.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container">
  <h1>Jogo "Descubra a Palavra"</h1>
  <a href="ex11_3.html" class="btn btn-primary">Cadastrar Palavras</a>
  <a href="ex11_3lista.html" class="btn btn-success">Listar Palavras</a>
  <a href="ex11_3jogo.html" class="btn btn-danger">Iniciar Jogo</a>
  <h2 class="mt-3">Listagem das Palavras</h2>
  <p class="text-end"><input type="checkbox" checked=""> Mostrar Palavras</p>
  <table class="table table-striped">
    <tr>
      <th>Palavra</th>
      <th>Dica</th>
      <th>Ações</th>
    </tr>
    </table>
  </div>
```

```
<script src="js/ex11_3lista.js"></script>
<!-- /body e /html -->
```

As palavras serão adicionadas pelo programa JavaScript na tabela definida no código HTML. Acrescente agora o arquivo ex11_3lista.js, responsável por gerenciar a exibição das palavras e permitir a exclusão de algumas delas. Observe, na segunda linha, uma das formas possíveis para criar uma referência ao checkbox da página.

Programa Javascript que controla a listagem e exclusão de palavras (js/ex11_3lista.js)

```
const tbPalavras = document.querySelector("table"); // cria referênci a tabela
const ckMostrar = document.querySelector("input[type='checkbox']");
// e ao checkbox

const montarTabela = () => {
    // se houver dados salvos em localStorage
    if (localStorage.getItem("jogoPalavra")) {
        // obtém conteúdo e converte em elementos de vetor (na ocorrência ";")
        const palavras = localStorage.getItem("jogoPalavra").split(";");
        const dicas = localStorage.getItem("jogoDica").split(":");

        // percorre elementos do vetor e os insere na tabela
        for (let i = 0; i < palavras.length; i++) {
            const linha = tbPalavras.insertRow(-1); // adiciona uma linha na tabela

            const col1 = linha.insertCell(0); // cria colunas na linha inserida
            const col2 = linha.insertCell(1);
            const col3 = linha.insertCell(2);

            col1.innerText = palavras[i]; // joga um conteúdo em cada célula
            col2.innerText = dicas[i];
            col3.innerHTML = "<i class='exclui' title='Excluir'>✖</i>";
        }
    }
};

// ocorre quando o checkbox é alterado. Exibe a lista se marcado, senão, recarrega
ckMostrar.addEventListener("change", () => {
    ckMostrar.checked ? montarTabela() : window.location.reload();
});

tbPalavras.addEventListener("click", (e) => {
```

```

// se a classe do elemento alvo clicado contém exclui
if (e.target.classList.contains("exclui")) {
    // acessa o "pai do pai" do elemento alvo, e obtém o texto do 1º filho
    const palavra = e.target.parentElement.parentElement.children[0].innerText;

    if (confirm(`Confirma Exclusão da Palavra: "${palavra}"?`)) {
        // remove a linha da tabela, correspondente ao símbolo de excluir clicado
        e.target.parentElement.parentElement.remove();

        localStorage.removeItem("jogoPalavra"); // exclui dados de localStorage
        localStorage.removeItem("jogoDica");

        const palavras = [];
        const dicas = [];

        // obtém os dados da tabela, acrescentando-os aos vetores
        for (let i = 1; i < tbPalavras.rows.length; i++) {
            palavras.push(tbPalavras.rows[i].cells[0].innerText);
            dicas.push(tbPalavras.rows[i].cells[1].innerText);
        }

        // salva o conteúdo dos vetores em localStorage (sem a linha removida)
        localStorage.setItem("jogoPalavra", palavras.join(";"));
        localStorage.setItem("jogoDica", dicas.join(";"));
    }
}
});

```

Como mencionado, o código desse programa é semelhante ao cadastro de filmes do capítulo anterior. Os dados armazenados em localStorage são separados em elementos de vetor e inseridos na tabela tbPalavras. A exibição ocorrerá apenas quando o usuário marcar o checkbox “Mostrar Palavras”.

11.3.3 Programação do jogo

Ao clicar sobre o botão **Iniciar Jogo**, diversas são as tarefas e verificações que devem ser realizadas pelo programa. Vamos destacar cada parte do jogo nessa seção. Começamos pela apresentação da página HTML, descrita a seguir.

Exemplo 11.3jogo – Página HTML que exibe o layout

inicial do jogo (ex11_3jogo.html)

```
<!-- doctype, html, head e body (conf. exemplo 9.1) -->
<div class="container">
    <h1>Jogo "Descubra a Palavra"</h1>
    <a href="ex11_3.html" class="btn btn-primary">Cadastrar Palavras</a>
    <a href="ex11_3lista.html" class="btn btn-success">Listar Palavras</a>
    <a href="ex11_3jogo.html" class="btn btn-danger">Iniciar Jogo</a>

    <form>
        <div class="row mt-3 corFundo py-2">
            <div class="col-sm-4">
                <h2 class="text-end mt-1">
                    <label for="inLetra">Informe uma letra:</label>
                </h2>
            </div>
            <div class="col-sm-2">
                <input type="text" class="form-control mt-1" id="inLetra" maxlength="1"
                    required autofocus>
            </div>
            <div class="col-sm-6">
                <input type="submit" class="btn btn-primary mt-1" value="Jogar" id="btJogar">
            </div>
        </div>
        <div class="row">
            <div class="col-sm-8">
                <h2 class="mt-2">Palavra:<br/>
                    <span id="outPalavra" class="entre-letras text-primary"></span>
                </h2>
                <h2>Nº Chances:<br/>
                    <span id="outChances" class="text-danger">4</span>
                </h2>
                <h2>Erros:<br/>
                    <span id="outErros" class="entre-letras text-danger"></span>
                </h2>
                <h2 id="outMensagemFinal"></h2>
            </div>
            <div class="col-sm-4">
                <h3 class="mt-2"> Situação do Jogador </h3>
                
            </div>
        </div>
    </form>
</div>
```

```

<div class="row corFundo p-3">
  <div class="col-sm-12">
    <input type="button" value="Ver Dica" class="btn btn-primary" id="btVerDica">
    <span id="outDica">* Custo: 1 chance</span>
  </div>
</div>
</form>
</div>
<script src="js/ex11_3jogo.js"></script>
<!-- /body e /html -->

```

O código do arquivo ex11_3jogo.html é responsável por criar o layout inicial da página, conforme ilustra a Figura 11.10.

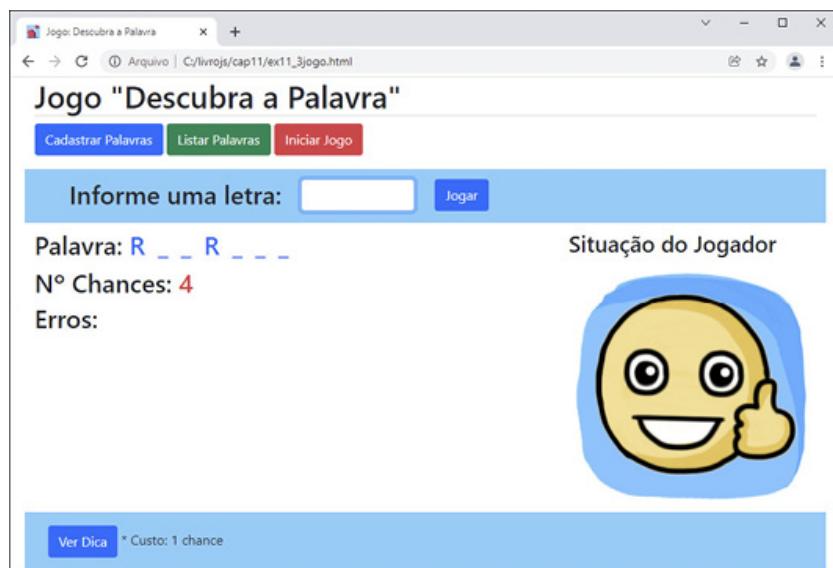


Figura 11.10 – A cada execução, uma palavra aleatória do cadastro é selecionada.

As tarefas do programa JavaScript para gerenciar a execução do jogo serão discutidas em partes. Crie o arquivo ex11_3jogo.js (na pasta js) e insira as seguintes linhas.

Programa JavaScript do Jogo “Descubra a Palavra” (js/ex11_3jogo.js)

```

const frm = document.querySelector("form"); // cria referência aos elementos HTML
const respPalavra = document.querySelector("#outPalavra");
const respErros = document.querySelector("#outErros");
const respDica = document.querySelector("#outDica");
const respChances = document.querySelector("#outChances");

```

```

const respMensagemFinal = document.querySelector("#outMensagemFinal");
const imgStatus = document.querySelector("img");

let palavraSorteada; // declara variáveis globais
let dicaSorteada;

window.addEventListener("load", () => {
    // se não há palavras cadastradas
    if (!localStorage.getItem("jogoPalavra")) {
        alert("Cadastre palavras para jogar"); // exibe alerta
        frm.inLetra.disabled = true; // desabilita inLetra e botões
        frm.btJogar.disabled = true;
        frm.btVerDica.disabled = true;
    }
}

// obtém conteúdo do localStorage e separa em elementos de vetor
const palavras = localStorage.getItem("jogoPalavra").split(";");
const dicas = localStorage.getItem("jogoDica").split(":");

const tam = palavras.length; // número de palavras cadastradas

// gera um número entre 0 e tam-1 (pois arredonda para baixo)
const numAleatorio = Math.floor(Math.random() * tam);

// obtém palavra (em letras maiúsculas) e dica na posição do nº aleatório gerado
palavraSorteada = palavras[numAleatorio].toUpperCase();
dicaSorteada = dicas[numAleatorio];
let novaPalavra = ""; // para montar palavra exibida (com letra inicial e "_")

// for para exibir a letra inicial e as demais ocorrências desta letra na palavra
for (const letra of palavraSorteada) {
    // se igual a letra inicial, acrescenta esta letra na exibição
    if (letra == palavraSorteada.charAt(0)) {
        novaPalavra += palavraSorteada.charAt(0);
    } else {
        novaPalavra += "_"; // senão, acrescenta "_"
    }
}

respPalavra.innerText = novaPalavra; // exibe a novaPalavra
});

```

O código inicia pela captura dos elementos da página a serem manipulados

pelo programa. No evento load, o primeiro cuidado que devemos ter é que o jogador pode clicar no botão **Iniciar Jogo** sem ter palavras cadastradas no jogo. Isso poderia causar alguns problemas, como ele clicar em **Ver Dica** e o programa exibir `undefined`, pois ainda não há conteúdo para as variáveis. Por isso iniciamos o programa verificando a existência de `jogoPalavra` em `localStorage`.

Observe, na sequência, que o programa gera um número aleatório de acordo com o tamanho do vetor com as palavras cadastradas. E, a partir de um laço de repetição, exibe a letra inicial (e suas ocorrências na palavra) e os símbolos de “_” para as demais letras que compõem a palavra. Para percorrer as letras que compõem uma palavra, também é possível utilizar o comando `for.. of`, como demonstrado. O resultado da execução dessa rotina é o ilustrado na Figura 11.10.

Caso o usuário clique no botão **Ver Dica**, o programa deve exibir a dica correspondente à palavra sorteada. Isso tem o custo de uma chance. Quando as chances diminuem, a situação do jogador, representada por uma imagem, muda. A Figura 11.11 apresenta a página quando o jogador clicou no botão logo no início do programa e perdeu uma chance.

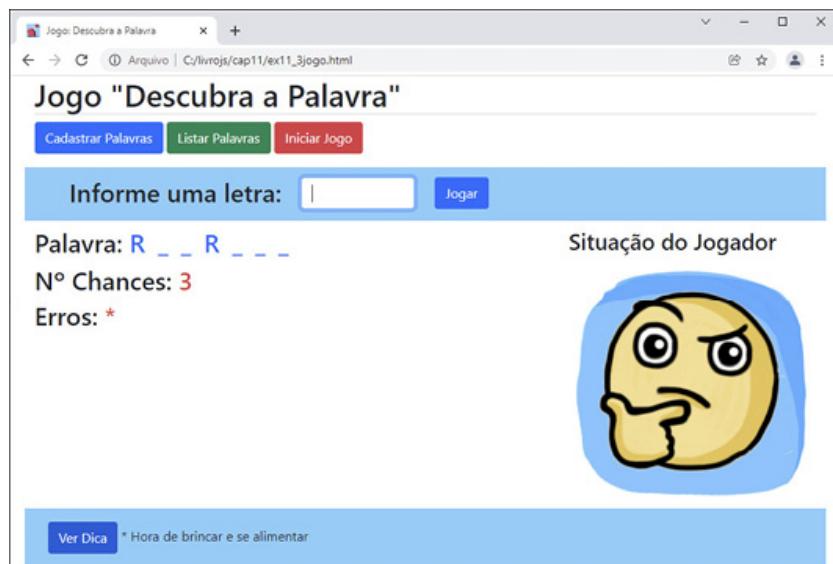


Figura 11.11 – Ao clicar em “Ver Dica”, o jogador perde uma chance e a imagem é modificada.

Acrescente ao programa `ex11_3jogo.cs` a programação associada ao evento click

do botão btVerDica descrita a seguir.

```
frm.btVerDica.addEventListener("click", () => {
    // verifica se o jogador já clicou anteriormente no botão
    if (respErros.innerText.includes("*")) {
        alert("Você já solicitou a dica... ");
        frm.inLetra.focus();
        return;
    }

    respDica.innerText = " * " + dicaSorteada; // exibe a dica
    respErros.innerText += "*"; // acrescenta "*" nos erros

    const chances = Number(respChances.innerText) - 1; // diminui 1 em chances
    respChances.innerText = chances; // mostra o nº de chances

    trocarStatus(chances); // troca a imagem

    verificarFim(); // verifica se atingiu limite de chances

    frm.inLetra.focus(); // joga o foco em inLetra
});
```

A função verifica se respErros já contém um “*”. Nesse caso, uma mensagem de alerta é exibida e o programa retorna à página. Na sequência, observe que a function executa as ações de exibir a dica, acrescentar “*” nos erros, diminuir e exibir o novo número de chances e trocar a imagem da situação do jogador.

Pode acontecer de o jogador clicar em **Ver Dica** quando ele só tem uma chance. Seria um erro, pois, como ele perde uma chance ao ver a dica, ele consequentemente também perderia o jogo. Essa tarefa é realizada pela função verificarFim() descrita no final desta seção.

A programação associada à function trocarStatus() é destacada a seguir. Ela recebe como argumento o número de chances que representa o status do jogador, o qual vai de 4 (feliz) até 1 (apavorado). As imagens dos bonecos, salvas na pasta img, devem possuir os nomes: status4.jpg, status3.jpg, status2.jpg e status1.jpg.

```
const trocarStatus = (num) => {
    if (num > 0) imgStatus.src = `img/status${num}.jpg`;
```

```
};
```

Vamos destacar a seguir a programação associada ao evento submit do formulário, que ocorre quando o usuário clica no botão **Jogar**. Algumas validações são realizadas pelo código HTML, como o preenchimento obrigatório do campo (required) e o tamanho máximo do seu conteúdo (maxlength="1").

```
frm.addEventListener("submit", e => {
  e.preventDefault(); // evita envio do form

  const letra = frm.inLetra.value.toUpperCase(); // obtém o conteúdo do campo inLetra

  let erros = respErros.innerText; // obtém o conteúdo dos elementos da página
  let palavra = respPalavra.innerText;

  // verifica se a letra apostada já consta em erros ou na palavra
  if (erros.includes(letra) || palavra.includes(letra)) {
    alert("Você já apostou esta letra");
    frm.inLetra.focus();
    return;
  }

  // se letra consta em palavraSorteada
  if (palavraSorteada.includes(letra)) {
    let novaPalavra = ""; // para compor novaPalavra
    // for para montar palavra a ser exibida
    for (let i = 0; i < palavraSorteada.length; i++) {
      // se igual a letra apostada, acrescenta esta letra na exibição
      if (palavraSorteada.charAt(i) == letra) {
        novaPalavra += letra;
      } else {
        novaPalavra += palavra.charAt(i); // senão, acrescenta letra ou "_" existente
      }
    }
    respPalavra.innerText = novaPalavra; // exibe a novaPalavra
  } else {
    respErros.innerText += letra; // acrescenta letra aos erros
    const chances = Number(respChances.innerText) - 1; // diminui nº de chances
    respChances.innerText = chances; // exibe novo nº de chances

    trocarStatus(chances); // troca imagem
  }
}
```

```

    verificarFim(); // verifica se já ganhou ou perdeu

    frm.inLetra.value = "";
    frm.inLetra.focus();
});
```

Analizando o programa por inteiro, ele pode parecer um pouco complexo. No entanto, se o dividirmos em partes, vamos verificar que as tarefas que ele realiza são simples. Essa é uma das vantagens da modularização de programas, discutida no Capítulo 8. A função associada ao evento submit (botão **Jogar**) obtém o conteúdo do campo `inLetra` e verifica se essa letra já foi apostada, ou seja, se ela já consta em erros ou na palavra montada pelo programa. Somente após esse teste é que verificamos se o jogador acertou ou errou a letra. Observe que a function `verificarFim()` é chamada e está fora do if, ou seja, deve ser executada quando o jogador acertar a letra (e pode ter ganhado o jogo) ou quando errar a letra (e pode ter perdido o jogo).

A Figura 11.12 exibe as modificações que podem ocorrer na página a cada letra que o jogador aposta. Se a letra estiver correta, o programa acrescenta a letra na palavra exibida. Se estiver incorreta, o jogador perde uma chance, a letra é exibida em erros e a situação do jogador se complica...

Vamos destacar a seguir a programação da function `verificarFim()` e `concluirJogo()` que finalizam esse programa. A Figura 11.13 ilustra uma situação em que o jogador perdeu o jogo. Já a Figura 11.14 apresenta a mensagem exibida quando o jogador acertar todas as letras e ganhar o jogo. Ambas as mensagens são exibidas a partir da execução dessas funções.

```

const verificarFim = () => {
    const chances = Number(respChances.innerText); // obtém número de chances

    if (chances == 0) {
        respMensagemFinal.className = "display-3 text-danger";
        respMensagemFinal.innerText = `Ah... é ${palavraSorteada}. Você Perdeu!`;
        concluirJogo();
    } else if (respPalavra.innerText == palavraSorteada) {
        respMensagemFinal.className = "display-3 text-primary";
        respMensagemFinal.innerText = "Parabéns!! Você Ganhou.";
        trocarStatus(4); // exibe a figura do "rostinho feliz"
```

```

        concluirJogo();
    }
}

// modifica o texto da dica e desabilita os botões de jogar
const concluirJogo = () => {
    respDica.innerText = "* Clique no botão 'Iniciar Jogo' para jogar novamente";
    frm.inLetra.disabled = true;
    frm.btJogar.disabled = true;
    frm.btVerDica.disabled = true;
}

```

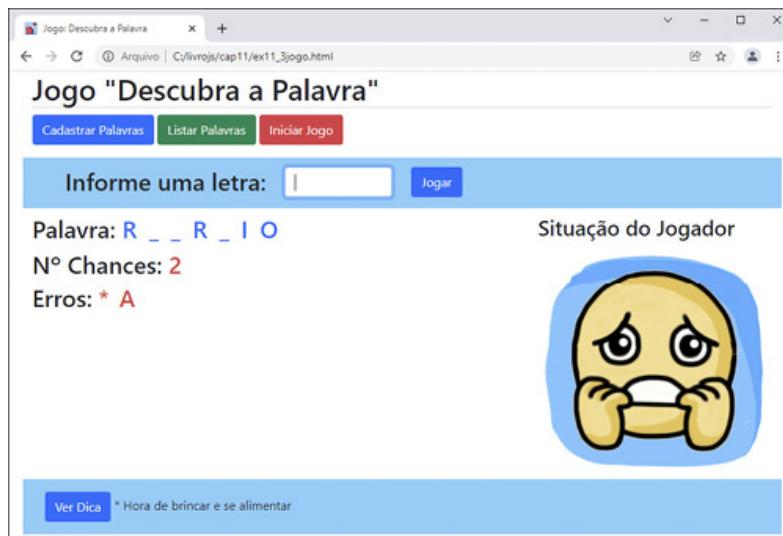


Figura 11.12 – As letras corretas são acrescentadas à palavra. Já os erros diminuem as chances e complicam a situação do jogador.

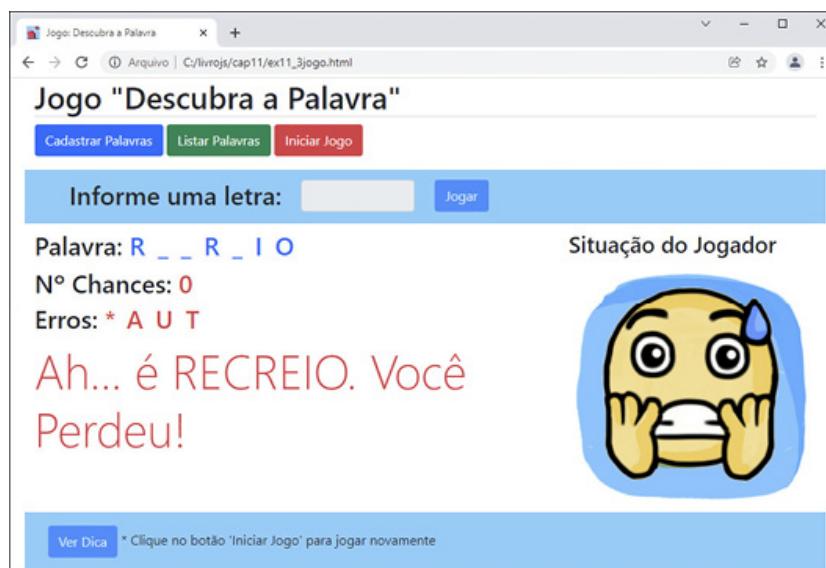


Figura 11.13 – Quando o jogador atingir o limite de chances, uma mensagem é exibida com a palavra correta.

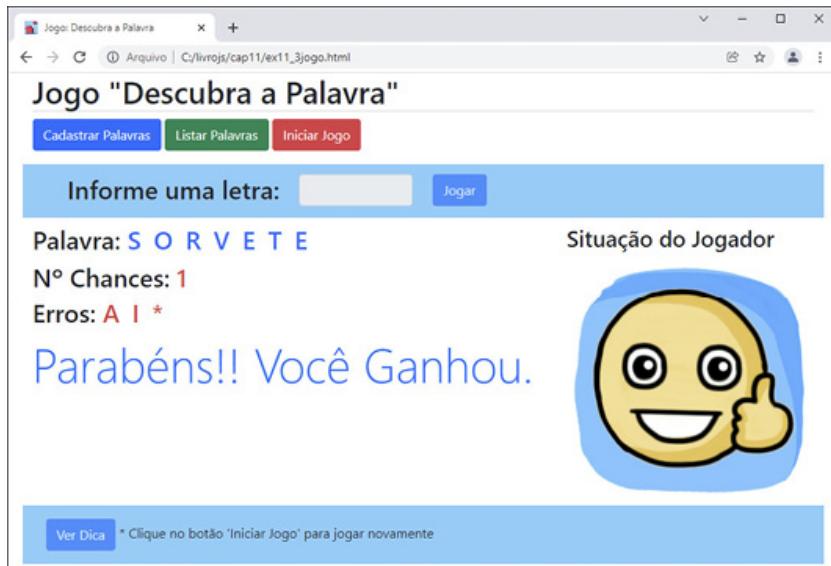


Figura 11.14 – Quando o jogador acertar a palavra, a carinha volta a ficar “feliz”.

Lembre-se de que os códigos desse e de todos os exemplos e exercícios do livro estão disponíveis para download no site da editora Novatec.

11.4 Considerações finais do capítulo

Neste capítulo, foram apresentados três programas. O primeiro, de controle de apostas de um Jockey Club, visou revisar os conceitos de modularização de programas com passagem e retorno de parâmetros. O segundo programa, de controle de poltronas ocupadas e disponíveis em um teatro, teve por propósito demonstrar a importância da linguagem JavaScript no processo de montagem do layout de páginas web. Já o intuito do terceiro e último exemplo do livro foi destacar que jogos completos, embora simples, podem ser criados apenas com HTML, CSS e JavaScript.

É possível ajustar os programas, melhorando-os. Faça alguns testes. Acrescente o cadastro dos cavalos participantes de um páreo no primeiro exemplo. Ou, então, estude sobre programação desenvolvida no lado servidor e crie um Web Services para receber a confirmação das reservas de

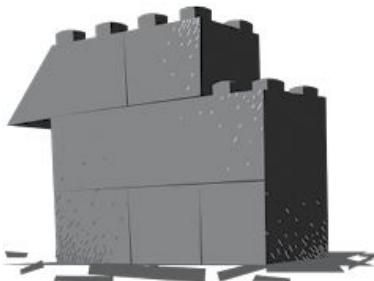
poltronas no programa do Teatro. Ou, ainda, salve uma senha na primeira inclusão no jogo “Descubra a Palavra” e só mostre as palavras cadastradas se o jogador informar a senha correta. No geral, sempre existem melhorias possíveis de serem implementadas nos programas, bem como diversas formas de desenvolver a programação de cada um deles.

O objetivo principal deste capítulo, porém, é demonstrar que, com o aprendizado das técnicas de programação discutidas ao longo do livro, você pode se aventurar e criar os próprios programas. No processo de construção de um programa, crie o hábito de testar cada nova implementação de código. Não deixe para verificar se o seu código está correto apenas no final do programa, pois dificulta a localização de possíveis erros. E lembre-se de utilizar o depurador de programas, discutido passo a passo no Capítulo 5.

Nos capítulos seguintes, vamos avançar um pouco mais e apresentar o passo a passo da criação de um app JavaScript que explora os conceitos de programação cliente x servidor, a partir do Express e React. Continue na sua leitura e execução dos nossos programas!

CAPÍTULO 12

Criação de um App: Back-end com Express



A ideia deste capítulo é apresentar os passos iniciais sobre o desenvolvimento de aplicações cliente-servidor com JavaScript. Os temas aqui abordados são amplos. Mas com o que já abordamos no livro é possível introduzir alguns novos conceitos que permitem um avanço gradual no seu processo de aprendizagem. O objetivo é dar uma visão geral sobre a criação de um Web Service que realiza as operações básicas de inclusão, alteração, consulta e exclusão de dados e uma página web que irá consumir esse Web Service. Esteja ciente de que há uma vasta gama de recursos que devem ser explorados sobre esses temas, além do que é abordado nestes dois últimos capítulos.

As aplicações web que utilizamos na internet e que contêm listas de produtos, formulários de cadastro ou campos de pesquisa geralmente interagem com um banco de dados. Esse banco de dados bem como os arquivos e programas que compõem um site ficam hospedados em um servidor web. Esses servidores ou provedores de conteúdo cobram uma taxa mensal para disponibilizar a sua aplicação 24h por dia na internet. Também há provedores de conteúdo que são gratuitos para um limite básico de

recursos. Além disso, se você quiser hospedar uma página na internet (para um cliente seu), terá de registrar um domínio. O site onde você pode verificar a disponibilidade e adquirir um domínio é o registro.br.

Mas voltemos ao nosso assunto de desenvolvimento de aplicações web. Alguns dos exemplos implementados no livro fizeram a inclusão e consulta de dados a partir da página que desenvolvemos: qual é a diferença deles para o que vamos abordar neste capítulo? A diferença é que, nos exemplos anteriores, a programação foi realizada na máquina do cliente. Os dados armazenados de apostas (no “Descubra o peso da Melancia”) ou de filmes, por exemplo, estavam vinculados ao navegador do usuário. Se esse usuário acessar a aplicação a partir de outra máquina, ou até mesmo outro navegador, não irá ter acesso a esses dados.

A programação que vamos realizar neste capítulo ocorre em um servidor web. A partir do Node.js é possível simular a existência de um servidor em nossa máquina e realizar todos os testes do projeto. Quando pronto, podemos “fazer o *deploy*”, ou seja, implantar a aplicação em um servidor real. A partir de então, os cadastros realizados passam a estar disponíveis para acesso a partir de qualquer equipamento que tenha acesso à internet (respeitados os devidos controles de segurança)

Há alguns anos, as aplicações desenvolvidas para rodar em um servidor web eram pensadas e construídas de forma conjunta. A mesma equipe ou profissional desenvolvia tanto a parte que iria manipular os dados no servidor (back-end), quanto a parte de apresentação desses dados (front-end). A linguagem PHP era soberana nesse formato de aplicações. Muitos projetos foram e são desenvolvidos em PHP nesse modelo, pois funcionam perfeitamente bem. Sim, eu também fazia parte do fã clube do PHP... :)

Diante da maior necessidade de as aplicações serem distribuídas, principalmente devido ao aumento da demanda por apps mobile, ganhou força a ideia de separar o back-end do front-end. Assim, é possível desenvolver um projeto back-end, que recebe e envia dados em um formato padrão, e receber requisições de aplicações web, mobile ou até mesmo desktop. Mesmo que uma aplicação, por exemplo, seja solicitada por uma

empresa para rodar apenas na web, ela pode expandir posteriormente para outras plataformas. O back-end recebe e envia dados, independentemente de qual seja a plataforma do software cliente.

E a linguagem JavaScript vem ampliando a sua utilização nesse segmento, a partir do Node.js. Uma das razões para esse avanço é justamente a possibilidade de a equipe utilizar a mesma tecnologia – tanto para o desenvolvimento do front-end, quanto do back-end. No Capítulo 3 foram apresentados os passos para instalar o Node.js e o pacote prompt-sync. Neste capítulo, o pacote que será utilizado é o Express, descrito a seguir.

12.1 Express

O Express é um framework para Node.js que fornece um conjunto de recursos para o desenvolvimento de aplicações web ou móveis. A ideia central de um framework é dispor de recursos que visam otimizar o processo de construção de aplicações. A documentação sobre o Express pode ser obtida no endereço: <https://expressjs.com/>. Gosto bastante de recorrer, também em minhas aulas, ao site oficial do software. Nele, é possível visualizar a descrição detalhada dos recursos do software e exemplos de códigos indicados pela equipe que o desenvolveu. Além disso, criar o hábito de consultar com frequência o site oficial do framework e dos pacotes utilizados em nossos projetos nos mantém atualizados sobre suas novas versões.

Após realizar a instalação do Node.js (processo descrito no Capítulo 3), é necessário adicionar o Express ao projeto. Crie a pasta `cap12`, dentro da pasta `livrojs`, como nos capítulos anteriores. Após, abra um prompt, acesse a pasta criada e execute o comando:

```
C:\livrojs\cap12>npm init -y
```

O comando `npm init` apresenta perguntas sobre o seu projeto, como nome, versão e autor. Com a opção `-y` os valores padrões são atribuídos. Após a conclusão desse comando, observe que um arquivo `package.json` é criado e seu conteúdo é exibido na tela. Ele contém as informações sobre o seu projeto, incluindo os pacotes necessários para que ele funcione corretamente. O

próximo comando é:

```
C:\livrojs\cap12>npm i express
```

O npm i express (ou npm install express) irá adicionar o pacote Express ao projeto. Após a sua instalação, a pasta node_modules é criada, assim como uma nova seção no arquivo package.json, indicando que o projeto necessita do Express para ser executado. Caso você disponibilize a sua aplicação para a comunidade no Github, por exemplo, quem baixá-la ficará sabendo quais pacotes são essenciais para o funcionamento da sua aplicação.

Vamos então criar o arquivo app.js com o conteúdo descrito a seguir. Ele contém, basicamente, os comandos indicados na seção “Getting Started” do pacote, com a tradução das mensagens. Trocamos a porta para 3001, visto que usaremos a porta 3000 para a aplicação de front-end a ser desenvolvida no capítulo seguinte.

Programa app.js (cap12/app.js)

```
const express = require('express');
const app = express();
const port = 3001;

app.get('/', (req, res) => {
  res.send('Olá... Bem-vindo!');
});

app.listen(port, () => {
  console.log(`Servidor rodando em http://localhost:${port}`);
});
```

No Express, as aplicações funcionam a partir da ideia de rotas – que são caminhos que podem ser acessados a partir do navegador web. Para rodar a aplicação, faça como nos demais exemplos de programas Node.js – volte ao prompt e execute o comando:

```
c:\livrojs\cap12>node app
```

Em resposta, é exibida a seguinte linha:

```
Servidor rodando em http://localhost:3001
```

Após, abra o navegador e digite a URL: http://localhost:3001. A Figura 12.1 exibe a tela com a resposta do programa ao acionamento dessa rota.

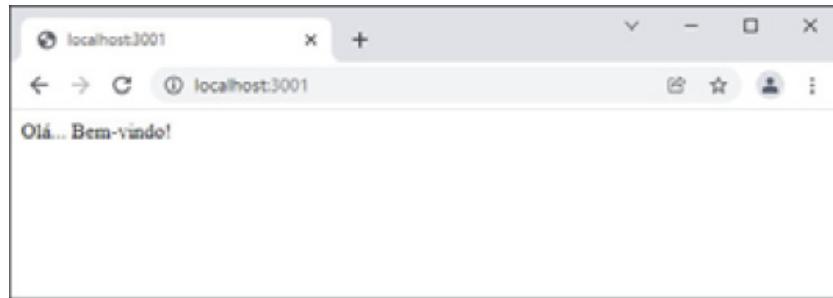


Figura 12.1 – A rota inicial é acionada e a mensagem, exibida.

Ou seja, ao acionar a rota raiz da aplicação, a partir do método Get, que é o método padrão do acesso a um endereço web pelo browser, a mensagem de retorno do método `res.send()` é exibida. Vamos acrescentar uma nova rota à nossa aplicação. Adicione o seguinte trecho de código ao arquivo `app.js` (antes do `app.listen()`):

```
app.get('/cap12', (req, res) => {
  res.send('<h2>Capítulo 12: Introdução ao Express</h2>');
});
```

Após salvar o arquivo, podemos imaginar que essa nova rota já esteja disponível para acesso no browser. Mas há um detalhe que precisa ser feito para que isso ocorra, e ele é descrito na próxima seção.

12.2 Nodemon

Quando executamos o comando `node app`, o Node.js compila o código salvo naquele momento. Ele não “escuta” as alterações realizadas no arquivo `app.js`. Dessa forma, teríamos de finalizar a execução do comando `node app` no terminal e, em seguida, executá-lo novamente. Esse processo não é nada prático.

Para resolver esse problema, podemos adicionar aos nossos projetos o pacote `nodemon`. Para instalá-lo, volte ao prompt de comandos, pare a execução do `node app` (com **Control+C** ou **Command+C**) e execute o seguinte comando:

```
c:\livrojs\cap12>npm i --save-dev nodemon
```

Observe que acrescentamos a opção `--save-dev` no comando. Para que ele serve? Para indicar que o `nodemon` não é uma dependência do projeto, e

sim um recurso adicionado para otimizar o desenvolvimento da aplicação pelo desenvolvedor. Se você observar o conteúdo do arquivo package.json após a inserção desse comando, seu conteúdo deve estar semelhante ao descrito a seguir:

```
{  
  "name": "cap12",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.2"  
  },  
  "devDependencies": {  
    "nodemon": "^2.0.15"  
  }  
}
```

Aproveite que o arquivo está aberto e ajuste o nome do nosso arquivo principal para app.js ("main": "app.js"). Salve o arquivo e retorne ao prompt de comandos. Agora, para testar a nova rota cap12 e fazer com que o Node.js fique na “escuta” de alterações no código, rode nodemon app no prompt de comandos:

```
C:\livrojs\cap12>nodemon app  
[nodemon] 2.0.7  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Servidor rodando em http://localhost:3001
```

Em seguida, teste a rota criada a partir do seu navegador web, inserindo a url <http://localhost:3001/cap12>.

12.3 Rotas POST e Formato JSON

No programa `app.js` é possível observar que cada arrow function associada a uma rota recebe dois parâmetros: `req` (request) e `res` (response). Eles podem receber qualquer nome, mas `req` e `res` são os geralmente utilizados. O parâmetro `req` contém as informações que são passadas para a rota – por quem a acionou. Já `res` refere-se à resposta da rota – como as mensagens exibidas nas duas rotas anteriormente criadas.

Vamos agora utilizar o parâmetro `request` e ver como é possível receber dados vindos de um programa web. Podemos imaginar que um dos programas anteriores que fez o cadastro de dados em `localStorage` agora vai enviar esses dados para a nossa aplicação que está rodando em um servidor web. Como podemos obter esses dados no servidor?

O primeiro passo é “combinar” com a página cliente um formato para a troca de informações. Um dos formatos mais utilizados na atualidade é o JSON (JavaScript Object Notation). Ele é semelhante à ideia de um registro em JavaScript. Se quisermos fazer isso no cadastro de filmes, do Capítulo 10, o programa JavaScript deve enviar os dados ao servidor no seguinte formato:

```
{  
  "titulo": "A Jornada de Vivo",  
  "genero": "Animação"  
}
```

O método JavaScript que deve ser utilizado no programa cliente para enviar os dados é o `fetch()` ou, então, podemos recorrer ao pacote `axios`. Mas esse assunto será tratado no Capítulo 13. Vamos ver agora como obter e exibir os dados que foram enviados. Acrescente o seguinte trecho de código ao arquivo `app.js` (antes do `app.listen()`):

```
// para reconhecer os dados recebidos como sendo um objeto no formato JSON  
app.use(express.json());  
app.post('/filmes', (req, res) => {  
  // const titulo = req.body.titulo;  
  // const genero = req.body.genero;  
  const { titulo, genero } = req.body;  
  res.send(`Filme: ${titulo} - Gênero: ${genero}, recebido...`);
```

});

A linha `app.use(express.json())` adiciona um middleware para reconhecer que o formato “combinado” para os dados a serem enviados pelo cliente e recebidos pelo programa no servidor é o JSON. Essa instrução é uma espécie de parser, como as utilizadas para converter formatos de variáveis. Na seção seguinte, retornaremos ao tema middleware.

Um primeiro detalhe na declaração da rota `/filmes` é a indicação do método POST (ou verbo POST). Os verbos HTTP servem para indicar o tipo de requisição a ser feita para um servidor. Os quatro principais verbos, conforme a metodologia RESTful, são descritos na Tabela 12.1.

Tabela 12.1 – Verbos HTTP na metodologia RESTful

Verbo	Ação requisitada pelo cliente ao servidor
GET	Consulta de dados
POST	Inclusão de dados
PUT	Alteração de dados
DELETE	Exclusão de dados

Para obter os dados vindos do form, utilizamos o parâmetro `req` (`request`). Como os dados vêm no formato de um array de objetos, pode-se obtê-los criando as variáveis linha a linha (forma comentada no exemplo) ou utilizando a atribuição via desestruturação – que se torna mais interessante. Falta, então, vermos como testar essa rota. Já vimos que as rotas que utilizam o verbo GET podem ser acessadas diretamente no navegador. As demais podem ser testadas a partir de nossos programas cliente ou, então, a partir de um software desenvolvido para essa finalidade – como o **Insomnia** ou o **Postman**.

Há também a opção de fazer isso diretamente pela web, em sites como o **reqbin.com**. Ao acessar uma rota localhost no reqbin.com, ele indica a necessidade de instalar a extensão do software no Chrome. Vou utilizar o reqbin.com – pela praticidade do online. Mas sinta-se à vontade caso preferir instalar outro software dessa categoria – você irá recorrer a ele muitas vezes durante o desenvolvimento do seu back-end. A Figura 12.2 demonstra o funcionamento da rota, com os dados passados pelo método POST.

Ok, já conseguimos recuperar as informações enviadas a partir do método POST. O próximo passo é inserir esses dados em um banco de dados – objetivo deste capítulo. Mas, antes, vamos discutir sobre alguns recursos envolvendo middleware e rotas com parâmetros.

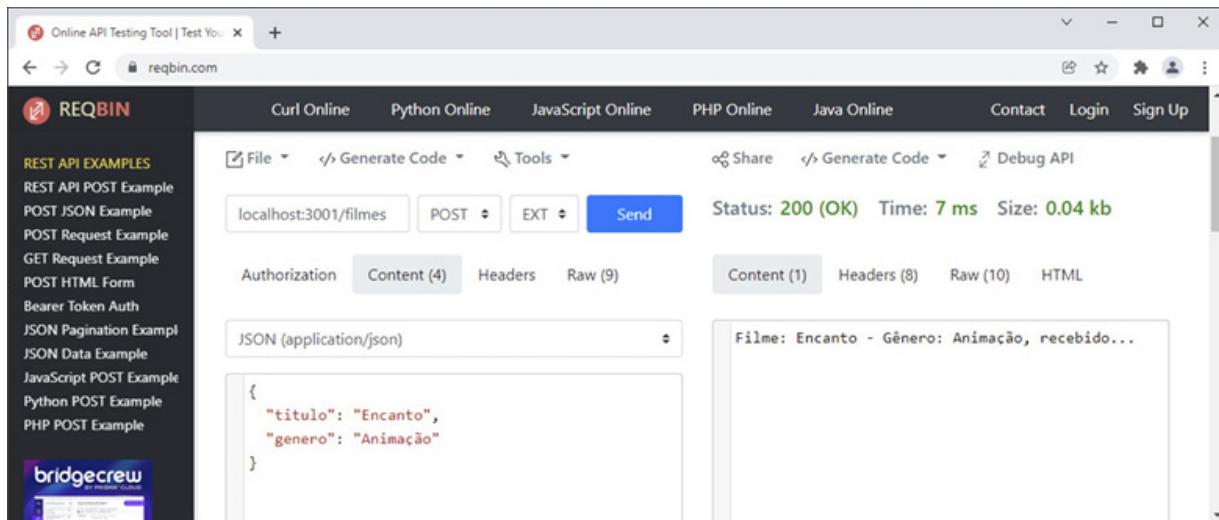


Figura 12.2 – Teste da rota que recebe os dados por POST e exibe uma mensagem.

12.4 Middlewares

Um middleware é uma espécie de mediador entre duas partes, algo que fica no meio (middle). Vamos construir um exemplo, para ilustrar uma possível funcionalidade para um middleware. Digamos que os administradores de um sistema gostariam de analisar quantas vezes ou em quais horários determinadas rotas estão sendo acionadas. Algo como um controle de logs, que poderia registrar ações como transferências financeiras, exclusões ou alterações de dados, por exemplo. Nesse caso, antes de acionar essas rotas, podemos fazer o sistema passar por um middleware que irá registrar essas ações. Acrescente o seguinte trecho ao app.js.

```
// Exemplo de Middleware
const log = (req, res, next) => {
  console.log(`..... Acessado em ${new Date()}`);
  next();
}
```

```
app.get('/transfere', log, (req, res) => {
  res.send("Ok! Valor transferido com sucesso... ");
});
```

Toda vez que a rota /transfere for acionada, você verá uma mensagem no prompt do sistema. O middleware tem acesso às variáveis passadas pela rota. O parâmetro next() serve para acionar a rota que se pretende acessar. Poderíamos, portanto, realizar alguma condição e, em alguns casos, não acessar a rota pretendida pelo usuário (caso o valor da transferência exceda o limite diário, por exemplo). Também é possível utilizar app.use(log), antes das rotas a que se deseja acrescentar o middleware (e removê-lo na declaração da rota). Todas as rotas abaixo de app.use(log) passam pelo middleware.

Um detalhe importante nesse último caso é que só podemos usar o app.use() para chamar uma arrow function após a sua declaração. Dessa forma, não podemos colocar app.use(log) no início do arquivo, por exemplo, e depois criar a function a partir do const. Para resolver isso, temos duas opções: a) declarar a const antes de fazer o app.use(); b) declarar a function da forma tradicional, como:

```
function log(req, res, next) {
  console.log(`..... Acessado em ${new Date()}`);
  next();
}
```

12.5 Use o Knex e escolha o banco de dados

Os bancos de dados permitem gerenciar os dados de um sistema, com inúmeros recursos visando garantir a segurança, integridade e disponibilidade dos dados. Armazenar as informações manipuladas por um sistema em um banco de dados é uma premissa de qualquer aplicativo profissional. SQLite, MySQL, PostgreSQL, Oracle e MS SQL são exemplos de sistemas gerenciadores de banco de dados.

Observe que o termo SQL é comum na maioria dos nomes dos bancos de dados. Isso porque SQL (Structured Query Language) é a linguagem padrão para a construção de consultas para bancos de dados relacionais.

As linguagens de programação dispõem de pacotes para permitir a conexão do sistema que você está desenvolvendo com um banco de dados. E o Node.js segue esse padrão. Assim, se quisermos conectar o nosso sistema com o MySQL, precisaremos adicionar o pacote do Node.js para o MySQL. Há pacotes disponíveis para conexão com os principais bancos de dados disponíveis no mercado.

Além da escolha do banco de dados, é necessário também definir qual a forma que você vai codificar o seu programa para interagir com o banco de dados. São três essas abordagens: a) inserir os comandos SQL diretamente no meio do código do programa; b) utilizar um pacote que converte códigos de programa em instruções SQL, conhecidos como Query Builders, como o pacote Knex; c) utilizar um package que define modelos de objetos associados às tabelas – o mapeamento objeto relacional (ORM), com destaque para o Sequelize.

Começamos pela escolha da forma de conexão. Diante do que já foi abordado neste livro, entendo que a utilização da segunda forma se mostra a mais adequada. O Knex possui uma ótima documentação, que nos auxilia em todas as etapas do desenvolvimento de um sistema, desde as criação das tabelas com as migrations até a construção de consultas SQL avançadas. O site onde você encontra essa documentação é o <https://knexjs.org>.

Uma das vantagens da utilização do Knex é a facilidade de migrar a sua aplicação de banco de dados. Para trocar do SQLite para o MySQL, por exemplo, é necessário realizar a instalação do pacote necessário para a conexão e de ajustes no arquivo que contém as especificações iniciais da conexão. Os comandos para criar as tabelas, inserir, alterar, consultar ou excluir os dados no programa são os mesmos.

Em nosso exemplo, vamos utilizar o SQLite, que é um banco de dados, como o nome sugere, mais leve e que não requer a instalação do sistema gerenciador de banco de dados em sua máquina. Começamos, então, pela instalação dos pacotes do Knex e do SQLite. Vá ao prompt de comandos da sua aplicação e insira os comandos:

```
C:\livrojs\cap12>npm i knex  
C:\livrojs\cap12>npm i sqlite3
```

Caso você opte por utilizar outro banco de dados, na documentação do Knex existe a indicação do driver adequado, bem como exemplos dos parâmetros de configuração da conexão. Todos os códigos desenvolvidos nas etapas seguintes deste capítulo foram testados com SQLite e MySQL – sem nenhuma alteração de código (exceto o arquivo de configuração da conexão).

12.6 Criação de tabelas com Migrations

Muito bem, após algumas considerações iniciais sobre bancos de dados e a instalação do pacote Knex e do driver para conexão, vamos ao próximo passo. Ainda no prompt, rode o seguinte comando:

```
C:\livrojs\cap12>npx knex init
```

E você irá receber como resposta: Created ./knexfile.js. Esse é o arquivo que vai indicar o driver de banco de dados a ser utilizado e as configurações da conexão. Edite o arquivo para que contenha o código:

```
module.exports = {
  development: {
    client: 'sqlite3',
    connection: {
      filename: './data/editora.db3'
    },
    useNullAsDefault: true,
    migrations: {
      directory: './data/migrations'
    },
    seeds: {
      directory: './data/seeds'
    }
  }
}
```

A etapa seguinte consiste na criação das tabelas do sistema. Uma tabela contém a definição dos campos ou atributos que devem existir em cada coleção de dados de um sistema. Por exemplo: em um sistema de controle de estoque, poderíamos ter as tabelas de produtos, marcas, clientes, funcionários, fornecedores, compras e vendas. Elas devem estar

relacionadas. Um produto deve conter o código de uma marca, uma venda deve conter o código do cliente e ainda se subdividir em itens da venda – que deve conter o código do produto etc.

Nosso projeto irá realizar o cadastro de livros da Editora Novatec. Então, vamos criar a tabela de livros. Para isso, vamos utilizar um conceito comum a outros frameworks como Laravel (PHP) e Flask (Python), que são as migrations. Um arquivo de migrations contém a definição da estrutura de cada tabela da aplicação e das modificações que ocorrem ao longo do desenvolvimento. Assim, caso você precise utilizar outro computador ou quando estiver com a aplicação pronta para disponibilizá-la na internet, basta rodar os arquivos de migrations e as tabelas serão criadas a partir de um único comando. As migrations também são úteis quando quisermos desfazer uma modificação na estrutura de alguma tabela do sistema. Para criar o arquivo que irá conter a definição da estrutura da tabela de livros, rode o seguinte comando no prompt de comandos:

```
C:\livrojs\cap12>npx knex migrate:make create_livros
```

Como resposta você irá obter a indicação de que o arquivo foi criado.

```
Using environment: development
```

```
Created Migration: C:\livrojs\cap12\data\migrations\20220107213419_create_livros.js
```

Foi criada a pasta `data\migrations` em seu projeto. Os números iniciais do nome do arquivo estão relacionados com data e hora. Edite esse arquivo com o Visual Studio Code para que contenha o código descrito a seguir:

```
exports.up = function (knex) {
  return knex.schema.createTable("livros", (table) => {
    table.increments();
    table.string("titulo", 80).notNullable();
    table.string("autor", 60).notNullable();
    table.integer("ano", 4).notNullable();
    table.decimal("preco", 9.2).notNullable();
    table.string("foto", 100).notNullable();
  });
}
```

```
exports.down = function (knex) {
  return knex.schema.dropTable("livros");
}
```

Salve o arquivo, retorne ao prompt e rode a migration com o comando:

```
C:\livrojs\cap12>npx knex migrate:latest
```

Pronto! Nosso banco de dados SQLite está criado, assim como a tabela de livros. Observe na pasta /data a existência do arquivo editora.db3.

12.7 Seeds: “semeando” dados iniciais

Durante o processo de desenvolvimento de aplicações, necessitamos que alguns registros sejam inseridos nas tabelas para a realização de testes. Esses registros também podem ser inseridos a partir de arquivos de códigos e executados utilizando o Knex. Rode o comando que cria esse arquivo:

```
C:\livrojs\cap12>npx knex seed:make 001_add_livros
```

Agora, edite esse arquivo – dentro da pasta ./data/seeds para inserir 5 livros na tabela anteriormente criada.

```
exports.seed = function (knex) {
  return knex("livros").del()
    .then(function () {
      return knex("livros").insert([
        {
          titulo: "Web Design Responsivo", autor: "Maurício Samy Silva", ano: 2014,
          preco: 73.0, foto: "https://s3.novatec.com.br/capas/9788575223925.jpg",
        },
        {
          titulo: "Proteção Moderna de Dados", autor: "W. Curtis Preston", ano: 2021,
          preco: 97.0, foto: "https://s3.novatec.com.br/capas/9786586057843.jpg",
        },
        {
          titulo: "SQL em 10 Minutos por Dia", autor: "Ben Forta", ano: 2021,
          preco: 79.0, foto: "https://s3.novatec.com.br/capas/9786586057447.jpg",
        },
        {
          titulo: "CSS Grid Layout", autor: "Maurício Samy Silva", ano: 2017,
          preco: 45.0, foto: "https://s3.novatec.com.br/capas/9788575226322.jpg",
        },
        {
          titulo: "Python para análise de dados", autor: "Wes McKinney", ano: 2018,
          preco: 132.0, foto: "https://s3.novatec.com.br/capas/9788575226476.jpg",
        },
      ]);
    });
}
```

```
});  
}
```

Para inserir os dados, volte ao prompt de comandos e execute a linha a seguir:

```
C:\livrojs\cap12>npx knex seed:run
```

Você receberá uma mensagem indicando que os dados foram “semeados”. Siga os próximos passos para ver que o nosso back-end irá exibir esses dados a partir da rota de consulta de dados. Caso queira baixar um programa para gerenciamento do banco de dados sqlite, você pode acessar a seção de download do site oficial do software (<https://www.sqlite.org/download.html>). Há versões para Windows, Mac e Linux, que incluem o “comamnd-line shell”. Basta descompactar os arquivos em uma pasta (como c:\sqlite3) e executar comandos SQL. Os seguintes comandos podem ser executados, no prompt, para listar os registros inseridos:

```
C:\livrojs\cap12>cd data  
C:\livrojs\cap12\data>\sqlite3\sqlite3 editora.db3  
SQLite version 3.37.0 2021-11-27 14:13:22  
Enter ".help" for usage hints.  
sqlite> select id, titulo, ano from livros;  
1|Web Design Responsivo|2014  
2|Proteção Moderna de Dados|2021  
3|SQL em 10 Minutos por Dia|2021  
4|CSS Grid Layout|2017  
5|Python para análise de dados|2018  
sqlite> .quit
```

12.8 Database Config e express.Router

O arquivo de configuração criado anteriormente (knexfile.js) serve para definir o driver do banco de dados e parâmetros da conexão utilizados no processo de criação das migrations e seeds da aplicação. Agora, precisamos criar um arquivo que vai conter esses detalhes de conexão com o banco de dados a serem utilizados pelo programa. Para isso, crie o arquivo db_config.js, dentro da pasta data, e insira as seguintes linhas:

```
const knex = require('knex');  
const config = require("../knexfile.js");
```

```
const dbKnex = knex(config.development);
module.exports = dbKnex;
```

Observe que estamos importando os dados de conexão do arquivo knexfile.js, que está na pasta raiz do projeto. Antes ainda de passarmos para a criação das funções contendo os métodos para realizar as operações de cadastro no banco de dados, vamos conversar sobre o express.Router. Nós inserimos algumas rotas no arquivo app.js para a realização de testes. Porém, quando a aplicação for crescendo em quantidade de recursos, dispor de todas essas rotas em um mesmo arquivo não é uma boa prática de programação.

O express.Router permite organizar as rotas de um projeto em arquivos distintos. Assim, as rotas relacionadas ao cadastro de livros, por exemplo, podem ficar em um arquivo chamado livros.js. Dessa forma, nosso projeto pode crescer e manter a organização do código. Crie, então, o arquivo livros.js na pasta raiz do projeto (cap12), com o seguinte conteúdo inicial:

```
const express = require("express"); // pacotes a serem utilizados
const router = express.Router();

const dbKnex = require("./data/db_config"); // dados de conexão com o banco de dados

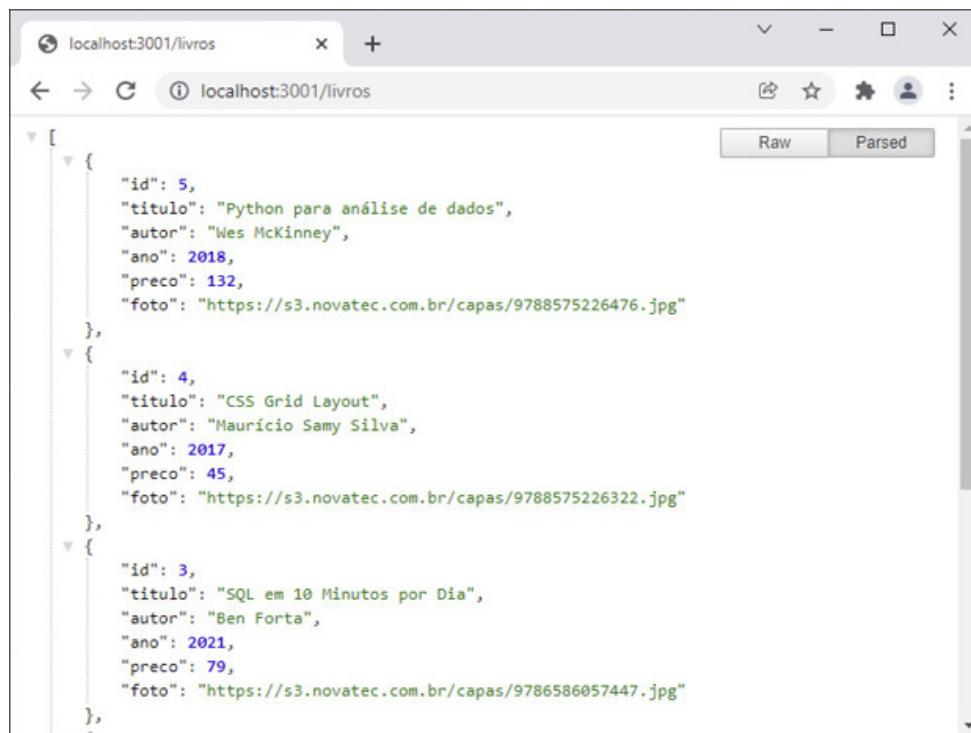
// método get é usado para consulta
router.get("/", async (req, res) => {
  try {
    // para obter os livros pode-se utilizar .select().orderBy() ou apenas .orderBy()
    const livros = await dbKnex("livros").orderBy("id", "desc");
    res.status(200).json(livros); // retorna statusCode ok e os dados
  } catch (error) {
    res.status(400).json({ msg: error.message }); // retorna status de erro e msg
  }
});
module.exports = router;
```

O próximo passo é importar esse arquivo no app.js e associar o nome da rota a ser utilizada para a realização do cadastro de livros. Abra o app.js e acrescente as linhas:

```
// Arquivo com rotas para o cadastro de livros
const livros = require('./livros');
```

```
app.use('/livros', livros); // identificação da rota e da const (require) associada
```

Salve os arquivos e vamos testar nossa nova rota! Como é uma requisição por GET, podemos acessar pelo próprio navegador. Não se esqueça de conferir se o servidor está rodando (nodemon app). A Figura 12.3 exibe os dados de retorno da rota com os livros no formato JSON (a extensão JSON Formatter do Chrome foi instalada para deixar os dados com uma formatação mais “elegante”).



```
[{"id": 5, "titulo": "Python para análise de dados", "autor": "Wes McKinney", "ano": 2018, "preco": 132, "foto": "https://s3.novatec.com.br/capas/9788575226476.jpg"}, {"id": 4, "titulo": "CSS Grid Layout", "autor": "Maurício Samy Silva", "ano": 2017, "preco": 45, "foto": "https://s3.novatec.com.br/capas/9788575226322.jpg"}, {"id": 3, "titulo": "SQL em 10 Minutos por Dia", "autor": "Ben Forta", "ano": 2021, "preco": 79, "foto": "https://s3.novatec.com.br/capas/9786586057447.jpg"}]
```

Figura 12.3 – A rota /livros retorna a lista dos livros cadastrados no formato JSON.

12.9 Async e await

Vamos comentar as novidades do primeiro método do arquivo livros.js, com destaque para o **async** e **await**. Observe que na declaração da função foi acrescentado um **async**. Ele deve ser utilizado em conjunto com o **await** e eles são necessários porque a programação no Node.js é assíncrona. Mas o que é uma programação assíncrona? De forma resumida, é uma programação em que os comandos ocorrem “sem sincronia” de conclusão.

Em linguagens síncronas, como Python e PHP, por exemplo, os comandos são executados uma linha após a outra e a linha seguinte só inicia após a conclusão da anterior. Para fazer com que dois processos sejam executados em paralelo, essas linguagens fazem uso de threads.

No Node.js, se tivermos as linhas A, B e C para serem executadas em sequência, mas a linha B demorar para responder, o programa já inicia a execução da linha C. Então, pode ocorrer que a resposta da execução da linha C ocorra antes da B. No geral, isso não causa problemas – pelo contrário, torna o programa mais eficiente. Mas, caso a execução da linha C dependa da conclusão da linha B, teremos, sim, um problema. Uma das formas de resolver isso é utilizando o `await`. Ele faz com que o programa aguarde a conclusão da linha precedida pela palavra `await`, para, então, executar a próxima.

Outra instrução ainda não utilizada em nossos programas é a `try... catch`. Ela funciona, basicamente, como uma condição para criar um bloco protegido de código. Se ocorrer algum erro nos comandos desse bloco, o programa não irá abortar, mas, sim, executará o comando alternativo. O bloco protegido deve estar dentro do `try`. O comando alternativo, no `catch`.

Assim, em nossa `function`, caso ocorra algum erro, como falha de conexão, o programa retorna uma mensagem, que deve ser mais significativa do que a gerada pela linguagem de programação.

Observe que, após todo o processo de instalação dos pacotes e definição das configurações da conexão com o banco de dados, a instrução para recuperar os registros cadastrados na tabela de livros, utilizando o Knex, é bastante simples: `dbKnex("livros").orderBy("id", "desc")`. Ela faz com que a lista de livros seja retornada em ordem decrescente de código – os últimos cadastros são exibidos primeiro. Caso quiséssemos obter a lista sem indicar uma ordenação, o comando deveria ser: `dbKnex("livros").select()`. O `.select()` pode ser omitido ao usar o `.orderBy()`. O primeiro parâmetro do `orderBy()` indica o campo da tabela a ser ordenado. O segundo ("`desc`") indica que a ordem será decrescente – sem esse parâmetro, os registros são recuperados em ordem crescente ou alfabética.

12.10 Status Code HTTP

Um último e importante detalhe de código, ainda nos comandos dessa função que retorna a listagem dos livros cadastrados no banco de dados, diz respeito aos status code retornados. Observe que há um `res.status(200)` e `res.status(400)`. O primeiro serve para indicar que a operação foi bem-sucedida, enquanto o segundo sinaliza um erro.

Esse códigos de status de resposta, fazem parte do protocolo HTTP. Os valores da faixa 100..199 são respostas de informação; da faixa 200..299, são respostas de sucesso; 300..399, usados para redirecionamentos; 400..499 indicam erros do cliente, e, por fim, 500..599 indicam erros do servidor.

Em nosso exemplo, os seguintes códigos de retorno serão utilizados: 200 (requisição foi bem-sucedida), 201 (requisição bem-sucedida e dados inseridos com sucesso) e 400 (erro – requisição inválida enviada ao servidor).

12.11 Rotas para a realização do CRUD

As operações básicas realizadas em uma tabela do banco de dados são também chamadas de CRUD (Create, Read, Update e Delete). Na primeira rota, foi realizada a consulta dos dados. Acrescente agora ao arquivo `livros.js` as seguintes linhas de código:

```
// Método post é usado para inclusão
router.post("/", async (req, res) => {
    // faz a desestruturação dos dados recebidos no corpo da requisição
    const { titulo, autor, ano, preco, foto } = req.body;

    // se algum dos campos não foi passado, irá enviar uma mensagem de erro e retornar
    if (!titulo || !autor || !ano || !preco || !foto) {
        res.status(400).json({ msg: "Enviar titulo, autor, ano, preco e foto do livro" });
        return;
    }

    // caso ocorra algum erro na inclusão, o programa irá capturar (catch) o erro
    try {
```

```

// insert, faz a inserção na tabela livros (e retorna o id do registro inserido)
const novo = await dbKnex("livros").insert({ titulo, autor, ano, preco, foto });
res.status(201).json({ id: novo[0] }); // statusCode indica Create
} catch (error) {
  res.status(400).json({ msg: error.message }); // retorna status de erro e msg
}
});

// Método put é usado para alteração. id indica o registro a ser alterado
router.put("/:id", async (req, res) => {
  const id = req.params.id; // ou const { id } = req.params
  const { preco } = req.body; // campo a ser alterado
  try {
    // altera o campo preco, no registro cujo id coincidir com o parâmetro passado
    await dbKnex("livros").update({ preco }).where("id", id); // ou .where({ id })
    res.status(200).json(); // statusCode indica Ok
  } catch (error) {
    res.status(400).json({ msg: error.message }); // retorna status de erro e msg
  }
});
});

// Método delete é usado para exclusão
router.delete("/:id", async (req, res) => {
  const { id } = req.params; // id do registro a ser excluído
  try {
    await dbKnex("livros").del().where({ id });
    res.status(200).json(); // statusCode indica Ok
  } catch (error) {
    res.status(400).json({ msg: error.message }); // retorna status de erro e msg
  }
});
};

module.exports = router;

```

A rota de inclusão, que recebe os dados pelo método POST, contém uma validação para verificar se as variáveis foram passadas com conteúdo. No capítulo seguinte, você vai ver que os campos de formulário do front-end também serão validados (required). De qualquer forma, é importante fazer a validação no back-end, pois, como comentado, outras aplicações podem vir a utilizar esse Web Service.

O processo de inclusão em si é realizado pela intrução `insert` do Knex. Caso a operação ocorra com sucesso, um código de status 201 é retornado, além do

id do registro inserido. A aplicação de front-end utiliza esses dados para exibir mensagem ao usuário do sistema. A Figura 12.4 ilustra a forma de acionar a rota de inclusão e o formato dos dados que devem ser enviados no formato JSON.

As rotas definidas para realizar a alteração (put) e a exclusão (delete) de dados contém o símbolo `/:id`. O símbolo “`:`” na frente do `id` serve para indicar que nesse local da rota deve ser passado um parâmetro. Então para fazer a exclusão do registro de código 10, por exemplo, deve-se indicar o método delete e chamar a rota da seguinte forma:

`http://localhost:3001/livros/10`

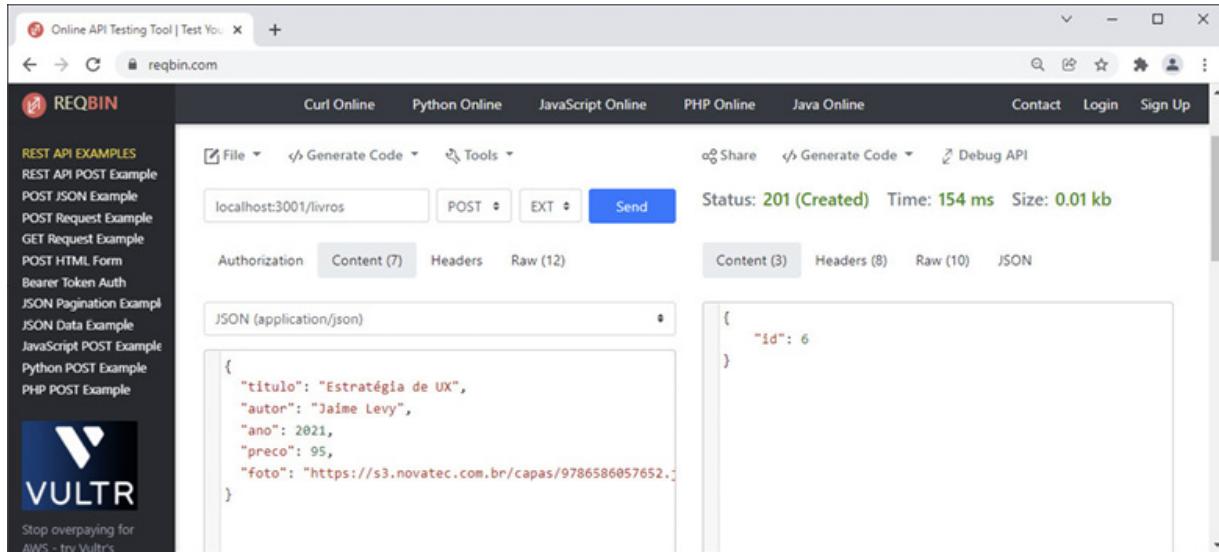


Figura 12.4 – Método de envio e formato dos dados a serem enviados para inclusão.

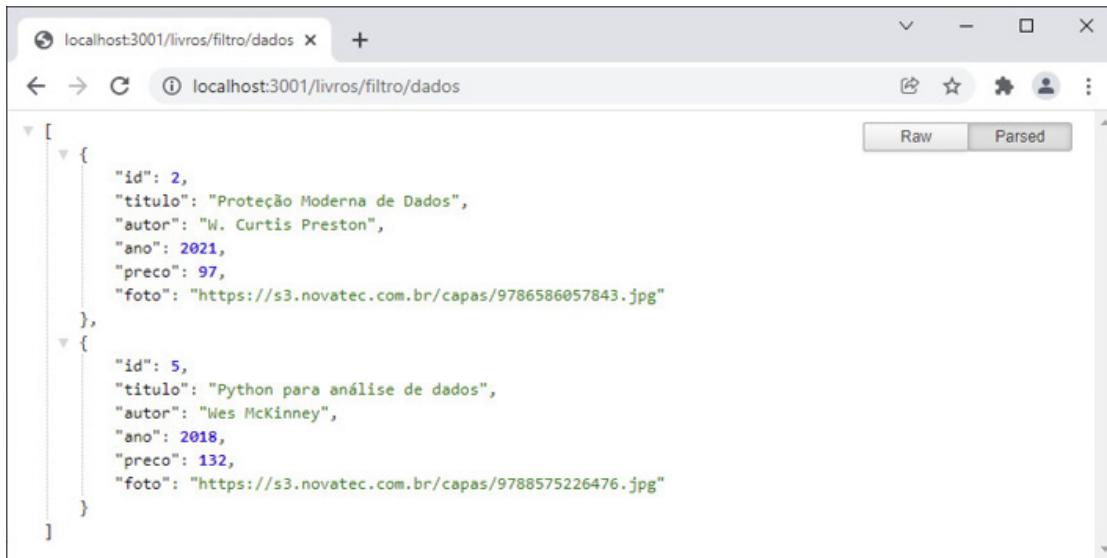
12.12 Filtros, totalizações e agrupamentos

Vamos acrescentar mais três novas rotas ao nosso back-end, que serão utilizadas pela aplicação web do capítulo seguinte. A primeira delas realiza um filtro, retornando apenas os livros que contenham a palavra-chave recebida como parâmetro. O filtro irá verificar se a palavra consta no título do livro ou no nome do autor. O operador like do SQL, com o uso de caracteres curingas, verifica se a palavra existe em qualquer local da string. Nas consultas SQL não há diferenciação entre caracteres maiúsculos ou

minúsculos nas comparações. Acrescente, antes do module.exports=router, as seguintes linhas de código:

```
// Filtro por título ou autor
router.get("/filtro/:palavra", async (req, res) => {
  const palavra = req.params.palavra; // palavra do título ou autor a pesquisar
  try {
    // para filtrar registros, utiliza-se .where(), com suas variantes
    const livros = await dbKnex("livros")
      .where("titulo", "like", `%${palavra}%`)
      .orWhere("autor", "like", `%${palavra}%`);
    res.status(200).json(livros); // retorna statusCode ok e os dados
  } catch (error) {
    res.status(400).json({ msg: error.message }); // retorna status de erro e msg
  }
});
```

A Figura 12.5 ilustra o funcionamento da rota de filtro, passando a palavra “dados” como parâmetro.



The screenshot shows a browser window with the URL `localhost:3001/livros/filtro/dados`. The page displays a JSON array of two book objects. Each book object contains fields: id, título, autor, ano, preço, and foto. The first book has id 2, titled "Proteção Moderna de Dados", and the second has id 5, titled "Python para análise de dados". Both books have their respective URLs for the book cover photo.

```
[{"id": 2, "título": "Proteção Moderna de Dados", "autor": "W. Curtis Preston", "ano": 2021, "preço": 97, "foto": "https://s3.novatec.com.br/capas/9786586057843.jpg"}, {"id": 5, "título": "Python para análise de dados", "autor": "Wes McKinney", "ano": 2018, "preço": 132, "foto": "https://s3.novatec.com.br/capas/9788575226476.jpg"}]
```

Figura 12.5 – Teste de funcionamento da rota de filtro.

Para concluir o nosso back-end, insira após a rota de filtro duas funções utilizadas para a obtenção de dados estatísticos do cadastro de livros.

```
// Resumo do cadastro de livros
router.get("/dados/resumo", async (req, res) => {
  try {
    // métodos que podem ser utilizados para obter dados estatísticos da tabela
    const consulta = await dbKnex("livros")
```

```

    .count({ num: "*" })
    .sum({ soma: "preco" })
    .max({ maior: "preco" })
    .avg({ media: "preco" });
const { num, soma, maior, media } = consulta[0];
res.status(200).json({ num, soma, maior, media: Number(media.toFixed(2)) });
} catch (error) {
  res.status(400).json({ msg: error.message }); // retorna status de erro e msg
}
});

// Soma dos preços, agrupados por ano
router.get("/dados/grafico", async (req, res) => {
  try {
    // obtém ano e soma do preco dos livros (com o nome total), agrupados por ano
    const totalPorAno = await dbKnex("livros").select("ano")
      .sum({ total: "preco" }).groupBy("ano");
    res.status(200).json(totalPorAno);
  } catch (error) {
    res.status(400).json({ msg: error.message }); // retorna status de erro e msg
  }
});

```

Count, sum, max e avg são funções SQL que realizam, respectivamente, a contagem, soma, obtenção do maior valor e cálculo da média. Já o groupBy faz o agrupamento dos registros por um campo e, no exemplo, retorna também a soma dos preços dos livros. A Figura 12.6 exibe o retorno de uma chamada à rota /livros/dados/grafico que será utilizada em nosso front-end para exibir um gráfico.

A screenshot of a browser window showing a JSON response. The URL is 'localhost:3001/livros/dados/grafico'. The response is a parsed JSON object with an array of objects. Each object has two properties: 'ano' (year) and 'total' (total value). The data is as follows:

```
[{"ano": 2014, "total": 73}, {"ano": 2017, "total": 45}, {"ano": 2018, "total": 132}, {"ano": 2021, "total": 271}]
```

Figura 12.6 – Soma do preço dos livros, agrupados por ano de publicação.

Esses processos de filtros, totalizações e agrupamentos também poderiam ser realizados no front-end, a partir de rotinas de programação, como as que desenvolvemos nos capítulos anteriores. Ou seja, obteríamos toda a lista de livros, percorreríamos os registros a partir de uma rotina de repetição para, então, calcular a soma ou a média de um dos campos, por exemplo. Mas, nesse caso, muitas informações precisariam ser enviadas do servidor ao cliente – o que implicaria um tempo maior de resposta do programa.

Contudo, nem sempre a equipe de desenvolvimento do back-end e a do front-end são as mesmas. E talvez você tenha pensado em um gráfico a ser implementado no front-end que poderia ser muito interessante para os administradores da empresa. Mas os dados para a criação do gráfico não estão no formato que você precisa. Nesse caso, você terá, sim, que obter todos ou um conjunto de dados filtrados do back-end e trabalhar esses dados para que eles fiquem no formato adequado.

12.13 CORS

Da forma como está, ao acessar a rota de consulta de dados, por exemplo, a partir de uma aplicação como a que vamos desenvolver no próximo capítulo, iríamos observar que os dados não seriam recuperados. Ao abrir o console do browser, nas ferramentas do desenvolvedor, iríamos deparar

com a mensagem:

```
Access to XMLHttpRequest at 'http://localhost:3001/livros' from origin 'http://localhost:3000' has  
been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the  
requested resource.
```

Isso ocorre porque servidores web impedem que requisições HTTP de scripts que estejam rodando em uma origem diferente – de plataforma, domínio ou porta, possam ter acesso aos recursos da aplicação nativa. Ou seja, a aplicação do site www.lojax.com.br não pode ter acesso aos dados do Web Service do site www.lojay.com.br. Trata-se, portanto, de um importante controle de segurança.

Contudo, para esse nosso Web Service de estudo, ou em aplicações que tenham por finalidade disponibilizar um conjunto de dados sem maiores preocupações com a segurança, como uma lista de produtos em oferta, por exemplo, podemos liberar o acesso a partir de outras origens. Para fazer isso, é necessário voltar ao prompt de comandos e adicionar o pacote CORS ao projeto.

```
C:\livrojs\cap12>npm i cors
```

Após, acrescente depois das duas primeiras linhas do arquivo `livros.js` o código a seguir:

```
const cors = require("cors");  
router.use(cors());
```

Também é possível indicar que apenas algumas rotas devam ser liberadas – acrescentando o CORS com a sintaxe de middleware anteriormente apresentada, ou então que somente requisições provenientes de domínios de empresas parceiras tenham acesso ao Web Service – a partir da personalização das configurações do CORS. Na documentação de instalação pacote (<https://www.npmjs.com/package/cors>) há detalhes para a implementação desses recursos.

12.14 Considerações finais do capítulo

Muito bem! Concluímos o desenvolvimento do nosso back-end. O sistema está pronto para interagir com qualquer outra aplicação cliente, seja ela desktop, mobile ou web. Esse é um diferencial importante dessa

metodologia. Podemos interagir com um sistema Delphi, que armazene seus dados em Oracle, ou um app Flutter com persistência de dados em Firebase ou, então, PHP com PostgreSQL. A única exigência é que o formato dos dados transacionados seja o JSON – amplamente utilizado e de fácil manipulação.

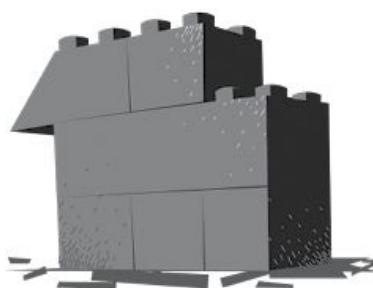
Neste capítulo, utilizamos o Express – um framework do Node.js para o desenvolvimento de aplicações de back-end. Vimos os detalhes iniciais de criação de um projeto Express, definição das primeiras rotas e inicialização do servidor web em nossa máquina.

Também acrescentamos o Knex, um pacote que simplifica o processo de construção de instruções SQL para a manipulação do banco de dados. Um dos benefícios de utilizar o Knex é que, enquanto estamos desenvolvendo a aplicação, podemos interagir com um banco de dados mais simples – como o SQLite. E, ao fazer o “deploy” da aplicação para um servidor web real, podemos migrar para outro banco de dados – como o MySQL ou PostgreSQL, apenas modificando o arquivo de configuração. Outro benefício do Knex é a possibilidade de criar e gerenciar as modificações nas tabelas do banco de dados a partir de arquivos de migrations, além da possibilidade de usar seeds para acrescentar dados iniciais às tabelas.

Após as configurações iniciais e criação da tabela a ser utilizada em nossa aplicação exemplo, criamos os códigos necessários para a realização das rotinas de cadastro, que incluem criação, consulta, alteração e exclusão de dados – também conhecidas como rotinas de CRUD (Create, Read, Update e Delete). Além disso, avançamos um pouco com o acréscimo de duas outras rotas, uma para a realização de filtros nos dados da tabela e outra que retorna dados gerenciais do sistema.

CAPÍTULO 13

Criação de um App: Front-end com React



E agora, a cereja do bolo! Depois de desenvolver os métodos para realizar o cadastro de livros, vamos construir a parte da aplicação que irá montar as telas de inclusão, listagem – incluindo botões para alterar, excluir e filtrar dados e, ainda, a exibição de dados estatísticos e um gráfico. Ou seja, a parte que irá interagir com o usuário. Muitos são os desenvolvedores que preferem trabalhar nesta área: o front-end.

Para fazer isso, vamos utilizar o React – um dos Frameworks JavaScript de maior destaque no mercado. Conforme a documentação oficial do React, disponível no site <https://reactjs.org>, o React é uma biblioteca para a criação de interfaces de usuário.

Na página de abertura do Framework, são destacadas duas características do React: declarativo e baseado em componentes. Sobre ser declarativo, há uma explicação indicando que, de forma eficiente, o React atualiza e renderiza os componentes apenas quando os dados mudam – a partir do uso dos controles de estado. Já em relação aos componentes, eles são um modelo que fazem a montagem de um layout em partes, que combinadas formam a interface da aplicação.

Bom, como o objetivo deste capítulo não é explorar todos os recursos do

Framework (há ótimos livros específicos sobre React), e, sim, construir uma aplicação considerando o que já foi abordado anteriormente, vamos ressaltar estes dois aspectos do Framework: organização em componentes e atualização eficiente da página. E, desse modo, incentivá-los a prosseguir nos seus estudos, seja na área do back-end ou do front-end (ou, quem sabe, nos dois – sendo um desenvolvedor full stack).

Antes de começar com os passos para a criação do projeto que irá interagir com o back-end, vamos pegar um exemplo já desenvolvido no livro e explorar o chamado “controle de estado” no React. Para isso, vamos utilizar o site <https://codesandbox.io/>. Ele permite criar projetos de programação em várias tecnologias. Após acessar o site, selecione Create Sandbox e escolha React. A Figura 13.1 apresenta a página do site, com as pastas e arquivos iniciais de uma aplicação React.

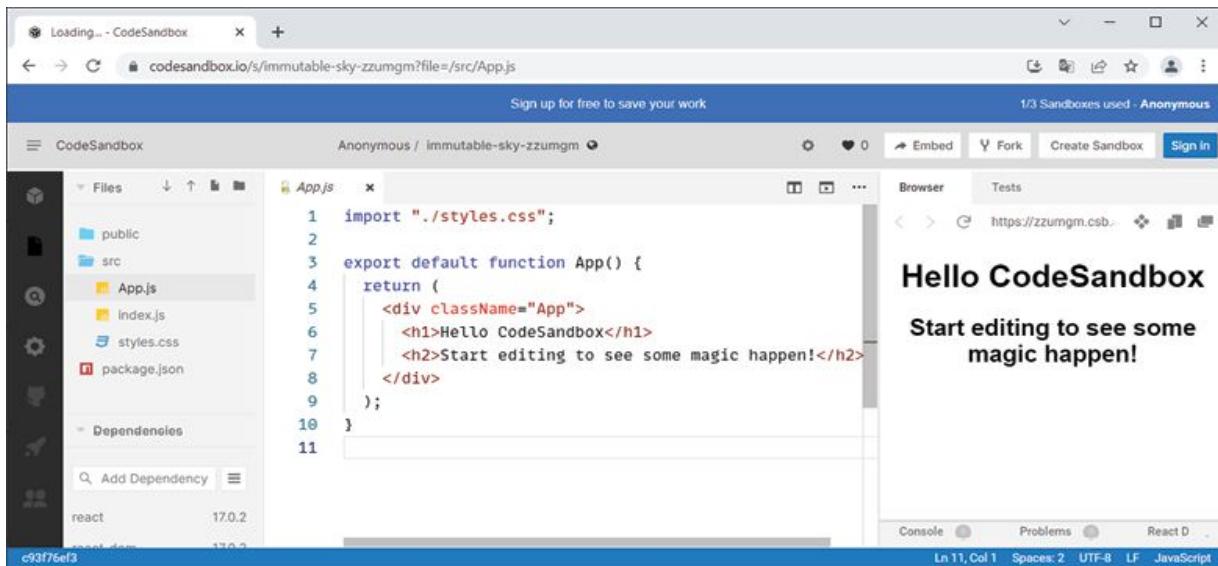


Figura 13.1 – Página do codesandbox.io com a estrutura inicial de pastas e arquivos do React.

Vamos explorar esses arquivos. Começamos pelo arquivo index.html, na pasta public. Abra esse arquivo e você irá observar que no corpo da página há apenas as tags noscript (com a mensagem a ser exibida, caso o JavaScript não esteja habilitado no browser do usuário) e <div id="root"></div>. Ou seja, nada é renderizado pelo index.html. Cabe ao programa JavaScript exibir o conteúdo da página e gerenciar a interação com o usuário.

O segundo arquivo a ser analisado é o `index.js`, da pasta `src` e descrito a seguir:

```
import { StrictMode } from "react";
import ReactDOM from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(
  <StrictMode>
    <App />
  </StrictMode>,
  rootElement
);
```

O Strict Mode, conforme a documentação, é uma ferramenta do React para indicar possíveis problemas no código – não interfere na renderização em si. Após, há as linhas que fazem a importação do ReactDOM e do componente a ser exibido, que é o App. Em seguida, é criada uma referência ao elemento root (do arquivo HTML) e, então, manda-se renderizar o componente App nesse elemento. Ou seja, nada de conteúdo ainda...

Quem vai conter as tags a serem exibidas é o `App.js`. Abra esse arquivo para ver o conteúdo descrito a seguir:

```
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </div>
  );
}
```

Veja que há um import para um arquivo de estilos e a função App. O retorno dessa function é o conteúdo exibido na Figura 13.1. Os seguintes cuidados sobre componentes devem ser observados:

- O retorno de cada componente deve conter um único elemento raiz (com os demais “dentro” desse elemento). Ou seja, se você retirar as tags `<div>`

e `</div>`, vai observar um erro. Caso não haja a necessidade de utilizar uma div, podem ser utilizados `<Fragment>` e `</Fragment>` (com `import {Fragment} from 'react'`) ou a forma abreviada `<>` e `</>`. Um componente pode retornar um form, uma table, ou uma tag nav, por exemplo. Mas não podem conter elementos de mesmo nível sem um elemento “pai”.

- Nomes de componentes devem iniciar por uma letra maiúscula (por padrão, utiliza-se o formato PascalCase).
- Todas as tags de retorno de um componente devem possuir uma abertura e fechamento. Tags que não possuem fechamento, como img, devem ser finalizadas com `>`. Esse código de retorno do elemento é chamado de JSX – que é uma extensão de sintaxe para o JavaScript, muito semelhante ao HTML.

Feitas as considerações iniciais sobre os componentes, vamos implementar um dos programas do livro em React. Trata-se do Programa 6.1, do centro odontológico. Copie o código HTML desse arquivo e cole-o dentro do return da função anterior. Vou ajustar também a sintaxe para o formato das arrow functions.

```
import { Fragment } from "react";
import "./styles.css";

const App = () => {
  return (
    <Fragment>
      <h1>Consultório Odontológico</h1>
      <form>
        <p>Paciente:</p>
        <input type="text" id="inPaciente" required autoFocus />
        <input type="submit" value="Adicionar" />
        <input type="button" value="Urgência" id="btUrgencia" />
        <input type="button" value="Atender" id="btAtender" />
      </p>
    </form>
    <h3>Em Atendimento:</h3>
    <span className="fonte-azul"></span>
  </Fragment>
<pre></pre>
```

```

);
}

export default App;

```

Ajuste o autoFocus (F deve ficar maiúsculo), o fechamento das tags input e substitua class por className. Após os ajustes, você verá o campo de formulário com os botões exibidos na caixa de renderização, como na Figura 13.2.

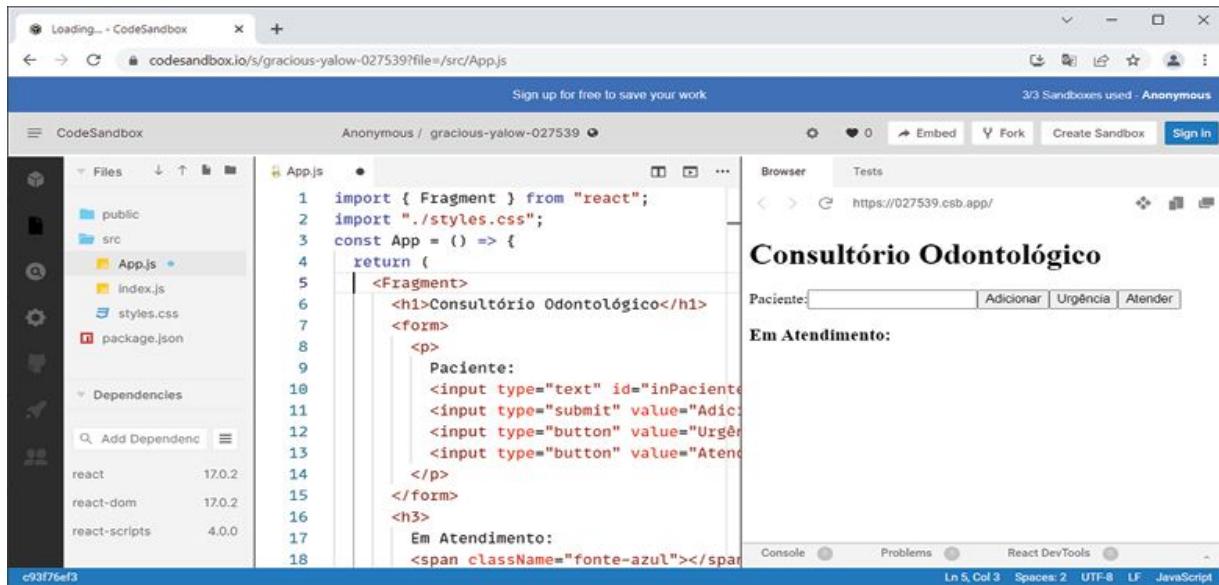


Figura 13.2 – Consultório Odontológico com o React.

Antes de prosseguir, edite o arquivo styles.css e insira os estilos utilizados neste exemplo:

```

.fonte-azul { color: blue; }
pre { font-size: 1.3em; }

```

13.1 Variáveis de Estado: useState()

A partir da versão 16.8 do React, foram adicionados os Hooks, que são funções especiais que nos permitem utilizar recursos do React com uma sintaxe mais simples. O primeiro Hook que iremos utilizar é o useState(). Ele serve para declarar uma variável que irá manter o seu conteúdo. Quando esse conteúdo for alterado em qualquer função do componente, a página é atualizada – ou, melhor, apenas o elemento que contém referência a essa variável. Três são as variáveis de estado que vamos utilizar em nosso

programa, acrescente-as antes do return (...).

```
const [pacientes, setPacientes] = useState([]);  
const [atendido, setAtendido] = useState("");  
const [nome, setNome] = useState("");
```

Altere a linha inicial, acrescentando o import do useState:

```
import { Fragment, useState } from "react";
```

É fundamental entender as partes que compõem a sintaxe da declaração das variáveis de estado: pacientes, atendido e nome são os nomes das variáveis; setPacientes, setAtendido e setNome são os métodos a serem utilizados para alterar o conteúdo dessas variáveis; [], "", "" são os conteúdos iniciais das variáveis.

Para que as variáveis sejam renderizadas, elas precisam ser inseridas no JSX do componente entre {}. Começamos inserindo pacientes e atendido nos seguintes locais do código:

```
<span className="fonte-azul">{atendido}</span>  
<pre>  
  {pacientes.map(paciente => paciente + "\n")}  
</pre>
```

Agora, experimente acrescentar “manualmente” alguns pacientes na lista:

```
const [pacientes, setPacientes] = useState(["Ana", "Carlos", "João"]);
```

Você verá que esses pacientes já são exibidos na página. Muito legal!

Vamos agora ver como controlar os campos de formulário. No React, usa-se a expressão “two-way binding”, uma espécie de vinculação bidirecional, e ela pode ser exemplificada ao trabalhar com campos de formulário: se alterarmos a variável, o campo de formulário é alterado; se alterarmos o campo de formulário, a variável irá conter o valor atualizado. Observe a sintaxe que pode ser utilizada no campo de formulário que recebe o nome do paciente.

```
<input type="text" value={nome} onChange={(e) => setNome(e.target.value)}  
      required autoFocus />
```

Assim, quando houver uma mudança no campo de formulário, setNome() faz com que a variável nome tenha o novo conteúdo. E esse novo conteúdo é, então, exibido pelo value.

13.2 Organização em componentes

Algo que ainda não destacamos é que, no React, tanto o código JavaScript quanto a declaração dos eventos que são programados ficam em um mesmo arquivo. Nos exemplos dos capítulos iniciais do livro, com JavaScript “puro”, organizamos o código do HTML, CSS e JS em arquivos distintos. No React, a ideia é montar a aplicação a partir de componentes funcionais. Então, cada componente é responsável por renderizar uma parte da página e também gerenciar a interação do usuário – que pode ocorrer a partir dos elementos desse componente. Assim, cada arquivo contém um componente – com o HTML (ou melhor, JSX) e JS. A estilização do componente fica em outro arquivo, geralmente com o mesmo nome do arquivo .js, mas com a extensão .css.

Neste primeiro exemplo, teremos apenas um componente responsável por exibir o formulário, os botões, a lista dos pacientes e o paciente em atendimento. No exemplo que irá consumir o Web Service do Capítulo 12, vamos explorar melhor esse recurso.

Então, para acionar um evento devemos acrescentar o onSubmit, onClick, onChange, onBlur, ... junto com o elemento HTML e a programação a ser executada pode ser acrescentada no mesmo arquivo. Vamos inserir cada chamada de evento e arrow function a seguir. Começamos pelo onSubmit. Acrescente a chamada ao método novoPaciente ao lado da tag form.

```
<form onSubmit={novoPaciente}>
```

E, após a declaração das variáveis de estado (e antes do return), insira a arrow function novoPaciente.

```
const novoPaciente = e => {
  e.preventDefault();
  setPacientes([...pacientes, nome]);
  setNome("");
}
```

Observe que não é necessário criar referência ao local da página onde a lista deve ser exibida, nem fazer uma atribuição com o innerText. Ao modificar o conteúdo da variável de estado, a partir do setPacientes e setNome, a lista é atualizada e o campo do formulário, limpo. É importante ressaltar que os

métodos utilizados para fazer a alteração são aqueles declarados lá no useState. E que todas as técnicas de programação estudadas seguem as mesmas – pois um framework não é uma nova linguagem e, sim, um conjunto de ferramentas que visam auxiliar no desenvolvimento de aplicações em uma linguagem, que no caso do React é o JavaScript.

Os métodos novoUrgente e atenderPaciente devem estar associados aos botões **Urgência** e **Atender**, no evento onClick.

```
<input type="button" value="Urgência" onClick={novoUrgente} />
<input type="button" value="Atender" onClick={atenderPaciente} />
```

A programação do novoUrgente adiciona o paciente no início da fila. Já o atenderPaciente verifica a existência de pacientes no vetor, altera o conteúdo da variável atendido – a partir de setAtendido(), e faz uma nova atribuição ao vetor pacientes, removendo o elemento da posição 0 (o slice(1) retorna a lista dos pacientes da posição 1 até o final).

```
const novoUrgente = () => {
  setPacientes([nome, ...pacientes]);
  setNome("");
}

const atenderPaciente = () => {
  if (!pacientes.length) {
    alert("Não há pacientes na fila de espera...")
    return
  }
  setAtendido(pacientes[0])
  setPacientes(pacientes.slice(1))
}
```

Pronto! Identificando o local onde as variáveis de estado devem ser renderizadas e alterando-as a partir do método declarado no useState() a “mágica” acontece. A Figura 13.3 exibe o programa em funcionamento.

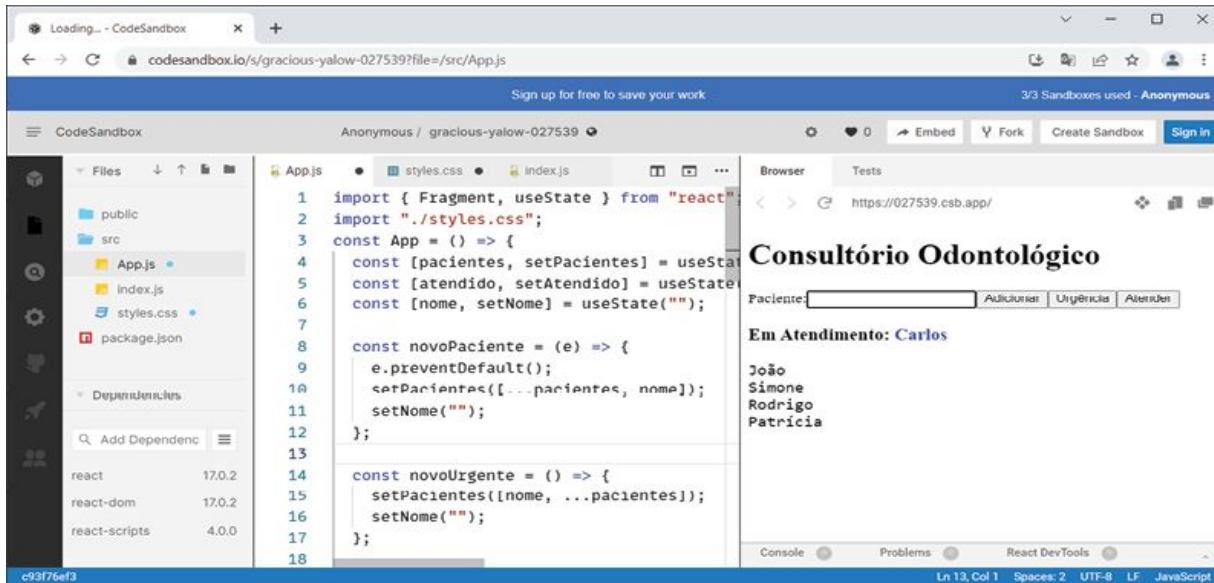


Figura 13.3 – Consultório Odontológico com o React.

13.3 Criação de um novo projeto React

Vamos agora iniciar o projeto que irá consumir o Web Service desenvolvido no Capítulo 12. Na pasta `livrojs`, execute no prompt o comando que cria um novo projeto React:

```
C:\livrojs>npx create-react-app cap13
```

Na primeira execução desse comando, você irá receber a seguinte mensagem:

Need to install the following packages:

create-react-app

Ok to proceed? (y)

Confirme a instalação e aguarde a conclusão do processo (que pode demorar alguns minutos). Após, entre nesta pasta (subdiretório) e abra o Visual Studio Code. Você verá que as pastas `public`, `src` e `node_modules` foram criadas pelo comando anterior.

```
C:\livrojs>cd cap13
```

```
C:\livrojs\cap13>code .
```

Abra o arquivo `index.html` da pasta `public`. Ajuste o arquivo, conforme o conteúdo a seguir:

```
<!DOCTYPE html>
```

```
<html lang="pt-br">
<head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/livrojs.png" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css">
  <title>Controle Pessoal de Livros</title>
</head>
<body>
  <noscript>Você precisa habilitar o JavaScript para executar este App</noscript>
  <div id="root"></div>
</body>
</html>
```

Copie o arquivo `livrojs.png` para a pasta `public` se desejar. Na pasta `src`, podemos manter apenas os arquivos `index.js` e `App.js` (como no projeto do CodeSandbox). Os demais arquivos podem ser removidos. Abra o `index.js` e mantenha apenas as linhas descritas a seguir:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
```

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Edite na sequência o conteúdo do `App.js`, para exibir uma mensagem e, em seguida, vamos rodar o projeto.

```
const App = () => {
  return <h1>Controle Pessoal de Livros</h1>;
};
export default App;
```

Não se esqueça de salvar os arquivos modificados. Para rodar ou dar o “start” no projeto, retorne ao prompt e execute o comando a seguir:

```
C:\livrojs\cap13>npm start
```

Após alguns instantes, você receberá uma mensagem indicando que o processo foi concluído e a aplicação está disponível para ser executada no endereço (padrão) `http://localhost:3000`. No geral, o seu navegador default é

aberto com a aplicação rodando. A Figura 13.4 exibe o resultado do que foi feito até o momento.

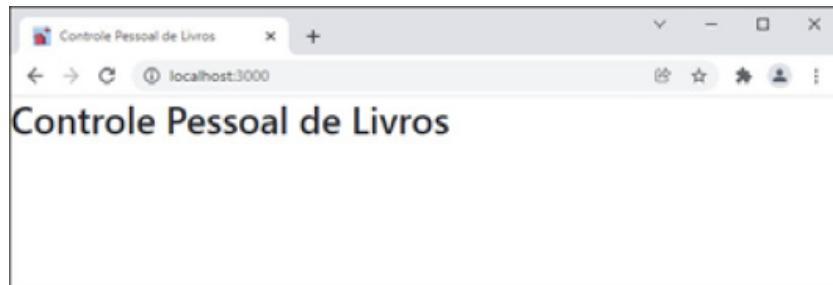


Figura 13.4 – Projeto em execução, com a mensagem retornada por App.js.

Vamos avançar um pouco. Crie a pasta components, dentro de src. E, dentro de components, crie os arquivos MenuSuperior.js e InclusaoLivros.js. Em MenuSuperior.js insira o seguinte código:

```
const MenuSuperior = () => {
  return (
    <nav className="navbar navbar-expand-sm bg-primary navbar-dark sticky-top">
      <div className="container">
        <a className="navbar-brand" href="#">Controle Pessoal de Livros</a>
        <ul className="navbar-nav">
          <li className="nav-item">
            <a className="nav-link" href="#">Inclusão</a>
          </li>
          <li className="nav-item">
            <a className="nav-link" href="#">Manutenção</a>
          </li>
          <li className="nav-item">
            <a className="nav-link" href="#">Resumo</a>
          </li>
        </ul>
      </div>
    </nav>
  );
};

export default MenuSuperior;
```

Provavelmente, o seu editor de código exibirá uma advertência sobre o uso da tag ``. Vamos ajustar isso na Seção 13.6. Observe que substituímos o atributo `class` por `className`, uma vez que `class` é uma palavra reservada do JavaScript. Os estilos são do Bootstrap 5 para a criação de

menus de navegação. Edite agora o arquivo InclusaoLivros.js para que contenha o código descrito a seguir:

```
const InclusaoLivros = () => {
  return (
    <div className="container">
      <h4 className="fst-italic mt-3">Inclusão </h4>
      <form>
        <div className="form-group">
          <label htmlFor="titulo">Título:</label>
          <input type="text" className="form-control" id="titulo" required
            autoFocus />
        </div>
        <div className="form-group mt-2">
          <label htmlFor="autor">Autor:</label>
          <input type="text" className="form-control" id="autor" required />
        </div>
        <div className="form-group mt-2">
          <label htmlFor="foto">URL da Foto:</label>
          <input type="url" className="form-control" id="foto" required />
        </div>
        <div className="row mt-2">
          <div className="col-sm-4">
            <div className="form-group">
              <label htmlFor="ano">Ano de Publicação:</label>
              <input type="number" className="form-control" id="ano" required />
            </div>
          </div>
          <div className="col-sm-8">
            <div className="form-group">
              <label htmlFor="preco">Preço R$:</label>
              <input type="number" className="form-control" id="preco"
                step="0.01" required />
            </div>
          </div>
        </div>
        <input type="submit" className="btn btn-primary mt-3" value="Enviar" />
        <input type="reset" className="btn btn-danger mt-3 ms-3" value="Limpar" />
      </form>
      <div className="alert"></div>
    </div>
  );
};
```

```
export default InclusaoLivros;
```

Muito bem, criamos os dois primeiros componentes de nosso projeto. Vamos, então, fazer com que esses componentes sejam renderizados a partir do App.js. Edite esse arquivo, para que contenha o seguinte:

```
import MenuSuperior from "./components/MenuSuperior";
import InclusaoLivros from "./components/InclusaoLivros";

const App = () => {
  return (
    <>
      <MenuSuperior />
      <InclusaoLivros />
    </>
  );
};

export default App;
```

Após salvar os arquivos, volte ao navegador e o resultado deve ser o que é exibido pela Figura 13.5. Que show! Muito bom esse React... Agora, os dois componentes são exibidos para compor a página web.

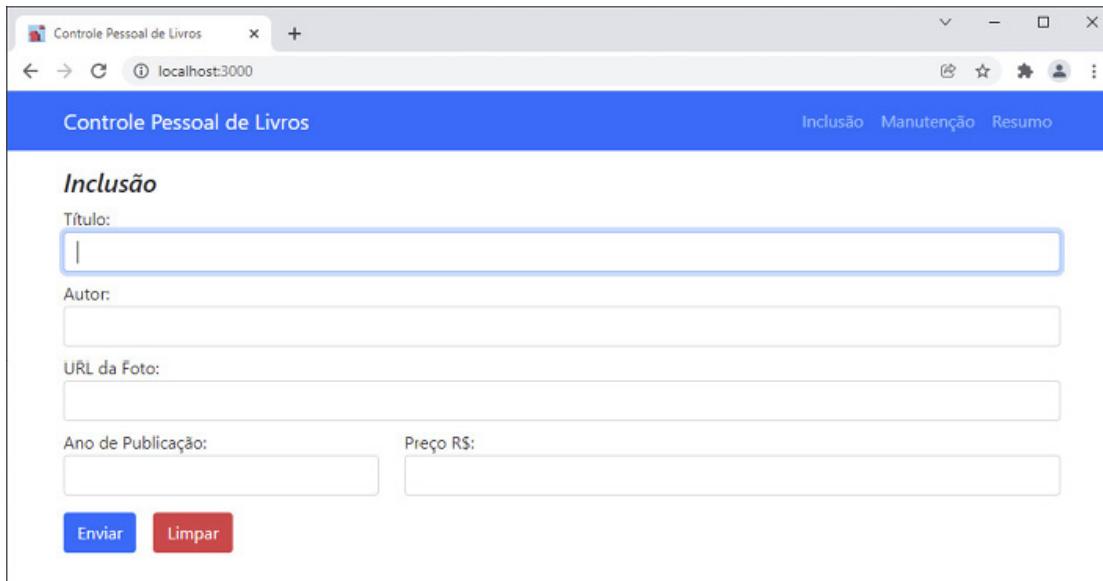


Figura 13.5 – Os componentes MenuSuperior e InclusaoLivros renderizados na página.

13.4 Simplificando o gerenciamento do form

com React Hook Form

No exemplo inicial deste capítulo, vimos que é possível controlar os campos de formulário com variáveis de estado. Esse processo pode ser simplificado com o uso do pacote React Hook Form (<https://react-hook-form.com/>). Inicialmente, volte ao prompt e adicione o pacote ao projeto.

```
C:\livrojs\cap13>npm i react-hook-form
```

Além de facilitar o gerenciamento de formulários, o React Hook Form também permite definir regras de validação no preenchimento dos campos. Esse recurso fica como sugestão para você explorar no futuro. Vamos focar na obtenção do conteúdo dos campos e envio dos dados. Faça os seguintes ajustes no arquivo InclusaoLivros.js:

```
import { useForm } from "react-hook-form";

const InclusaoLivros = () => {
  // register serve para definir os nomes dos campos do form (e validações)
  // handleSubmit, para indicar o método a ser acionado no evento onSubmit do form
  const { register, handleSubmit } = useForm();

  // método chamado ao enviar o form (onSubmit)
  const salvar = (campos) => {
    // JSON.stringify() converte um objeto JavaScript para uma String JSON
    alert(JSON.stringify(campos));
  }

  ...
}
```

Após, no return() da arrow function acrescente na tag form:

```
<form onSubmit={handleSubmit(salvar)}>
```

Dessa forma, indicamos que a função salvar será chamada quando o usuário enviar o formulário. O próximo ajuste é acrescentar o register para cada campo do formulário:

```
<input type="text" className="form-control" id="titulo" required
       autoFocus {...register("titulo")}>
```

E nos demais:

```
{...register("autor")}, {...register("foto")}, {...register("ano")}, {...register("preco")}
```

Para o correto funcionamento da nossa aplicação é fundamental que os

campos de formulário tenham o mesmo nome dos campos da tabela do banco de dados. Isso evita que tenhamos de ajustar os dados no momento da chamada ao Web Service.

Salve o arquivo e faça um teste preenchendo os campos de formulário. Ao clicar no botão **Enviar** você irá receber um `alert()` com os dados de um objeto JavaScript (convertidos para o formato String JSON), exatamente como esperado pelo Web Service – conforme ilustra a Figura 13.6. Vamos então realizar as etapas para enviar esses dados para o back-end.

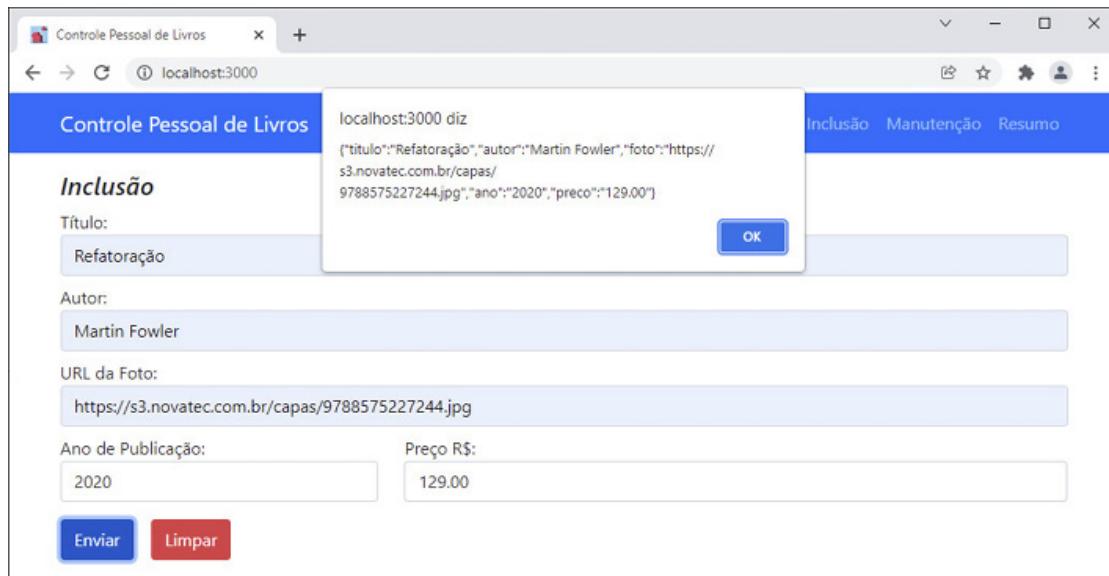


Figura 13.6 – Campos de formulário são recuperados com os métodos do React Hook Form.

13.5 Axios para comunicar com o Web Service

O Axios é um dos principais pacotes do JavaScript para enviar e receber informações entre o cliente e o servidor web. Para utilizá-lo precisamos acrescentar o package ao nosso projeto. Retorne ao prompt e execute o comando:

```
C:\livrojs\cap13>npm i axios
```

O Axios utiliza o conceito das Promises do JavaScript – aquela ideia destacada na Seção 12.9, na qual uma ação no código pode demandar um

tempo maior para a sua execução. Então, faremos uso de funções com `async` e `await`. Como iremos informar a URL do Web Service em diversos locais do código, uma boa prática é criar um arquivo com o endereço dessa URL. Assim, caso ela modifique – quando publicarmos nosso projeto, por exemplo, basta fazer a troca nesse arquivo. Crie, então, o arquivo `config_axios.js`, na raiz do projeto, com as linhas descritas a seguir:

```
import axios from 'axios';
// cria uma instance Axios com a URL base do Web Service a ser acessado pelo app
export const inAxios = axios.create({ baseURL: 'http://localhost:3001/'});
```

Nesse exemplo, não utilizamos o `export default` apenas para demonstrar a diferença no `import` que deve ser utilizado no arquivo que for utilizar a instância do Axios. Observe a sintaxe:

```
import { inAxios } from "../config_axios";
```

Ou seja, quando não houver um `export default`, é necessário indicar entre `{ }` qual ou quais variáveis ou funções queremos importar. A seguir, o código do `InclusaoLivros.js` com os ajustes finais de programação:

```
import { useState } from "react";
import { useForm } from "react-hook-form";
import { inAxios } from "../config_axios";

const InclusaoLivros = () => {
    // register serve para definir os nomes dos campos do form (e validações)
    // handleSubmit, para indicar o método a ser acionado no evento onSubmit do form
    // reset, para atribuir um valor para os campos registrados (para limpar o form)
    const { register, handleSubmit, reset } = useForm();

    const [aviso, setAviso] = useState("");

    // método chamado ao enviar o form (onSubmit)
    const salvar = async (campos) => {
        try {
            const response = await inAxios.post("livros", campos);
            setAviso(`Ok! Livro cadastrado com código ${response.data.id}`);
        } catch (error) {
            setAviso(`Erro... Livro não cadastrado: ${error}`);
        }
    }

    // setTimeout: executa o comando após o tempo indicado (em milissegundos)
```

```

setTimeout(() => {
  setAviso("");
}, 5000);

// limpa os campos de formulário para uma nova inclusão
reset({titulo: "", autor: "", foto: "", ano: "", preco: ""});
};

return (...)

}

```

Importante: para testar a inclusão de dados, você precisa deixar rodando (node app) o back-end desenvolvido no Capítulo 12.

Vamos aos comentários sobre os acréscimos nesse programa. O primeiro deles foi a inclusão do reset no import do useForm(). Esse método serve para atribuir um valor para as variáveis registradas nos campos de formulário. Útil para limpar os campos após a inclusão de dados.

Na sequência do código, declaramos a variável de estado aviso. Essa variável inicia com uma string vazia “” e é modificada de acordo com o sucesso ou não da inclusão de dados. Após, observe o uso do método setTimeout() que aguarda um tempo (em milésimos de segundo) para, então, executar um ou mais comandos. Então, após 5 segundos, a variável aviso volta a ter uma string vazia. Ajuste o código JSX do return para o correto funcionamento da exibição dessa mensagem. A linha:

```
<div className="alert"></div>
```

Deve ser substituída por:

```
<div className={aviso.startsWith("Ok!") ? "alert alert-success" :
aviso.startsWith("Erro") ? "alert alert-danger" : ""}> {aviso}
</div>
```

Entendeu a sintaxe utilizada nessa tag? Ela diz o seguinte: se o conteúdo da variável aviso iniciar com “Ok!” aplique o estilo “alert alert-success”, senão, e se aviso iniciar com “Erro”, aplique o estilo “alert alert-danger”. Caso contrário, deixe o estilo vazio. {aviso} indica o local onde a mensagem será exibida.

Desse modo, essa div não é inicialmente exibida (pois não há estilo nem conteúdo para ela). Quando a arrow function salvar for executada, ocorre a mudança no conteúdo de aviso e a div passa a ser exibida. Na sequência, após

5 segundos, o conteúdo de aviso volta a ser vazio e a mensagem desaparece. Muito legal!

O método `startsWith` é semelhante ao `includes`, porém `startsWith` retorna verdadeiro se a string iniciar pelos caracteres indicados, enquanto `includes` verifica a existência da substring em qualquer local da string.

Analizando ainda as mudanças no código do programa, temos dentro da função `salvar` o envio dos dados para o Web Service a partir da instância do Axios. Observe a sintaxe desse comando, dentro do `try`:

```
const response = await inAxios.post("livros", campos);
```

Ele indica que os dados serão passados pelo método POST, que o complemento da URL (definida no `config_axios.js`) é `livros` e que o objeto JSON a ser transmitido está em `campos`. A resposta da chamada a esse método é atribuída ao `response`, que contém em `.data` os dados retornados pelo Web Service. A Figura 13.7 exemplifica o cadastro de um livro com o aviso em exibição.

The screenshot shows a web application titled "Controle Pessoal de Livros" with a blue header bar containing the title and navigation links for "Inclusão", "Manutenção", and "Resumo". The main content area is titled "Inclusão". It contains several input fields: "Título" with the value "React Aprenda Praticando", "Autor" with the value "Maurício Samy Silva", "URL da Foto" with the value "https://s3.novatec.com.br/capas/9786586057393.jpg", "Ano de Publicação" with the value "2021", and "Preço R\$" with the value "69.00". At the bottom left are two buttons: a blue "Enviar" button and a red "Limpar" button. A green success message at the bottom right states "Ok! Livro cadastrado com código 8".

Figura 13.7 – Os dados do livro são enviados ao Web Service que realiza a inclusão.

13.6 Criação de Rotas com o React Router

Na Figura 13.6 é possível observar que a nossa aplicação de front-end possui um menu superior e um formulário. O menu é renderizado a partir do componente `MenuSuperior` e o formulário, a partir do componente `InclusaoLivros`. Queremos agora manter o menu superior e substituir a parte que exibe o formulário por um componente que exiba a listagem de livros ou o resumo do cadastro. Para fazer isso, vamos utilizar o React Router, que está na versão 6.

Como de costume, precisamos iniciar pelo acréscimo desse pacote ao projeto. Retorne ao prompt e execute o seguinte comando:

```
C:\livrojs\cap13>npm i react-router-dom@6
```

Vamos seguir os passos indicados na documentação do pacote (<https://reactrouter.com/>). O primeiro ajuste deve ser feito no arquivo `index.js`. Acrescente as linhas em negrito:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from "react-router-dom";
import App from './App';

ReactDOM.render(
  <BrowserRouter>
  <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

Para testar, precisamos criar os dois novos componentes. Podemos acrescentar um conteúdo inicial básico. Comece pelo arquivo `ManutencaoLivros.js` (dentro da pasta `components`):

```
const ManutencaoLivros = () => {
  return <h2>Manutenção</h2>;
};
export default ManutencaoLivros;
```

E, em seguida, o arquivo `ResumoLivros.js` (também dentro de `components`):

```
const ResumoLivros = () => {
  return <h2>Resumo</h2>;
};
```

```
};

export default ResumoLivros;
```

Após, abra o arquivo App.js, e adicione as linhas indicadas em negrito:

```
import { Routes, Route } from "react-router-dom";
import MenuSuperior from "./components/MenuSuperior";
import InclusaoLivros from "./components/InclusaoLivros";
import ManutencaoLivros from "./components/ManutencaoLivros";
import ResumoLivros from "./components/ResumoLivros";

const App = () => {
  return (
    <>
      <MenuSuperior />
      <Routes>
        <Route path="/" element={<InclusaoLivros />} />
        <Route path="manut" element={<ManutencaoLivros />} />
        <Route path="resumo" element={<ResumoLivros />} />
      </Routes>
    </>
  );
};

export default App;
```

Observe que o componente que deve ser mantido em todas as páginas (MenuSuperior) fica acima da tag Routes. E que os demais componentes possuem uma Route path (caminho de rota), ou seja, sua exibição fica vinculada a uma rota. Quando essa rota for acionada, apenas esse componente é renderizado – sendo que o componente inicial é indicado pela rota “/”. Muito bacana!

O último passo é ajustar a tag que cria o link para essas rotas. Elas estão no componente MenuSuperior. A tag `` deve ser substituída pela tag `<Link to=...>`, conforme exemplificado a seguir:

```
import { Link } from "react-router-dom";

const MenuSuperior = () => {
  return (
    <nav className="navbar navbar-expand-sm bg-primary navbar-dark sticky-top">
      <div className="container">
        <Link to="/" className="navbar-brand">Controle Pessoal de Livros</Link>
        <ul className="navbar-nav">
```

```

<li className="nav-item">
  <Link to="/" className="nav-link">Inclusão</Link>
</li>
<li className="nav-item">
  <Link to="/lista" className="nav-link">Manutenção</Link>
</li>
<li className="nav-item">
  <Link to="/controle" className="nav-link">Controle</Link>
</li>
</ul>
</div>
</nav>
);
};

export default MenuSuperior;

```

Certifique-se de que todos os arquivos estão salvos e retorne ao navegador. Observe que os links já estão funcionais, com a vantagem da melhora de performance – pois apenas o que muda é atualizado no browser. A Figura 13.8 exibe o menu superior e um dos componentes com o conteúdo inicial exibido.

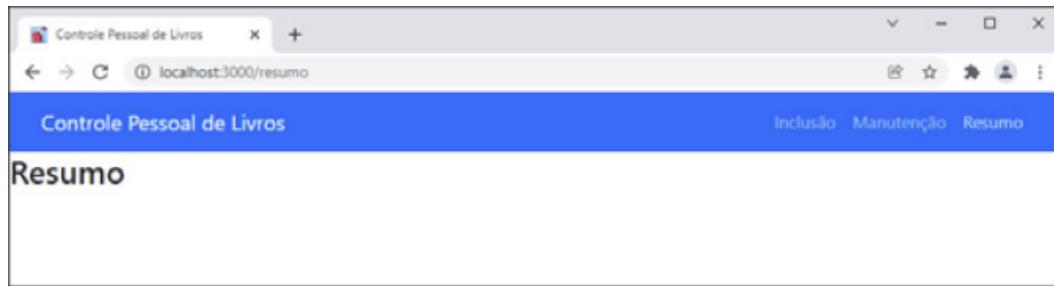


Figura 13.8 – React Router em uso para navegação entre os componentes.

13.7 useEffect()

Lembram do `window.addEventListener("load")` que utilizamos em diversos exemplos no livro? O objetivo desse ouvinte de evento é indicar uma programação a ser executada assim que a página concluir a carga de seus elementos. De forma semelhante, com o Hook `useEffect` do React, conseguimos executar algum trecho de código assim que ocorrer a primeira renderização do componente. Também é possível vincular essa execução à

alteração de uma variável de estado.

Vamos destacar a seguir o código inicial do componente `ManutencaoLivros` e, na sequência, novas funcionalidades serão adicionadas a ele. Esse será um componente bastante versátil, responsável pela execução de diversas tarefas, como filtro, alteração e exclusão de livros.

```
import { useState, useEffect } from "react";
import { inAxios } from "../config_axios";
import ItemLista from "./ItemLista";

const ManutencaoLivros = () => {
  const [livros, setLivros] = useState([]);

  const obterLista = async () => {
    try {
      const lista = await inAxios.get("livros");
      setLivros(lista.data);
    } catch (error) {
      alert(`Erro... Não foi possível obter os dados: ${error}`);
    }
  };

  // define o método que será executado assim que o componente for renderizado
  useEffect(() => {
    obterLista();
  }, []);

  return (
    <div className="container">
      <div className="row">
        <div className="col-sm-7">
          <h4 className="fst-italic mt-3">Manutenção</h4>
        </div>
        <div className="col-sm-5">
          <form>
            <div className="input-group mt-3">
              <input type="text" className="form-control"
                placeholder="Título ou autor" required />
              <input type="submit" className="btn btn-primary" value="Pesquisar" />
              <input type="button" className="btn btn-danger" value="Todos" />
            </div>
          </form>
        </div>
      </div>
    </div>
  );
}
```

```

        </div>
    </div>

<table className="table table-striped mt-3">
    <thead>
        <tr>
            <th>Cód.</th>
            <th>Título</th>
            <th>Autor</th>
            <th>Ano</th>
            <th>Preço R$</th>
            <th>Foto</th>
            <th>Ações</th>
        </tr>
    </thead>
    <tbody>
        {livros.map((livro) => (
            <ItemLista key={livro.id} id={livro.id} titulo={livro.titulo}
                autor={livro.autor} ano={livro.ano} preco={livro.preco}
                foto={livro.foto} />
        ))}
    </tbody>
</table>
</div>
);

};

export default ManutencaoLivros;

```

Esse componente renderiza um campo de formulário, a ser utilizado para a pesquisa de dados e o cabeçalho de uma tabela. A exibição de cada item na tabela fica a cargo de um novo componente: ItemLista. Vamos criar este arquivo ItemLista.js, também na pasta components, e depois retornamos com as explicações sobre todo o código.

```

import "./ItemLista.css";

const ItemLista = (props) => {
    return (
        <tr>
            <td>{props.id}</td>
            <td>{props.titulo}</td>
            <td>{props.autor}</td>
            <td>{props.ano}</td>

```

```

<td class="text-end">
  {Number(props.preco).toLocaleString("pt-br", {minimumFractionDigits: 2})}
</td>
<td class="text-center">
  <img src={props.foto} alt="Capa do Livro" width="75" />
</td>
<td class="text-center">
  <i className="exclui text-danger fw-bold" title="Excluir">#10008;</i>
  <i className="altera text-success fw-bold ms-2" title="Alterar">#36;</i>
</td>
</tr>
);
};

export default ItemLista;

```

O arquivo `ItemLista.css`, que é importado no início do código, deve conter a linha a seguir:

```
.altera, .exclui { cursor: pointer; }
```

Salve todos os arquivos. A página da manutenção de livros deve estar semelhante à imagem da Figura 13.9.

Ainda falta acrescentar as funcionalidades para a realização dos filtros, da alteração e exclusão de registros. Mas vamos começar analisando o código do componente `ManutencaoLivros`. Observe que é declarada uma variável de estado que irá conter a relação dos livros retornados pelo Web Service. E que fizemos uso do `useEffect`, ou seja, após o componente ser renderizado, irá ocorrer uma chamada ao método `obterLista`. Nele, há uma chamada, via GET, ao endereço `http://localhost:3001/livros`.

A lista de dados que é retornada pelo Web Service é, então, utilizada para modificar o conteúdo do array `livros`, a partir do `setLivros`. Isso faz com que a tabela seja atualizada, pois toda alteração em uma variável de estado gera uma nova renderização da página (apenas do elemento JSX que contém referência à variável). Essa é a magia do React!

Cód.	Título	Autor	Ano	Preço R\$	Foto	Ações
8	React Aprenda Praticando	Maurício Samy Silva	2021	69,00		X \$
7	Refatoração	Martin Fowler	2020	129,00		X \$
6	Estratégia de UX	Jaime Levy	2021	95,00		X \$
5	Python para análise de dados	Wes McKinney	2018	132,00		X \$

Figura 13.9 – Lista dos livros cadastrados e opções de manutenção da tabela.

E para apresentar cada livro na tabela, seguimos a recomendação da equipe de desenvolvimento do React, que sugere que os elementos que compõem uma UI (user interface) sejam divididos em partes independentes e reutilizáveis. Os dados de cada livro são passados como props para o componente filho, que retorna um objeto React. Observe que há o atributo especial `key` nos parâmetros passados. Ele ajuda o React a identificar modificações na lista e deve conter valores únicos (não duplicados) para dar uma identidade estável aos elementos.

O componente `ItemLista` recebe as propriedades de dados de cada elemento a ser renderizado em `props`. Poderíamos utilizar a atribuição via desestruturação no lugar de `props` da seguinte forma:

```
const ItemLista = ({id, titulo, autor, ano, preco, foto}) => {...}
```

Desse modo, não é necessário preceder cada nome de variável por `props` no código.

13.8 Filtrando os registros da listagem

Vamos acrescentar agora a programação necessária para filtrar os dados da lista de livros. Como se trata de gerenciar um formulário, comece pelo acréscimo do import do React Hook Form.

```
import { useForm } from "react-hook-form";
```

O próximo passo é indicar os métodos do React Hook Form a serem utilizados. Dentro da arrow function ManutencaoLivros, insira a linha:

```
const { register, handleSubmit, reset } = useForm();
```

Em seguida, edite a linha do form:

```
<form onSubmit={handleSubmit(filtrarLista)}>
```

E acrescente o register no campo de formulário

```
<input type="text" className="form-control"  
placeholder="Título ou autor" required {...register("palavra")}>
```

Por fim, vamos à programação do método filtrarLista, descrito a seguir:

```
const filtrarLista = async (campos) => {  
  try {  
    const lista = await inAxios.get(`livros/filtro/${campos.palavra}`);  
    lista.data.length  
      ? setLivros(lista.data)  
      : alert("Não há livros com a palavra-chave pesquisada...");  
  } catch (error) {  
    alert(`Erro... Não foi possível obter os dados: ${error}`);  
  }  
};
```

Ou seja, para realizar um filtro na listagem, faremos uma chamada ao Web Service, na rota que criamos lá na Seção 12.12. Se houver elementos no array (lista.data.length), uma nova atribuição a livros é feita (o que irá forçar uma “re-renderização” da tabela). Caso contrário, uma mensagem de alerta é exibida.

Também poderíamos fazer esse filtro de um segundo modo: utilizando o método filter do JavaScript. Contudo um cuidado para esse modo seria necessário. Se um usuário filtrar pela palavra “CSS” e, em seguida, pela palavra “Python”, por exemplo, haveria um problema, pois a lista sobre a qual o último filtro seria realizado não iria conter todos os livros.

Falta-nos ainda fazer a programação associada ao clique no botão **Todos**.

Ela deve limpar o conteúdo do campo de formulário e obter novamente a lista de livros. Vamos implementar essas ações diretamente no evento onClick do botão:

```
<input type="button" className="btn btn-danger" value="Todos"  
onClick={() => { reset({palavra: ""}); obterLista(); }} />
```

A Figura 13.10 ilustra o funcionamento do filtro na tela de manutenção de livros.

Cód.	Título	Autor	Ano	Preço R\$	Foto	Ações
3	SQL em 10 Minutos por Dia	Ben Forta	2021	79,00		X \$

Figura 13.10 – Filtro retorna apenas os livros que contenham a palavra-chave informada.

13.9 Exclusão, alteração e passagem de funções como props

Conforme ilustra a Figura 13.8, cada registro exibido na tabela contém dois botões, que permitem excluir ou alterar o livro. Começamos demonstrando o processo necessário para realizar a exclusão de um registro. Edite o arquivo ManutencaoLivros.js e insira o seguinte código:

```
const excluir = async (id, titulo) => {  
  if (!window.confirm(`Confirma a exclusão do livro "${titulo}"?`)) {  
    return;  
  }  
  try {  
    await inAxios.delete(`livros/${id}`);  
    setLivros(livros.filter((livro) => livro.id !== id));  
  } catch (err) {  
    console.error(err);  
  }  
};
```

```
    } catch (error) {
      alert(`Erro... Não foi possível excluir este livro: ${error}`);
    }
};
```

Esta arrow function solicita que o usuário confirme a exclusão do livro. Lembre-se de que o sinal de exclamação (!) antes do window.confirm() significa negação. Logo, o return ocorre quando ele não confirmar. Para excluir o registro, basta acionar a rota de exclusão usando o verbo delete, invocado a partir do Axios. Se a exclusão for bem-sucedida, setLivros() faz uma atualização no array aplicando um filtro – que obtém todos os registros cujo id seja diferente daquele passado como parâmetro.

Mas essa função excluir está no ManutencaoLivros.js e o botão para acionar a exclusão está lá no componente ItemLista. Como isso vai funcionar?

A resposta é: passando a função excluir como uma propriedade para o componente ItemLista. Edite a linha que faz a chamada ao ItemLista. Ela deve ficar assim:

```
<ItemLista key={livro.id} id={livro.id} titulo={livro.titulo}
  autor={livro.autor} ano={livro.ano} preco={livro.preco}
  foto={livro.foto} excluirClick={() => excluir(livro.id, livro.titulo)} />
```

Esta props deve então ser recebida em ItemLista. Se você fez a desestruturação de props, acrescente esta variável:

```
const ItemLista = ({id, titulo, autor, ano, preco, foto, excluirClick}) => {
```

E, por fim, no ícone de excluir em ItemLista deve ser acrescentado o evento onClick:

```
<i className="exclui text-danger fw-bold" title="Excluir"
  onClick={excluirClick}>#10008;</i>
```

Muito bacana! Então, a partir de ManutencaoLivros nós passamos como propriedade de ItemLista a função que será acionada quando o usuário clicar no ícone de excluir.

Entendida a exclusão? Tente você fazer agora a alteração! Lembre-se de que optamos por alterar apenas o preço do livro. Então você pode recorrer ao método prompt() para solicitar o novo preço e usar a mesma ideia implementada na exclusão. Pausa na leitura... para você tentar!

Ok, vamos pensar juntos algumas questões relacionadas com a alteração. Receber o id e o título do livro a ser alterado e realizar a chamada ao Web Service, acredito que seja tranquilo de implementar. A questão mais complicada é alterar a lista de modo que o novo preço seja atualizado na tabela. Claro que poderíamos chamar novamente o `obterLivros()`, mas não vamos recorrer a ele.

Uma forma de tornar eficiente a alteração é acrescentar como parâmetro o índice de cada elemento do array, na chamada ao método alterar. Vamos acrescentar isso em `ManutencaoLivros.js`:

```
{livros.map((livro, index) => (
  <ItemLista key={livro.id} id={livro.id} titulo={livro.titulo}
    autor={livro.autor} ano={livro.ano} preco={livro.preco} foto={livro.foto}
    excluirClick={() => excluir(livro.id, livro.titulo)}
    alterarClick={() => alterar(livro.id, livro.titulo, index)} />
))}
```

Assim, sabendo o índice do elemento a ser alterado podemos modificar o atributo preço em um novo array e chamar o `setLivros` (que, por sua vez, fará com que a tabela seja atualizada). Vamos ao código!

```
const alterar = async (id, titulo, index) => {
  const novoPreco = Number(prompt(`Informe o novo preço do livro "${titulo}"`));
  if (isNaN(novoPreco) || novoPreco === 0) {
    return;
  }
  try {
    await inAxios.put(`livros/${id}`, { preco: novoPreco });
    const livrosAlteracao = [...livros];
    livrosAlteracao[index].preco = novoPreco;
    setLivros(livrosAlteracao);
  } catch (error) {
    alert(`Erro... Não foi possível alterar o preço: ${error}`);
  }
};
```

Não se esqueça de realizar as alterações em `ItemLista`: acrescentar o `alterarClick` nos parâmetros e o `onClick` no ícone alterar. A Figura 13.11 exibe o processo de alteração em andamento.

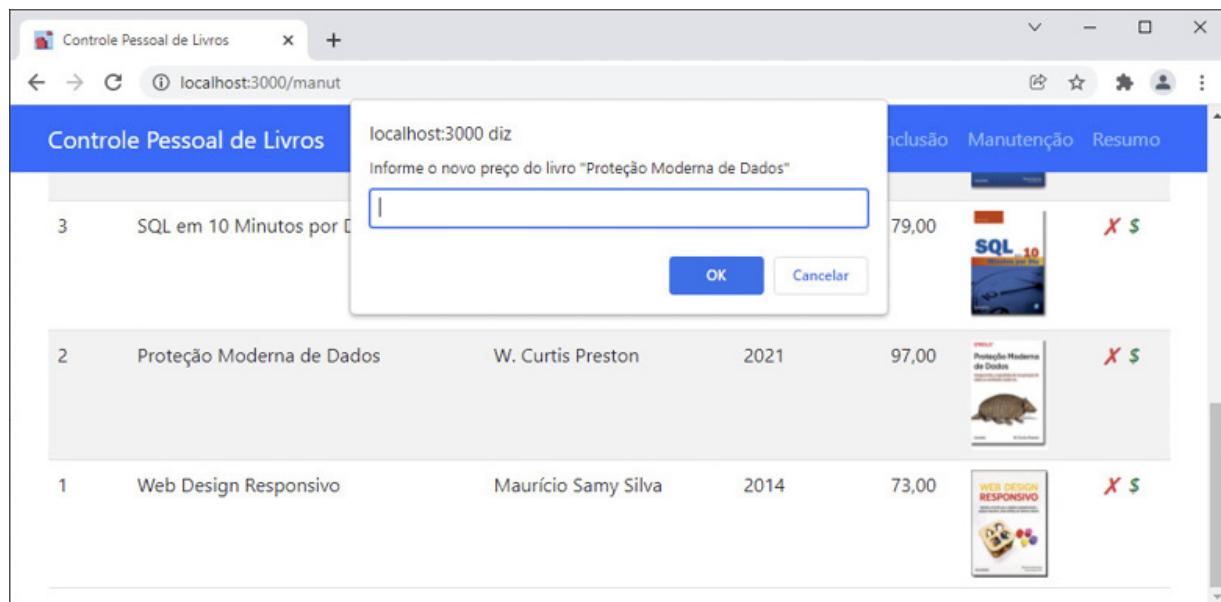


Figura 13.11 – Ao clicar sobre o ícone “\$” o programa solicita o novo preço do livro.

13.10 Resumo com gráfico ou dashboard do sistema

O terceiro componente a ser exibido após o menu de navegação de nosso projeto tem por objetivo exibir dados estatísticos do cadastro de livros. Ele bem que poderia ser a página inicial de nossa aplicação, como uma Dashboard – termo utilizado para designar um painel que fornece uma visão geral do sistema. Caso queira fazer isso, ajuste as rotas no arquivo App.js. Vou apresentar inicialmente a visualização desta página, na Figura 13.12, e na sequência destacar os recursos de programação necessários para implementá-la.



Figura 13.12 – Dados estatísticos e gráfico do cadastro de livros.

Para a construção do gráfico, vamos utilizar o pacote React Google Charts. Existem outros pacotes do React que permitem criar gráficos. Simplicidade, variedade de tipos de gráfico e boa documentação são os destaques do React Google Charts. A documentação está disponível no endereço <https://react-google-charts.com/>. Começamos, então, adicionando este pacote ao projeto:

```
C:\livrojs\cap13>npm i react-google-charts
```

As rotas do back-end com as funções a serem acionadas para a exibição de dados estatísticos e do gráfico foram apresentadas na Seção 12.12. O código apresentado a seguir contém chamadas às duas rotas que retornam dados estatísticos do cadastro de livros. Para preencher os valores exibidos nos quadros destacados na Figura 13.10, é necessário criar uma variável de estado – um objeto com os atributos fornecidos pelo Web Service, e posicionar corretamente cada atributo desse objeto no local do código JSX onde ele deve ser renderizado.

Já para a criação do gráfico é preciso ter o cuidado de ajustar o formato dos dados de acordo com o esperado pelo React Google Charts. Conforme pode

ser visto na documentação do pacote, os dados precisam estar no formato de vetores, sendo que o primeiro elemento do vetor deve conter o nome das colunas utilizadas na construção do gráfico. Por isso, recorremos ao método `.map()` – que percorre os elementos do objeto JSON retornado e adiciona cada elemento ao vetor.

Abra o arquivo `ResumoLivros.js` – que foi anteriormente criado com um único retorno de teste, e insira o seguinte código:

```
import { useState, useEffect } from "react";
import { Chart } from "react-google-charts";
import { inAxios } from "./config_axios";

const ResumoLivros = () => {
  // return <h2>Resumo de Livros</h2>;
  const [resumo, setResumo] = useState([]);
  const [grafico, setGrafico] = useState([]);

  const obterDados = async () => {
    try {
      const dadosResumo = await inAxios.get("livros/dados/resumo");
      setResumo(dadosResumo.data);

      const dadosGrafico = await inAxios.get("livros/dados/grafico");
      // cria um array e adiciona a primeira linha
      const arrayGrafico = [["Ano", "R$ Total"]];
      // percorre cada linha do JSON e adiciona ao array
      dadosGrafico.data.map(( dado ) =>
        arrayGrafico.push([dado.ano.toString(), dado.total])
      );
      setGrafico(arrayGrafico);
    } catch (error) {
      alert(`Erro... Não foi possível obter os dados: ${error}`);
    }
  };

  // define o método que será executado assim que o componente for renderizado
  useEffect(() => {
    obterDados();
  }, []);

  return (
    <div>
      <h2>Resumo de Livros</h2>
      <p>Este é o resumo dos livros disponíveis:</p>
      <table border="1">
        <thead>
          <tr>
            <th>Ano</th>
            <th>Preço Total (R$)</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>2010</td>
            <td>100000</td>
          </tr>
          <tr>
            <td>2011</td>
            <td>120000</td>
          </tr>
          <tr>
            <td>2012</td>
            <td>140000</td>
          </tr>
          <tr>
            <td>2013</td>
            <td>160000</td>
          </tr>
          <tr>
            <td>2014</td>
            <td>180000</td>
          </tr>
          <tr>
            <td>2015</td>
            <td>200000</td>
          </tr>
          <tr>
            <td>2016</td>
            <td>220000</td>
          </tr>
          <tr>
            <td>2017</td>
            <td>240000</td>
          </tr>
          <tr>
            <td>2018</td>
            <td>260000</td>
          </tr>
          <tr>
            <td>2019</td>
            <td>280000</td>
          </tr>
          <tr>
            <td>2020</td>
            <td>300000</td>
          </tr>
        </tbody>
      </table>
      <div>
        <h3>Gráfico de Vendas por Ano</h3>
        <Chart
          chartType="Line"
          data={[
            { Ano: 2010, Total: 100000 },
            { Ano: 2011, Total: 120000 },
            { Ano: 2012, Total: 140000 },
            { Ano: 2013, Total: 160000 },
            {Ano: 2014, Total: 180000},
            {Ano: 2015, Total: 200000},
            {Ano: 2016, Total: 220000},
            {Ano: 2017, Total: 240000},
            {Ano: 2018, Total: 260000},
            {Ano: 2019, Total: 280000},
            {Ano: 2020, Total: 300000}
          ]}
        />
      </div>
    </div>
  );
}
```

```

<div className="container">
  <h4 className="mt-3">Resumo</h4>
  <span className="btn btn-outline-primary btn-lg">
    <p className="badge bg-danger">{resumo.num}</p>
    <p>Nº de Livros Cadastrados</p>
  </span>
  <span className="btn btn-outline-primary btn-lg mx-2">
    <p className="badge bg-danger">
      {Number(resumo.soma).toLocaleString("pt-br", {minimumFractionDigits: 2})}
    </p>
    <p>Total Investido em Livros</p>
  </span>
  <span className="btn btn-outline-primary btn-lg me-2">
    <p className="badge bg-danger">
      {Number(resumo.maior).toLocaleString("pt-br", {minimumFractionDigits: 2})}
    </p>
    <p>Maior Preço Cadastrado</p>
  </span>
  <span className="btn btn-outline-primary btn-lg">
    <p className="badge bg-danger">
      {Number(resumo.media).toLocaleString("pt-br", {minimumFractionDigits: 2})}
    </p>
    <p>Preço Médio dos Livros</p>
  </span>

  <div className="d-flex justify-content-center">
    <Chart
      width={1000}
      height={420}
      chartType="ColumnChart"
      loader={<div>Carregando Gráfico...</div>}
      data={grafico}
      options={{{
        title: "Total de Investimentos em Livros - por Ano de Publicação",
        chartArea: { width: "80%" },
        hAxis: { title: "Ano de Publicação" },
        vAxis: { title: "Preço Acumulado R$" },
        legend: { position: "none" },
      }}}
    />
  </div>
</div>
);

```

```
};  
export default ResumoLivros;
```

O componente Chart é o responsável por renderizar o gráfico. Os parâmetros fornecidos seguem a documentação do pacote e são bem intuitivos. O cuidado maior é justamente com o formato dos dados passados em data. Após a conversão realizada a partir do método `.map()`, os dados ficam no formato:

```
[['Ano', 'R$ Total'], ['2014', 73], ['2017', 45], ...]
```

Faça o cadastro de novos livros, teste os filtros, alterações, exclusões, resumo e o gráfico. Ficou bonito o nosso sistema! Claro que ele sempre pode ser aprimorado. Melhorar a interface. Trocar as mensagens de alerta por janelas modais... Talvez o próximo passo seja adicionar novas tabelas e entender como funciona o processo de relacionamento entre duas ou mais tabelas. Poderíamos, por exemplo, adicionar dados referentes a comentários sobre os livro. Esses dados deveriam, portanto, ser armazenados na tabela comentários, contendo em sua estrutura chaves estrangeiras para as tabelas livros e usuários. E, assim, segue a brincadeira...

13.11 Considerações finais

Concluída a leitura deste livro, acredito que você deu um grande passo para ingressar na área de TI. O que fazer agora? Bom, vai depender dos seus objetivos. Quer avançar nos estudos em programação web ou desenvolver sistemas para aplicativos móveis? Busque novos livros sobre os temas. Contudo, se pretende ser um profissional da área de Computação, além de estudar a partir de livros, fazer um curso superior é muito importante. Na faculdade, você vai adquirir conhecimentos fundamentais em diversas áreas, como banco de dados, análise de sistemas, gerência de projetos, segurança, entre outras, além de programação. Vai passar pelas experiências de apresentar os seus projetos, defender o TCC (Trabalho de Conclusão de Curso), ter a oportunidade de participar de feiras, maratonas de programação, projetos de pesquisa... Além disso, vai conviver com vários colegas que já estão trabalhando e professores, que frequentemente são solicitados a indicar alunos para vagas de emprego - muitas delas para

empresas gerenciadas por seus ex-alunos.

E, para finalizar, desejo o melhor da vida para cada um de vocês. Sejam, acima de tudo, boas pessoas, pois boas pessoas, com caráter, humildade e determinação, cedo ou tarde alcançam os seus objetivos. Que Deus acompanhe a todos. Um grande abraço!

Referências

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi. *Fundamentos da Programação de Computadores: Algoritmos, Pascal, C/C++ e Java*. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

DUCKETT, Jon. *JavaScript & jQuery: desenvolvimento de interfaces web interativas*. Rio de Janeiro: Alta Books, 2015.

Express: *Getting Started*. Disponível em: <<https://expressjs.com/pt-br/starter/hello-world.html>>.

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. *Lógica de Programação – A construção de Algoritmos e Estruturas de Dados*. São Paulo: Makron Books, 1993.

Knex.js SQL Query Builder. Disponível em: <<https://knexjs.org/>>.

LECHETA, Ricardo R. *Node Essencial*. São Paulo: Novatec, 2018.

Mozilla Developer Network. Disponível em: <<https://developer.mozilla.org/pt-BR>>.

Node.js Docs. Disponível em: <<https://nodejs.org/en/docs/>>.

POWERS, Shelley. *Aprendendo JavaScript*. São Paulo: Novatec; Califórnia, EUA: O'Reilly, 2010.

React Docs. Disponível em: <<https://reactjs.org/docs/getting-started.html>>.

SILVA, Maurício Samy. *Bootstrap 3.3.5*. São Paulo: Novatec, 2015.

SILVA, Maurício Samy. *Fundamentos de HTML5 e CSS3*. São Paulo: Novatec, 2015.

W3Schools.com. Disponível em: <<https://www.w3schools.com>>.

Marcos Abe

MANUAL DE
**ANÁLISE
TÉCNICA**

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por

meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)

O'REILLY®



Microsserviços prontos para a produção



CONSTRUINDO SISTEMAS PADRONIZADOS EM UMA
ORGANIZAÇÃO DE ENGENHARIA DE SOFTWARE



novatec

Susan J. Fowler

Microsserviços prontos para a produção

Fowler, Susan J.

9788575227473

224 páginas

[Compre agora e leia](#)

Um dos maiores desafios para as empresas que adotaram a arquitetura de microsserviços é a falta de padronização de arquitetura – operacional e organizacional. Depois de dividir uma aplicação monolítica ou construir um ecossistema de microsserviços a partir do zero, muitos engenheiros se perguntam o que vem a seguir. Neste livro prático, a autora Susan Fowler apresenta com profundidade um conjunto de padrões de microsserviço, aproveitando sua experiência de padronização de mais de mil microsserviços do Uber. Você aprenderá a projetar microsserviços que são estáveis, confiáveis, escaláveis, tolerantes a falhas, de alto desempenho, monitorados, documentados e preparados para qualquer catástrofe. Explore os padrões de disponibilidade de produção, incluindo: Estabilidade e confiabilidade – desenvolva, implante, introduza e descontinue microsserviços; proteja-se contra falhas de dependência. Escalabilidade e desempenho – conheça os

componentes essenciais para alcançar mais eficiência do microsserviço. Tolerância a falhas e prontidão para catástrofes – garanta a disponibilidade forçandoativamente os microsserviços a falhar em tempo real. Monitoramento – aprenda como monitorar, gravar logs e exibir as principais métricas; estabeleça procedimentos de alerta e de prontidão. Documentação e compreensão – atenua os efeitos negativos das contrapartidas que acompanham a adoção dos microsserviços, incluindo a dispersão organizacional e a defasagem técnica.

[Compre agora e leia](#)

Entendendo algoritmos

Um guia *ilustrado* para programadores
e outros curiosos

Aditya Y. Bhargava



novatec

MANNING

Entendendo Algoritmos

Bhargava, Aditya Y.

9788575226629

264 páginas

[Compre agora e leia](#)

Um guia ilustrado para programadores e outros curiosos. Um algoritmo nada mais é do que um procedimento passo a passo para a resolução de um problema. Os algoritmos que você mais utilizará como um programador já foram descobertos, testados e provados. Se você quer entendê-los, mas se recusa a estudar páginas e mais páginas de provas, este é o livro certo. Este guia cativante e completamente ilustrado torna simples aprender como utilizar os principais algoritmos nos seus programas. O livro Entendendo Algoritmos apresenta uma abordagem agradável para esse tópico essencial da ciência da computação. Nele, você aprenderá como aplicar algoritmos comuns nos problemas de programação enfrentados diariamente. Você começará com tarefas básicas como a ordenação e a pesquisa. Com a prática, você enfrentará problemas mais complexos, como a compressão de dados e a inteligência artificial. Cada exemplo é apresentado em

detalhes e inclui diagramas e códigos completos em Python. Ao final deste livro, você terá dominado algoritmos amplamente aplicáveis e saberá quando e onde utilizá-los. O que este livro inclui A abordagem de algoritmos de pesquisa, ordenação e algoritmos gráficos Mais de 400 imagens com descrições detalhadas Comparações de desempenho entre algoritmos Exemplos de código em Python Este livro de fácil leitura e repleto de imagens é destinado a programadores autodidatas, engenheiros ou pessoas que gostariam de recordar o assunto.

[Compre agora e leia](#)

Loiane Groner

Estruturas de dados e algoritmos com JavaScript

2^a Edição

Escreva um código JavaScript complexo e eficaz usando
a mais recente ECMAScript

novatec

Packt

Estruturas de dados e algoritmos com JavaScript

Groner, Loiane

9788575227282

408 páginas

[Compre agora e leia](#)

Uma estrutura de dados é uma maneira particular de organizar dados em um computador com o intuito de usar os recursos de modo eficaz. As estruturas de dados e os algoritmos são a base de todas as soluções para qualquer problema de programação. Com este livro, você aprenderá a escrever códigos complexos e eficazes usando os recursos mais recentes da ES 2017. O livro Estruturas de dados e algoritmos com JavaScript começa abordando o básico sobre JavaScript e apresenta a ECMAScript 2017, antes de passar gradualmente para as estruturas de dados mais importantes, como arrays, filas, pilhas e listas ligadas. Você adquirirá um conhecimento profundo sobre como as tabelas hash e as estruturas de dados para conjuntos funcionam, assim como de que modo as árvores e os mapas hash podem ser usados para buscar arquivos em um disco rígido

ou para representar um banco de dados. Este livro serve como um caminho para você mergulhar mais fundo no JavaScript. Você também terá uma melhor compreensão de como e por que os grafos – uma das estruturas de dados mais complexas que há – são amplamente usados em sistemas de navegação por GPS e em redes sociais.

Próximo ao final do livro, você descobrirá como todas as teorias apresentadas podem ser aplicadas para solucionar problemas do mundo real, trabalhando com as próprias redes de computador e com pesquisas no Facebook. Você aprenderá a:

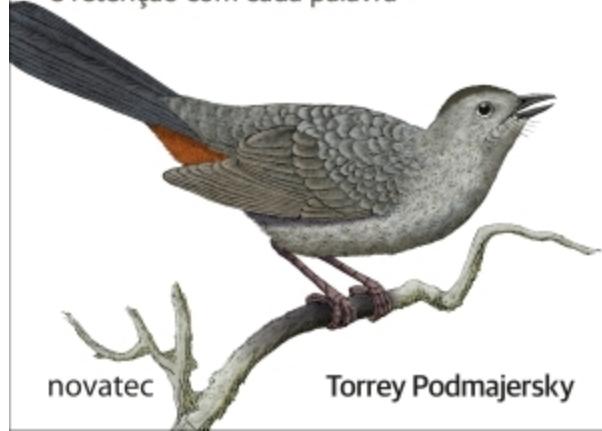
- declarar, inicializar, adicionar e remover itens de arrays, pilhas e filas;
- criar e usar listas ligadas, duplamente ligadas e ligadas circulares;
- armazenar elementos únicos em tabelas hash, dicionários e conjuntos;
- explorar o uso de árvores binárias e árvores binárias de busca;
- ordenar estruturas de dados usando algoritmos como bubble sort, selection sort, insertion sort, merge sort e quick sort;
- pesquisar elementos em estruturas de dados usando ordenação sequencial e busca binária

[Compre agora e leia](#)

O'REILLY®

Redação Estratégica para UX

Aumente engajamento, conversão
e retenção com cada palavra



Torrey Podmajersky

Redação Estratégica para UX

Podmajersky, Torrey

9788575228135

184 páginas

[Compre agora e leia](#)

Quando uma organização depende de seres humanos comportando-se de maneira específica – comprando ingressos, jogando ou usando o transporte público –, as palavras bem colocadas são mais eficientes. Como escolher as melhores palavras? E como saber se vão funcionar? Com este livro prático, você vai aprender como redigir de forma estratégica para UX usando ferramentas para construir os alicerces do texto das interfaces com os usuários e a estratégia de voz da marca para UX. A estrategista de conteúdo UX Torrey Podmajersky discute estratégias para conversão, engajamento e suporte aos usuários, além de maneiras de fazê-los retornar à experiência. Você vai usar frameworks e padrões para conteúdo, métodos para avaliar sua eficiência e processos para criar a colaboração necessária ao sucesso. Também vai estruturar toda a voz da marca de modo que a marca seja facilmente reconhecida por seu público.

•Aprenda como funciona o conteúdo UX ao

longo do ciclo de vida de desenvolvimento de software. •Use um framework para alinhar o conteúdo UX aos princípios do produto. •Explore o design focado no conteúdo para basear o texto UX no diálogo. •Aprenda como os padrões de texto UX funcionam em diferentes vozes. •Escreva um texto significativo, conciso, dialógico e claro.

[Compre agora e leia](#)



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se singlelogin.re go-to-zlibrary.se single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>