

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

Edécio Fernando Iepsen

Novatec

© Novatec Editora Ltda. 2018.

Todos os direitos reservados e protegidos pela Lei 9610 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

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

Ilustrações: Carolina Kuwabata

Capa: Carolina Kuwabata

ISBN: 978-85-7522-657-5

Histórico de edições impressas:

Março/2018 Primeira edição

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: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

*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 e Marina.*

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().

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 Introdução a eventos e funções

2.4 Método getElementById().

2.5 Propriedades textContent, innerHTML e value

2.6 Formas de adicionar JavaScript ao HTML

- [2.6.1 Uso de rotinas de tratamento de eventos HTML](#)
- [2.6.2 Uso de rotinas de tratamento de eventos DOM](#)
- [2.6.3 Uso dos listeners \(ouvintes\) de eventos](#)
- [2.7 Nomenclatura e escopo das variáveis](#)
- [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 ■ Condições

- [3.1 If... else](#)
- [3.2 Operadores relacionais](#)
- [3.3 Operadores lógicos](#)
- [3.4 Switch... Case](#)
- [3.5 Operador ternário](#)
- [3.6 Exemplos](#)
- [3.7 Exercícios](#)
- [3.8 Considerações finais do Capítulo](#)

Capítulo 4 ■ Repetições

- [4.1 Repetição com variável de controle: laços for](#)
- [4.2 Repetição com teste no início: laços while](#)
- [4.3 Repetição com teste no final: laços do.. while](#)
- [4.4 Interrupções nos laços \(break e continue\).](#)
- [4.5 Contadores e acumuladores](#)
- [4.6 Depurar programas \(detectar erros\).](#)
- [4.7 Exercícios](#)
- [4.8 Considerações finais do Capítulo](#)

Capítulo 5 ■ Vetores

- [5.1 Inclusão e exclusão de itens](#)
- [5.2 Tamanho do vetor e exibição dos itens](#)
- [5.3 Localizar conteúdo](#)
- [5.4 Vetores de objetos](#)
- [5.5 Pesquisar e filtrar dados](#)
- [5.6 Classificar os itens do vetor](#)

[5.7 Exercícios](#)

[5.8 Considerações finais do Capítulo](#)

[Capítulo 6 ■ Strings e datas](#)

[6.1 Percorrer os caracteres de uma string](#)

[6.2 Converter para letras maiúsculas ou minúsculas](#)

[6.3 Cópia de caracteres da string](#)

[6.4 Localizar um ou mais caracteres na string](#)

[6.5 Dividir a string em elementos de vetor](#)

[6.6 Validar senhas com o método match\(\)](#)

[6.7 Substituição de caracteres](#)

[6.8 Manipulação de datas](#)

[6.9 Exercícios](#)

[6.10 Considerações finais do Capítulo](#)

[Capítulo 7 ■ Funções e eventos](#)

[7.1 Funções com passagem de parâmetros](#)

[7.2 Funções com retorno de valor](#)

[7.3 Funções anônimas](#)

[7.4 Funções atribuídas para variáveis](#)

[7.5 Eventos JavaScript](#)

[7.6 Exercícios](#)

[7.7 Considerações finais do Capítulo](#)

[Capítulo 8 ■ Persistência de dados com localStorage](#)

[8.1 Salvar e recuperar dados](#)

[8.2 Uma “pitada” de Bootstrap](#)

[8.3 Remover dados do localStorage](#)

[8.4 Uso do getElementByTagName\(\)](#)

[8.5 Manipular listas no localStorage](#)

[8.6 Exercícios](#)

[8.7 Considerações finais do Capítulo](#)

[Capítulo 9 ■ Inserir elementos HTML via JavaScript](#)

[9.1 Inserir e manipular elementos de texto](#)

[9.2 Inserir imagens](#)

[93 Manipular tabelas HTML](#)

[94 Exercícios](#)

[95 Considerações finais do Capítulo](#)

[Capítulo 10 ■ É muito bom programar... Programe!](#)

[10.1 Programa Jockey Club](#)

[10.2 Programa Reserva de Poltronas em Teatro](#)

[10.3 Jogo “Descubra a Palavra”](#)

[10.3.1 Cadastro de palavras](#)

[10.3.2 Listagem de palavras](#)

[10.3.3 Programação do jogo](#)

[10.4 Considerações finais do Capítulo](#)

[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.

Às empresas e às pessoas envolvidas com o Realiza Digital, um belo projeto social que visa dar formação e oportunidades de estágio na área de TI para jovens estudantes de escolas públicas de Pelotas.

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, objetivo maior da busca pela qualificação técnica e didática.

A todos que me apoiaram em diversos momentos de minha vida. Muito obrigado!

Sobre o autor

Edécio Fernando Iepson 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 Matemática Aplicada, Programação para Internet, Programação Android, Programação para Redes e Segurança de Sistemas.

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.

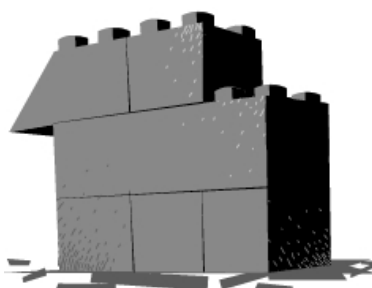
Escrever um livro compartilhando a experiência acumulada no ensino de Algoritmos, com os exemplos em JavaScript – uma linguagem em constante ascensão no mercado, foi para mim uma tarefa muito agradável. Espero que gostem do livro da mesma forma como eu gostei de escrevê-lo. Desejo que ele seja uma importante 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://www.novatec.com.br/livros/logica-programacao-algoritmos-com-javascript/>

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.

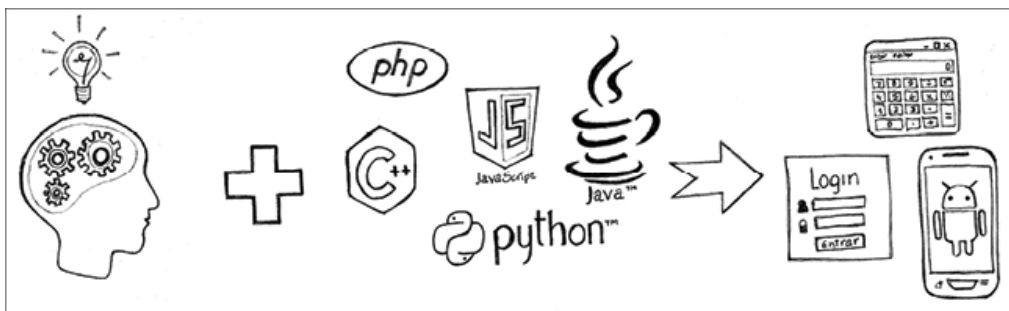


Figura 1.1 – Os conhecimentos de lógica de programação aplicados em uma linguagem permitem criar um aplicativo.

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 precisamos 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 extraídas 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 no item 1.1.6.

1.1.4 Analisar outras possibilidades de solução

Você tem 3 amigos e precisa somar a idade dos 2 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.

Esses 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 no item 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 ocorrer 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. Atualmente, destacam-se as versões ECMAScript 6 e ECMAScript 7.

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 8.1).
- Em um site de uma pizzeria, 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 7.6).
- 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 10.2).
- 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 10.3).

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. As figuras 1.2 e 1.3 destacam duas 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.

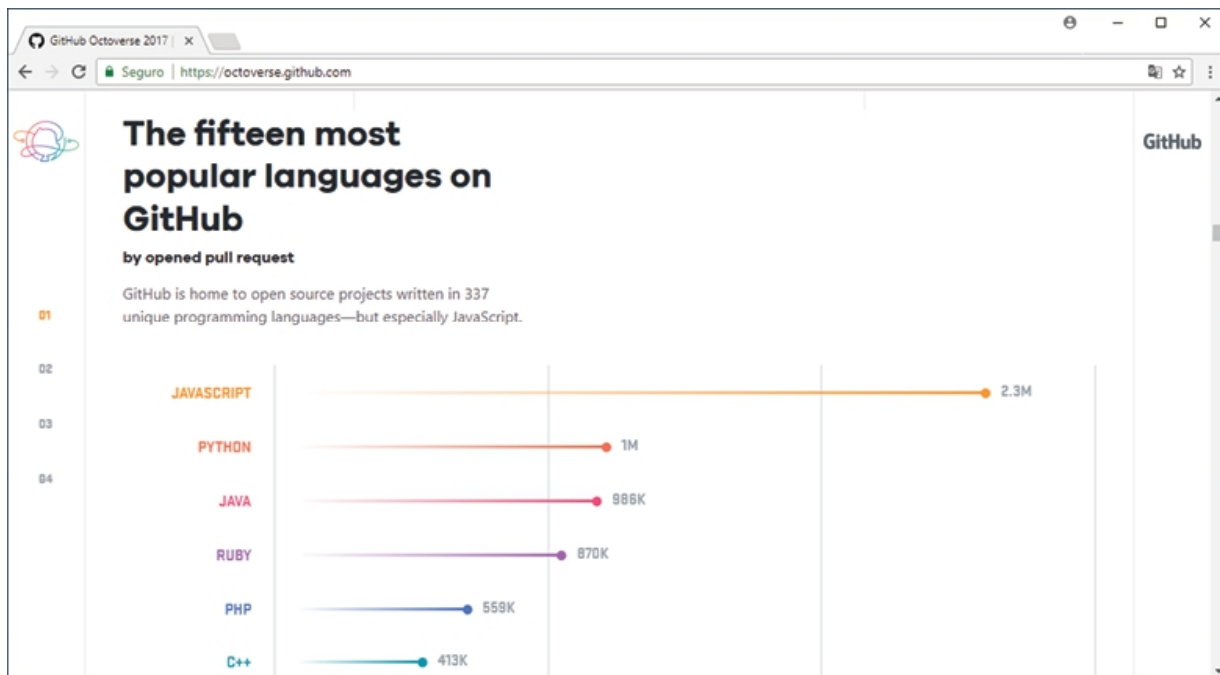


Figura 1.2 – Linguagens de Programação mais populares no GitHub
(<https://octoverse.github.com/>).

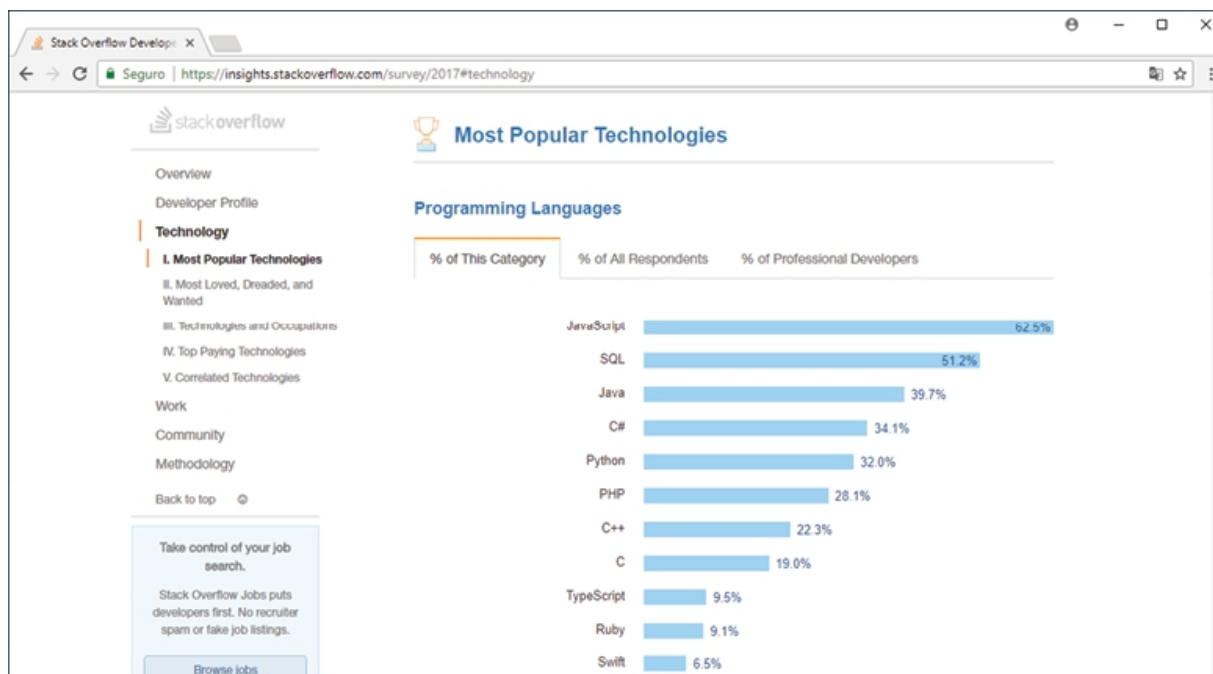


Figura 1.3 – Tecnologias mais populares no Stack Overflow
(<https://insights.stackoverflow.com/survey/2017>).

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, js.do e jsbin.com. 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.

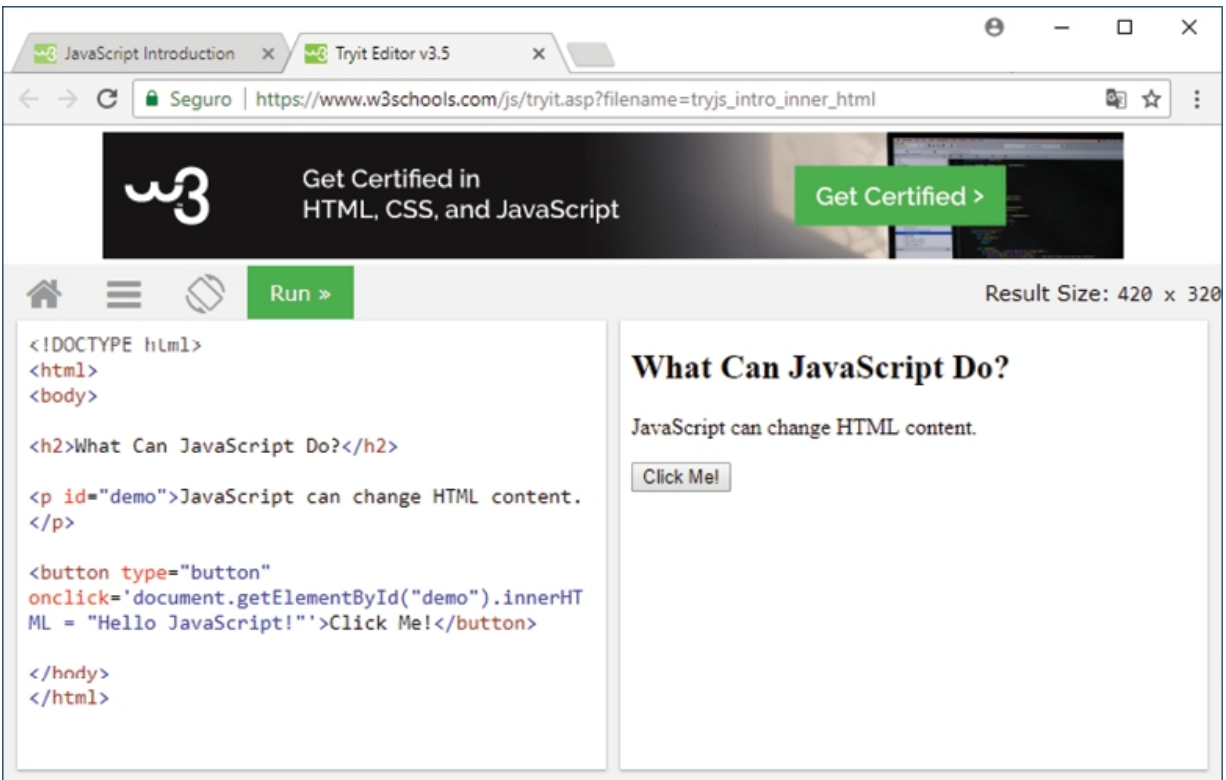


Figura 1.4 – Site da w3schools.com.

Além dos editores online, outro software que você pode utilizar para realizar pequenas experiências de programação JavaScript é o Scratchpad, um editor que acompanha o Mozilla Firefox. Para acessar o Scratchpad, pressione Shift+F4 (no Windows, Mac ou Linux), ou selecione o menu **Desenvolvedor Web** e depois Scratchpad. A tela inicial do programa é exibida na Figura 1.5.

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 1.6. Esse software será utilizado nos exemplos deste livro.

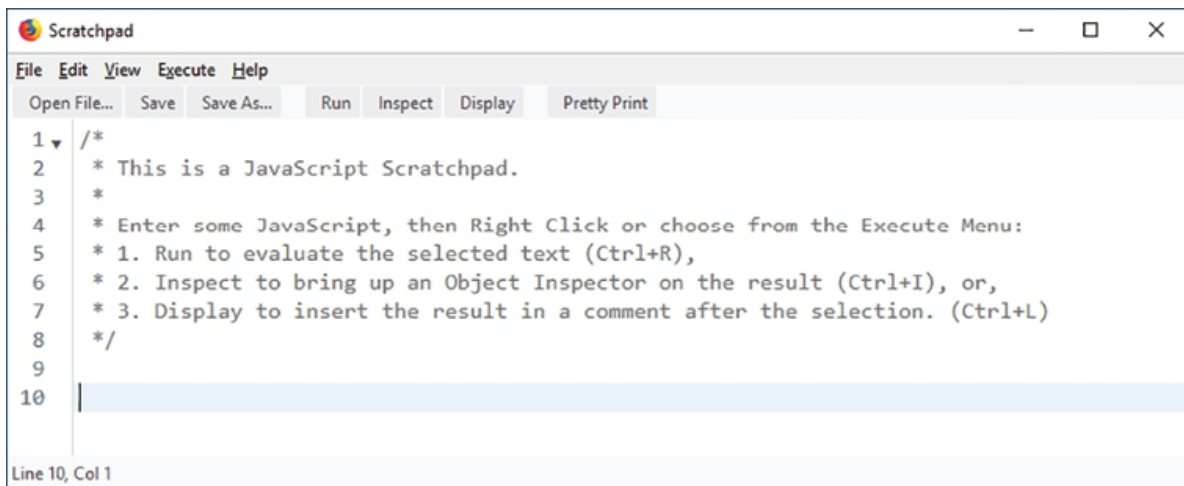


Figura 1.5 – Scratchpad que acompanha o Mozilla Firefox.

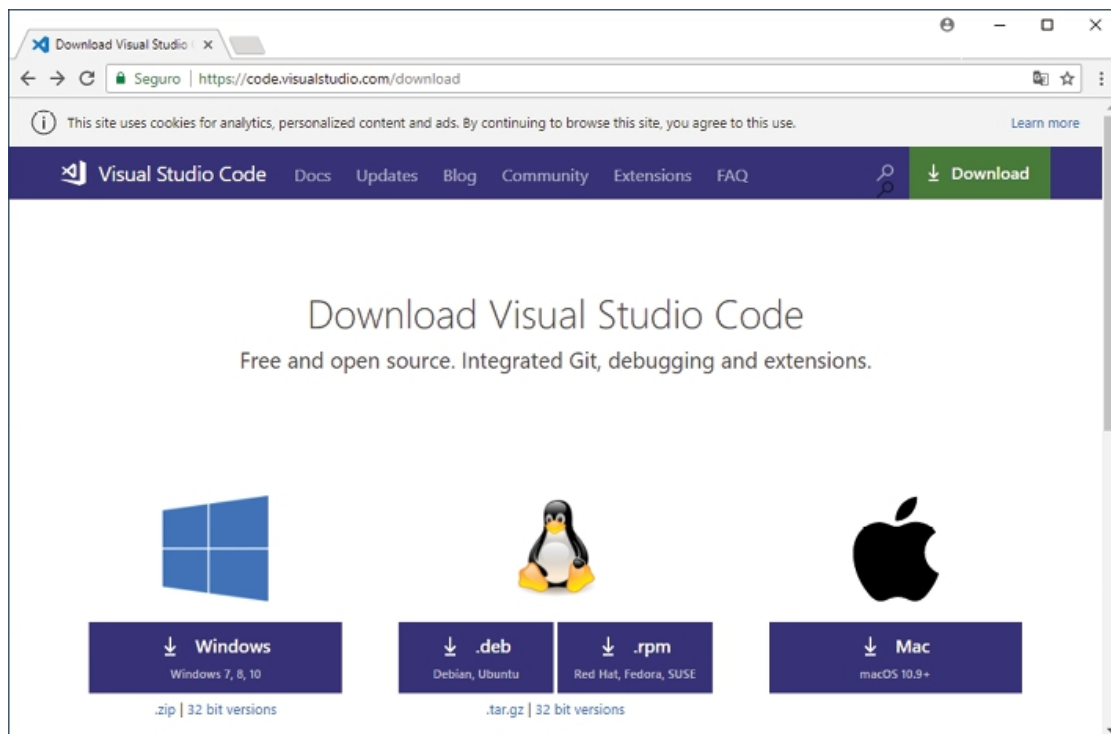


Figura 1.6 – Página de Download do Visual Studio Code.

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. Selecione o idioma, clique **Next** na tela de boas-vindas (Welcome), aceite os termos de licença, informe o local onde o Visual Studio Code será instalado, **Next**, **Next** e **Finish**.

Na primeira execução do Visual Studio Code, é apresentada a tela de boas-vindas do editor (Figura 1.7). Nela, você deve instalar o suporte para

a linguagem JavaScript. Para isso, clique sobre JavaScript em **Install support for ...** no quadro **Tools and languages**.

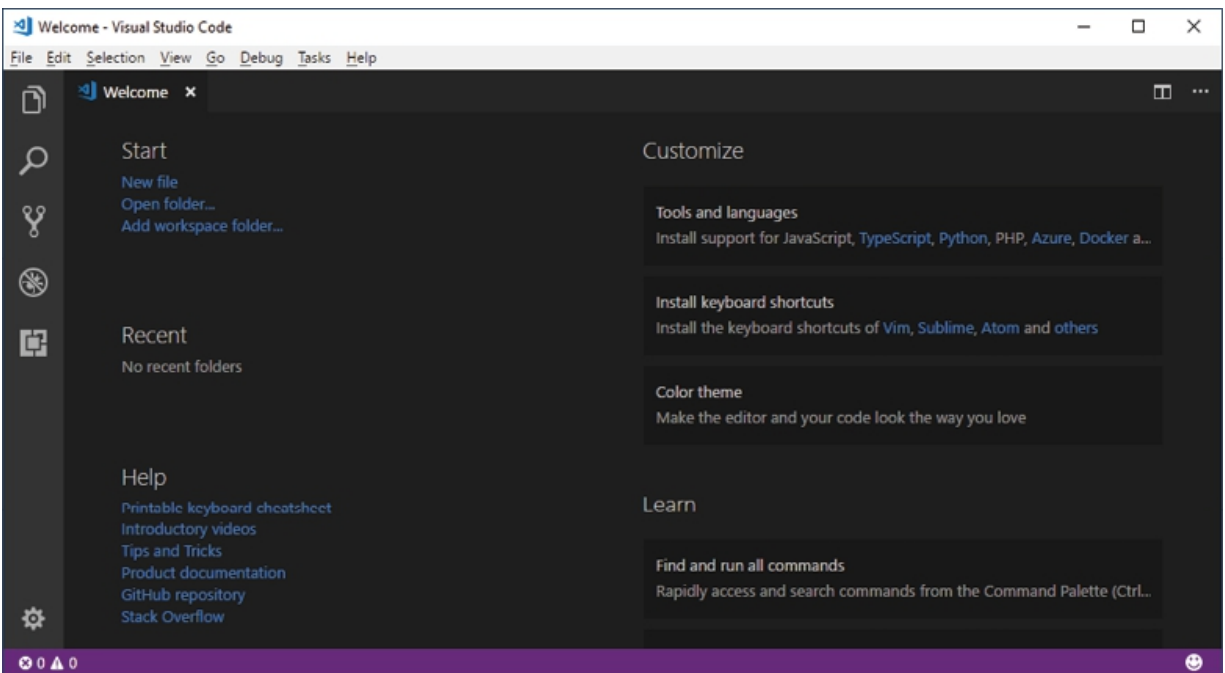


Figura 1.7 – Tela de boas-vindas do Visual Studio Code com link para suporte ao JavaScript.

Caso queira mudar as cores do editor, selecione **File / Preferences / Color Theme** e escolha um novo padrão, como na Figura 1.8. Para começar um novo arquivo, clique no menu **File / New File (Ctrl+N ou Cmd+N)**. Uma janela com um documento em branco é exibida, conforme ilustra a Figura 1.9.

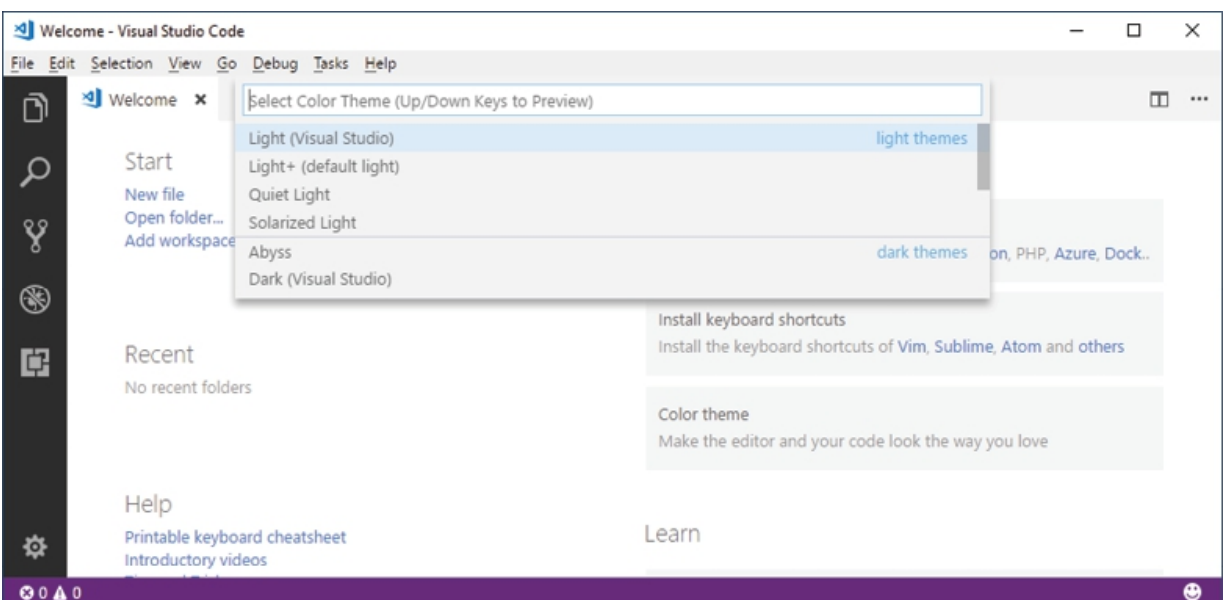


Figura 1.8 – Escolha do tema do layout do editor.

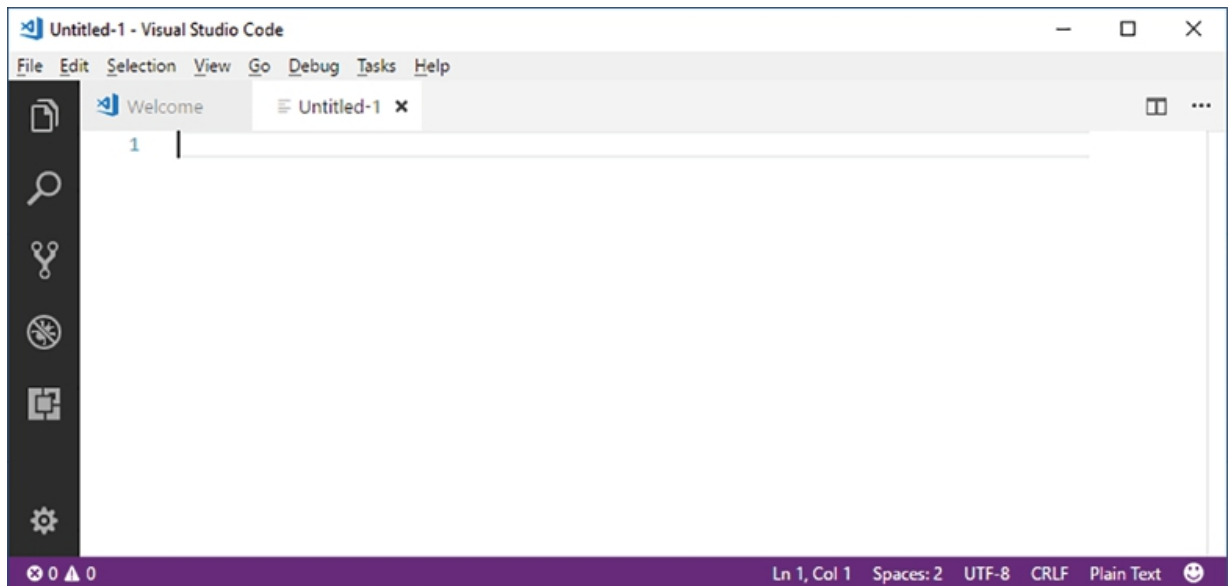


Figura 1.9 – Novo documento no Visual Studio Code.

1.5 Saída de dados com alert()

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 `cap1`. 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. Ou, então, utilizar o Scratchpad que acompanha o navegador Mozilla Firefox.

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 **Plain Text**, uma lista de tipos de arquivos é apresentada, conforme ilustra a Figura 1.10. 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`, como na Figura 1.11.

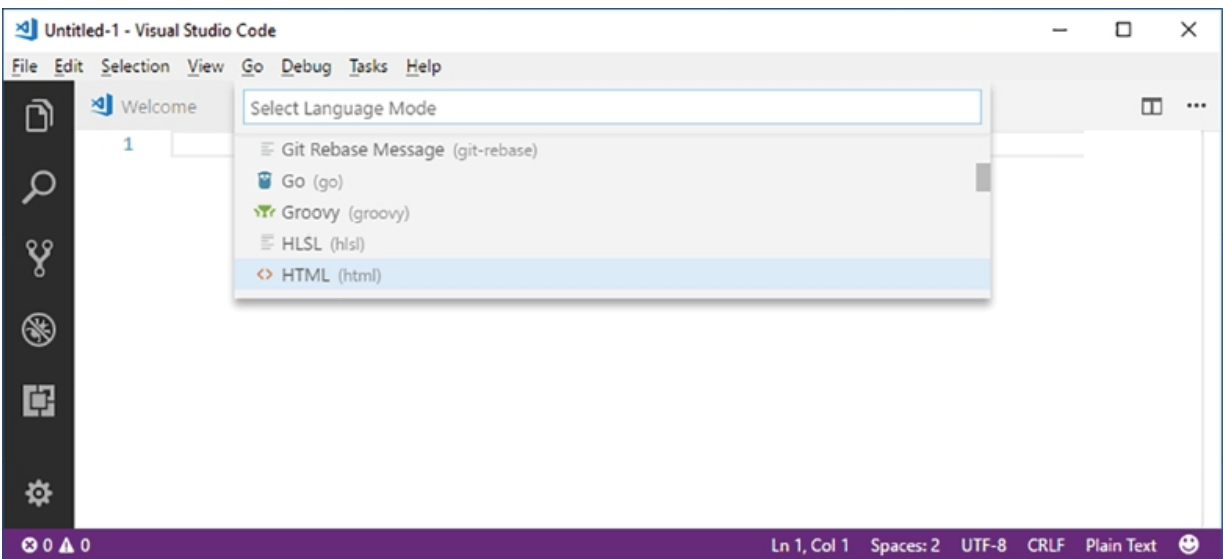


Figura 1.10 – Ao clicar em “Plain Text” na barra de status, pode-se selecionar o tipo do arquivo.

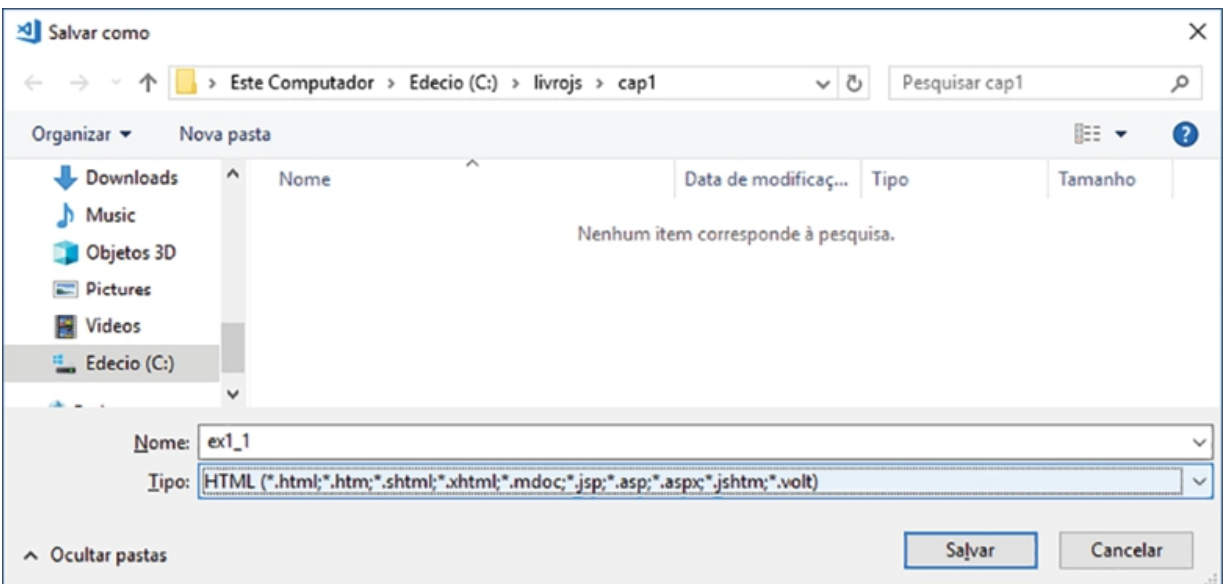


Figura 1.11 – Ou indicar o tipo do arquivo ao salvá-lo. Isso pode ser feito logo depois de criar o documento.

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!");  
</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\cap1\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 1.12.



Figura 1.12 – Exemplo de saída de dados em uma caixa de alerta.

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 4, 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, utiliza-se o comando:

```
var nome;
```

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 `var` e do sinal de atribuição da seguinte forma:

```
var idade = 18;
```

As novas versões dos navegadores web permitem a declaração de constantes. Diferente das variáveis, uma constante não pode ter o seu conteúdo alterado no decorrer do programa. Mas, então, por que declarar uma constante? Vamos utilizar um exemplo para ilustrar os benefícios de trabalhar com constantes. Imagine o programa anteriormente citado de montagem do layout com as poltronas de um teatro. Em vez de colocar o número total de poltronas diretamente no código, podemos declarar uma constante para indicar esse número. Assim, se por algum motivo esse número tiver de ser alterado, não é preciso procurá-lo em todo o código, basta modificar o valor atribuído à constante.

Para declarar uma constante em um programa, utilize a palavra reservada `const` seguida pelo nome da constante e da atribuição de valor a ela. É padrão utilizar letras maiúsculas para nomear as constantes.

```
const POLTRONAS = 240;
```

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>
  var nome = prompt("Qual é o seu nome?");
  alert("Olá " + nome);
</script>
```

Observe a execução do programa nas figuras 1.13 e 1.14. Inicialmente, é solicitado o nome do usuário, a partir do método `prompt` (Figura 1.13). Após, a mensagem de “Olá “ seguida do nome digitado é apresentada (Figura 1.14).

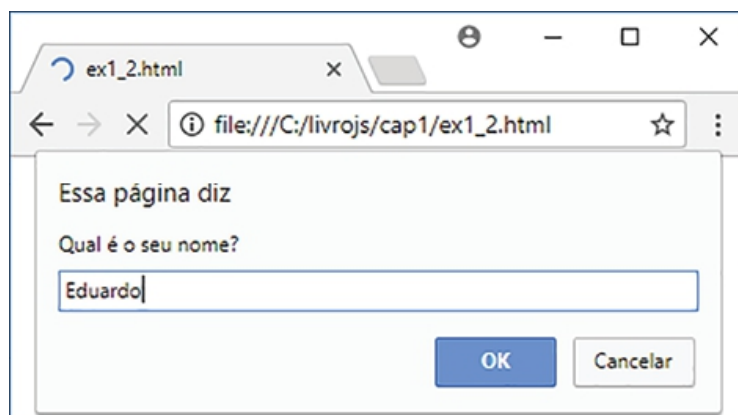


Figura 1.13 – Execução do método `prompt`.

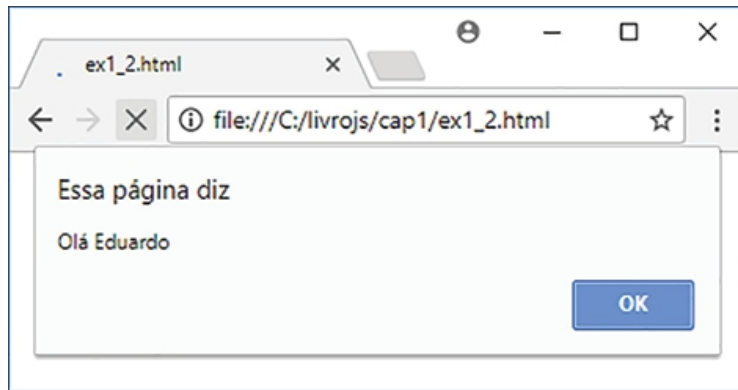


Figura 1.14 – Execução do alert com um texto concatenado com uma variável.

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:

```
var 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 1.14.

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 várias linhas
*/
```

O Exemplo 1.3 contém a inserção desses dois tipos de comentários.

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>
  /*
    Operações envolvendo strings e números
  */
  var a = "20";
  var b = a * 2; // b = 40;
  var c = a / 2; // c = 10;
  var d = a - 2; // d = 18
  var 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” (`var 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. No Exemplo 1.4, é acrescentado o uso de `Number()`, que converte o texto em número e realiza a soma entre os dois valores.

Exemplo 1.4 – String convertida em número (ex1_4.html)

```
<script>
  /*
    Operações envolvendo strings e números
  */
  var a = "20";
  var b = a * 2; // b = 40
  var c = a / 2; // c = 10
  var d = a - 2; // d = 18
  var e = a + 2; // e = 202
  var f = Number(a);
  var g = f + 2; // g = 22
  alert("g: " + g); // exibe o valor de uma variável
</script>
```

Em JavaScript não é necessário 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 1.5 exibe os valores das variáveis declaradas no script, destacados na Figura 1.15.

Exemplo 1.5 – Tipos de variáveis (ex1_5.html)

```
<script>
  var fruta = "Banana";
  var preco = 3.50;
  var levar = true;
  var novoValor;
  alert(fruta + " " + preco + " " + levar + " " + novoValor);
</script>
```

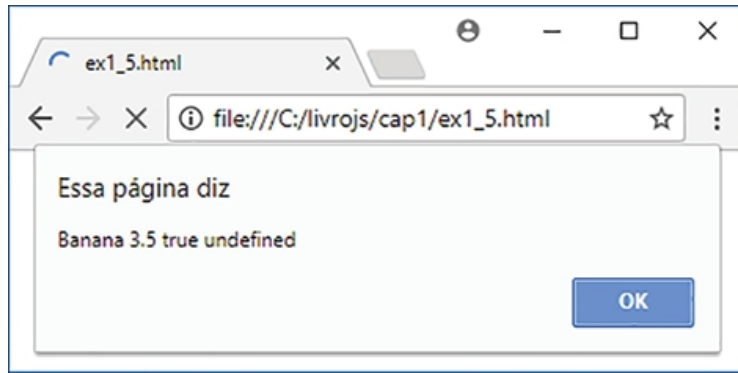


Figura 1.15 – Principais tipos de dados JavaScript.

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 uma forma 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 1.6 – Cálculo do dobro de um número (ex1_6.html)

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

A execução desse script apresenta as telas ilustradas nas figuras 1.16 e 1.17.

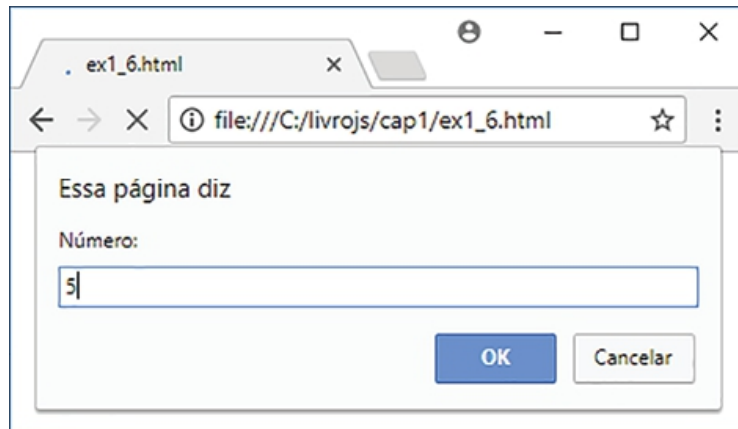


Figura 1.16 – Leitura do número no programa dobro.

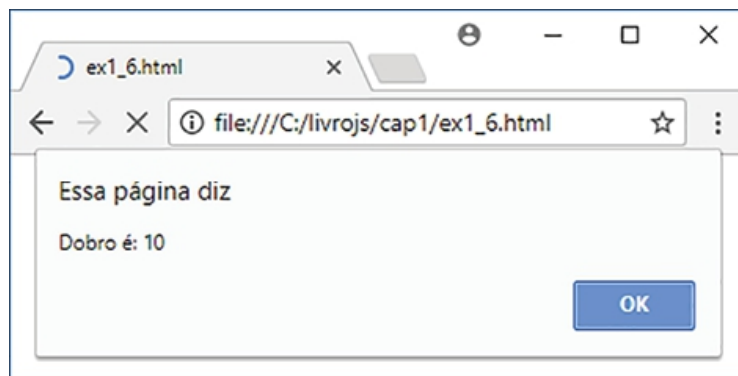


Figura 1.17 – Apresentação do dobro do número.

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. O Exemplo 1.7 implementa essa operação. Observe que a conversão pode ser feita junto com a entrada de dados. Aqui poderia ocorrer um erro como o usuário digitar uma letra no lugar de um número. Esse assunto será abordado no Capítulo 3. Por ora, vamos imaginar que o nosso usuário é bonzinho e digita um número como solicitado... O resultado da execução é o mesmo apresentado nas figuras 1.16 e 1.17.

Exemplo 1.7 – Conversão da entrada para número (ex1_7.html)

```
<meta charset="utf-8">
<script>
```

```
// lê um dado de entrada
var num = Number(prompt("Número: "));
// calcula o dobro
var dobro = num + num;
// exibe a resposta
alert("Dobro é: " + dobro);
</script>
```

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 1.8 – Soma de dois números (ex1_8.html)

```
<meta charset="utf-8">
<script>
// lê os números
var num1 = Number(prompt("1º Número: "));
var num2 = Number(prompt("2º Número: "));
// calcula a soma
var soma = num1 + num2;
// exibe o resultado
alert("Soma é: " + soma);
</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 1.9 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 1.9 – Cálculo do valor do jantar (ex1_9.html)

```
<meta charset="utf-8">
<script>
  // lê o valor do jantar
  var jantar = Number(prompt("Valor do Jantar R$: "));
  // calcula os dados
  var garcom = jantar * 0.10;
  var total = jantar + garcom;
  // apresenta as respostas
  alert("Taxa Garçom R$: " + garcom.toFixed(2) + "\nTotal R$: " +
total.toFixed(2));
</script>
```

A Figura 1.18 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 1.18 – 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:

```
var total = jantar + (jantar * 0.10); // ou então,  
var 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 1.10 – Cálculo da duração de horas de uma viagem (ex1_10.html)

```
<meta charset="utf-8">
<script>
  // lê os dados de entrada
  var dias = Number(prompt("Nº Dias: "));
  var horas = Number(prompt("Nº Horas: "));
  // calcula a duração
  var total = (dias * 24) + horas;
  // exibe o total
  alert("Total de Horas: " + 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 `cap1`, 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:

1ª Nota: 7.0

2ª 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.

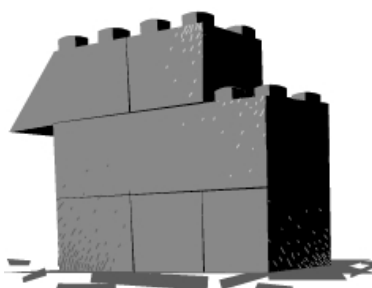
- Lógica de programação é algo que se aprende com treinamento. Ao criar um algoritmo, vamos “ensinar” o computador a realizar uma tarefa. Com a compreensão e a prática dos exercícios, passamos a assimilar a forma como as estruturas de um programa devem estar organizadas. Usamos a lógica para montar corretamente 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.
- 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 (HyperText 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 **cap2**, 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 (**File / New File**), salve o documento como sendo do tipo HTML (dentro da pasta **cap2**, 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.

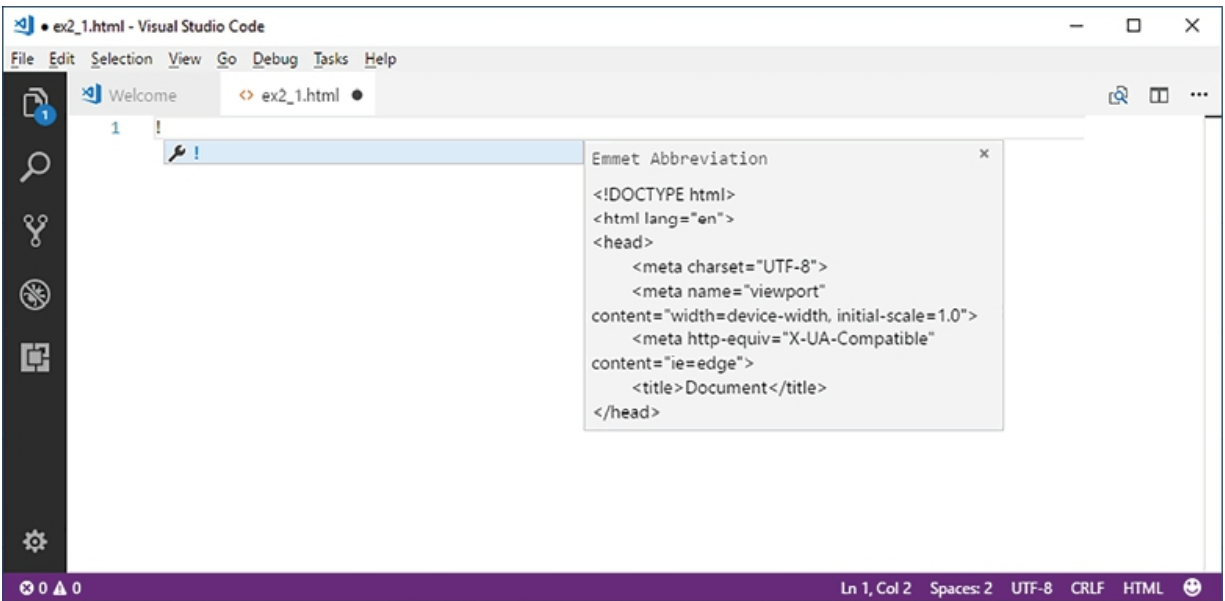


Figura 2.1 – Depois de salvar o documento como tipo HTML, digite !.

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.

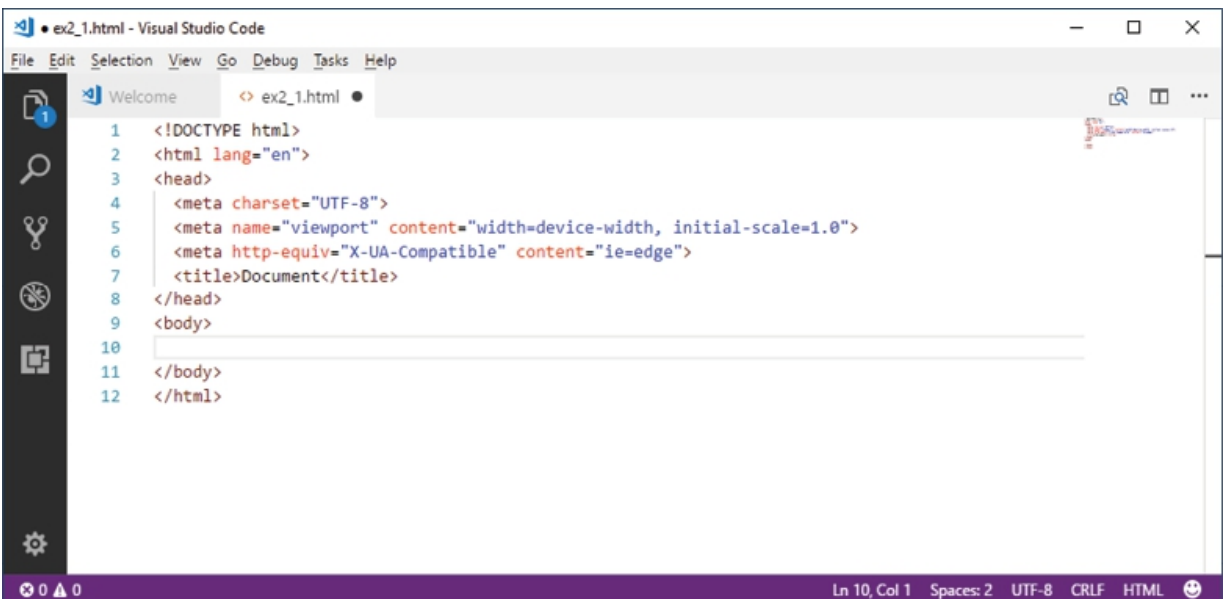


Figura 2.2 – Ao pressionar Tab ou Enter, os comandos básicos de um documento HTML são inseridos.

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>
<p> Nome: <input type="text" id="nome">
    <input type="button" value="Mostrar"></p>
<p id="resposta"></p>
```

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. A tag HTML que cria um campo de formulário para digitação de dados é `<input>`. Cada campo deve possuir um identificador (id) a ser utilizado no código JavaScript para obter o conteúdo do campo. E a tag `<input type="button" ...>`, como o nome sugere, cria um botão geralmente utilizado para acionar um programa JavaScript. Esses dois últimos comandos estão dentro de um parágrafo criado com as tags `<p>` e `</p>`. A última linha cria um novo parágrafo no documento, que será utilizado 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 name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Exemplo 2.1</title>
</head>

<body>
  <h1> Programa Olá Você! </h1>
  <p> Nome: <input type="text" id="nome">
    <input type="button" value="Mostrar"> </p>
  <p id="resposta"></p>
</body>

</html>
```

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 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 programadas em uma função. O evento mais comum de ser programado é o clique no botão. Mas há diversos outros, como modificar o conteúdo de um campo, enviar os dados de um formulário, sair de um campo, carregar a página, entre outros. Mais detalhes sobre os eventos serão apresentados no Capítulo 7.

Para criar um evento e definir qual função será acionada quando esse evento ocorrer, deve-se utilizar uma palavra reservada para indicar para qual evento a linguagem ficará “na escuta”, seguido do nome da função a ser acionada. A palavra reservada pode ser, por exemplo, (on)click, (on)change, (on)submit, (on)blur ou (on)load. O uso do “on” para preceder o evento vai depender da forma a ser utilizada para criar o link entre o HTML e o JavaScript (visto nas próximas seções).

Já as funções JavaScript são declaradas a partir da palavra-chave **function** seguida do nome da função e dos parênteses (). Uma função contém um conjunto de comandos que realizam uma ação. Os exemplos do Capítulo 1, de calcular o dobro de um número, a média de notas de um aluno ou o valor total de um jantar com a taxa do garçom, serão agora inseridos dentro de uma função 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.

Os comandos que pertencem a uma função devem estar delimitados pelas chaves { }. A função que obtém o conteúdo do campo nome de um formulário e a exibe precedida pela palavra “Olá” em um parágrafo com id="resposta" é a seguinte:

```
function mostrarOla() {
```

```
var nome = document.getElementById("nome").value;  
document.getElementById("resposta").textContent = "Olá " + nome;  
}
```

Como uma função executa um conjunto de ações, é uma boa prática de programação dar um nome para a função começando por um verbo. O uso do camelcase (com letras maiúsculas no meio do nome para indicar uma nova palavra) é também um padrão recomendado e facilita a compreensão da leitura do nome.

2.4 Método `getElementById()`

Para referenciar um elemento HTML identificado no documento, deve-se utilizar o método `getElementById()`. Esse método permite referenciar qualquer elemento da página, como um campo de formulário, um parágrafo, um botão, uma imagem, entre outros. Para que um elemento HTML seja referenciado, ele precisa conter um atributo `id`.

Podemos armazenar a referência a um elemento em uma variável e depois obter a sua propriedade, como no exemplo a seguir:

```
var inputNome = document.getElementById("nome");  
var nome = inputNome.value;
```

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

```
var nome = document.getElementById("nome").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.

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 visto anteriormente, utilizamos o objeto **document**, que a partir da execução do método **getElementById()** pode referenciar os elementos identificados pelos ids “nome” e “resposta”. Já a propriedade **value** é utilizada para obter o conteúdo digitado no campo de formulário. A propriedade **textContent**, por sua vez, altera um atributo do documento, que é o conteúdo do parágrafo identificado por “resposta”.

2.5 Propriedades **textContent**, **innerHTML** e **value**

Na função **mostrarOla()**, são utilizadas as propriedades **value** e **textContent**. Elas 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 junto com o método **getElementById()** que faz uma referência a um campo de formulário identificado no código HTML.

Já a propriedade **textContent** 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 **textContent** 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 **textContent**, **innerHTML** e **value**.

*Tabela 2.1 – Comparativo entre as propriedades **textContent**, **innerHTML** e **value***

textContent	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. Nos exemplos do livro, utilizaremos as propriedades `value` e `textContent`. O inconveniente da `textContent` é que, no Internet Explorer, ela só passou a ser suportada na versão 9. Caso necessite rodar nossos exemplos nas versões antigas do Internet Explorer, substitua a `textContent` por `innerHTML`.

2.6 Formas de adicionar JavaScript ao HTML

Nas seções anteriores, vimos como criar uma página HTML básica e uma função JavaScript. Veremos agora como criar um “link” entre elas.

2.6.1 Uso de rotinas de tratamento de eventos HTML

Uma das formas de indicar qual função JavaScript será executada quando um determinado evento ocorrer é informar isso no próprio elemento HTML que vai acionar a function. Assim, a tag HTML programada para chamar o script deve conter a palavra-chave que identifica o evento (`onclick`, `onchange`, `onmouseover`) associado ao nome da função a ser executada. No Exemplo 2.1, a tag `<input type="button"...>` ficaria da seguinte forma:

```
<input type="button" value="Mostrar" onclick="mostrarOla()">
```

Nesse modelo, os códigos JavaScript seriam inseridos no documento HTML a partir do acréscimo das tags `<script>` e `</script>` e da function `mostrarOla()`, vista na seção anterior.

O uso desse tipo de vinculação do JavaScript com o documento HTML não é considerado uma boa prática de programação, visto que o papel do HTML (conteúdo e semântica) e do JavaScript (comportamento) são diferentes no processo de construção de sites web. E misturá-los num mesmo arquivo com referências definidas nas próprias tags HTML remete a uma prática antiga de construir sites de uma forma desorganizada, em que HTML, CSS e JavaScript ocupavam um mesmo espaço. Dar manutenção em um sistema web construído dessa forma causa “arrepios” em qualquer equipe de desenvolvimento...

2.6.2 Uso de rotinas de tratamento de eventos DOM

Separar os arquivos contendo programas JavaScript dos arquivos contendo as tags HTML é um modo melhor de organizar os documentos de um site. Para criar o vínculo entre os arquivos, deve-se acrescentar no documento HTML a seguinte tag:

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

E adicionar um identificador para o botão que vai controlar o evento `onClick` para que ele seja referenciado.

```
<input type="button" value="Mostrar" id="mostrar">
```

No arquivo `.js`, as funções JavaScript devem estar no início do arquivo. Em seguida, é necessário referenciar o elemento programado para acionar o script. Observe no exemplo a seguir as regras de sintaxe do arquivo `.js` desse modelo.

```
function mostrarOla() {  
    var nome = document.getElementById("nome").value;  
    document.getElementById("resposta").innerHTML = "Olá " + nome;  
}  
var mostrar = document.getElementById("mostrar");  
mostrar.onclick = mostrarOla;
```

Essa abordagem é conhecida como rotina de tratamento de eventos DOM. A DOM (Document Object Model) permite acessar cada elemento de uma página HTML como uma estrutura hierárquica – semelhante à árvore genealógica de uma família. No Capítulo 9, vamos realizar operações de inserção, acesso e remoção de elementos de um site utilizando a DOM.

Esse modo de vinculação entre os documentos HTML e JavaScript é suportado pelos principais navegadores web. Possui apenas um inconveniente, que é a impossibilidade de anexar mais de uma função a um mesmo evento.

2.6.3 Uso dos listeners (ouvintes) de eventos

A abordagem recomendada para vincular um arquivo `.js` ao documento HTML é a utilização dos chamados listeners (ouvintes) de eventos ou modelos de eventos DOM nível 2. Nesse modelo, é possível registrar

múltiplas funções para um mesmo elemento HTML em um mesmo evento. Essa será a forma utilizada nos exemplos do livro.

Para criar um listener, deve-se utilizar o método `addEventListener`, cuja sintaxe é:

```
elemento.addEventListener('evento', função);
```

O nome do evento não necessita ser precedido pela sigla 'on', como nos modelos anteriores. Após o nome da função, pode ainda ser informado um terceiro parâmetro. Ele é opcional e indica a forma de propagação do evento.

No Internet Explorer, esse método passou a ser suportado na versão 9. Caso necessite desenvolver para versões antigas desse navegador, pesquise sobre o método `attachEvent()`. Com ele, é possível criar uma condição para verificar aspectos de compatibilidade entre os navegadores.

Vamos então criar esse arquivo para depois rodar o nosso programa. Para melhor organização, vamos colocar os arquivos JavaScript dentro de uma nova pasta chamada `js` dentro de `cap2`. Crie a pasta. Inicie um novo programa no Visual Studio Code, salve o arquivo dentro da pasta `js` com o nome `ex2_1.js`. Lembre-se de indicar que o arquivo deve ser do tipo JavaScript. Digite o código a seguir, sendo que as linhas que iniciam pelas `//` são comentários e, portanto, opcionais para o funcionamento do programa.

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

```
// declara a função mostrarOla
function mostrarOla() {
    // obtém o conteúdo do campo (com id=) nome
    var nome = document.getElementById("nome").value;
    // exhibe no parágrafo (resposta): "Olá " e o nome informado
    document.getElementById("resposta").textContent = "Olá " + nome;
}
// cria uma referência ao botão (com id=) mostrar
var mostrar = document.getElementById("mostrar");
// registra para o botão "mostrar" um ouvinte para o evento click,
// que ao ser clicado irá chamar a função mostrarOla
mostrar.addEventListener("click", mostrarOla);
```

Neste capítulo, foram adicionados novos conceitos e instruções de programação. Não se preocupe... 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 `getElementById()`. Para concluir a digitação do método, pressione **Tab** ou **Enter**. É possível também obter informações sobre o método ao clicar no ícone ao final de cada linha.

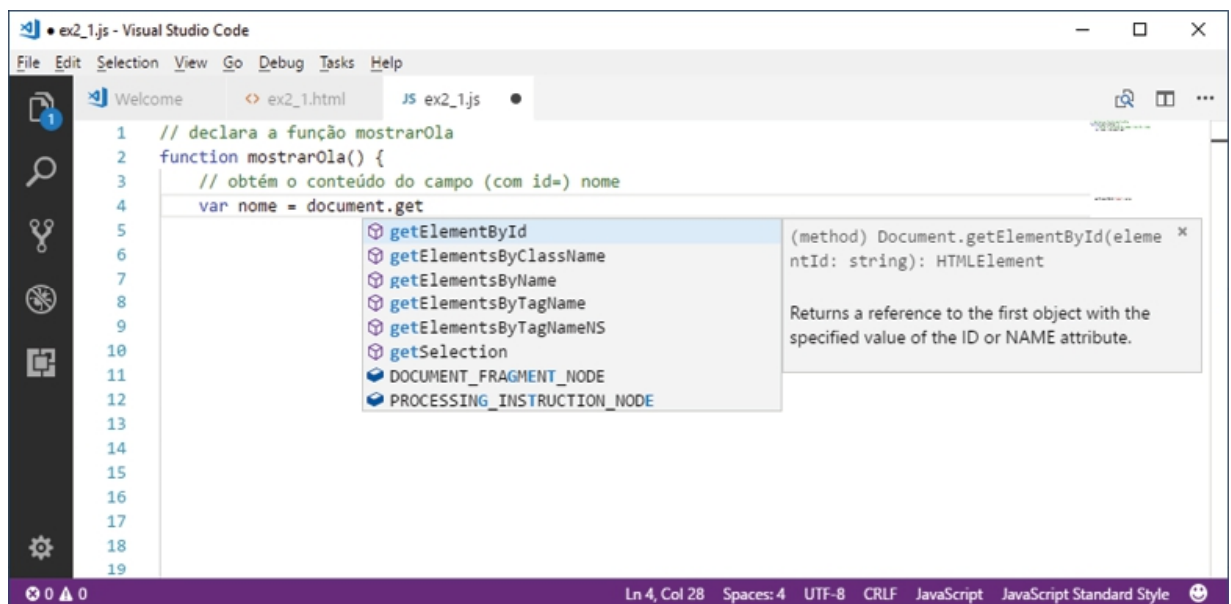
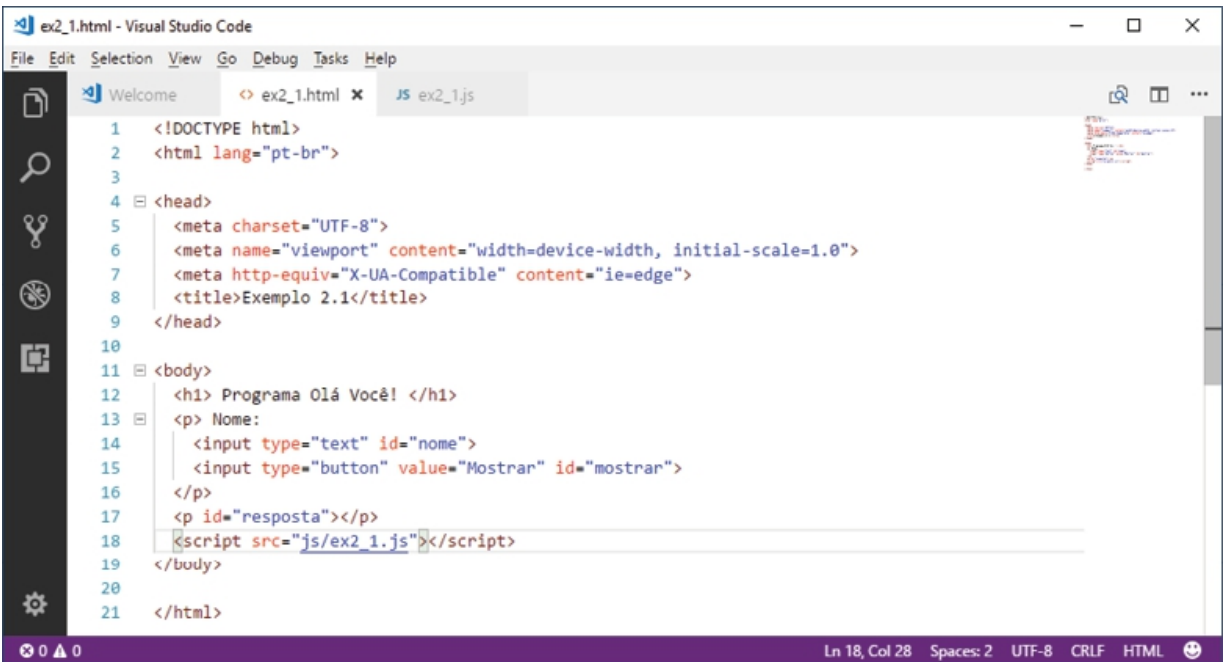


Figura 2.4 – IntelliSense no Visual Studio Code.

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 **Help / Keyboard Shortcuts Reference**. Na Figura 2.5, é exibida a tela, com os comandos do arquivo HTML formatados pelo editor.



```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <meta http-equiv="X-UA-Compatible" content="ie=edge">
8   <title>Exemplo 2.1</title>
9 </head>
10
11 <body>
12   <h1> Programa Olá Você! </h1>
13   <p> Nome:
14     <input type="text" id="nome">
15     <input type="button" value="Mostrar" id="mostrar">
16   </p>
17   <p id="resposta"></p>
18   <script src="js/ex2_1.js"></script>
19 </body>
20
21 </html>
```

Figura 2.5 – Indentação do código facilita a compreensão e manutenção de seus programas.

Vamos testar o nosso programa! Não se esqueça de alterar o documento HTML com o acréscimo do id para o botão (linha 15 da Figura 2.5) e da linha que referencia o arquivo JavaScript (linha 18 da Figura 2.5). Depois de realizar a alteração desses comandos, salve o arquivo HTML (e também o arquivo .js) e recarregue a página no seu browser favorito. A Figura 2.6 exibe a tela com a execução do script.



Figura 2.6 – Ao clicar no botão Mostrar, a mensagem Olá seguida do nome é exibida.

2.7 Nomenclatura e escopo das variáveis

No exemplo `ex2_1.js`, declaramos duas variáveis: uma para receber o texto digitado no campo de formulário com `id="nome"`, e outra para fazer

referência ao elemento `button` do formulário com `id="mostrar"`. Em programas maiores, o número de variáveis vai naturalmente ser maior também. O que significa que se torna importante adotar algumas regras para facilitar a compreensão dessas variáveis.

Se analisarmos o exemplo `ex2_1.js`, vamos perceber que as variáveis são utilizadas para fins diferentes: texto de entrada e referência ao elemento `button`. Em outros exemplos, vamos também utilizar variáveis para referenciar elementos de saída. A partir do próximo exemplo, vamos definir os nomes dos elementos HTML precedidos por letras que nos indicam o seu uso no programa: `in` (input) para os campos de entrada de texto, `bt` (button) para os botões e `out` (output) para os locais de saída de dados, como `inNome`, `btMostrar` e `outResposta`. E, nos programas JavaScript, vamos adotar uma pequena regra: ao referenciar esses elementos, vamos manter seu nome. Já as variáveis que recebem o conteúdo de um campo não utilizam esses caracteres iniciais. Esse nosso padrão visa facilitar o processo de aprendizagem. Como definir regras para nomes de variáveis e elementos é algo bem particular de cada programador, você pode criar as próprias regras e, inclusive, aplicá-las nos exemplos deste livro. Basta substituir os nomes no momento em que for implementar os programas.

Outro detalhe importante sobre as variáveis é que elas têm um “escopo” de abrangência (de validade no programa). As variáveis declaradas dentro de uma função são denominadas variáveis locais e, como o nome sugere, só têm validade nesse local (nessa função). Já as variáveis declaradas fora das funções são consideradas variáveis globais, e valem para todo o programa. Sempre que possível, dê preferência por utilizar variáveis locais, pois o espaço alocado por uma variável local é liberado após a conclusão da função. Além disso, diversos outros problemas podem ser evitados, como redefinir na função o valor de uma variável global já existente.

Um cuidado com relação à declaração das variáveis JavaScript: não se esqueça de utilizar a palavra `var` antes do nome da variável. Declarar variáveis sem a palavra `var` faz com que ela seja entendida como uma variável global, mesmo se declarada dentro de uma função. As boas práticas de programação recomendam o uso do `var` antes do nome da variável, e devemos seguir essa recomendação.

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:

```
var a = 5 % 2; // a = 1
var 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 será convertido para positivo. Se positivo, o valor permanece o mesmo. Exemplo: Math.abs(-3) => 3 Veja uma aplicação de Math.abs() no Exemplo 8.3
Math.ceil(num)	Arredonda o valor para cima. Dessa forma, se o valor possuir decimais, retorna o próximo número inteiro do valor analisado. Exemplo: Math.ceil(4.2) => 5 Veja uma aplicação de Math.ceil() no Exemplo 9.2
Math.floor(num))	Arredonda o valor para baixo, retornando a parte inteira do número. Exemplo: Math.floor(7.9) => 7 Veja uma aplicação de Math.floor() no Exemplo 2.2
Math.pow(base, exp)	Retorna a base elevada ao expoente. Exemplo: Math.pow(3, 2) => 9 Veja uma aplicação de Math.pow() no Exemplo 3.2
Math.random()	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.

	Exemplo: <code>Math.random()</code> => 0.6501314074022906 Veja uma aplicação de <code>Math.random()</code> no jogo do Exemplo 5.2
<code>Math.round(num)</code>	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. Exemplo: <code>Math.round(2.7)</code> => 3 Veja uma aplicação de <code>Math.round()</code> no Exemplo 6.10
<code>Math.sqrt(num)</code>	Retorna a raiz quadrada do número (square root). Exemplo: <code>Math.sqrt(16)</code> => 4 Veja uma aplicação de <code>Math.sqrt()</code> no Exemplo 3.5

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

```
var media1 = (nota1 + nota2) / 2;
var 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...: $\text{Math.sqrt}(9) * 8 / 2$

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 1.10, do primeiro capítulo:

```
var 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>
  var 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 uma Vídeo Locadora, 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 2.7.

Crie um novo arquivo (**File / New File**) 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**). Para o `p`, faça da mesma forma. 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, conforme ilustra a Figura 2.8. Para finalizar a seleção de

um comando, pressione **Tab** (ou **Enter**).



Figura 2.7 – Programa deve converter a duração do filme.

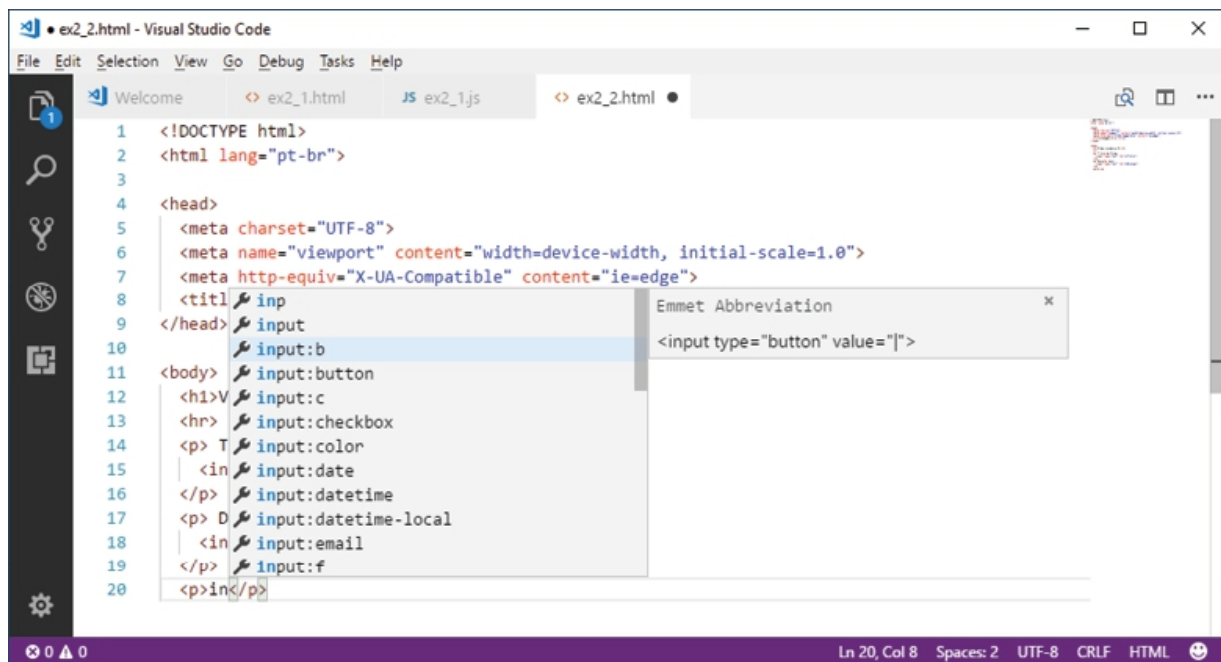


Figura 2.8 – Atalhos para a digitação das tags HTML.

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

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
```

```

scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Exemplo 2.2</title>
</head>

<body>
  <h1>Vídeo Locadora JS</h1>
  <hr>
  <p> Título do Filme:
    <input type="text" id="inTitulo">
  </p>
  <p> Duração (min):
    <input type="text" id="inDuracao">
  </p>
  <p>
    <input type="button" value="Converter em Horas e Minutos"
id="btConverter">
  </p>
  <p id="outTitulo"></p>
  <p id="outResposta"></p>
  <script src="js/ex2_2.js"></script>
</body>
</html>

```

O código HTML contém as tags de definição da estrutura da página, dos campos de digitação dos dados do filme e dos parágrafos para a exibição da resposta, a ser posteriormente preenchida pelo programa JavaScript.

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

```

function converterDuracao() {
  // cria referência aos elementos da página
  var inTitulo = document.getElementById("inTitulo");
  var inDuracao = document.getElementById("inDuracao");
  var outTitulo = document.getElementById("outTitulo");
  var outResposta = document.getElementById("outResposta")

  // obtém conteúdos dos campos de entrada
  var titulo = inTitulo.value;
  var duracao = Number(inDuracao.value);

```



```

// arredonda para baixo o resultado da divisão
var horas = Math.floor(duracao / 60);
// obtém o resto da divisão entre os números
var minutos = duracao % 60;
// altera o conteúdo dos parágrafos de resposta
outTitulo.textContent = titulo;
outResposta.textContent = horas + " hora(s) e " + minutos + "
minuto(s)";
}
// cria uma referência ao elemento btConverter (botão)
var btConverter = document.getElementById("btConverter");
// registra um evento associado ao botão, para carregar uma função
btConverter.addEventListener("click", converterDuracao);

```

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 2.9 ilustra uma execução desse programa.



Exemplo 2.3

file:///C:/livrojs/cap2/ex2_3.html

Revenda de Veículos JS

Veículo: Sanderio 2016

Preço RS: 30000.00

Ver Promoção

Promoção: Sanderio 2016

Entrada de RS: 15000.00

+ 12x de RS: 1250.00

Figura 2.9 – Exemplo de dados do programa revenda de veículos.

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

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Exemplo 2.3</title>
</head>

<body>
  <h1>Revenda de Veículos JS</h1>
  <p> Veículo:
    <input type="text" id="inVeiculo"></p>
  <p> Preço R$:
    <input type="text" id="inPreco"></p>
  <p> <input type="button" value="Ver Promoção" id="btVerPromocao">
</p>
  <h3 id="outVeiculo"></h3>
  <h3 id="outEntrada"></h3>
  <h3 id="outParcela"></h3>
  <script src="js/ex2_3.js"></script>
</body>

</html>
```

O código HTML segue o padrão do exemplo anterior. Ele deve conter os ids dos elementos que serão referenciados pelo programa JavaScript vinculado a esse documento a partir da tag <script src...>.

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

```
function mostrarPromocao() {
  // cria referência aos elementos manipulados pelo programa
  var inVeiculo = document.getElementById("inVeiculo");
  var inPreco = document.getElementById("inPreco");
  var outVeiculo = document.getElementById("outVeiculo");
  var outEntrada = document.getElementById("outEntrada");
```

```

var outParcela = document.getElementById("outParcela");

// obtém conteúdo dos campos de entrada
var veiculo = inVeiculo.value;
var preco = Number(inPreco.value);

// calcula valor da entrada e das parcelas
var entrada = preco * 0.50;
var parcela = (preco * 0.50) / 12;
// altera o conteúdo dos parágrafos de resposta
outVeiculo.textContent = "Promoção: " + veiculo;
outEntrada.textContent = "Entrada de R$: " + entrada.toFixed(2);
outParcela.textContent = "+ 12x de R$: " + parcela.toFixed(2);
}
// cria uma referência ao elemento btVerPromocao (botão)
var btVerPromocao = document.getElementById("btVerPromocao");
// registra um evento associado ao botão, para carregar uma função
btVerPromocao.addEventListener("click", mostrarPromocao);

```

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 `preco` por 2. Para o cálculo das parcelas, também poderíamos utilizar a própria variável `entrada` e dividir por 12.

O programa inicia pela criação das variáveis associadas aos elementos da página. Na sequência, obtêm-se os conteúdos digitados nos campos de formulário da página HTML. Em seguida, é realizado o cálculo dos valores a serem apresentados como resposta. E, no final da `function`, são apresentadas as informações nos respectivos elementos de saída da página HTML. A propriedade `textContent` recebe os dados a serem exibidos.

Observe que os programas seguem um padrão. Os exemplos HTML devem conter as tags básicas de estruturação da página e do conteúdo a ser exibido. Os programas JavaScript iniciam pela declaração da função. Nesta ocorrem as etapas destacadas no capítulo anterior: entrada, processamento e saída. E, por fim, a forma de referenciar o botão identificado na página e registrar um evento associado a esse botão.

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 2.10.



Exemplo 2.4

file:///C:/livrojs/cap2/ex2_4.html

Restaurante JS

Buffet por Quilo R\$:

Consumo do Cliente (gr):

Valor a pagar R\$: 28.00

Figura 2.10 – Programa Restaurante: dados de exemplo.

Exemplo 2.4 – Código HTML do programa Revenda de Veículos (ex2_4.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Exemplo 2.4</title>
</head>

<body>
  <h1>Restaurante JS</h1>
  <p>Buffet por Quilo R$:
    <input type="text" id="inQuilo">
  </p>
  <p>Consumo do Cliente (gr):
    <input type="text" id="inConsumo">
  </p>
  <p>
    <input type="button" value="Calcular Preço" id="btCalcular">
  </p>
  <h3 id="outValor"></h3>
  <script src="js/ex2_4.js"></script>
```

```
</body>
```

```
</html>
```

Observe que o código HTML modifica pouco em relação aos exemplos anteriores. O arquivo contendo o código JavaScript para implementar o cálculo e a exibição do valor a ser pago pelo cliente é exibido no programa a seguir.

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

```
function calcularPreco() {  
    // cria referência aos elementos da página  
    var inQuilo = document.getElementById("inQuilo");  
    var inConsumo = document.getElementById("inConsumo");  
    var outValor = document.getElementById("outValor")  
  
    // obtém conteúdo dos campos de entrada  
    var quilo = Number(inQuilo.value);  
    var consumo = Number(inConsumo.value);  
  
    // calcula valor a ser pago  
    var valor = (quilo / 1000) * consumo;  
    // altera o conteúdo da linha de resposta  
    outValor.textContent = "Valor a pagar R$: " + valor.toFixed(2);  
}  
// cria referência ao elemento btCalcular  
var btCalcular = document.getElementById("btCalcular");  
// registra um evento associado ao botão, para carregar uma função  
btCalcular.addEventListener("click", calcularPreco);
```

A função 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 2.11 apresenta a tela com um exemplo de dados de entrada e saída do programa.



Exercício 2.a

file:///C:/livrojs/cap2/resp2_a.html

Farmácia JS

Medicamento:

Preço RS:

Promoção de Aspirina

Leve 2 por apenas RS: 14.00

Figura 2.11 – Exemplo dos dados de entrada e saída do programa Farmácia.

- b) **Elaborar um programa para uma lan 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 2.12 exibe um exemplo com dados do programa.

A screenshot of a web browser window titled 'Exercício 2.b'. The address bar shows the file path 'file:///C:/livrojs/cap2/resp2_b.html'. The page content includes the title 'Lan House JS', a label 'Valor por 15min de Uso RS:' followed by an input field containing '2.00', another label 'Tempo de Uso do Cliente:' followed by an input field containing '35', a button labeled 'Calcular Valor a Pagar', and a final result 'Valor a Pagar RS: 6.00'.

Figura 2.12 – Dados de exemplo do programa Lan House JS.

- 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 2.13.

A screenshot of a web browser window titled 'Exercício 2.c'. The address bar shows the file path 'file:///C:/livrojs/cap2/resp2_c.html'. The page content includes the title 'Supermercado JS', a label 'Produto:' followed by an input field containing 'Escova Dental', another label 'Preço RS:' followed by an input field containing '4.00', a button labeled 'Ver Promoção', and two lines of promotional text: 'Escova Dental - Promoção: Leve 3 por RS: 10.00' and 'O 3° produto custa apenas RS: 2.00'.

Figura 2.13 – 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 interpretador 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 de parágrafos 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 o método `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 `textContent`).

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 registrar 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

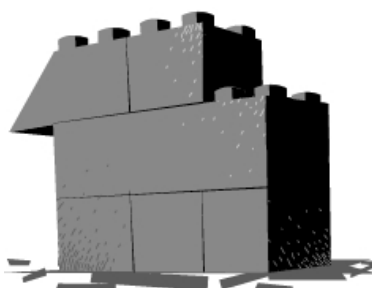
JavaScript têm papéis diferentes no processo de construção de um site. A ligação do HTML com o JavaScript, nesse modelo, ocorre a partir da inserção da tag `<script src="nomearq.js"></script>` no documento HTML.

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

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).

3.1 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 3.1 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.

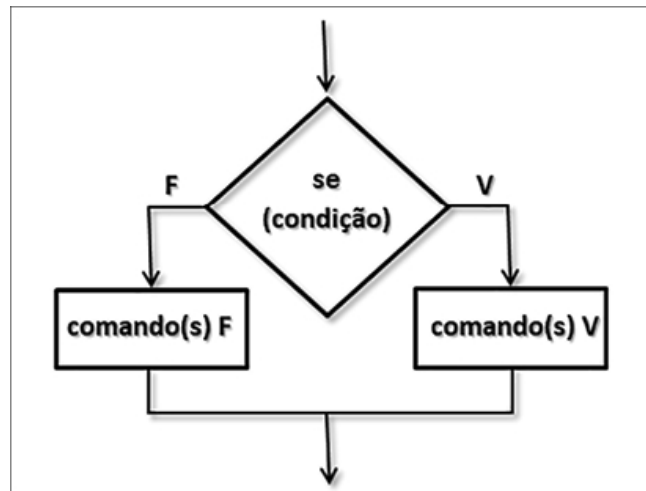


Figura 3.1 – Representação do fluxo dos comandos em uma estrutura condicional.

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.

3.2 Operadores relacionais

Para definir as condições utilizadas nas estruturas condicionais, deve-se fazer uso dos operadores relacionais. A Tabela 3.1 apresenta os operadores relacionais utilizados em JavaScript. Quando inseridas em um programa, cada comparação deve retornar true (verdadeiro) ou false (falso).

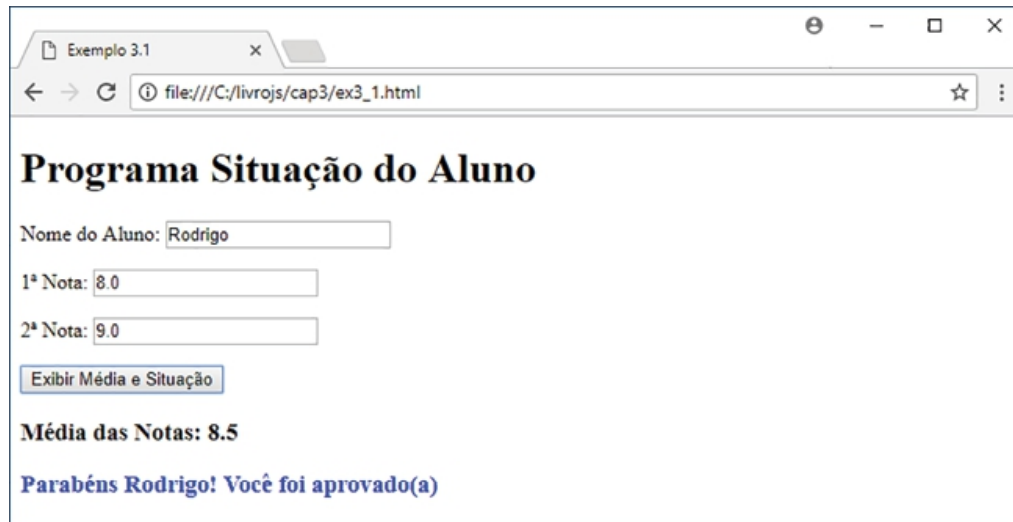
Tabela 3.1 – Operadores relacionais

Símbol o	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 comparam 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 3.2 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 3.1

file:///C:/livrojs/cap3/ex3_1.html

Programa Situação do Aluno

Nome do Aluno:

1ª Nota:

2ª Nota:

Média das Notas: 8.5

Parabéns Rodrigo! Você foi aprovado(a)

Figura 3.2 – Programa calcula a média e, a partir de uma condição, exibe a situação do aluno.

Como nos capítulos anteriores, vamos agora criar a pasta **cap3** 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 **cap3** com o nome **ex3_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 3.1 – Código HTML do programa Situação do Aluno (ex3_1.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Exemplo 3.1</title>
</head>

<body>
```

```

<h1>Programa Situação do Aluno</h1>
<p>Nome do Aluno:
  <input type="text" id="inNome">
</p>
<p>1ª Nota:
  <input type="text" id="inNota1">
</p>
<p>2ª Nota:
  <input type="text" id="inNota2">
</p>
<p>
  <input type="button" value="Exibir Média e Situação"
id="btResultado">
</p>
<h3 id="outMedia"></h3>
<h3 id="outSituacao"></h3>
<script src="js/ex3_1.js"></script>
</body>

</html>

```

O código HTML contém as tags básicas de estruturação da página, dos campos de entrada e das linhas h3 que serão utilizadas para exibição das respostas do programa. Além deles, há também a referência ao arquivo contendo o programa JavaScript para manipular os dados da página. O programa que vai manipular esses dados é descrito a seguir e deve ser salvo dentro da pasta js (livrojs/cap3/js/ex3_1.js).

Código JavaScript do programa Situação do Aluno (js/ex3_1.js)

```

function calcularMedia() {
  // cria referência aos elementos da página
  var inNome = document.getElementById("inNome");
  var inNota1 = document.getElementById("inNota1");
  var inNota2 = document.getElementById("inNota2");
  var outSituacao = document.getElementById("outSituacao");
  var outMedia = document.getElementById("outMedia");

  // obtém os conteúdos dos campos de edição da página
  var nome = inNome.value;

```

```

var nota1 = Number(inNota1.value);
var nota2 = Number(inNota2.value);

// calcula a média das notas
var media = (nota1 + nota2) / 2;

// apresenta a média (altera o conteúdo do elemento outMedia)
outMedia.textContent = "Média das Notas: " + media.toFixed(1);

// cria a condição
if (media >= 7) {
    // altera o texto e estilo da cor do elemento outSituacao
    outSituacao.textContent = "Parabéns " + nome + "! Você foi aprovado(a)";
    outSituacao.style.color = "blue";
} else {
    outSituacao.textContent = "Ops " + nome + "... Você foi reprovado(a)";
    outSituacao.style.color = "red";
}
}
// cria uma referência ao elemento btResultado (botão)
var btResultado = document.getElementById("btResultado");
// registra um evento associado ao botão, para carregar uma função
btResultado.addEventListener("click", calcularMedia);

```

Nesse programa, a condição foi definida na saída de dados, já que a leitura dos campos do formulário e o cálculo da média são os mesmos.

Observe que a função `calcularMedia()` inicia com a criação de referências a todos os elementos manipulados pela `function`, a partir do método `getElementById()`. Vamos manter esse padrão em todos os programas do livro para facilitar a compreensão: primeiro criamos referência aos elementos da página e, em seguida, acessamos ou modificamos alguma de suas propriedades ou executamos algum de seus métodos.

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) {  
    outSituacao.textContent = "Parabéns " + nome + "! Você foi  
    aprovado(a)";  
    outSituacao.style.color = "blue";  
} else if (media >= 4) {  
    outSituacao.textContent = "Atenção " + nome + ". Você está em  
    exame";  
    outSituacao.style.color = "green";  
} else {  
    outSituacao.textContent = "Ops " + nome + "... Você foi  
    reprovado(a)";  
    outSituacao.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 uma mesma função.

3.3 Operadores lógicos

No exercício anterior, foi criada uma condição com base em uma comparação (`media >= 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 3.2 apresenta os principais operadores lógicos disponíveis em JavaScript.

Tabela 3.2 – 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”).

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

A Tabela 3.3 apresenta a tabela verdade da negação, representada pelo símbolo (!). Ela pode ser aplicada a apenas uma proposição.

Tabela 3.3 – 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 3.4 expressa os valores resultantes para as

proposições p e q, usando a conjunção.

Tabela 3.4 – Conjunção (&&)

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

Na conjunção, como observado na Tabela 3.4, 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 3.5 representa a disjunção.

Tabela 3.5 – Disjunção (||)

p	q	p 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 ||.

São exemplos de condições que utilizam o operador `||`.

```
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 (`||`). 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: $22 * altura^2$ (para homens); e $21 * altura^2$ (para mulheres).

A página deve ficar semelhante à ilustrada na Figura 3.3. 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 3.3 – 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 3.3. A tag `<link rel="stylesheet" href="css/estilos.css">` inserida no código HTML define a ligação do documento HTML 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.

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 na tag ``. O código HTML da página exibida na Figura 3.3 pode ser conferido no Exemplo 3.2. 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 3.2 – Código HTML (ex3_2.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 3.2</title>
</head>

<body>
  
  <h1>Programa Cálculo do Peso Ideal</h1>
  <p>Nome:
    <input type="text" id="inNome">
  </p>
  <p>Sexo:
    <input type="radio" name="sexo" id="rbMasculino"> Masculino
    <input type="radio" name="sexo" id="rbFeminino"> Feminino</p>
  <p>Altura:
    <input type="text" id="inAltura">
  </p>
  <p>
    <input type="button" value="Calcular Peso Ideal" id="btCalcular">
    <input type="button" value="Limpar Campos" id="btLimpar">
  </p>
</body>
</html>
```

```

</p>
<h3 id="outResposta"></h3>
<script src="js/ex3_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/ex3_2.js)

```

function calcularPeso() {
    // cria referência aos elementos manipulados pela function
    var inNome = document.getElementById("inNome");
    var rbMasculino = document.getElementById("rbMasculino");
    var rbFeminino = document.getElementById("rbFeminino");
    var inAltura = document.getElementById("inAltura");
    var outResposta = document.getElementById("outResposta");

    // obtém o conteúdo dos campos de edição da página
    var nome = inNome.value;
    var masculino = rbMasculino.checked;
    var feminino = rbFeminino.checked;
    var altura = Number(inAltura.value);

    // verifica se nome foi preenchido e sexo selecionado
    if (nome == "" || (masculino == false && feminino == false)) {
        alert("Por favor, informe o nome e selecione o sexo...");
        inNome.focus(); // posiciona (joga o foco) no campo de edição
        inNome
        return;
    }

    // se altura vazio (0) ou NaN: Not-a-Number (um texto for
    informado, por exemplo)
    if (altura == 0 || isNaN(altura)) {
        alert("Por favor, informe a altura corretamente...");
        inAltura.focus();
        return;
    }
}

```

```

// se masculino (significa se masculino == true)
if (masculino) {
    var peso = 22 * Math.pow(altura, 2); // Math.pow eleva ao quadrado
} else {
    var peso = 21 * Math.pow(altura, 2);
}

// apresenta a resposta (altera o conteúdo da linha outResposta)
outResposta.textContent = nome + ": Seu peso ideal é " +
peso.toFixed(3) + " kg";
}
// cria referência ao elemento btCalcular e registra evento
associado a calcularPeso
var btResultado = document.getElementById("btCalcular");
btCalcular.addEventListener("click", calcularPeso);

```

Vamos analisar essa primeira function. Inicialmente, foi feita uma referência a todos os elementos utilizados na function e, após, obtidos os seus conteúdos. Utilizaram-se os operadores lógicos para verificar se os campos foram preenchidos corretamente. Nesse programa, essa verificação foi realizada a partir de duas condições. A primeira verifica o preenchimento do campo nome e do sexo do usuário. Caso não informados, é exibida uma mensagem de advertência, posiciona-se no campo `inNome` e há ainda um comando `return`. Mas retornar para onde?

No item 2.3, foi feita uma introdução às funções e aos eventos JavaScript. Nela, destacou-se que uma página está estática, aguardando uma ação (evento) do usuário. Quando um evento programado ocorre, uma função é chamada. E quando, nessa função, um comando `return` é executado, ele retorna para o local que originou a sua chamada – nesse caso, a própria página. Portanto, o programa é interrompido e fica no aguardo de uma nova ação (evento) programado no sistema – que poderá ser um novo clique nesse botão.

Já a segunda condição valida o preenchimento do campo `altura`. Um detalhe nessa condição é que o método `Number()`, se aplicado a um conteúdo vazio, retorna o valor 0. Caso o usuário informe uma altura inválida (um texto ou um número com vírgula, por exemplo), a conversão realizada pelo método `Number()` resultará em `"NaN"` (Not-a-Number). A

condição, portanto, faz essa verificação e, caso isso ocorra, também exibe uma mensagem, posiciona no campo `altura` e retorna à página.

Realizadas as validações, é hora de calcular o peso ideal seguindo a fórmula indicada. 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`.

Observe que nesse programa apenas o cálculo da fórmula está dentro da condição. A entrada e saída de dados é a mesma para pessoas do sexo masculino ou feminino.

Para limpar o conteúdo dos campos, foi criada a função `limparCampos()`. Esse processo pode ser realizado de duas formas principais. No exemplo de resolução exibido a seguir, é realizado o acesso aos elementos da página e atribuído um valor vazio ou falso a cada campo.

```
function limparCampos() {  
  // Limpa os conteúdos dos elementos  
  document.getElementById("inNome").value = "";  
  document.getElementById("rbMasculino").checked = false;  
  document.getElementById("rbFeminino").checked = false;  
  document.getElementById("inAltura").value = "";  
  document.getElementById("outResposta").textContent = "";  
  // posiciona (joga o foco) no elemento inNome  
  document.getElementById("inNome").focus();  
}  
var btLimpar = document.getElementById("btLimpar");  
btLimpar.addEventListener("click", limparCampos);
```

Observe que a propriedade `value`, antes utilizada para obter o conteúdo de um campo, é agora empregada para atribuir um conteúdo vazio para os campos do formulário. Para desmarcar os campos do tipo `RadioButton`, a propriedade `checked` recebe o valor `false`. Já a propriedade `textContent`, por sua vez, é utilizada para limpar a linha da resposta. No final, capturamos o segundo botão (`btLimpar`) e adicionamos um “ouvinte” para o evento `click` que vai carregar a função `limparCampos`.

Outra forma de limpar os campos de edição é executando uma chamada ao método `location.reload()` que recarrega a página atual. Dessa forma, a página volta ao seu estado inicial, não sendo necessário alterar o conteúdo de cada elemento da página. A seguir, a programação com o uso do método `location.reload()`.

```
function limparCampos() {  
    // recarrega a página  
    location.reload();  
    // posiciona (joga o foco) no elemento inNome  
    document.getElementById("inNome").focus();  
}  
var btLimpar = document.getElementById("btLimpar");  
btLimpar.addEventListener("click", limparCampos);
```

3.4 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 3.3 – Condições com switch... case (ex3_3.html)

```
<script>  
var bairro = prompt("Bairro de Entrega: ");  
var 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).

3.5 Operador ternário

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

```
var 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**.

```

if (idade >= 18) {
    var categoria = "Adulto";
} else {
    var categoria = "Juvenil";
}

```

Em razão da sua simplicidade, o operador ternário é bastante utilizado e está disponível na maioria das linguagens de programação da atualidade.

3.6 Exemplos

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 3.4 ilustra a tela com dados de entrada e saída do programa.



Figura 3.4 – Exemplo de dados de entrada e saída do programa fuso horário.

Exemplo 3.4 – Código HTML do programa Fuso Horário (ex3_4.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 3.4</title>
</head>

<body>
  
```

```

<h1>Programa Fuso Horário</h1>
<p>Hora no Brasil (h.m):
  <input type="text" id="inHoraBrasil">
</p>
<p>
  <input type="button" value="Exibir Hora na França" id="btExibir">
</p>
<h3 id="outHoraFranca"></h3>
<script src="js/ex3_4.js"></script>
</body>

</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 3.4 (js/ex3_4.js)

```

function calcularFuso() {
  // cria uma referência aos elementos da página
  var inHoraBrasil = document.getElementById("inHoraBrasil");
  var outHoraFranca = document.getElementById("outHoraFranca");

  // obtém e converte o conteúdo do campo inHoraBrasil
  var horaBrasil = Number(inHoraBrasil.value);

  // se não preencheu ou Not-a-Number (NaN)
  if (inHoraBrasil.value == "" || isNaN(horaBrasil)) {
    alert("Informe a hora no Brasil corretamente"); // exibe alerta
    inHoraBrasil.focus(); // posiciona em inHoraBrasil
    return;
  }

  var horaFranca = horaBrasil + 5; // calcula o horário na França

  // se passar das 24 horas na Franca ...
  if (horaFranca > 24) {

```

```

        horaFranca = horaFranca - 24; // ... subtrai 24
    }
    // exibe resposta (altera conteúdo do elemento outHoraFranca)
    outHoraFranca.textContent = "Hora na França: " +
    horaFranca.toFixed(2);
}
// cria uma referência ao elemento btExibir e registra evento
// associado a function
var btExibir = document.getElementById("btExibir");
btExibir.addEventListener("click", calcularFuso);

```

A `function` `calcularFuso()` segue o padrão de código do Exemplo 3.2. Novamente a `function` começa pela criação de variáveis que referenciam os elementos manipulados pelo programa. Após, obtém-se o conteúdo do campo `inHoraBrasil`, convertido para número. Caso o usuário não tenha informado a hora no Brasil ou digite um valor inválido, uma mensagem de advertência é exibida, o programa joga o foco no campo `inHoraBrasil` e retorna à página.

O próximo passo é calcular o horário na França. Para isso, 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. Após a condição, o valor da variável `horaFranca` é atribuído à propriedade `textContent`, junto com um texto inicial, de modo a exibir o valor calculado.

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 3.5 ilustra uma execução desse programa.



Figura 3.5 – Exemplo de execução do programa raiz quadrada.

Exemplo 3.5 – Programa Raiz Quadrada – código HTML (ex3_5.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 3.5</title>
</head>

<body>
  
  <h1>Programa Raiz Quadrada</h1>
  <p>Número:
    <input type="text" id="inNumero">
  </p>
  <p>
    <input type="button" value="Exibir Raiz Quadrada" id="btExibir">
  </p>
  <h3 id="outResposta"></h3>
  <script src="js/ex3_5.js"></script>
</body>
```

</html>

Programa Raiz Quadrada – código JavaScript (js/ex3_5.js)

```
function calcularRaiz() {  
    // cria uma referência aos elementos da página  
    var inNumero = document.getElementById("inNumero");  
    var outResposta = document.getElementById("outResposta");  
  
    var numero = Number(inNumero.value); // obtém conteúdo do campo  
    inNumero  
  
    // se não preencheu ou Not-a-Number (NaN)  
    if (numero == 0 || isNaN(numero)) {  
        alert("Informe um número válido..."); // exibe alerta  
        inNumero.focus(); // posiciona no campo inNumero  
        return; // retorna  
    }  
  
    var raiz = Math.sqrt(numero); // calcula raiz quadrada do número  
  
    // se valor da variável raiz igual a este valor arredondado para  
    baixo...  
    if (raiz == Math.floor(raiz)) {  
        outResposta.textContent = "Raiz: " + raiz; // mostra a raiz  
    } else {  
        // senão, exibe mensagem indicando que não há raiz exata  
        outResposta.textContent = "Não há raiz exata para " + numero;  
    }  
}  
// cria referência ao elemento btExibir e registra evento que irá  
carregar function  
var btExibir = document.getElementById("btExibir");  
btExibir.addEventListener("click", calcularRaiz);
```

No programa `ex3_5.js`, é realizada a captura dos elementos e a validação dos dados já discutida 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 com decimais ou sem decimais. Por exemplo, a raiz de 16 é 4.0, já a raiz de 15 é 3.872983...

Para verificar se o número não têm decimais, podemos submetê-lo à

função `Math.floor()`, que arredonda o valor para baixo, ou seja, remove as decimais do número. Se a raiz for igual à aplicação da `Math.floor()` sobre a raiz, significa que não há decimais, pois 4.0 é igual a `Math.floor(4.0)`, por exemplo. Já 3.872983... é diferente de `Math.floor(3.872983...)` – que retorna 3.0. Há outras formas de 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, `4.0 % 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 3.6 exemplifica uma execução do programa.*



Figura 3.6 – Programa caixa eletrônico: exemplo de saque.

Exemplo 3.6 – Programa Caixa Eletrônico – código HTML (ex3_6.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
```

```

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<link rel="stylesheet" href="css/estilos.css">
<title>Exemplo 3.6</title>
</head>
<body>

<h1>Programa Caixa Eletrônico</h1>
<p>Valor do Saque R$:
  <input type="text" id="inSaque">
</p>
<p>
  <input type="button" value="Exibir Notas para Saque"
id="btExibir">
</p>
<h3 id="outNotasCem"></h3>
<h3 id="outNotasCinquenta"></h3>
<h3 id="outNotasDez"></h3>
<script src="js/ex3_6.js"></script>
</body>
</html>

```

Observe que, no código HTML do Exemplo 3.6, 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 ex3_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/ex3_6.js)

```

function calcularNotas() {
  // cria uma referência aos elementos da página
  var inSaque = document.getElementById("inSaque");
  var outNotasCem = document.getElementById("outNotasCem");
  var outNotasCinquenta =
document.getElementById("outNotasCinquenta");
  var outNotasDez = document.getElementById("outNotasDez");

```

```

// Limpa mensagens (caso, segunda execução)
outNotasCem.textContent = "";
outNotasCinquenta.textContent = "";
outNotasDez.textContent = "";

var saque = Number(inSaque.value); // converte conteúdo do campo
inSaque

// se não preencheu ou Not-a-Number (NaN)
if (saque == 0 || isNaN(saque)) {
    alert("Informe o valor do saque corretamente"); // exibe alerta
    inSaque.focus(); // posiciona em inSaque
    return;
}

// verifica se saque não é múltiplo de 10
if (saque % 10 != 0) {
    alert("Valor inválido para notas disponíveis (R$ 10, 50, 100)");
    inSaque.focus();
    return;
}

// calcula notas de 100, 50 e 10
var notasCem = Math.floor(saque / 100);
var resto = saque % 100;
var notasCinquenta = Math.floor(resto / 50);
resto = resto % 50;
var notasDez = Math.floor(resto / 10);
// exibe as notas apenas se houver
if (notasCem > 0) {
    outNotasCem.textContent = "Notas de R$ 100: " + notasCem;
}
if (notasCinquenta > 0) {
    outNotasCinquenta.textContent = "Notas de R$ 50: " +
notasCinquenta;
}
if (notasDez > 0) {
    outNotasDez.textContent = "Notas de R$ 10: " + notasDez;
}

```

```
}  
// cria referência ao elemento btExibir e associa function ao evento  
click  
var btExibir = document.getElementById("btExibir");  
btExibir.addEventListener("click", calcularNotas);
```

Nesse exemplo, recorreremos 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 da função 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.

3.7 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. Exemplos de respostas estão no site da editora. Mas lembre-se: tente resolver primeiro os programas com os cuidados destacados nesses primeiros capítulos antes de ver o exemplo de resolução do exercício. 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. A Figura 3.7 ilustra a tela de execução do programa. Para os exercícios, foi utilizada uma figura padrão, mas você pode substituí-la caso tenha interesse.*



Figura 3.7 – 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 “Multa Leve”. E, se a velocidade for superior a 20% da velocidade permitida, exiba “Multa Grave” – conforme ilustra a Figura 3.8.*

Programa Verifica Velocidade

Velocidade Permitida:

Velocidade do Condutor:

Situação: Multa Leve

Figura 3.8 – 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 3.9. Se o valor for inferior ao tempo mínimo, exiba a mensagem: “Valor Insuficiente”. Considerar os valores/tempos da Tabela 3.6 (o máximo é 120 min).

Tabela 3.6 – Valores do Parquímetro.

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



Figura 3.9 – 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 3.10 exibe um exemplo de execução do exercício.



Figura 3.10 – Exemplo dos dados do programa lados de um triângulo.

3.8 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”.

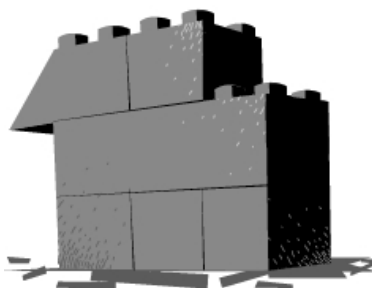
Para definir uma condição em JavaScript, podem ser utilizados os comandos `if.. else` ou `switch.. case`. Quando o número de opções for grande e baseado no conteúdo de uma variável, é recomendável utilizar o `switch.. case`. Os demais casos podem ser resolvidos com a estrutura condicional `if.. else`. 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 (`||`).

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 limpar os conteúdos e 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 4

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 10.2, 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`.

4.1 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 4.1 destaca a sintaxe de um comando `for` com valores de exemplo atribuídos a uma variável de controle `i`.

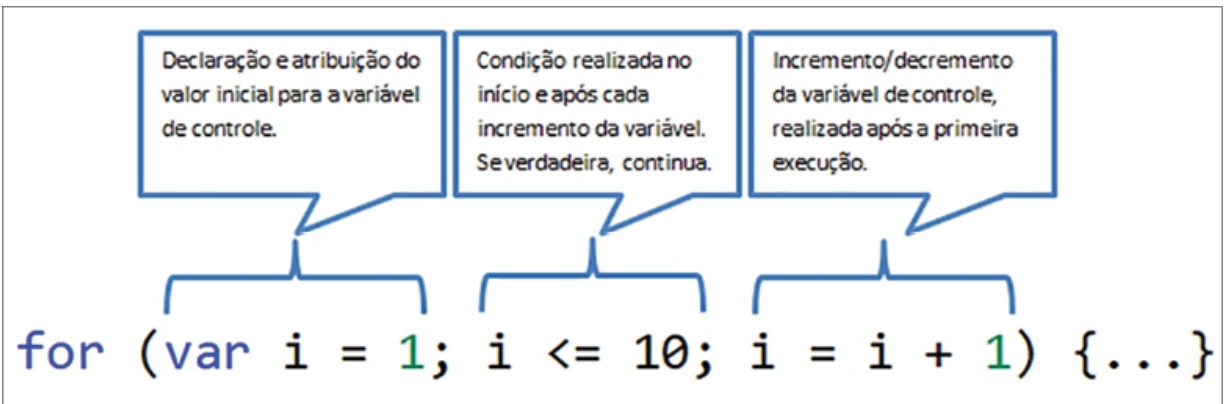


Figura 4.1 – Sintaxe do comando `for` com o significado de cada parte.

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 4.1, 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 4.2 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.

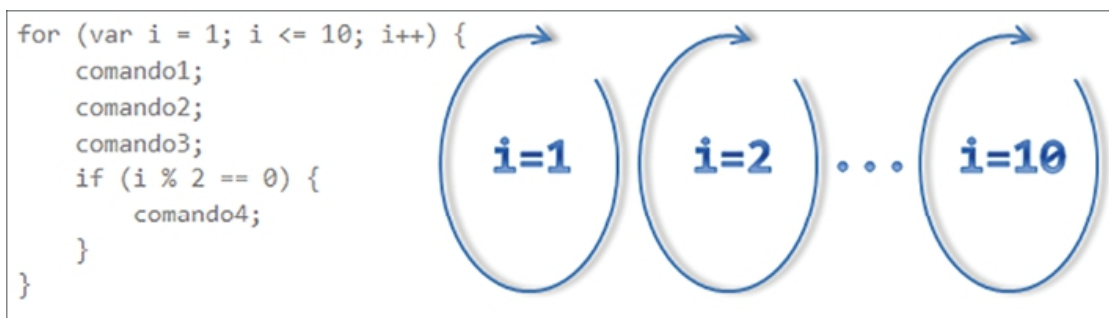


Figura 4.2 – A cada loop, a variável de controle “`i`” é incrementada em um.

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 4.3.

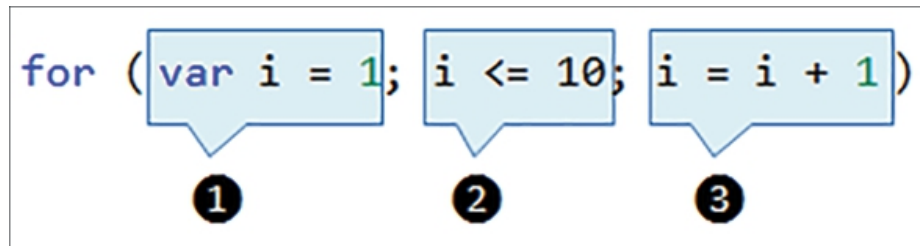


Figura 4.3 – Ao entender a sequência de execução das partes que compõem o comando `for`, você poderá montar diversas variações de valores para a variável de controle.

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.

```
var numeros = "";  
for (var i = 1; i < 4; i = i + 1) {  
    numeros = numeros + i + " ";  
}  
outResposta.textContent = 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 mesmo + `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 mesmo + `i`: `numeros = "1 2 "`.
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 = "1 2 3 "`.
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 **cap4** e, dentro dela, crie as pastas **css**, **img** e **js** (da mesma forma como no Capítulo 3). 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 4.1, para gerar uma página de acordo com a ilustração da execução do programa da Figura 4.4.

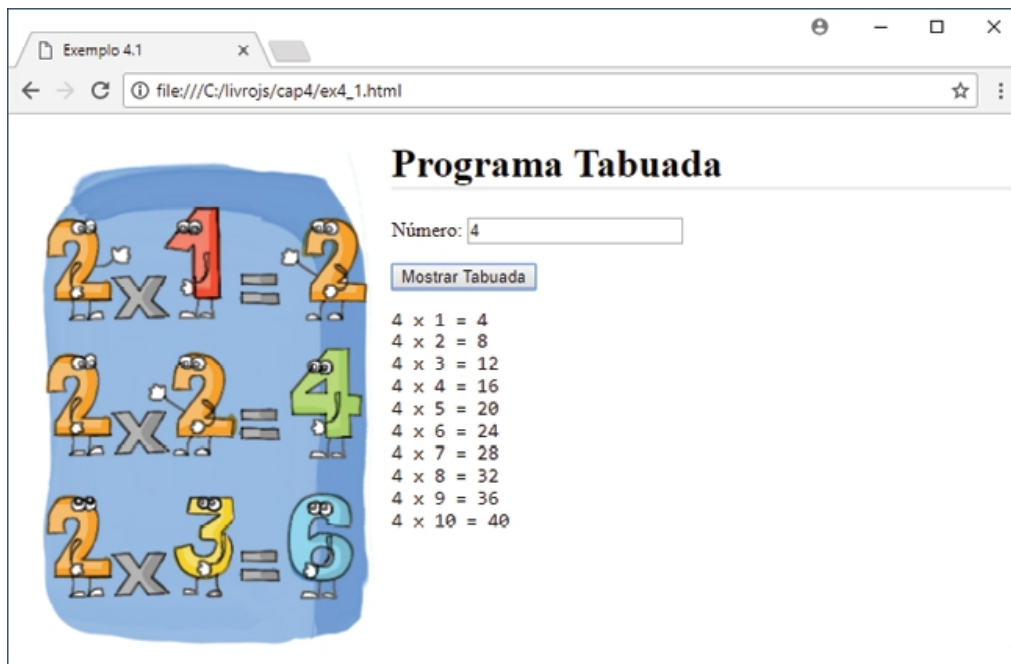


Figura 4.4 – Programa Tabuada: aplicação do comando for.

Exemplo 4.1 – Código HTML do Programa Tabuada (ex4_1.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 4.1</title>
</head>

<body>
  
  <h1> Programa Tabuada </h1>
  <p>Número:
    <input type="text" id="inNumero">
  </p>
  <input type="button" value="Mostrar Tabuada" id="btMostrar">
  <pre id="outTabuada"></pre>
  <script src="js/ex4_1.js"></script>
</body>
```

</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 `<p>` e `<h3>` utilizadas nos exemplos anteriores.

Uma dica do editor Visual Studio Code: para criar o arquivo JavaScript indicado na tag `<script src="js/ex4_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 **Create File**. 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/ex4_1.js)

```
function mostrarTabuada() {  
    // cria referência aos elementos da página  
    var inNumero = document.getElementById("inNumero");  
    var outTabuada = document.getElementById("outTabuada");  
  
    // converte conteúdo do campo inNumero  
    var numero = Number(inNumero.value);  
    // valida o número  
    if (numero == 0 || isNaN(numero)) {  
        alert("Informe um número válido...");  
        inNumero.focus();  
        return;  
    }  
  
    // cria uma variável do tipo String, que irá concatenar a resposta  
    var resposta = "";  
  
    // cria um laço de repetição  
    for (var i = 1; i <= 10; i++) {  
        // a variável resposta vai acumulando os novos conteúdos
```

```

    resposta = resposta + numero + " x " + i + " = " + numero * i +
"\n";
}

// o conteúdo da tag pre é alterado para exibir a tabuada do num
outTabuada.textContent = resposta;
}
// cria referência ao botão e após associa function ao evento click
var btMostrar = document.getElementById("btMostrar");
btMostrar.addEventListener("click", mostrarTabuada);

```

A programação do início da *function* é semelhante aos exemplos dos primeiros capítulos: captura dos elementos e validação do preenchimento. Em seguida, implementamos uma estrutura de repetição para acumular em uma string (*resposta*) a tabuada do número – como 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.

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 4.1, 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 4.1, 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 4.5 apresenta uma página com a demonstração da resposta do programa.



Figura 4.5 – Lista de números em ordem decrescente com o valor inicial informado pelo usuário.

Exemplo 4.2 – Código HTML do Programa Números Decrescentes (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">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 4.2</title>
</head>

<body>
  
  <h1>Programa Números Decrescentes</h1>
  <p>Número:
    <input type="text" id="inNumero">
  </p>
  <input type="button" value="Decrescer Até 1" id="btDecrescer">
  <h3 id="outResposta"></h3>
  <script src="js/ex4_2.js"></script>
</body>

</html>
```

O código do programa JavaScript 4.2 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/ex4_2.js)

```
function listarNumeros() {  
    // cria referência aos elementos que a function irá manipular  
    var inNumero = document.getElementById("inNumero");  
    var outResposta = document.getElementById("outResposta");  
  
    var numero = Number(inNumero.value); // obtém o número informado  
  
    // verifica validade do número  
    if (numero == 0 || isNaN(numero)) {  
        alert("Informe um número válido...");  
        inNumero.focus();  
        return;  
    }  
  
    // inicializa variável resposta  
    var resposta = "Entre " + numero + " e 1: ";  
  
    // cria um for decrescente  
    for (var i = numero; i > 0; i = i - 1) {  
        // resposta vai acumulando números (e vírgulas)  
        resposta = resposta + i + ", ";  
    }  
  
    // altera o conteúdo de outResposta  
    outResposta.textContent = resposta;  
}  
  
// referencia elemento e após associa function ao evento click  
var btDecrescer = document.getElementById("btDecrescer");  
btDecrescer.addEventListener("click", listarNumeros);
```

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 4.5. 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). Podemos criar uma condição para verificar se o número da repetição é o último a ser exibido, isolar o último número ou, ainda, isolar o primeiro número. Observe como poderia ficar o comando `for` com o primeiro exemplo de solução.

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

Em JavaScript, a variável `i` declarada no `for` é uma variável local que existe na função. Após o término do laço, ela mantém o valor do último incremento/decremento da repetição. Portanto, ao referenciar o `i` fora do laço, o conteúdo "1" + "." é acrescentado à resposta.

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 deixa de fora esse valor. As atribuições

também devem ser modificadas. Observe as mudanças.

```
var resposta = "Entre " + numero + " e 1: " + numero;  
  
for (var 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. Caso queira utilizar essa forma e acrescentar um ponto no final, adicione o comando **resposta = resposta + "."** após o laço de repetição.

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**.

4.2 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:

```
// declara e inicializa a variável i
var i = numero;
// enquanto i maior que 0
while (i > 0) {
    // acumula em resposta os números em ordem decrescente
    resposta = resposta + i + ", ";
    // subtrai 1 no valor da variável i (i=i-1)
    i--;
}
```

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 (e o programa aceite esse valor para o número), a condição já é falsa na primeira verificação e o programa não entra no laço de repetição.

4.3 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 4.3, 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 4.6 apresenta um exemplo de saída de dados desse script.

Exemplo 4.3 – Programa JavaScript para ilustrar o funcionamento do laço `do.. while` (ex4_3.html)

```
<script>
// solicita o número e repete a leitura enquanto inválido
do {
    var num = Number(prompt("Número: "));
    if (num == 0 || isNaN(num)) {
        alert("Digite um número válido...");
    }
} while (num == 0 || isNaN(num));

// declara e inicializa variável que irá exibir pares
var pares = "Pares entre 1 e " + num + ": ";

// isola o primeiro par (para evitar última vírgula)
if (num > 1) {
    pares = pares + "2";
}

// laço para acumular pares (inicia em 4, pois o 2 já foi
atribuído)
for (var i = 4; i <= num; i = i + 2) {
    pares = pares + ", " + i;
}
// exibe lista dos números pares
```

```
alert(pares);  
</script>
```

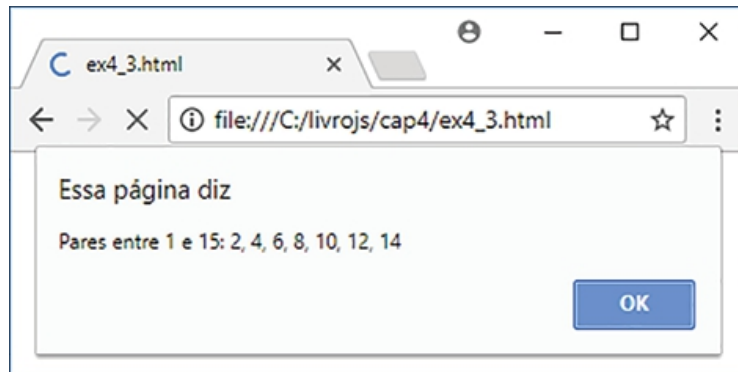


Figura 4.6 – Lista dos pares entre 1 e 15 (número informado pelo usuário).

Para evitar a exibição da última vírgula, foi utilizada a técnica de isolar o primeiro número. Dessa forma, o laço do comando **for** está programado para iniciar em 4. Isso significa que ele poderá não iniciar, caso a condição seja inicialmente falsa, isto é, o valor da variável **i** ser maior que o valor da variável **num**.

4.4 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 que 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 4.7, o que ocorre no laço de repetição no momento em que os comandos **continue** e **break** são executados.

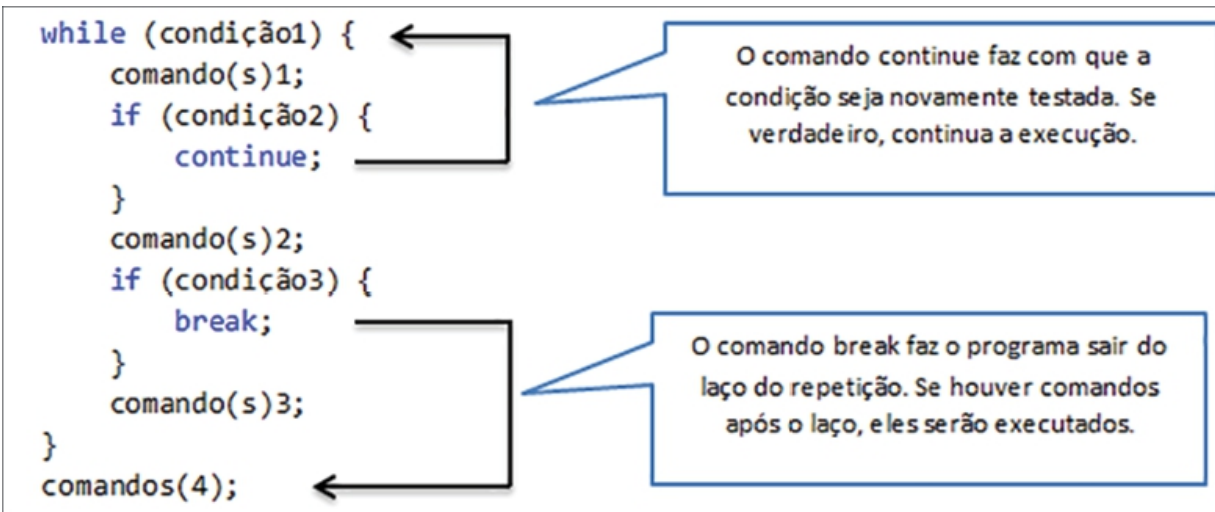


Figura 4.7 – Comandos continue e break modificam o fluxo dos comandos da repetição.

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 decremento da variável de controle ocorre normalmente, como se o laço tivesse sido executado até o seu final.

O Exemplo 4.4 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 4.4 – Uso dos comandos break e continue em um laço de repetição (ex4_4.html)

```
<script>  
    alert("Digite 0 para sair");  
    // início da repetição  
    do {  
        var num = Number(prompt("Número: "));  
  
        if (num == 0 || isNaN(num)) {  
            var sair = confirm("Confirma saída?"); // solicita confirmação  
do usuário  
            if (sair) {  
                break; // sai da repetição  
            } else {
```



```

        continue; // volta ao início do laço
    }
}
// se par, mostra o dobro; ímpar, mostra o triplo
if (num % 2 == 0) {
    alert("O dobro de " + num + " é: " + num * 2);
} else {
    alert("O triplo de " + num + " é: " + num * 3);
}
} 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 4.8 ilustra a saída do método `confirm()`.

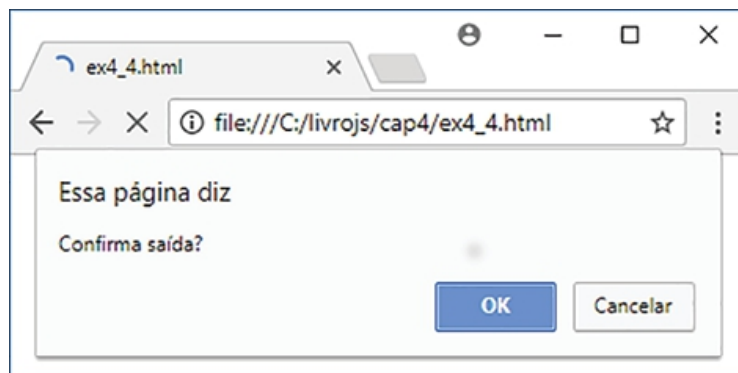


Figura 4.8 – Retorno do método `confirm()` vai definir a execução do `break` ou `continue`.

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`.

4.5 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.

O programa 4.5 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 4.9 exibe uma tela com algumas contas digitadas na execução do programa.

Exemplo 4.5 – Código HTML do programa Contas do Mês (ex4_5.html)

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 4.5</title>
</head>
<body>
  
  <h1>Programa Contas do Mês</h1>
  <p>Descrição da Conta:
    <input type="text" id="inDescricao">
```

```

</p>
<p>Valor a Pagar R$:
  <input type="text" id="inValor">
</p>
<input type="button" value="Registrar Conta" id="btRegistrar">
<pre id="outListaContas" ></pre>
<pre id="outTotal" ></pre>
<script src="js/ex4_5.js"></script>
</body>
</html>

```



Figura 4.9 – Exemplo de contadores e acumuladores.

Código JavaScript do programa Contas do Mês (js/ex4_5.js)

```

// declara e inicializa contadores e acumuladores (variáveis
globais)
var numContas = 0;
var valTotal = 0;

// variável string que acumula as contas
var resposta = "";

function registrarConta() {

```

```

// cria referência aos elementos da página manipulados pela
function
var inDescricao = document.getElementById("inDescricao");
var inValor = document.getElementById("inValor");
var outListaContas = document.getElementById("outListaContas");
var outTotal = document.getElementById("outTotal");

// obtém conteúdo dos campos
var descricao = inDescricao.value;
var valor = Number(inValor.value);

// verifica preenchimento dos campos
if (descricao == "" || valor == 0 || isNaN(valor)) {
    alert("Informe os dados corretamente...");
    inDescricao.focus();
    return;
}

// adiciona valores ao contador e acumulador
numContas++;
valTotal = valTotal + valor;

// concatena as contas
resposta = resposta + descricao + " - R$: " + valor.toFixed(2) +
"\n";

// altera o conteúdo das tags de resposta
outListaContas.textContent = resposta + "-----"
-----";
outTotal.textContent = numContas + " Conta(s) - Total R$: " +
valTotal.toFixed(2);

// limpa campos e posiciona cursor em inDescricao
inDescricao.value = "";
inValor.value = "";
inDescricao.focus();
}

// referencia elemento e após associa function ao evento click
var btRegistrar = document.getElementById("btRegistrar");
btRegistrar.addEventListener("click", registrarConta);

```

Observe que nesse programa fizemos uso de variáveis globais. Como discutido na seção 2.7, uma variável global continua armazenada em memória após a execução de uma função. 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 4.6 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 4.10 exibe a tela do programa números primos.



Figura 4.10 – Programa Números Primos.

Exemplo 4.6 – Código HTML para exibir o layout do programa Números Primos (ex4_6.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```

<meta http-equiv="X-UA-Compatible" content="ie=edge">
<link rel="stylesheet" href="css/estilos.css">
<title>Exemplo 4.6</title>
</head>
<body>
  
  <h1>Programa Números Primos</h1>
  <p>Número:
    <input type="text" id="inNumero">
  </p>
  <input type="button" value="Verificar se é Primo"
id="btVerificarPrimo">
  <h3 id="outResposta"></h3>
  <script src="js/ex4_6.js"></script>
</body>

</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 ex4_6.js a seguir, contém apenas a comparação `if (numDivisores == 2)`.

Código JavaScript do programa Números Primos (js/ex4_6.js)

```

function verificarPrimo() {
  // cria referência aos elementos da página
  var inNumero = document.getElementById("inNumero");
  var outResposta = document.getElementById("outResposta")

  var num = Number(inNumero.value); // obtém o número informado

  // verifica se preencheu corretamente o campo inNumero
  if (num == 0 || isNaN(num)) {
    alert("Número Inválido...");
    inNumero.focus();
    return;
  }

  var numDivisores = 0; // declara e inicializa contador

```

```

// percorre todos os possíveis divisores do número
for (var i = 1; i <= num; i++) {
    // verifica se i (1, 2, 3...) é divisor do num
    if (num % i == 0) {
        numDivisores++; // se é, incrementa contador
    }
}

// se possui apenas 2 divisores, é primo
if (numDivisores == 2) {
    outResposta.textContent = num + " É primo";
} else {
    outResposta.textContent = num + " Não é primo";
}
}

// referencia elemento e após associa function ao evento click
var btVerificarPrimo = document.getElementById("btVerificarPrimo");
btVerificarPrimo.addEventListener("click", verificarPrimo);

```

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.

```

// declara e inicializa a variável tipo flag

```

```

var temDivisor = 0;

// percorre os possíveis divisores do num
for (var i = 2; i <= num / 2; i++) {
    if (num % i == 0) {
        temDivisor = 1; // muda o flag
        break; // sai da repetição
    }
}
// se num > 1 e não possui divisor
if (num > 1 && !temDivisor) {
    outResposta.textContent = num + " É primo";
} else {
    outResposta.textContent = 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.

4.6 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 4.11 – 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.

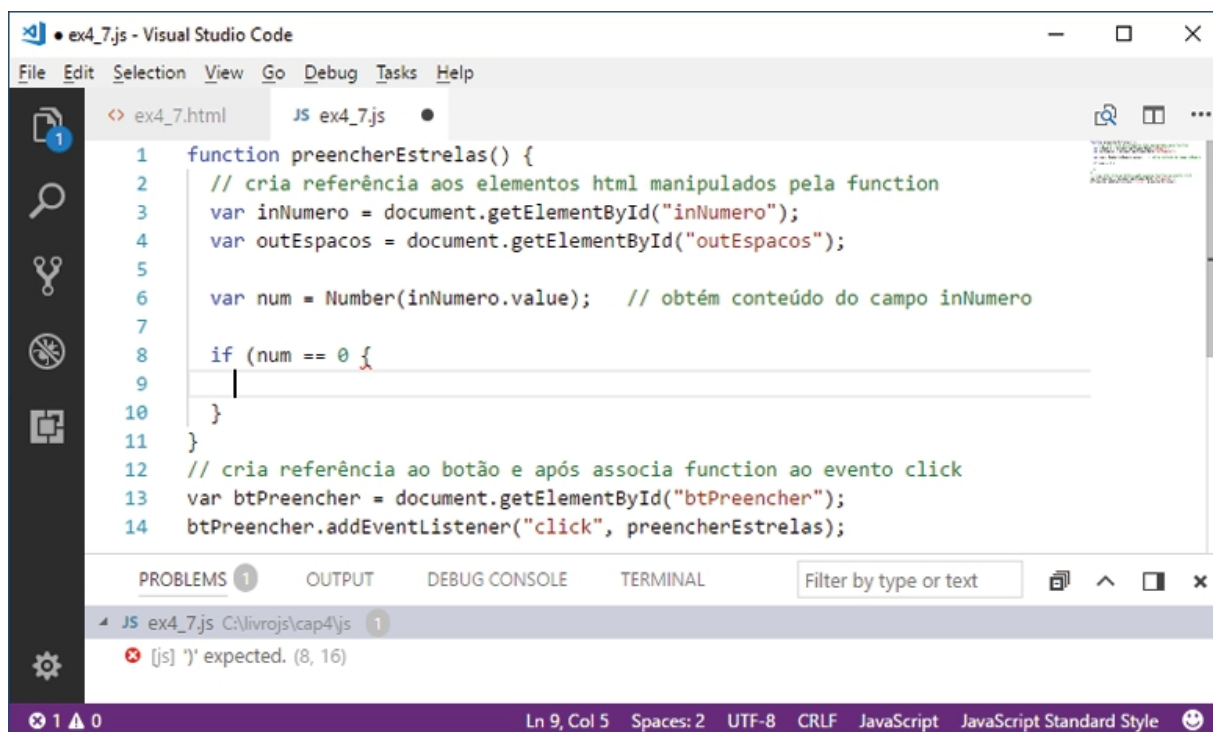


Figura 4.11 – Alguns erros de sintaxe são detectados pelo próprio editor.

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 4.12 ilustra uma execução desse programa.



Figura 4.12 – Programa Fábrica de Estrelas que será depurado no navegador.

Exemplo 4.7 – Código HTML do programa Fábrica de Estrelas (ex4_7.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 4.7</title>
</head>

<body>
  
  <h1>Fábrica de Estrelas</h1>
  <p>Número de Símbolos:
    <input type="text" id="inNumero">
  </p>
```

```

<input type="button" value="Preencher Espaço" id="btPreencher">
<h3 id="outEspacos"></h3>
<script src="js/ex4_7.js"></script>
</body>
</html>

```

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

Código JavaScript do programa Fábrica de Estrelas (js/ex4_7.js)

```

function preencherEstrelas() {
    // cria referência aos elementos html manipulados pela function
    var inNumero = document.getElementById("inNumero");
    var outEspacos = document.getElementById("outEspacos");

    var num = Number(inNumero.value); // obtém conteúdo do campo
    inNumero

    if (num == 0 || isNaN(num)) {
        alert("Número Inválido...");
        inNumero.focus();
        return;
    }
    var estrelas = ""; // declara variável que irá concatenar as
    estrelas/traços
    // cria um laço de repetição de 1 até o número informado
    for (var i = 1; i <= num; i++) {
        if (i % 2 == 1) {
            estrelas = estrelas + "*"; // na posição ímpar do i: *
        } else {
            estrelas = estrelas + "-"; // na posição par: -
        }
    }
    outEspacos.textContent = estrelas; // exibe as estrelas
}
// cria referência ao botão e após associa function ao evento click
var btPreencher = document.getElementById("btPreencher");
btPreencher.addEventListener("click", preencherEstrelas);

```

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 4.7 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 **ex4_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 4.13.

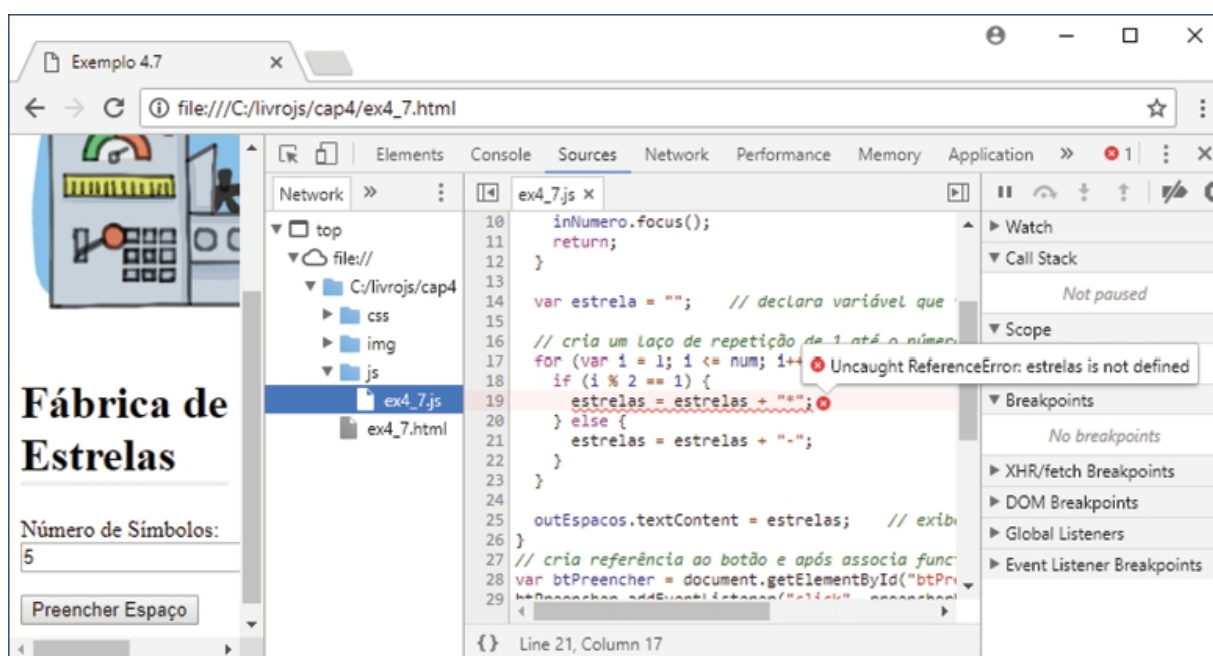


Figura 4.13 – Depurador posiciona na linha com erro e exibe descrição do problema verificado.

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 “+” ao lado de Watch. A Figura 4.14 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.

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.

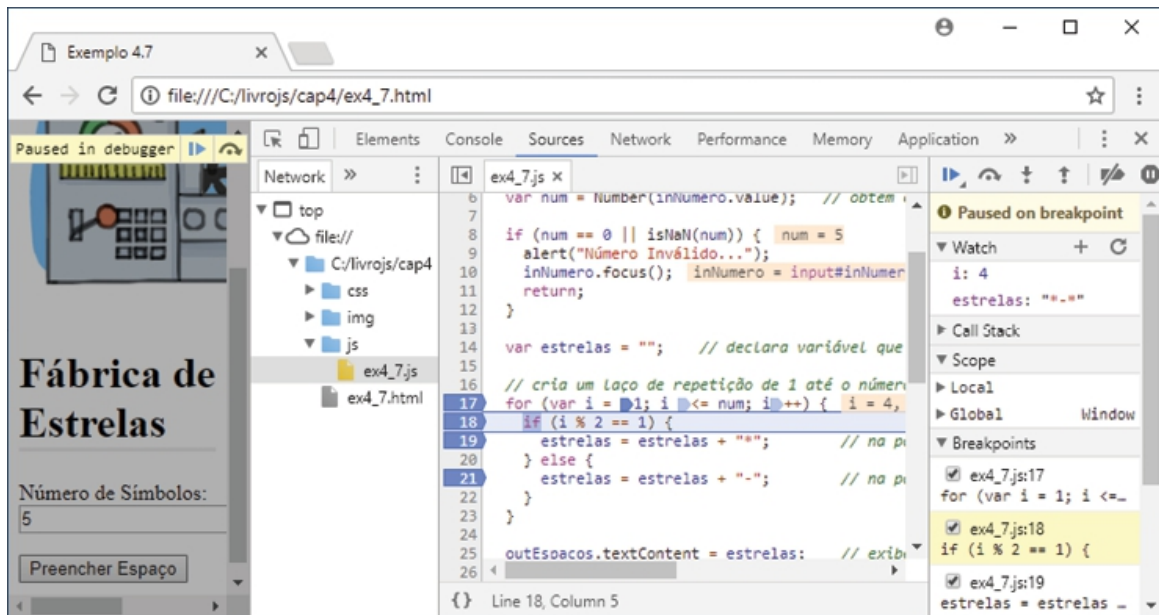


Figura 4.14 – Depurador do Chrome em ação.

4.7 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 dos conteúdos abordados neste capítulo, sugeridos a seguir.

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 4.15 ilustra a execução do programa.



Figura 4.15 – 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. Validar a entrada para que o número inicial de chinchilas seja maior ou igual a 2 (um casal). A Figura 4.16 exibe a página com um exemplo de saída do programa.



Figura 4.16 – 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 4.17 exibe a página do

programa com um exemplo de número perfeito.



Figura 4.17 – Programa Números Perfeitos deve utilizar estruturas de repetição e acumuladores.

d) Elaborar um programa que leia um número e exiba estrelas na página, em linhas diferentes. A cada nova linha, o número de estrelas deve ser incrementado. Observação: caso você informe um valor alto, as linhas podem ultrapassar o tamanho da imagem e se alinhar à margem esquerda da página. Para evitar que isso ocorra, crie uma nova classe no arquivo estilos.css, que defina uma flutuação à direita para essa imagem. A Figura 4.18 ilustra uma execução do programa desse exercício.



Figura 4.18 – As estrelas devem ser incrementadas a cada linha.

4.8 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 decremento 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

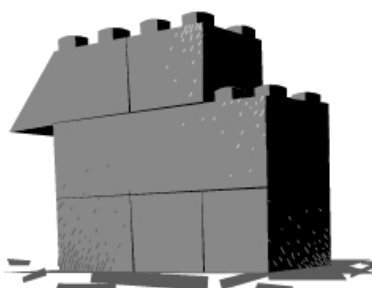
comando `for`). Esses comandos conferem ao programador maior controle do fluxo de execução dos comandos na repetição.

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 5

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 5.1 contém uma lista de itens de um supermercado armazenada no vetor produtos.

Tabela 5.1 – 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:

```
var produtos = [];  
var 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:

```
var produtos = ["Arroz", "Feijão", "Iogurte"];
```

Talvez você esteja se perguntando: no Exemplo 4.5, 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.

```
var 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.

```
var idade = [];  
idade[0] = 18;  
idade[1] = 15;
```

No Exemplo 5.1, 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 como exibir os elementos do vetor.

5.1 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 5.2.

Tabela 5.2 – 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 5.2.

```
<script>
var cidades = ["Pelotas"]; // declara e define conteúdo inicial do
vetor
// cidades[0] = "Pelotas"

cidades.push("São Lourenço"); // adiciona cidade ao final do vetor
// cidades[0] = "Pelotas"
// cidades[1] = "São Lourenço";

cidades.unshift("Porto Alegre"); // adiciona ao início e desloca as
demais
// cidades[0] = "Porto Alegre"
// cidades[1] = "Pelotas"
// cidades[2] = "São Lourenço";

var retirada = cidades.pop(); // remove a última cidade do vetor
// cidades[0] = "Porto Alegre"
// cidades[1] = "Pelotas"

retirada = cidades.shift(); // remove a primeira e "sobe" as demais
```

```
// cidades[0] = "Pelotas"  
</script>
```

Note, no código anterior, que as operações de exclusão de itens são atribuídas à variável `retirada`. Essa variável recebe o conteúdo do elemento removido do vetor.

Também é possível inserir elementos em uma posição específica do vetor. Não vamos realizar essa operação em nossos exemplos e exercícios. Caso necessite inserir elementos no meio do vetor, pesquise sobre o método JavaScript `splice()`. Com o método `splice()`, também é possível remover elementos do meio do vetor.

5.2 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`. Observe o script a seguir:

```
<script>  
  var cidades = ["Pelotas", "São Lourenço", "Porto Alegre"];  
  
  for (var i = 0; i < cidades.length; i++) {  
    alert(cidades[i]);  
  }  
</script>
```

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. Referenciar apenas o nome do vetor ou o nome seguido pelo método `join()` sem o caractere de separação gera uma saída idêntica à `toString()`. No script a seguir, os métodos `toString()` e `join()` são exemplificados:

```
<script>
```

```
var cidades = ["Pelotas", "São Lourenço", "Porto Alegre"];  
alert(cidades.toString() + "\n\n" + cidades.join(" - "));  
</script>
```

A Figura 5.1 exibe a saída do método `alert()` gerada por esse script.

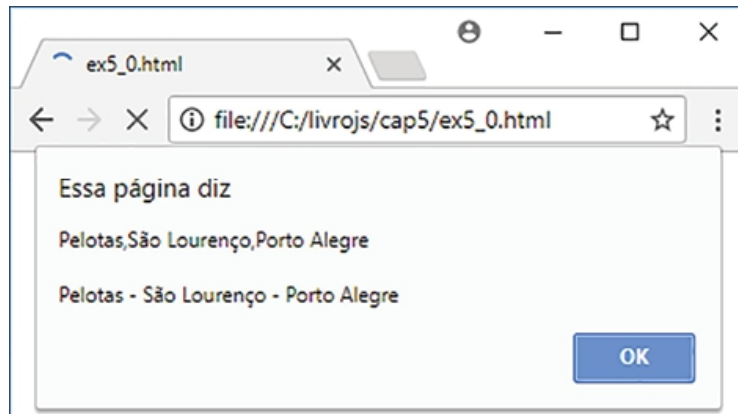


Figura 5.1 – Uso dos métodos `toString()` e `join()` em vetores.

Vamos criar um exemplo para ilustrar a aplicação dos métodos de inclusão e exclusão de elementos no 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 5.2 ilustra a página exibida pelo programa em execução.



Figura 5.2 – Um vetor gerencia a ordem dos atendimentos.

Como nos capítulos anteriores, crie uma nova pasta **cap5**, 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; }
.fonteAzul { color: blue; }
.oculta { display: none; }
.exibe { display: inline; }
.italico { font-style: italic; }
```

A classe fonteAzul será utilizada nesse primeiro exemplo para destacar os pacientes em atendimento. Vamos, então, criar a página HTML do Exemplo 5.1, descrita a seguir:

Exemplo 5.1 – Código HTML do programa Consultório Odontológico (ex5_1.html)

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 5.1</title>
</head>

<body>
  
  <h1>Consultório Odontológico</h1>
  <p>Paciente:
    <input type="text" id="inPaciente" autofocus>
    <input type="button" value="Adicionar" id="btAdicionar">
    <input type="button" value="Urgência" id="btUrgencia">
    <input type="button" value="Atender" id="btAtender">
  </p>
  <h3>Em Atendimento:
```



```

    <span id="outAtendimento" class="fonteAzul"></span>
</h3>
<pre id="outLista"></pre>
<script src="js/ex5_1.js"></script>
</body>

</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 (que foi adicionado no HTML 5), 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: ".

O programa JavaScript possui três **functions**: uma para realizar a inclusão dos pacientes na lista de espera, outra para inserir um atendimento de urgência e outra para chamar um paciente para o atendimento. A seguir, a descrição da primeira **function** que insere um novo paciente ao final do vetor. Crie o arquivo **ex5_1.js** dentro da pasta **js** e adicione nele essa primeira **function**.

Código JavaScript do programa Consultório Odontológico (js/ex5_1.js)

```

var pacientes = []; // declara vetor global

function adicionarPaciente() {
    // cria referência aos elementos de entrada e saída de dados da
    página
    var inPaciente = document.getElementById("inPaciente");
    var outLista = document.getElementById("outLista");

    var nome = inPaciente.value; // obtém nome do paciente

    // verifica preenchimento do nome do paciente
    if (nome == "") {
        alert("Informe o nome do paciente");
        inPaciente.focus();
    }
}

```

```

    return;
}

pacientes.push(nome); // adiciona o nome no final do vetor

var lista = ""; // string para concatenar pacientes

// percorre os elementos do vetor
for (i = 0; i < pacientes.length; i++) {
    lista += (i + 1) + ". " + pacientes[i] + "\n";
}

// altera o conteúdo da tag outLista
outLista.textContent = lista;

// limpa campo e posiciona cursor em inPaciente
inPaciente.value = "";
inPaciente.focus();
}

// cria referência ao btAdicionar e associa function ao evento click
var btAdicionar = document.getElementById("btAdicionar");
btAdicionar.addEventListener("click", adicionarPaciente);

```

No início do programa JavaScript, declaramos um vetor de escopo global, que será utilizado pelas três funções do programa. Na **function** **adicionarPaciente()**, começamos pelas instruções necessárias para referenciar os elementos da página, obter o nome do paciente e validar o seu preenchimento. Na sequência, o método **push()** é utilizado para inserir 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.

Outro detalhe acrescentado nesse programa é o uso do operador **+=** na atribuição da variável **lista** dentro do comando **for**. Esse operador de atribuição indica que a variável vai receber o próprio conteúdo da variável “+” um novo valor. Assim, as seguintes linhas são equivalentes.

```

lista = lista + pacientes[i];
lista += pacientes[i];

```

Já a segunda função desse programa deve inserir um atendimento de urgência, no qual o nome do paciente será inserido no início da fila. Você pode copiar e colar a primeira `function` no final do script e alterar o seu nome. A única instrução que precisa ser alterada é a linha que define o método de inserção do elemento no vetor. Portanto, substitua a linha contendo uma chamada ao método `push()` pela seguinte instrução:

```
pacientes.unshift(nome); // adiciona o nome no início do vetor
```

Em seguida, não se esqueça de referenciar o botão `btUrgencia` e criar um `addEventListener()` para o evento `click`, associando-o à nova `function`.

Falta agora a programação a ser executada no `click` do botão **Atender**. Ela está descrita a seguir.

```
function atenderPaciente() {  
    // verifica se vetor pacientes está vazio  
    if (pacientes.length == 0) {  
        alert("Não há pacientes na lista de espera");  
        inPaciente.focus();  
        return;  
    }  
  
    // cria referência aos elementos de saída de dados  
    var outAtendimento = document.getElementById("outAtendimento");  
    var outLista = document.getElementById("outLista");  
  
    // remove paciente do início da fila (e obtém nome)  
    var atender = pacientes.shift();  
    // exibe nome do paciente em atendimento  
    outAtendimento.textContent = atender;  
  
    // string para concatenar pacientes  
    var lista = "";  
  
    // percorre os elementos do vetor  
    for (i = 0; i < pacientes.length; i++) {  
        lista += (i + 1) + ". " + pacientes[i] + "\n";  
    }  
    // altera o conteúdo da tag outLista  
    outLista.textContent = lista;
```

```

}
var btAtender = document.getElementById("btAtender");
btAtender.addEventListener("click", atenderPaciente);

```

A function `atenderPaciente()` 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. A apresentação dos pacientes em espera contém as mesmas instruções realizadas nas funções anteriores.

5.3 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()` e `lastIndexOf()` 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 pelo método. Observe o exemplo a seguir, contendo um vetor com a idade de crianças, que ilustra o funcionamento desses métodos.

```

<script>
var idades = [5, 6, 8, 3, 6, 9];
alert(idades.indexOf(6)); // retorna 1
alert(idades.lastIndexOf(6)); // retorna 4
alert(idades.indexOf(7)); // retorna -1
</script>

```

Lembre-se de que o vetor começa na posição 0. Então, o “desenho” dos seus elementos é a representação ilustrada na Tabela 5.3.

Tabela 5.3 – Posição dos elementos no vetor idades

idades	
0	5
1	6

2	8
3	3
4	6
5	9

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, na última busca do exemplo anterior, um valor inexistente é passado. Assim, o valor -1 é retornado pelo método.

Vamos construir um segundo exemplo, em que o método `indexOf()` é utilizado. Trata-se do “Jogo Descubra o Número” ilustrado na Figura 5.3.



Figura 5.3 – 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 `indexOf()`. 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 5.2 – Código HTML do Programa Descubra o Número (ex5_2.html)

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 5.2</title>
</head>

<body>
  
  <h1>Descubra o Número</h1>
  <p>Número:
    <input type="text" id="inNumero" autofocus>
    <input type="button" value="Apostar" id="btApostar">
    <input type="button" value="Jogar Novamente" id="btJogar"
class="oculta">
  </p>
  <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/ex5_2.js"></script>
</body>
</html>
```

Código JavaScript do Programa Descubra o Número (js/ex5_2.js)

```
// declara vetor de escopo global que irá conter os números já
apostados
var erros = [];
```

```
// gera um número aleatório entre 1 e 100
var sorteado = Math.floor(Math.random() * 100) + 1;

// declara constante com o número de chances
const CHANCES = 6;

function apostarNumero() {
  // cria referência ao campo de entrada e obtém seu conteúdo
  var inNumero = document.getElementById("inNumero");
  var numero = Number(inNumero.value);

  // valida o número
  if (numero <= 0 || numero > 100 || isNaN(numero)) {
    alert("Informe um número válido...");
    inNumero.focus();
    return;
  }

  // referencia espaços das saídas de dados
  var outDica = document.getElementById("outDica");
  var outErros = document.getElementById("outErros");
  var outChances = document.getElementById("outChances");

  // se aposta do jogador for igual ao número sorteado
  if (numero == sorteado) {
    alert("Parabéns!! Você Acertou!!!");
    // troca status dos botões
    btApostar.disabled = true;
    btJogar.className = "exibe";
    outDica.textContent = "Parabéns!! Número sorteado: " + sorteado;
  } else {
    // se número existe no vetor erros
    if (erros.indexOf(numero) >= 0) {
      alert("Você já apostou o número " + numero + ". Tente outro...");
    } else {
      erros.push(numero); // adiciona número ao vetor
      var numErros = erros.length; // obtém tamanho do vetor
      var numChances = CHANCES - numErros; // calcula nº de chances
    }
  }
}
```

```

// exibe nº de erros, conteúdo do vetor e nº de chances
outErros.textContent = numErros + " (" + erros.join(", ") + ")";
outChances.textContent = numChances;
if (numChances == 0) {
    alert("Suas chances acabaram...");
    btApostar.disabled = true;
    btJogar.className = "exibe";
    outDica.textContent = "Game Over!! Número Sorteado: " +
sorteado;
} else {
    // usa operador ternário (condicional) para mensagem da dica
    var dica = numero < sorteado ? "maior" : "menor";
    outDica.textContent = "Dica: Tente um número " + dica + " que "
+ numero;
}
}
}
// Limpa campo de entrada e posiciona cursor neste campo
inNumero.value = "";
inNumero.focus();
}
var btApostar = document.getElementById("btApostar");
btApostar.addEventListener("click", apostarNumero);

```

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 da função **apostarNumero()**, 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.

A função `apostarNumero()` começa pela obtenção e validação do número apostado pelo jogador. Como o número deve estar no intervalo de 1 a 100, acrescentamos um teste para evitar que o jogador informe um número superior a 100. Em seguida, recuperamos os elementos da página que exibem as mensagens do programa.

Vamos aos testes do programa. Nosso primeiro teste verifica se o jogador acertou o número. Caso ok, as mensagens de parabéns são exibidas e os botões trocam de status. Caso o número apostado não corresponda ao sorteado, utilizamos o método `indexOf()` discutido nesta seção. Se o retorno do `indexOf()` for maior ou igual a zero, significa que o número apostado já consta no vetor e a mensagem de alerta é exibida – conforme ilustra a Figura 5.4.

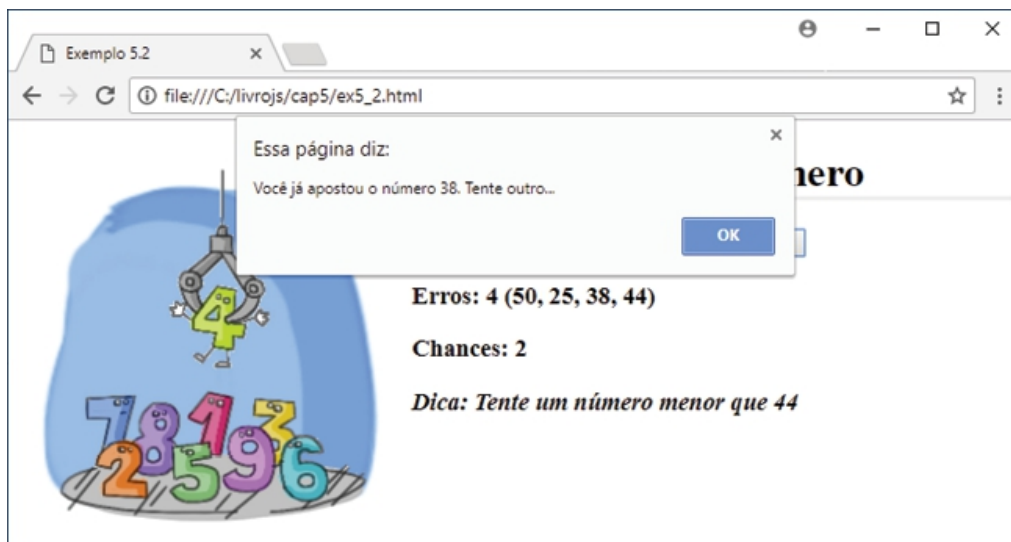


Figura 5.4 – O método `indexOf()` é executado para verificar se um número já consta no vetor erros.

A sequência do programa trata de um erro “válido” do jogador. Nesse caso, o número apostado é inserido no vetor, o número de erros é obtido a partir da propriedade `length` e o número de chances é calculado. Esses valores são exibidos na página. Caso o número de chances seja igual a 0, as instruções de final de jogo são executadas. Após a exibição da caixa de alerta, os botões trocam de status – conforme ilustra a Figura 5.5. Isso impede que o jogador continue apostando outros números mesmo com as chances zeradas. Apenas o botão “Jogar Novamente” permanece disponível para o usuário.

A programação do 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.

```
function jogarNovamente() {  
    location.reload(); // recarrega a página  
}  
var btJogar = document.getElementById("btJogar");  
btJogar.addEventListener("click", jogarNovamente);
```



Figura 5.5 – Troca de status dos botões impedem novas apostas.

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!

5.4 Vetores de objetos

Um vetor pode conter uma lista de nomes, como no Exemplo 5.1, ou de números, como no Exemplo 5.2. 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>
  var carros = [];
  carros.push({ modelo: "Fusca", preco: 6500 });
  carros.push({ modelo: "Escort", preco: 7800 });

  for (var i = 0; i < carros.length; i++) {
    alert(carros[i].modelo + " - R$: " + carros[i].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 valor que será atribuído a ele. Na sequência do script, utilizamos o comando `for` para percorrer os elementos do vetor e apresentar o conteúdo de cada um dos seus atributos.

Os próximos dois exemplos deste capítulo vão manipular vetores de objetos. Antes desses exemplos, vamos discutir sobre dois processos importantes que podem ser realizados sobre listas, que são: pesquisa de dados e classificação dos elementos do vetor.

5.5 Pesquisar e filtrar dados

Depois de possuímos 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 videolocadora e há um campo para pesquisar pelo título do filme. Você escreve uma palavra-chave e fica aguardando a lista dos filmes. Caso o sistema não localize filmes 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>
  var idades = [12, 20, 15, 17, 14];
  for (var i = 0; i < idades.length; i++) {
    if (idades[i] >= 18) {
      alert(idades[i]);
    }
  }
</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. 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>
  var idades = [12, 16, 15, 17, 14];

  var maiores = false;

  for (var i = 0; i < idades.length; i++) {
    if (idades[i] >= 18) {
      alert(idades[i]);
      maiores = true;
    }
  }

  if (!maiores) {
    alert("Não há idades maiores que 18 na lista");
  }
</script>
```

Também poderíamos atribuir 0 (valor inicial) e 1 (dentro do laço) para a variável `maiores`. Vamos construir um exemplo para explorar esse recurso de filtragem de dados em uma lista. Nosso programa é para a “Revenda Herbie”, que vai armazenar em um vetor de objetos o modelo e o preço dos carros disponíveis na revenda. Um botão de “**Filtrar por Preço**” vai solicitar o valor máximo de que um determinado cliente dispõe para a compra do veículo. A página do programa, já com alguns veículos listados, é exibida na Figura 5.6.

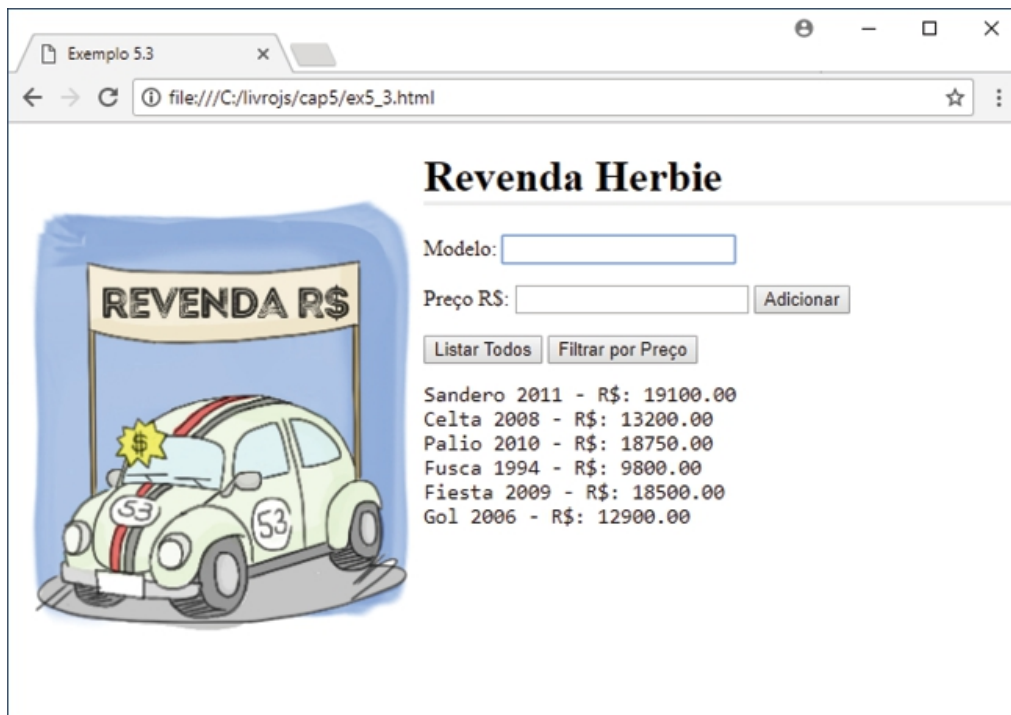


Figura 5.6 – Um vetor de objetos armazena modelo e preço dos veículos.

Exemplo 5.3 – Código HTML do programa Revenda Herbie (ex5_3.html)

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 5.3</title>
</head>

<body>
  
  <h1>Revenda Herbie</h1>
  <p>Modelo:
    <input type="text" id="inModelo" autofocus> </p>
  <p>Preço R$:
    <input type="text" id="inPreco">
    <input type="button" value="Adicionar" id="btAdicionar">
```

```

</p>
<input type="button" value="Listar Todos" id="btListar">
<input type="button" value="Filtrar por Preço" id="btFiltrar">
<pre id="outLista"></pre>
<script src="js/ex5_3.js"></script>
</body>
</html>

```

O programa JavaScript apresentado a seguir contém três funções: adicionar um carro no vetor de objetos, listar os carros e filtrar por preço. Observe que, no final do `adicionarCarros()`, há uma chamada para a `function` `listarCarros()`. Essa `function` também é acionada quando o evento `click` do botão **Listar Carros** ocorre. Assim, já estamos modularizando nosso programa, ou seja, fazendo com que o programa seja dividido em blocos menores (módulos). Para chamar uma `function` a partir de outra, basta colocar o nome da `function` no local onde ela deve ser executada. Esse assunto será discutido com mais detalhes no Capítulo 7.

Código JavaScript do programa Revenda Herbie (js/ex5_3.js)

```

var carros = []; // declara vetor global

function adicionarCarros() {
    // Cria referência aos elementos contendo os dados de entrada
    var inModelo = document.getElementById("inModelo");
    var inPreco = document.getElementById("inPreco");

    var modelo = inModelo.value; // obtém conteúdo dos campos
    var preco = Number(inPreco.value);

    // verifica preenchimento dos campos
    if (modelo == "" || preco == 0 || isNaN(preco)) {
        alert("Informe corretamente os dados");
        inModelo.focus();
        return;
    }
    // adiciona dados ao vetor de objetos
    carros.push({ modelo: modelo, preco: preco });

    // limpa campos e posiciona cursor em inModelo

```

```
inModelo.value = "";
inPreco.value = "";
inModelo.focus();
```

```
    listarCarros(); // chama function que lista os carros
}
// cria referência ao btAdicionar e associa function ao evento click
deste botão
var btAdicionar = document.getElementById("btAdicionar");
btAdicionar.addEventListener("click", adicionarCarros);
```

A função `listarCarros` percorre o vetor a partir de um laço de repetição e apresenta o conteúdo dos dois atributos (modelo e preço) do vetor de objetos carros. Observe que nessa `function` foi utilizada a propriedade `textContent` diretamente na instrução que referencia o elemento a ter o conteúdo desse atributo modificado (`document.getElementById("outLista").textContent = lista;`).

```
function listarCarros() {
    // verifica se vetor está vazio
    if (carros.length == 0) {
        alert("Não há carros na lista");
        return;
    }

    var lista = ""; // para concatenar lista de carros

    // percorre os elementos do vetor
    for (var i = 0; i < carros.length; i++) {
        // adiciona à lista, cada objeto do vetor
        lista += carros[i].modelo + " - R$: " + carros[i].preco.toFixed(2)
+ "\n";
    }

    // referencia elemento e altera conteúdo exibido
    document.getElementById("outLista").textContent = lista;
}
var btListar = document.getElementById("btListar");
btListar.addEventListener("click", listarCarros);
```

Já a função `filtrarCarros()` solicita inicialmente o valor a ser utilizado

como filtro nos dados da lista, conforme ilustra a Figura 5.7. No laço de repetição, uma condição verifica se cada veículo do vetor possui preço inferior ou igual ao informado pelo usuário. Os veículos que obedecem ao filtro são selecionados e exibidos (Figura 5.8).

```
function filtrarCarros() {
    // faz a leitura do valor máximo a partir do método prompt
    var maximo = Number(prompt("Qual o valor máximo que o cliente
deseja pagar?"));

    // se não preencheu ou conteúdo inválido ...
    if (maximo == 0 || isNaN(maximo)) {
        return; // ... retorna
    }

    // para concatenar lista de carros que obedecem ao critério de
pesquisa / filtro
    var lista = "";

    // percorre todos os elementos do vetor
    for (var i = 0; i < carros.length; i++) {
        // verifica se o preço é inferior (ou igual) ao máximo
        if (carros[i].preco <= maximo) {
            lista += carros[i].modelo + " - R$: " +
carros[i].preco.toFixed(2) + "\n";
        }
    }

    var outLista = document.getElementById("outLista"); // cria
referência a outLista

    // se a lista esta vazia, significa que nenhum veículo foi
encontrado (no for)
    if (lista == "") {
        outLista.textContent = "Não há carros com preço até R$ " +
maximo.toFixed(2);
    } else {
        // senão, mostra os veículos obtidos
        outLista.textContent = "Carros até R$ " + maximo.toFixed(2) +
"\n-----\n" + lista;
    }
}
```

```

    }
}
var btFiltrar = document.getElementById("btFiltrar");
btFiltrar.addEventListener("click", filtrarCarros);

```

Observe que, nesse programa, aproveitamos a variável `lista` para sinalizar a existência de veículos com preço inferior ao informado. Ela vai conter a lista de veículos ou manterá o valor inicial (vazio), realizando, portanto, a tarefa da flag, anteriormente discutida.



Figura 5.7 – O programa solicita o valor limite, para então realizar o filtro.



Figura 5.8 – Apenas os veículos que obedecem ao critério de pesquisa são exibidos.

5.6 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 uma função os dados ordenados, é possível criar uma cópia do vetor original a partir do método `slice()`. Esse método – que significa fatia, pedaço ou porção em inglês – obtém um conjunto de elementos de um outro vetor. Sem parâmetros, ele obtém uma cópia com todos os elementos do vetor original. Observação: cuidado para não confundir os métodos `slice()` e `splice()` – comentado no final da seção 5.1.

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>
  var nomes = ["Pedro", "Ana", "João"];

  nomes.sort();
  alert(nomes.join(", ")); // Ana, João, Pedro

  nomes.reverse();
  alert(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 7. 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>
  var numeros = [50, 100, 2];

  numeros.sort();
  alert(numeros.join(", ")); // 100, 2, 50
  numeros.sort(function (a, b) { return a - b });
  alert(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 5.9 exibe a página com uma lista de crianças inseridas no vetor.



Figura 5.9 – Os dados inseridos serão ordenados por idade para obter-se o resumo da lista.

Exemplo 5.4 – Código HTML do programa Brinquedoteca (ex5_4.html)

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 5.4</title>
</head>

<body>
  
  <h1>Programa Brinquedoteca</h1>
```

```

<p>Nome da Criança:
  <input type="text" id="inNome" autofocus>
</p>
<p>Idade:
  <input type="text" id="inIdade">
  <input type="button" value="Adicionar" id="btAdicionar">
</p>
<p>
  <input type="button" value="Listar Todos" id="btListar">
  <input type="button" value="Resumir por Idade" id="btResumir">
</p>
<pre id="outLista"></pre>
<script src="js/ex5_4.js"></script>
</body>

</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 `function resumirLista()`. Vamos, inicialmente, ver o código das duas primeiras funções do programa `ex5_4.js` destacadas a seguir.

Código JavaScript do programa Brinquedoteca (js/ex5_4.js)

```

var criancas = []; // declara vetor global

function adicionarCrianças() {
  // cria referência aos elementos (campos) de entrada de dados
  var inNome = document.getElementById("inNome");
  var inIdade = document.getElementById("inIdade");

  var nome = inNome.value; // obtém conteúdo dos campos
  var idade = Number(inIdade.value);
  // verifica preenchimento dos campos
  if (nome == "" || inIdade.value == "" || isNaN(idade)) {
    alert("Informe corretamente os dados");
    inNome.focus();
    return;
  }
}

```

```

}

// adiciona dados ao vetor de objetos
criancas.push({ nome: nome, idade: idade });

// limpa campos e posiciona cursor em inNome
inNome.value = "";
inIdade.value = "";
inNome.focus();

listarCrianças(); // chama function que lista as crianças
}
// cria referência ao btAdicionar e associa function ao evento click
deste botão
var btAdicionar = document.getElementById("btAdicionar");
btAdicionar.addEventListener("click", adicionarCrianças);

function listarCrianças() {
    // verifica se vetor está vazio
    if (criancas.length == 0) {
        alert("Não há crianças na lista");
        return;
    }

    var lista = ""; // para concatenar lista de crianças

    // percorre os elementos do vetor
    for (var i = 0; i < criancas.length; i++) {
        lista += criancas[i].nome + " - " + criancas[i].idade + " anos\n";
    }

    // exibe a lista (em uma única instrução)
    document.getElementById("outLista").textContent = lista;
}
var btListar = document.getElementById("btListar");
btListar.addEventListener("click", listarCrianças);

```

Já a function `resumirLista()` implementa os recursos apresentados nessa seção. Ela deve produzir uma saída conforme a ilustração da Figura 5.10. Analise com atenção o código dessa function, apresentado na sequência.

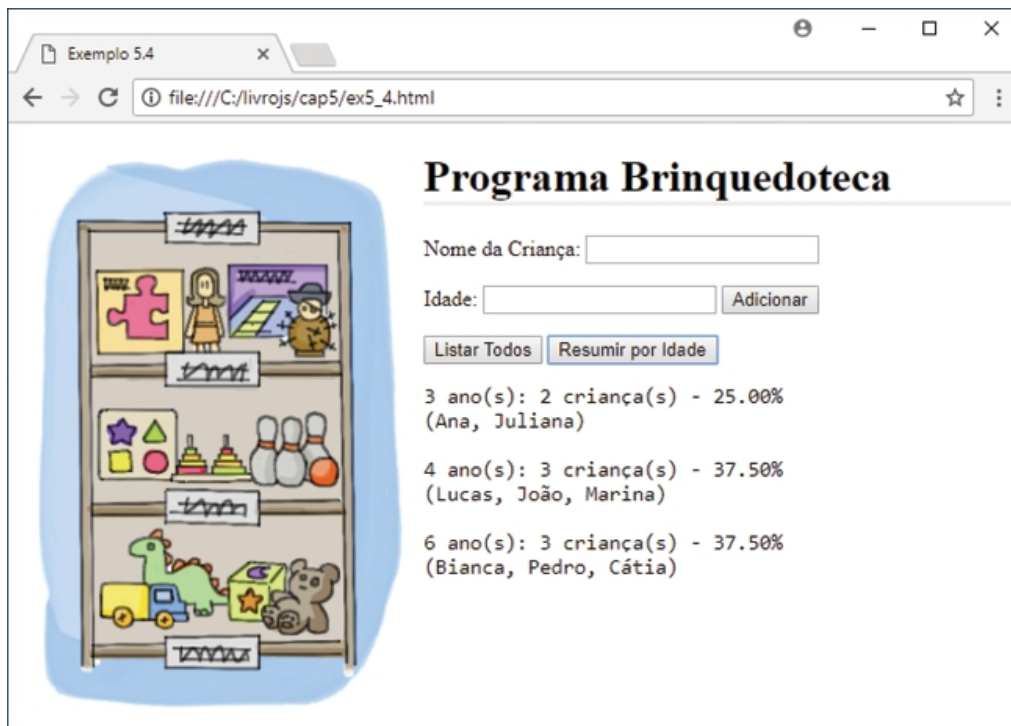


Figura 5.10 – Para se obterem os dados de saída, foi utilizado o método `sort()`.

```
function resumirLista() {
    // verifica se vetor está vazio
    if (criancas.length == 0) {
        alert("Não há crianças na lista");
        return;
    }

    // cria uma cópia do vetor crianças
    var copia = criancas.slice();

    // ordena o vetor copia pela idade
    copia.sort(function (a, b) { return a.idade - b.idade });

    var resumo = ""; // para concatenar saída

    var aux = copia[0].idade; // menor idade do vetor ordenado
    var nomes = []; // vetor para inserir nomes de cada idade

    // percorre os elementos do vetor (classificado por idade)
    for (var i = 0; i < copia.length; i++) {
        // se é da mesma idade auxiliar, adiciona ao vetor
        if (copia[i].idade == aux) {
```



```

        nomes.push(copia[i].nome);
    } else {
        // senão, adiciona ao resumo, dados e nomes inseridos em nomes[]
        resumo += aux + " ano(s): " + nomes.length + " criança(s) - ";
        resumo += (nomes.length / copia.length * 100).toFixed(2) +
"%\n";
        resumo += "(" + nomes.join(", ") + ")\n\n";
        aux = copia[i].idade; // obtém a nova idade na ordem
        nomes = []; // limpa o vetor dos nomes
        nomes.push(copia[i].nome); // adiciona o primeiro da nova idade
    }
}
// adiciona os nomes da última idade ordenada
resumo += aux + " ano(s): " + nomes.length + " criança(s) - ";
resumo += (nomes.length / copia.length * 100).toFixed(2) + "%\n";
resumo += "(" + nomes.join(", ") + ")\n\n";

// altera conteúdo de outLista
document.getElementById("outLista").textContent = resumo;
}
var btResumir = document.getElementById("btResumir");
btResumir.addEventListener("click", resumirLista);

```

Para resolver um problema diferente, como o do Exemplo 5.4, 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 5.9, nossa lista apresenta o desenho conforme ilustra a Tabela 5.4:

Tabela 5.4 – 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!

5.7 Exercícios

Desenvolva os seguintes programas que manipulam listas de dados a serem armazenados em vetores.

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. As

figuras 5.11 e 5.12 ilustram as páginas do programa.



Figura 5.11 – Página exibida após a inclusão de um clube e ao clicar no botão “Listar Clubes”.



Figura 5.12 – 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 5.13 demonstra um exemplo de execução do programa.



Figura 5.13 – 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 2ª Fase**, ler o número de acertos para aprovação dos candidatos para a 2ª fase do concurso, conforme ilustra a Figura 5.14. 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 5.15). Caso nenhum candidato tenha sido aprovado, exibir mensagem.



Figura 5.14 – Ao clicar no botão “Aprovados 2ª Etapa”, ler n° de acertos para aprovação.

Exercício 5.c

file:///C:/livrojs/cap5/resp5_c.html

Programa Concurso

Candidato:

Nº Acertos:

Eduardo - 36 acertos
André - 35 acertos
Débora - 33 acertos
Giovana - 31 acertos

Figura 5.15 – Listar candidatos aprovados em ordem decrescente de n° de acertos.

d) Elaborar um programa que leia as últimas notícias da sua região e armazene-as em um vetor. Exibir na página, a cada inclusão, somente a quantidade de notícias cadastradas. O programa deve conter um botão que solicite o número de notícias que o usuário deseja visualizar e, então, liste as notícias em ordem inversa da inclusão. Ou seja, apenas as notícias mais recentes devem ser listadas, de acordo com o número informado pelo usuário. Validar esse número para que seja inferior ou igual à quantidade de notícias cadastradas. A Figura 5.16 ilustra a página do exercício com um exemplo em que o usuário solicitou as 3 últimas notícias.

Exercício 5.d

file:///C:/livrojs/cap5/resp5_d.html

Programa Últimas Notícias

Noticia:

Notícias Cadastradas: 10

3 Últimas Notícias

10ª) Semana deve ser de calor na cidade de Pelotas
9ª) Praia de São Lourenço esteve movimentada no final de semana
8ª) Pelotas vence mais uma e agora é líder do Gauchão

Figura 5.16 – Página deve exibir as últimas notícias de acordo com o número informado

pelo usuário.

5.8 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:

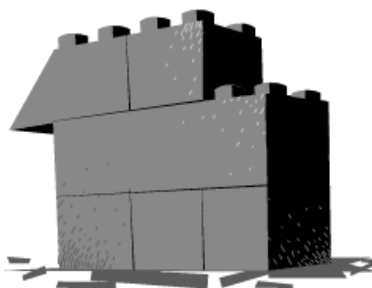
- **Inserir e remover elementos no início e no final do vetor** – Para a inserção, utilize os métodos `unshift()` e `push()`, que inserem, respectivamente, um novo elemento no início e no final do vetor. Já a exclusão pode ser realizada com os métodos `shift()` e `pop()`, para remover o primeiro e o 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. O método `join()` também é frequentemente utilizado para consultar a 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 do método `indexOf()`, que retorna a posição da primeira ocorrência do conteúdo pesquisado no vetor, e do método `lastIndexOf()`, que realiza a pesquisa do final para o início, também retornando a posição do elemento encontrado. Caso o conteúdo não exista no vetor, o valor `-1` é retornado em ambos os métodos.
- **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.

CAPÍTULO 6

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 para, por exemplo, 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()` e `lastIndexOf()`, 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.

6.1 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 6.1.

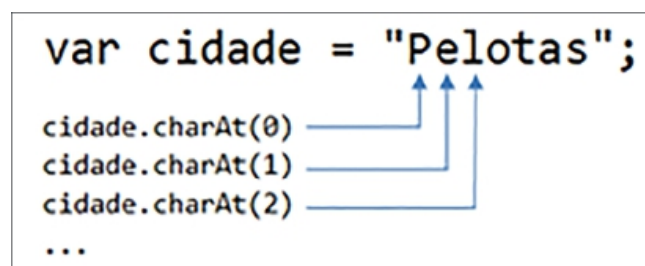


Figura 6.1 – Método `charAt()` obtém cada um dos caracteres de uma string.

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 (**cap6**) e, dentro dela, as pastas **css**, **img** e **js**. Em seguida, crie o arquivo **ex6_1.html**. Para alguns exemplos simples deste capítulo vamos trabalhar com os métodos `alert()` e `prompt()`.

Exemplo 6.1 – Programa que lê um anúncio e informa número de palavras (ex6_1.html)

```
<script>
  var anuncio = prompt("Anúncio: "); // lê o anúncio
  var tam = anuncio.length; // obtém o tamanho
  var numPalavras = 0; // inicializa contador
  for (var 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 6.2 exibe o `alert()` desse script, com um exemplo de anúncio informado pelo usuário.

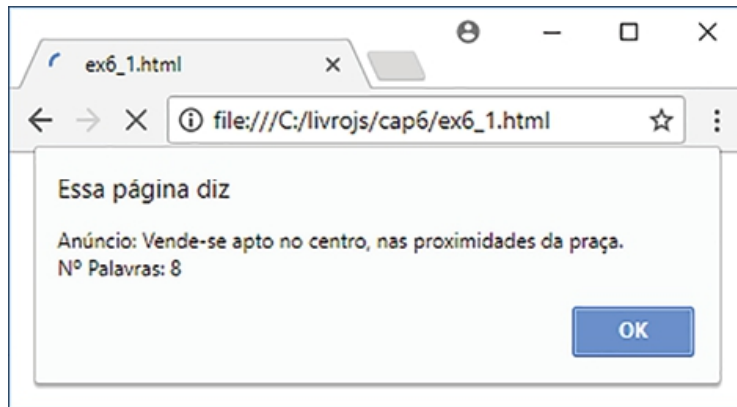


Figura 6.2 – Uso do método `charAt()` para percorrer os caracteres de uma string.

Vamos construir outro exemplo que explora o método `charAt()`. O programa “Qual é a fruta?” deve ler uma palavra (sugere-se uma fruta) e exibir, após o clique no botão **Montar 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 6.3. Esse jogo completo será desenvolvido no Capítulo 10. Por enquanto, vamos apenas entender como montar a tela inicial do jogo.



Figura 6.3 – 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; }
.entreLetras { letter-spacing: 0.5em; }
.aDireita { text-align: right; }
```

Em seguida, crie os arquivos `ex6_2.html` e `ex6_2.js` (na pasta `js`) com os códigos descritos a seguir.

Exemplo 6.2 – Código HTML do Programa “Qual é a fruta?” (`ex6_2.html`)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 6.2</title>
</head>

<body>
  
  <h1> Programa Qual é a Fruta? </h1>
  <p> Fruta:
    <input type="text" id="inFruta">
    <input type="button" value="Montar Dica" id="btMontar">
  </p>
  <h3> Dica:
    <span id="outDica" class="entreLetras"></span>
  </h3>
  <script src="js/ex6_2.js"></script>
</body>

</html>
```

Código JavaScript do programa Qual é a fruta? (`js/ex6_2.js`)

```
function montarDica() {
  // cria referência aos elementos da página
```

```

var inFruta = document.getElementById("inFruta");
var outDica = document.getElementById("outDica");

// obtém conteúdo do campo de entrada
var fruta = inFruta.value;

// se vazio, exibe alerta, posiciona em inFruta e retorna
if (fruta == "") {
    alert("Informe a fruta...");
    inFruta.focus();
    return;
}

var resposta = fruta.charAt(0); // letra inicial da fruta
var estrelas = "*"; // vai concatenar *
var tam = fruta.length; // obtém tamanho da fruta

// percorre os demais caracteres da fruta
for (var i = 1; i < tam; i++) {
    // se a letra da posição da variável de controle for igual à
    primeira...
    if (fruta.charAt(i) == fruta.charAt(0)) {
        resposta += fruta.charAt(0); // adiciona a letra inicial
    } else { // senão,
        resposta += "_"; // adiciona o sublinhado
    }
    estrelas += "*"; // concatena *
}

// exibe a resposta e exibe estrelas (*) em inFruta
outDica.textContent = resposta;
inFruta.value = estrelas;
}

// cria referência ao botão e após associa function ao evento click
var btMontar = document.getElementById("btMontar");
btMontar.addEventListener("click", montarDica);

```

Nosso programa segue o padrão dos exemplos dos capítulos anteriores. Há uma função que será chamada quando o evento `click` do botão ocorrer. Na função, criamos referência aos elementos que serão

manipulados por ela e, em seguida, obtemos a palavra informada pelo usuário. 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 o caractere da posição da variável de controle (*i*) 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, inicializa-se uma variável (*estrelas*) e dentro da repetição acrescenta-se “*” nessa variável em cada volta do laço de repetição. Ao final, modifica-se a propriedade *value* desse elemento.

6.2 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 ...
    var 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()*. No Exemplo 6.2, que verifica se a letra inicial consta em outras posições da palavra, também é importante

adicionar o método `toUpperCase()`, pois, se o usuário informar a letra inicial maiúscula e as demais, minúsculas, o programa não funciona corretamente. Assim, substitua o comando `var fruta = inFruta.value` pelo comando:

```
var fruta = inFruta.value.toUpperCase();
```

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 6.4 ilustra uma execução desse programa.

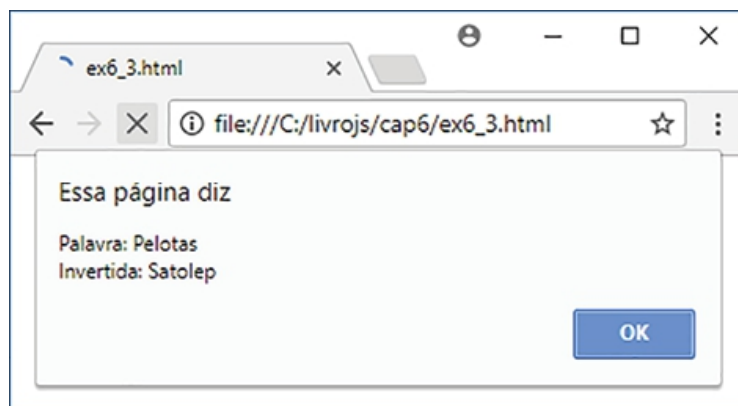


Figura 6.4 – Na inversão, apenas a letra inicial deve ficar em caixa alta.

Exemplo 6.3 – Programa Inverte Palavra (ex6_3.html)

```
<script>
var palavra = prompt("Palavra: "); // lê a palavra
var tam = palavra.length; // obtém o tamanho

// inverso inicia com a última letra da palavra em caixa alta
var inverso = palavra.charAt(tam-1).toUpperCase();
// for decrescente percorre as demais letras e ...
for (var 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 6.3, 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.

6.3 Cópia de caracteres 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>
var palavra = "saladas";
var copia1 = palavra.substr(2); // obtém "ladas"
var copia2 = palavra.substr(2, 4); // obtém "lada"
var copia3 = palavra.substr(0, palavra.length-1); // obtém "salada"
var 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 4.

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:

```
var ultima1 = palavra.substr(-1);
var 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()`.

6.4 Localizar um ou mais caracteres na string

Estão lembrados dos métodos `indexOf()` e `lastIndexOf()` 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. Confira os exemplos a seguir:

```
<script>
var palavra = "saladas";
var posicao1 = palavra.indexOf("a"); // retorna 1
var posicao2 = palavra.lastIndexOf("a"); // retorna 5
var posicao3 = palavra.indexOf("sal"); // retorna 0
var posicao4 = palavra.indexOf("e"); // retorna -1
</script>
```

Assim como nos vetores, a pesquisa retorna 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 6.5 exemplifica uma entrada e saída desse programa.



Figura 6.5 – Programa “Nome no Crachá” utiliza os métodos discutidos nas seções 6.3 e 6.4.

Exemplo 6.4 – Código HTML do Programa Nome no Crachá (ex6_4.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 6.4</title>
</head>

<body>
  
  <h1>Programa Nome no Crachá</h1>
  <p>Nome do Participante:
    <input type="text" id="inNome">
  </p>
  <input type="button" value="Gerar Crachá" id="btGerar">
  <h3 id="outCracha"></h3>
  <script src="js/ex6_4.js"></script>
</body>
```

</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, na validação, é adicionada uma condição para verificar se o nome não possui espaço, ou seja, está incompleto.

Programa JavaScript que exibe primeiro e último nome (js/ex6_4.js)

```
function gerarCracha() {  
    // cria referência aos elementos da página  
    var inNome = document.getElementById("inNome");  
    var outCracha = document.getElementById("outCracha");  
  
    // obtém o conteúdo do campo de entrada  
    var nome = inNome.value;  
  
    // se vazio ou se não há espaço no nome  
    if (nome == "" || nome.indexOf(" ") == -1) {  
        alert("Informe o nome completo do participante...");  
        inNome.focus();  
        return;  
    }  
  
    var priEspaco = nome.indexOf(" "); // posição do primeiro espaço  
    var ultEspaco = nome.lastIndexOf(" "); // posição do último espaço  
  
    // copia nome e sobrenome usando os parâmetros do substr()  
    var cracha = nome.substr(0, priEspaco) + nome.substr(ultEspaco);  
  
    // altera o texto identificado na página por outCracha  
    outCracha.textContent = "Crachá: " + cracha;  
}  
// cria referência ao botão e após associa function ao evento click  
var btGerar = document.getElementById("btGerar");  
btGerar.addEventListener("click", gerarCracha);
```

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.

6.5 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()`.

```
var sabores = "calabresa, 4 queijos, atum, frango";  
var 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 6.6 ilustra um exemplo de execução desse programa.



Figura 6.6 – Programa gera um e-mail com as iniciais do nome e o sobrenome do funcionário.

Exemplo 6.5 – Código HTML do Programa E-mail Institucional (ex6_5.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 6.5</title>
</head>

<body>
  
  <h1>Programa E-mail Institucional</h1>
  <p>Funcionário:
    <input type="text" id="inFuncionario">
  </p>
  <input type="button" value="Gerar E-mail do Funcionário"
id="btGerar">
  <h3 id="outEmail"></h3>
  <script src="js/ex6_5.js"></script>
```

```
</body>
</html>
```

Programa JavaScript que gera e-mail com iniciais e sobrenome (js/ex6_5.js)

```
function gerarEmail() {
    // cria referência aos elementos da página
    var inFuncionario = document.getElementById("inFuncionario");
    var outEmail = document.getElementById("outEmail");

    // obtém conteúdo do campo de entrada
    var funcionario = inFuncionario.value;

    // se vazio ou se não há espaço no nome do funcionario
    if (funcionario == "" || funcionario.indexOf(" ") == -1) {
        alert("Informe o nome completo do funcionário...");
        inFuncionario.focus();
        return;
    }

    // divide o nome em itens de vetor, criados a cada ocorrência do
    // espaço
    var partes = funcionario.split(" ");
    var email = ""; // vai concatenar letras
    var tam = partes.length; // obtém nº itens do vetor partes

    // percorre os itens do vetor (exceto o último) ...
    for (var i = 0; i < tam - 1; i++) {
        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";

    // exibe e-mail em letras minúsculas
    outEmail.textContent = "E-mail: " + email.toLowerCase();
}

// cria referência ao botão e após associa function ao evento click
var btGerar = document.getElementById("btGerar");
btGerar.addEventListener("click", gerarEmail);
```

O programa `ex6_5.js` obtém o nome do funcionário e, depois de realizar a validação do preenchimento do nome, 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.

6.6 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:

```
var 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>
var palavra = "#Senha_123!";
var vetor1 = palavra.match(/[a-z]/g); // e,n,h
var vetor2 = palavra.match(/[A-Z]/g); // S,A
var vetor3 = palavra.match(/[0-9]/g); // 1,2,3
var vetor4 = palavra.match(/\W|_/g); // #,_,!
var 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. As figuras 6.7 e 6.8 ilustram o funcionamento desse programa.

Exemplo 6.6 – Código HTML do Programa Valida Senha (ex6_6.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 6.6</title>
</head>

<body>
  
  <h1> Programa Valida Senha </h1>
```



```

<p> Senha:
  <input type="text" id="inSenha">
</p>
<input type="button" value="Verificar Validade da Senha"
id="btVerificar">
  <h3 id="outResposta"></h3>
  <script src="js/ex6_6.js"></script>
</body>

</html>

```

A página HTML contém um campo `<input type="text" ...>` que permite visualizar o que é digitado. Você pode substituir `text` por `password` nesse comando para ofuscar os caracteres das senhas. O browser exibe um círculo (Chrome, Firefox, Safari e Edge) ou um asterisco (Opera) para cada caractere digitado.

Programa JavaScript de validação da senha (js/ex6_6.js)

```

function verSenha() {
  // cria referência aos elementos da página
  var inSenha = document.getElementById("inSenha");
  var outResposta = document.getElementById("outResposta");

  // obtém o conteúdo do campo de entrada
  var senha = inSenha.value;

  var 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(/[0-9]/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) {
    outResposta.textContent = "Ok! Senha Válida"
} else {
    outResposta.textContent = "Erro... A senha deve " + erros.join(",
");
}
}

// cria referência ao botão e após associa function ao evento click
var btVerificar = document.getElementById("btVerificar");
btVerificar.addEventListener("click", verSenha);

```

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 6.7 – Exemplo com mensagens de erro.



Figura 6.8 – Exemplo de senha válida.

6.7 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:

```
var novaStr = str.replace(caracterePesquisado, novoCaractere);
```

Portanto, se a variável `senha` tiver o conteúdo indicado a seguir:

```
var senha = "ABACAD";
```

A execução do método `replace()` sobre essa variável vai produzir os seguintes novos conteúdos:

```
var senha1 = senha.replace("A", "X"); // XBACAD  
var 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:

```
<script>  
var app = "App Controle Financeiro";  
var app2 = app.replace(" ", ""); // AppControle Financeiro  
var app3 = app.replace(/ /g, ""); // AppControleFinanceiro  
var app4 = app.replace(/ /g, "").toLowerCase(); //  
appcontrolefinanceiro  
</script>
```

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 6.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.

6.8 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:

```
var 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 construir pequenos scripts para exemplificar a aplicação desses métodos. Começamos pelo script `ex6_7.html` destacado a seguir.

Exemplo 6.7 – Programa exibe a data atual e seguinte (ex6_7.html)

```
<meta charset="utf-8">
<script>
  var hoje = new Date();
  var amanha = new Date();

  var dia = amanha.getDate();
  amanha.setDate(dia + 1);

  alert("Hoje: " + hoje + "\nAmanhã: " + amanha);
</script>
```

Observe na Figura 6.9 que o programa 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.

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.

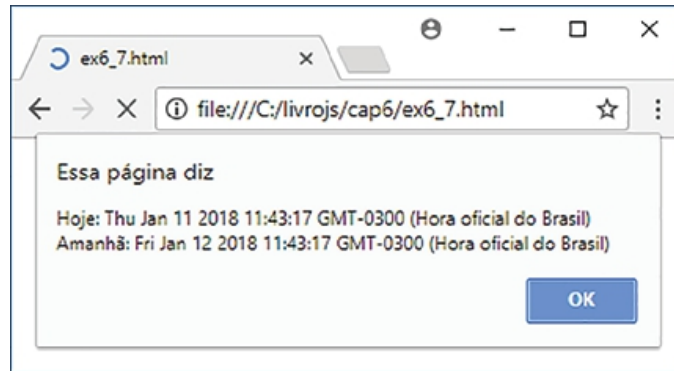


Figura 6.9 – Data atual e adição de mais um dia à data.

Com os métodos `toDateString()` e `toTimeString()`, podemos dividir a exibição completa da data em dados da data e da hora, conforme apresentado na Figura 6.10 (script `ex6_8.html`), decorrente da execução dos seguintes comandos.

Exemplo 6.8 – Programa que exibe data e hora atual (`ex6_8.html`)

```
<script>
  var hoje = new Date();
  alert("Dia: " + hoje.toDateString() + "\nHora: " +
    hoje.toTimeString());
</script>
```

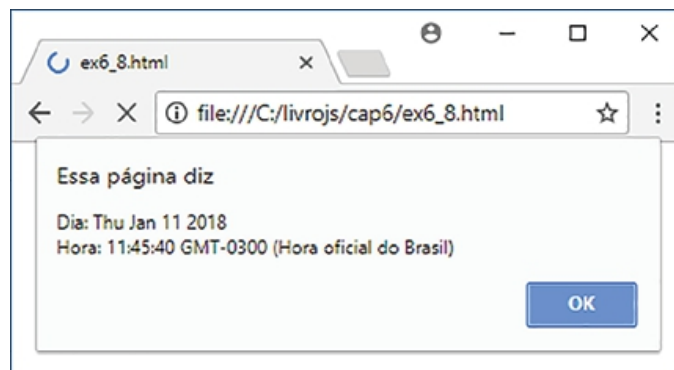


Figura 6.10 – Métodos `toDateString()` e `toTimeString()`.

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 (`ex6_9.html`), extraímos o ano atual para calcular o ano de nascimento de uma pessoa.

Exemplo 6.9 – Programa que calcula ano de nascimento de uma pessoa (`ex6_9.html`)

```

<meta charset="utf-8">
<script>
    var anoAtual = new Date().getFullYear();
    var idade = prompt("Quantos anos você comemora em " + anoAtual + "?");
    var 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 `ex9_10.html` e, na sequência, vamos destacar esses detalhes.

Exemplo 6.10 – Código HTML do Programa Caixa da Loja (ex6_10.html)

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="css/estilos.css">
    <title>Exemplo 6.10</title>

```

```

</head>

<body>
  
  <h1> Programa Caixa da Loja </h1>
  <p>Data de Vencimento:
    <input type="date" id="inDataVenc" autofocus>
  </p>
  <p>Valor da Conta R$:
    <input type="text" id="inValor" class="aDireita">
    <input type="button" value="Calcular Multa e Juros"
id="btCalcular">
  </p>
  <p>Valor da Multa R$:
    <input type="text" id="outMulta" class="aDireita" readonly>
  </p>
  <p>Valor do Juros R$.:
    <input type="text" id="outJuros" class="aDireita" readonly>
  </p>
  <p>Total a Pagar R$....:
    <input type="text" id="outTotal" class="aDireita" readonly>
    <input type="button" value="Nova Conta" id="btNovaConta">
  </p>
  <script src="js/ex6_10.js"></script>
</body>

</html>

```

Para receber a data no programa 6.10, foi utilizada a tag `<input type="date">`. Esse tipo de campo de entrada foi acrescentado no HTML 5 e demorou algum tempo para ser renderizado da mesma forma por todos os navegadores. O Mozilla Firefox o exibiu como um campo de texto normal, mas nas suas últimas versões seguiu o padrão dos demais navegadores. Na Figura 6.11, é exibida a caixa da tag `<input type="date">` renderizada no Google Chrome.



Figura 6.11 – Campo de entrada do tipo date.

Há ainda outra mudança nesse código HTML em relação aos exemplos anteriores. Utilizamos campos `<input type="text">` 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 `ex6_10.js` para verificar se a conta está em atraso e calcular os valores da conta.

Programa JavaScript para calcular multa e juros de contas em atraso (js/ex6_10.js)

```
const TAXA_MULTA = 2 / 100; // multa por atraso
const TAXA_JUROS = 0.33 / 100; // juros por dia de atraso

function calcularMultaJuros() {
    // cria referência aos elementos da página
    var inDataVenc = document.getElementById("inDataVenc");
    var inValor = document.getElementById("inValor");
    var outMulta = document.getElementById("outMulta");
    var outJuros = document.getElementById("outJuros");
    var outTotal = document.getElementById("outTotal");

    // obtém os conteúdos dos campos de entrada
    var dataVenc = inDataVenc.value;
    var valor = Number(inValor.value);

    // valida o preenchimento dos campos
```

```

if (dataVenc == "" || valor == 0 || isNaN(valor)) {
    alert("Informe corretamente os dados da conta...");
    inData.focus();
    return;
}

// cria duas variáveis (instancia dois objetos) tipo data
var hoje = new Date();
var vencTo = new Date();

// a data vem no formato aaaa-mm-dd
var partes = dataVenc.split("-");
vencTo.setDate(Number(partes[2]));
vencTo.setMonth(Number(partes[1]) - 1);
vencTo.setFullYear(Number(partes[0]));

// calcula a diferença de dias entre as datas (em milissegundos)
var atraso = hoje - vencTo;

var multa = 0; // inicializa multa e juros com 0
var juros = 0;

// se conta estiver em atraso ...
if (atraso > 0) {
    // converte os milissegundos da diferença em dias
    // (1 dia = 24hor x 60min x 60seg x 1000mseg: 86400000)
    // round(): necessário para períodos envolvendo horário de verão
    var dias = Math.round(atraso / 86400000);

    multa = valor * TAXA_MULTA; // calcula multa e juros
    juros = (valor * TAXA_JUROS) * dias;
}

var total = valor + multa + juros; // calcula o total

outMulta.value = multa.toFixed(2); // exibe os valores com 2
decimais
outJuros.value = juros.toFixed(2);
outTotal.value = total.toFixed(2);
}

```

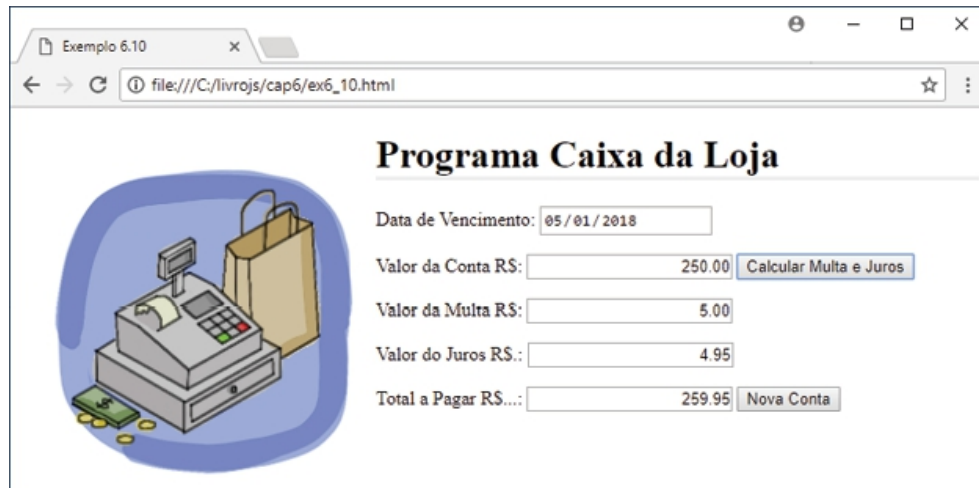
```
// cria referência ao botão e após associa function ao evento click
var btCalcular = document.getElementById("btCalcular");
btCalcular.addEventListener("click", calcularMultaJuros);
```

Vamos aos comentários sobre esse programa. Iniciamos pela definição das constantes que definem a taxa da multa e dos juros a ser aplicada sobre as contas em atraso. Na sequência, mantemos a forma padrão que adotamos neste livro, criando uma referência aos elementos que são manipulados pelo script e validando o preenchimento dos campos de entrada. Em seguida, 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, na sequência, 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 6.12 ilustra o funcionamento do programa.

Para inserir uma nova conta, podemos recorrer ao método `reload()`, já utilizado em outros exemplos. Ele recarrega a página, portanto limpa os campos de entrada e posiciona o foco na data de vencimento (a partir do atributo `autofocus` desse campo). Acrescente o seguinte código ao programa `ex6_10.js`.

```
function limparCampos() {
    location.reload();
}
var btNovaConta = document.getElementById("btNovaConta");
btNovaConta.addEventListener("click", limparCampos);
```



Programa Caixa da Loja

Data de Vencimento: 05/01/2018

Valor da Conta R\$: 250.00

Valor da Multa R\$: 5.00

Valor do Juros R\$: 4.95

Total a Pagar R\$: 259.95

Figura 6.12 – Cálculo da multa e juros de uma conta em atraso.

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. Note no script a seguir (ex6_11.html) e na Figura 6.13 que o mês de janeiro é exibido como 0.

Exemplo 6.11 – Programa que exibe dia, mês e ano de uma data (ex6_11.html)

```
<script>
var hoje = new Date();
var dia = hoje.getDate();
var mes = hoje.getMonth();
var ano = hoje.getFullYear();
alert(hoje+"\nData: " + dia + "/" + mes + "/" + ano);
</script>
```

Portanto, para exibir o valor numérico do mês corretamente nesse formato, deve-se adicionar 1 à variável relativa ao mês de uma data.

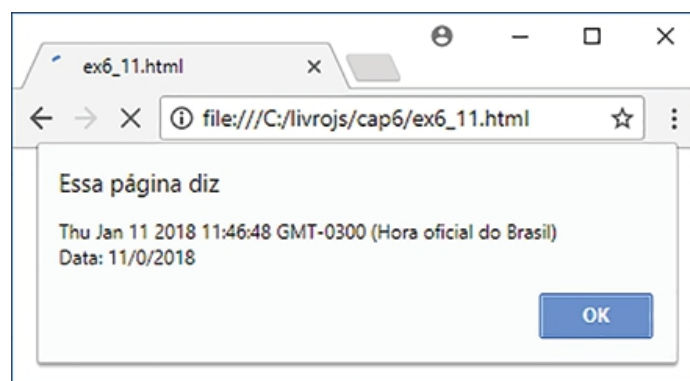


Figura 6.13 – O número do mês varia de 0 a 11.

Vamos construir um último exemplo para destacar as vantagens de trabalhar com datas em JavaScript e também para reforçar esse cuidado com a exibição do mês. O programa solicita o número de parcelas que devem ser geradas e calcula a data de cada uma dessas parcelas, uma para cada mês. A Figura 6.14 exibe o `alert()` com as datas geradas.

Exemplo 6.12 – Código HTML do Programa Data das Parcelas (ex6_12.html)

```
<script>
  var parcelas = Number(prompt("Quantas Parcelas? "));

  var lista = ""; // irá acumular as parcelas
  var dia, mes, ano, diaZero, mesZero;

  var data = new Date(); // data atual

  for (var i = 1; i <= parcelas; i++) {
    data.setMonth(data.getMonth() + 1); // aumenta um mês na data
    dia = data.getDate();
    mes = data.getMonth() + 1; // mês varia de 0 a 11, por isso, +1
    ano = data.getFullYear();

    diaZero = dia < 10 ? "0" + dia : dia; // acrescenta 0 se dia menor
    que 10
    mesZero = mes < 10 ? "0" + mes : mes; // acrescenta 0 se mês menor
    que 10

    // concatena as parcelas na variável linha, com uma quebra de
    linha ao final
    lista += i + "ª Parcela: " + diaZero + "/" + mesZero + "/" + ano +
    "\n";
  }
  alert(lista); // exibe as parcelas
</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.

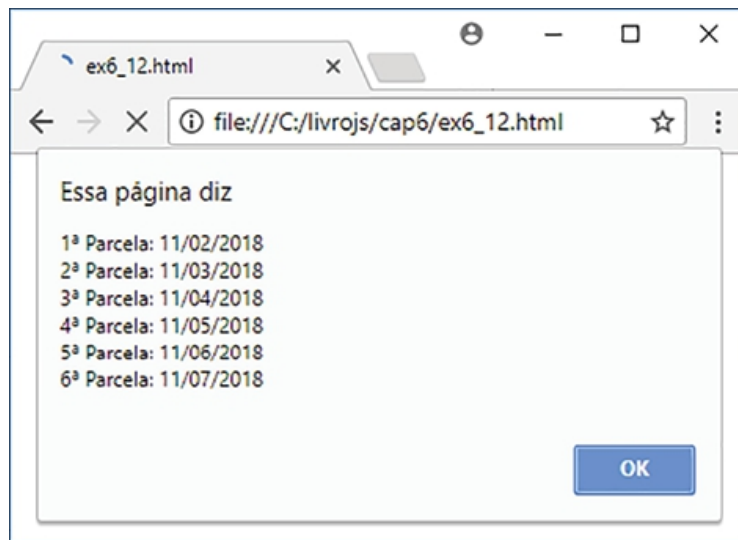


Figura 6.14 – Datas de vencimento das parcelas de uma conta.

6.9 Exercícios

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 6.15 exibe a mensagem criptografada. O programa deve conter ainda um botão para decryptografar a mensagem, ou seja, retornar a mensagem original a partir do texto cifrado (Figura 6.16).

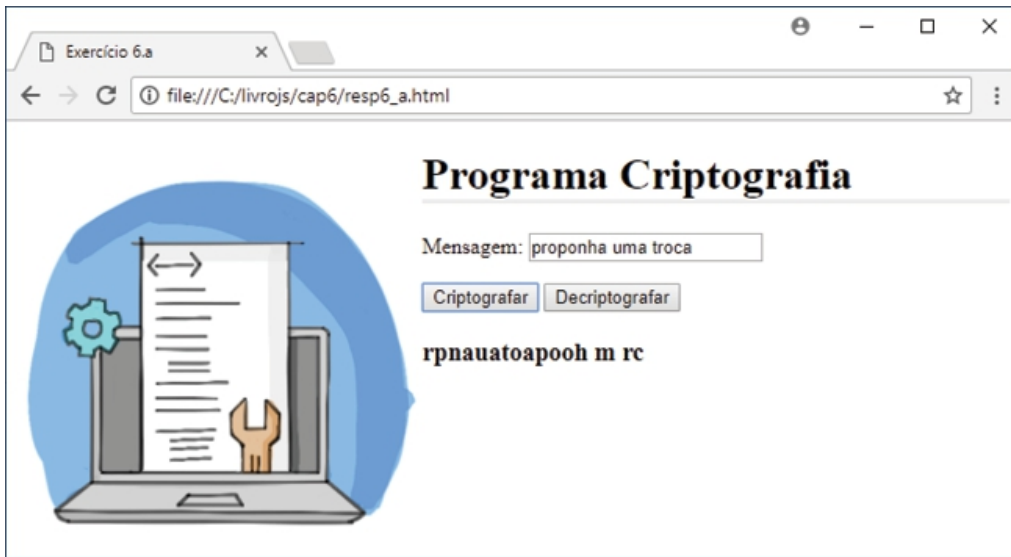


Figura 6.15 – Exemplo da mensagem criptografada.

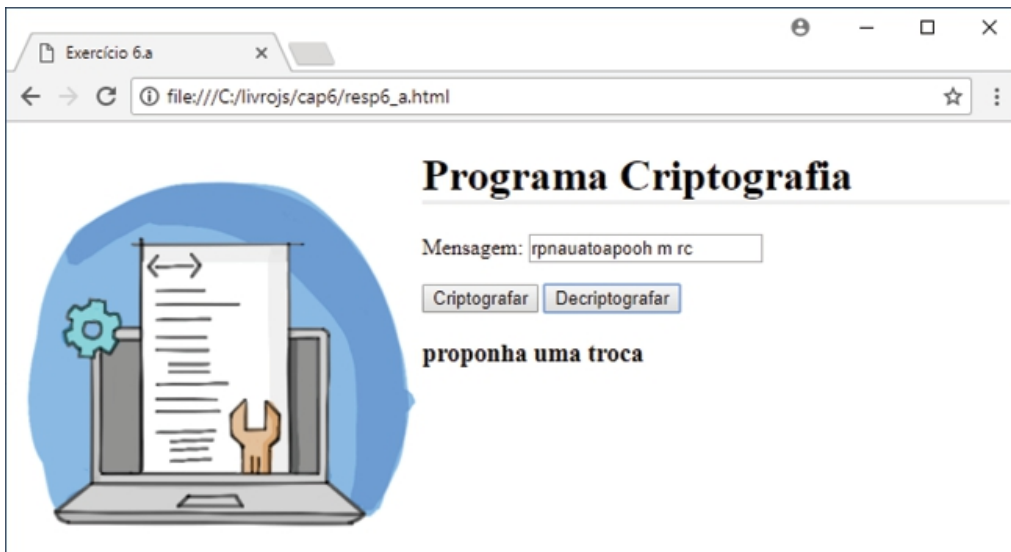


Figura 6.16 – O programa deve remontar a mensagem original a partir do texto cifrado.

b) Elaborar um programa que leia o nome de um autor de um livro. Em seguida, exiba-o no formato de uma citação bibliográfica com o sobrenome, seguido pelas iniciais do nome, conforme o exemplo ilustrado na Figura 6.17.



Figura 6.17 – O programa deve compor a citação bibliográfica, a partir do nome do autor.

c) 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 6.18 apresenta uma frase que é um palíndromo.



Figura 6.18 – Um palíndromo pode ser lido nos dois sentidos.

d) 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 6.19 ilustra o funcionamento do programa.



Figura 6.19 – Programa multa de trânsito deve utilizar os métodos de manipulação de datas.

6.10 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

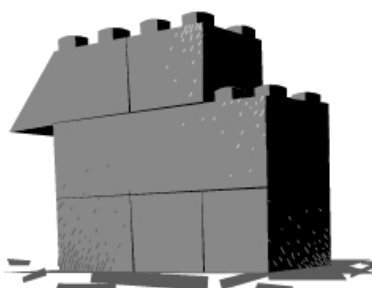
métodos `toUpperCase()` e `toLowerCase()` realizam, respectivamente, essas tarefas e podem ser aplicados sobre uma palavra ou um caractere da palavra.

- **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 `var 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 `var data = new Date()`.

CAPÍTULO 7

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. 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 precisamos executar esse trecho de código, chamamos o módulo, sem a necessidade de reescrever seus comandos.

Os eventos também podem ser mais bem explorados, pois o único evento que trabalhamos nos exemplos do livro até aqui foi o evento click. Há vários outros: 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.

7.1 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 realizamos a chamada dela. Ele serve para ampliar o uso 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. Novamente, neste capítulo, vamos desenvolver pequenos scripts para exemplificar os temas abordados e alguns programas maiores. Crie a estrutura de pastas como nos capítulos anteriores (`cap7` e, dentro dela, `css`, `img` e `js`). O primeiro exemplo é descrito a seguir:

Exemplo 7.1 – Exemplo de função com passagem de parâmetro (ex7_1.html)

```
<script>
// função recebe 2 parâmetros: nota e media
function exibirSituacao(nota, media) {
    if (nota >= media) { // compara o valor das variáveis
        alert("Aprovado");
    } else {
        alert("Reprovado");
    }
}

var prova1 = Number(prompt("Qual Nota: ")); // lê uma nota

// chama a função exibirSituacao() passando 2 argumentos
exibirSituacao(prova1, 7);
</script>
```

Observe que a função `exibirSituacao()` recebe os parâmetros `nota` e `media`. 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`.

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.

É possível também obter o conteúdo das variáveis passadas como parâmetro, ou seja, os argumentos da função utilizando a palavra reservada **arguments**. Ela funciona como um vetor que contém a lista dos valores passados na chamada da função. Dessa forma, pode-se omitir a declaração dos parâmetros no nome da função e recuperá-los utilizando **arguments**. Observe o exemplo a seguir:

Exemplo 7.2 – Uso de arguments para obter valores passados à função (ex7_2.html)

```
<script>
function calcularMedia() {

    if (arguments.length == 0) { // se não foram passados argumentos
        alert("Informe, no mínimo, uma nota");
        return;
    }
    var soma = 0; // vai acumular a soma das notas
    var numNotas = arguments.length; // obtém a quantidade de
    argumentos

    for (var i = 0; i < numNotas; i++) { // percorre os argumentos
        soma += arguments[i]; // soma o valor dos argumentos
    }

    var media = soma / numNotas; // calcula a média
    alert("Média das Notas: " + media.toFixed(1));
}

// exemplos de chamada de calcularMedia() com nº de argumentos
diferentes
calcularMedia(5, 6, 8);
calcularMedia(2, 10);
calcularMedia(7.5, 10, 8, 9.5);
calcularMedia();
</script>
```

A função inicia com um teste que exibe um alerta caso o número de argumentos seja igual a zero. Como `arguments` é um vetor, utilizamos a propriedade `length` para recuperar o número de elementos desse vetor. Em seguida, é possível utilizar uma estrutura de repetição com o comando `for` 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.

7.2 Funções com retorno de valor

Nos exemplos 7.1 e 7.2, as funções receberam parâmetros e então apresentaram dentro de cada 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 7.1 retorne um valor indicativo da situação do aluno, poderíamos modificá-la da seguinte forma:

```
function obterSituacao(nota, media) {  
    var situacao = (nota >= media) ? "Aprovado" : "Reprovado";  
    return situacao;  
}
```

Também poderíamos utilizar o operador condicional na função original do Exemplo 7.1. Observe, porém, que a função recebe os parâmetros e com base neles define o conteúdo da variável `situacao`. 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:

```
var resposta = obterSituacao(prova1, 7);  
outResposta.textContent = "Situação: " + resposta;
```

Ou, então, utilizar o próprio retorno da função como parte ou parâmetro de outro método, como na chamada ao método `alert()` seguinte:

```
alert("Situação: " + obterSituacao(prova1, 7));
```

Vamos agora construir um exemplo com uso de funções com retorno de valor. Nosso programa será para a Revenda Herbie (a mesma do Exemplo

5.3). Ele deve ler modelo, ano de fabricação e preço do veículo (o valor que o proprietário deseja receber pelo 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 preço de venda do veículo com um acréscimo de 8% no preço dos veículos novos ou de 10% no preço dos veículos classificados como seminovos ou usados. Para a classificação e o cálculo do preço de venda do veículo, serão utilizadas funções com retorno de valor. A Figura 7.1 ilustra a página com um veículo para exemplificar os dados de entrada e saída do programa.



Figura 7.1 – 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; }
.aDireita { text-align: right; }
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 7.3 – Programa vai classificar e calcular o preço de venda do veículo (ex7_3.html)

```
<!DOCTYPE html>  
<html lang="pt-BR">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-  
scale=1.0">  
  <meta http-equiv="X-UA-Compatible" content="ie=edge">  
  <link rel="stylesheet" href="css/estilos.css">  
  <title>Exemplo 7.3</title>  
</head>  
<body>  
    
  <h1>Revenda Herbie - Consignados</h1>  
  <p>Modelo do Veículo:  
    <input type="text" id="inModelo" autofocus>  
  </p>  
  <p>Ano de Fabricação:  
    <input type="text" id="inAno" class="aDireita">  
  </p>  
  <p>Preço Proprietário R$:  
    <input type="text" id="inPreco" class="aDireita">  
  </p>  
  <input type="button" value="Classificar/Calcular Preço"  
id="btCalcular">  
  <h3 id="outClassif"></h3>  
  <h3 id="outPrecoVenda"></h3>  
  <script src="js/ex7_3.js"></script>  
</body>  
</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 utilizados no programa, tanto para obter o conteúdo informado pelo usuário, quanto para exibir a resposta ou associar uma função ao evento `click` do botão.

Código JavaScript que chama funções com retorno de valor (js/ex7_3.js)

```
function mostrarDados() {  
    // cria referência aos elementos da página  
    var inModelo = document.getElementById("inModelo");  
    var inAno = document.getElementById("inAno");  
    var inPreco = document.getElementById("inPreco");  
    var outClassif = document.getElementById("outClassif");  
    var outPrecoVenda = document.getElementById("outPrecoVenda");  
  
    // obtém o conteúdo dos campos de entrada de dados  
    var modelo = inModelo.value;  
    var ano = Number(inAno.value);  
    var preco = Number(inPreco.value);  
  
    // valida o preenchimento dos campos  
    if (modelo == "" || ano == 0 || preco == 0 || isNaN(ano) ||  
        isNaN(preco)) {  
        alert("Informe corretamente os dados do veículo");  
        inModelo.focus();  
        return;  
    }  
  
    // chama e atribui o retorno das funções às variáveis  
    var classificacao = classificarVeiculo(ano);  
    var precoVenda = calcularVenda(preco, classificacao);  
  
    // exibe as respostas  
    outClassif.textContent = modelo + " - " + classificacao;  
    outPrecoVenda.textContent = "Preço de Venda R$: " +  
    precoVenda.toFixed(2);  
}  
  
// cria referência ao botão e associa ao evento click a função  
mostrarDados  
var btCalcular = document.getElementById("btCalcular");
```

```
btCalcular.addEventListener("click", mostrarDados);
```

Observe que depois de validar o preenchimento dos campos são feitas chamadas às funções `classificarVeiculo()` e `calcularVenda()` com parâmetros sendo passados. O retorno dessas funções é atribuído para as variáveis `classificacao` e `precoVenda` 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
function classificarVeiculo(ano) {
    var anoAtual = new Date().getFullYear(); // obtém o ano atual
    var classif;
    // condições para definir a classificação do veículo
    if (ano == anoAtual) {
        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 `calcularVenda()`, descrita a seguir:

```
// função recebe valor e status do veículo como parâmetro
function calcularVenda(valor, status) {
    var prVenda = (status == "Novo") ? valor * 1.08 : valor * 1.10;
    return prVenda;
}
```

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. O valor a ser atribuído para a variável **prVenda** depende da condição realizada com base no conteúdo da variável **status**. Ao final, retorna-se o valor da variável **prVenda**.

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.

7.3 Funções anônimas

As funções anônimas permitem definir a programação de um evento sem atribuir um nome para a função. Apenas se utiliza a palavra reservada **function()** seguida do trecho de programa que queremos executar. No geral, são utilizadas para a inserção de pequenos trechos de código. Contudo, todos os recursos de programação podem ser normalmente inseridos em uma função anônima.

Vamos construir um exemplo para demonstrar a sintaxe da criação de uma função anônima. Nosso programa deve ler o preço de um produto qualquer e exibir as formas de pagamento disponíveis na loja, que variam de 1x até 10x. Dessa forma, realizamos uma breve revisão das rotinas de repetição abordadas no Capítulo 4. A Figura 7.2 ilustra o funcionamento do programa.



Figura 7.2 – Função anônima vai realizar o cálculo das parcelas.

O código HTML da página exibida na Figura 7.2 é apresentado a seguir.

Exemplo 7.4 – Programa cálculo das parcelas (ex7_4.html)

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 7.4</title>
</head>
<body>
  
  <h1>Programa Cálculo das Parcelas</h1>
  <p>Preço R$:
    <input type="text" id="inPreco" class="numero" autofocus>
    <input type="button" value="Exibir Formas de Pagto" id="btExibir">
  </p>
  <pre id="outParcelas"></pre>
  <script src="js/ex7_4.js"></script>
</body>
</html>
```

Na sequência, crie o código JavaScript para calcular e exibir o valor das parcelas com o uso de uma função anônima.

Código do programa JavaScript contendo declaração de função anônima (js/ex7_4.js)

```
// cria uma referência ao botão
var btExibir = document.getElementById("btExibir");

// ao clicar no botão, a programação da função anônima é executada
btExibir.addEventListener("click", function () {
    // cria referência aos elementos da página
    var inPreco = document.getElementById("inPreco");
    var outParcelas = document.getElementById("outParcelas");

    // obtém o conteúdo do campo de entrada
    var preco = Number(inPreco.value);

    // vai acumular as formas de pagamento
    var lista = "";

    // cria uma repetição para montar as opções de pagamento
    for (var i = 1; i <= 10; i++) {
        // acumula em lista o nº da parcela e o cálculo do valor
        lista += i + "x de R$: " + (preco / i).toFixed(2) + "\n";
    }

    // exibe a lista no elemento outParcelas
    outParcelas.textContent = "Opções de Pagamento\n\n" + lista;
});
```

Percebe-se uma mudança logo no início do programa, que começa pela captura do elemento `btExibir` da página HTML. Em seguida, na declaração do método `addEventListener()`, deve-se indicar normalmente o evento ao qual a linguagem ficará na “escuta”. A modificação vem a seguir: informa-se a palavra reservada `function()` e `"{"` para indicar que a programação da função anônima será desenvolvida na sequência. No restante, realizam-se as mesmas operações de uma função nomeada: cria-se referência aos elementos da página, obtém-se o seu conteúdo e, para esse programa, utiliza-se uma rotina de repetição para montar a resposta a ser exibida na página.

7.4 Funções atribuídas para variáveis

A linguagem JavaScript permite que uma função seja atribuída a uma variável. Ou seja, no lugar de se atribuir uma expressão para a variável, realizamos a atribuição de uma função. Por isso, esse formato é também conhecido como expressão de função. Para chamar a função, devemos informar o nome da variável e, como nos outros formatos, é possível declarar parâmetros a serem manipulados pela *function*. Observe o exemplo a seguir:

Exemplo 7.5 – Função atribuída à variável pode também receber parâmetros (ex7_5.html)

```
<script>
  var dobro = function(a) {
    return a * 2;
  }
  var num = Number(prompt("Número: "));
  alert("O dobro é: " + dobro(num));
</script>
```

A linha desse exemplo que chama a *function*, passa o valor de *num* como argumento e exibe o retorno da função é destacada a seguir:

```
  alert("O dobro é: " + dobro(num));
```

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.

7.5 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 8.1), 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 7.6 explora a programação de novos eventos (além do `click`) visando dar maior interatividade a um sistema de controle de pedidos de uma pizzeria. 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 7.3 apresenta esse recurso, em que a troca para bebidas modificou os itens do campo `select`.



Figura 7.3 – 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 de 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 7.4 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 7.4 – A dica é alterada ao posicionar no campo de edição, conforme o tamanho da pizza.

Além deles, vamos também facilitar a inserção de um novo item no pedido, a partir da verificação das teclas pressionadas na edição do

conteúdo do campo **Detalhes do Item**. Caso o usuário (garçom) pressione a tecla **enter** no campo, o item deve ser inserido no pedido. Esse recurso pode ser adicionado aos diversos programas implementados no livro. A seguir o código HTML da página da Pizzaria Avenida.

Exemplo 7.6 – Código HTML da página da Pizzaria Avenida (ex7_6.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 7.6</title>
</head>
<body>
  
  <h1>Pizzaria Avenida - Controle de Pedidos</h1>
  <p>Item do Pedido:
    <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:
    <input type="text" id="inDetalhes" class="detalhes">
    <input type="button" value="Adicionar" id="btAdicionar">
```

```
</p>
<pre id="outPedido"></pre>
<script src="js/ex7_6.js"></script>
</body>
</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/ex7_6.js)

```
var itens = []; // vetor global para armazenar os itens do pedido

function trocarItem() {
    // cria referência aos elementos select
    var inPizza = document.getElementById("inPizza");
    var inBebida = document.getElementById("inBebida");

    // se estiver marcado
    if (rbPizza.checked) {
        inBebida.className = "oculta"; // oculta select das bebidas
        inPizza.className = "exibe"; // exhibe select das pizzas
    } else {
        inPizza.className = "oculta"; // oculta as pizzas
        inBebida.className = "exibe"; // exhibe as bebidas
    }
}

// cria referência aos radiobutton's e associa função ao evento
change
var rbPizza = document.getElementById("rbPizza");
rbPizza.addEventListener("change", trocarItem);

var rbBebida = document.getElementById("rbBebida");
rbBebida.addEventListener("change", trocarItem);
```

O programa começa pela declaração de um vetor global para armazenar os itens do pedido. Na sequência foi criada a função `trocarItem()`. Ela é acionada pelo evento `change` dos dois botões do tipo `radio` da página HTML. Ou seja, quando o garçom trocar de pizza para bebida, ou vice-versa, a função será chamada. E o que essa função faz? Captura os `selects` da página e, depois de verificar se `rbPizza` está marcado, 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 `ex7_6.js` está relacionado ao campo “Detalhes do Item”. Adicione os seguintes comandos nesse programa.

```
function mostrarNumSabores() {  
    // se radiobutton rbPizza estiver marcado  
    if (rbPizza.checked) {  
        var pizza = inPizza.value; // obtém value do item selecionado  
        // uso o operador condicional (cap. 3), para indicar o número de  
        sabores  
        var num = (pizza == "media") ? 2 : (pizza == "grande") ? 3 : 4;  
        // atributo placeholder exibe uma dica de preenchimento do campo  
        inDetalhes.placeholder = "Até " + num + " sabores";  
    }  
}  
// cria referência ao elemento e associa função ao evento focus  
var inDetalhes = document.getElementById("inDetalhes");  
inDetalhes.addEventListener("focus", mostrarNumSabores);  
  
// cria função (anônima) ao evento blur (quando o campo perde o  
foco)  
inDetalhes.addEventListener("blur", function() {  
    inDetalhes.placeholder = ""; // limpa a dica de preenchimento  
});
```

A função `mostrarNumSabores()` é chamada toda vez que o campo `inDetalhes` recebe o foco. Ela 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. Uma observação: como já definimos uma referência global ao elemento `rbPizza` (para programar a troca entre pizza e bebida), não é necessário referenciar novamente esse elemento na `function`. O mesmo vale para o elemento `inDetalhes`.

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`, criamos uma função anônima, já que precisamos de apenas um comando quando esse evento ocorrer, que é a retirada da dica de preenchimento (importante para quando o garçom sair do campo e voltar ao `radiobutton`, por exemplo).

O próximo trecho de código também cria uma função anônima, agora associada ao evento `keypress` do campo `inDetalhes`. Esse é o terceiro evento associado a esse elemento da página HTML.

```
// cria função (anônima) associada ao evento keypress (tecla
pressionada)
inDetalhes.addEventListener("keypress", function (tecla) {
  // se pressionou tecla de código 13 (enter)
  if (tecla.keyCode == 13) {
    adicionarItem(); // irá adicionar item no pedido
  }
});
```

Observe que a função recebe o parâmetro `tecla`. Caso a tecla pressionada seja a de código 13, que é o código da tecla `enter`, nós realizamos uma chamada à função `adicionarItem()` que também será acionada pelo evento `click` do botão **Adicionar**. Ou seja, mesmo em um sistema simples, já é possível aproveitar os benefícios da modularização de programas.

Na sequência, destacamos a função principal do programa, que é a `adicionarItem()`. Essa função cria referência aos dois campos do tipo `select`, mas apenas um deles será utilizado quando a função for acionada. Para verificar se devemos obter o tamanho da pizza ou o tipo de bebida, 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 function.

```
function adicionarItem() {  
    // cria referência aos elementos da página (ainda não  
    referenciados)  
    var inPizza = document.getElementById("inPizza");  
    var inBebida = document.getElementById("inBebida");  
    var outPedido = document.getElementById("outPedido");  
  
    // se radiobutton Pizza estiver marcado  
    if (rbPizza.checked) {  
        var num = inPizza.selectedIndex; // obtém nº do item selecionado  
        var produto = inPizza.options[num].text; // texto do item  
        selecionado  
    } else {  
        var num = inBebida.selectedIndex;  
        var produto = inBebida.options[num].text;  
    }  
    var detalhes = inDetalhes.value; // conteúdo do inDetalhes  
    itens.push(produto + " (" + detalhes + ")"); // adiciona ao vetor  
    outPedido.textContent = itens.join("\n"); // exibe em outPedidos  
    limparCampos(); // limpa conteúdo dos campos  
}  
// cria referência ao elemento e associa função ao evento click  
var btAdicionar = document.getElementById("btAdicionar");  
btAdicionar.addEventListener("click", adicionarItem);
```

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 `outPedido`.

Observe que no final da function `adicionarItem()` há uma chamada para a function `limparCampos()`. Agrupar os comandos que realizam uma tarefa em específico é uma boa prática para termos um código mais organizado.

```
function limparCampos() {  
    rbPizza.checked = true; // marca (seleciona) rbPizza  
    inBebida.className = "oculta"; // oculta select das Bebidas
```

```
inPizza.className = "exibe"; // exibe select das Pizzas
inPizza.selectedIndex = 0; // seleciona 1º item (posição 0)
inDetalhes.value = ""; // limpa input Detalhes
rbPizza.focus(); // "joga o foco" no rbPizza
}
```

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 seção 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 10, continuaremos a explorar os eventos e as funções com passagem e retorno de parâmetros.

Vamos agora desenvolver alguns exercícios sobre os assuntos destacados neste capítulo? 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.

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!

7.6 Exercícios

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 7.5 ilustra os dados de entrada e saída do programa.



Figura 7.5 – Os tra os 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 ser  o sobrenome seguido pelo n mero de vogais do nome completo dele. O programa deve conter as fun  es:

- `validarNome()` – que receba um nome como par metro e retorne `true` (nome completo) ou `false` (nome incompleto).
- `obterSobrenome()` – que receba um nome como par metro e retorne o  ltimo nome do aluno em letras min sculas.
- `contarVogais()` – que receba um nome e retorne o n mero de vogais

deste, com dois dígitos.

A Figura 7.6 destaca um exemplo de execução desse exercício.



Figura 7.6 – 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 7.7 apresenta a tela inicial do programa para um cliente sem convênio. Já a Figura 7.8 exibe a página para um cliente que possui convênio. Observe que a caixa de seleção não deve ser exibida no início do programa.



Figura 7.7 – A opção “Não” (em “Possui Convênio”) deve estar inicialmente selecionada na página.



Figura 7.8 – Ao modificar para “Sim”, um campo select com os convênios deve ser exibido.

7.7 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, conseqüentemente, a manutenção do programa,

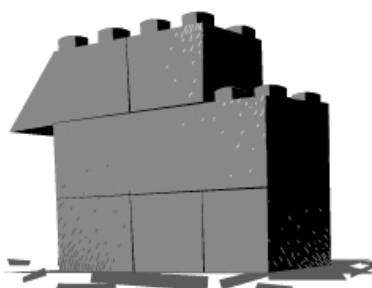
além de permitir a reutilização de módulos já desenvolvidos. Dessa forma, uma função criada e testada uma vez pode ser novamente utilizada em outra parte do programa. Isso também economiza tempo, produz sistemas mais confiáveis e permite gerenciar melhor o trabalho em equipe. Esses benefícios, contudo, 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 8

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 reinicializar 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 sejamos 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).

Observação: Nos navegadores Internet Explorer e Edge, o uso do `localStorage` funciona corretamente em páginas hospedadas em servidores web. Contudo, pelo menos até a conclusão deste livro, havia restrições para o armazenamento de dados local (`file://`). Nos demais navegadores, você pode testar normalmente os exemplos deste capítulo salvando os arquivos em seu computador.

8.1 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 da w3schools.com, para os nossos primeiros exemplos deste capítulo. Acesse o endereço do site, clique em JavaScript e em algum dos botões **Try It Yourself**. Para salvar de forma persistente um conteúdo para a variável `idade`, limpe os códigos do programa de exemplo do site e 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**. 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 8.1.

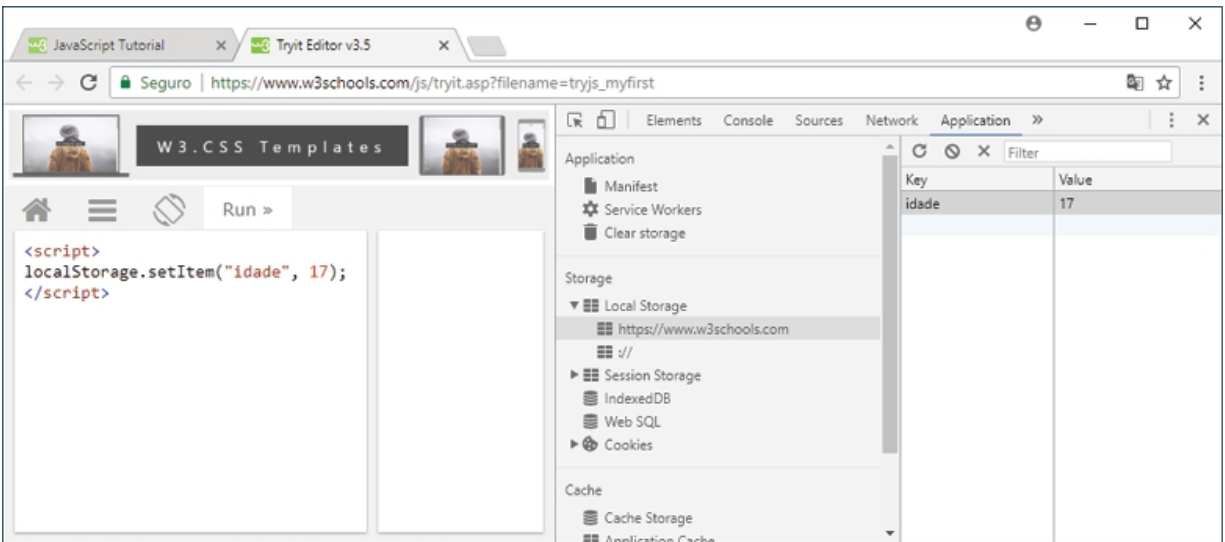


Figura 8.1 – Dados salvos no localStorage podem ser manipulados no navegador.

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.

```
var 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>
  var idade = localStorage.getItem("idade");
  var 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:

```
var 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 `cap8` 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:

```
img { max-width: 100%; }
pre { font-size: 1.2em; }
.oculta { display: none; }
.exibe { display: inline; }
.coresBrasil { color: black; background-color: red; }
.coresPelotas { color: blue; background-color: yellow; }
.coresFarroupilha { color: green; background-color: white; }
```

Porém, antes de digitar o código do primeiro programa do Capítulo 8, vamos conversar sobre um aliado importante que podemos utilizar na construção do layout de nossas páginas: o Bootstrap.

8.2 Uma “pitada” de Bootstrap

Como salientamos nos capítulos iniciais, não temos a intenção neste livro de destacar questões relacionadas ao design da página. Nosso 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 3, 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*.

Em nossos exemplos, vamos utilizar apenas recursos básicos do Bootstrap. Para saber mais sobre ele, baixe o livro do Maurício Samy Silva, disponível gratuitamente no site da Editora Novatec. Pode também consultar o site da [W3Schools](http://w3schools.com), na seção que aborda essa tecnologia.

Para utilizar o Bootstrap em seus projetos, baixe os arquivos do site da biblioteca ou então referencie um CDN (*Content Delivery Network* – Rede de Distribuição de Conteúdo), um site onde os arquivos do framework estão disponíveis. Para baixar a biblioteca, acesse o endereço www.getbootstrap.com e clique no botão **download**, conforme ilustra a Figura 8.2. Do arquivo `.zip` baixado, necessitamos, para os nossos exemplos, apenas do arquivo `bootstrap.min.css`. Copie-o para a pasta `css` em que você vai criar os exemplos deste capítulo.

Caso opte por referenciar diretamente o arquivo da biblioteca de um endereço web, você pode consultar o site da W3Schools que, em seus exemplos, utilizam essa forma. Um dos endereços válidos que contém o Bootstrap é o seguinte:

`https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css`

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.

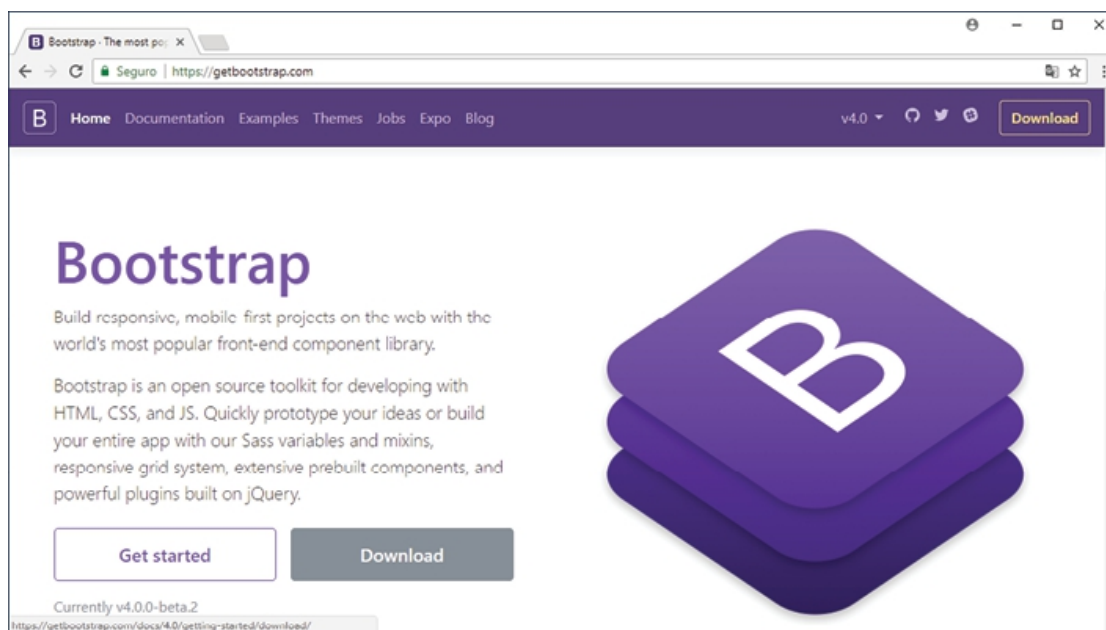


Figura 8.2 – Site para download do Bootstrap.

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 8.3. Naturalmente, você pode modificar esse programa inserindo os clubes de sua cidade ou estado.

O código HTML do Exemplo 8.1 destacado a seguir exibe a página da loja de esportes sem apresentar, inicialmente, a imagem do clube do cliente.

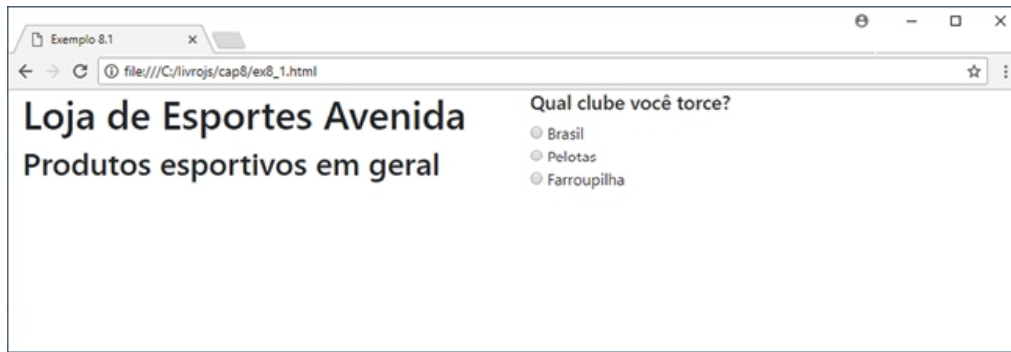


Figura 8.3 – Página inicial da loja de esportes com os estilos do Bootstrap inseridos.

Exemplo 8.1 – Código HTML do programa Loja de Esportes (ex8_1.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 8.1</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>
        <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
      </div>
    </div>
  </div>
```

```

<div class="col-sm-2">
  <img id="imgClube" class="oculta">
</div>
</div>
</div>
<script src="js/ex8_1.js"></script>
</body>
</html>

```

Observe que no cabeçalho da página `ex8_1.html` acrescentamos a referência ao arquivo baixado do site do Bootstrap. 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 8.1, 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 (também `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 8.4 apresenta a página com um dos clubes selecionados.



Figura 8.4 – 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 function `trocarClube` descrita a seguir.

Código JavaScript do programa Loja de Esportes (js/ex8_1.js)

```
function trocarClube() {  
    // cria referência aos elementos da página  
    var imgClube = document.getElementById("imgClube");  
    var divTitulo = document.getElementById("divTitulo");  
  
    var clube; // define variável que irá receber o nome do clube  
  
    // verifica qual radiobutton está selecionado  
    if (rbBrasil.checked) {  
        clube = "Brasil";  
    } else if (rbPelotas.checked) {  
        clube = "Pelotas";  
    } else {  
        clube = "Farroupilha";  
    }  
  
    // define as classes de divTitulo: row e cores do clube  
    divTitulo.className = "row cores"+clube;  
  
    // modifica a imagem de acordo com a seleção do cliente  
    imgClube.src = "img/" + clube.toLowerCase() + ".png";  
    imgClube.className = "exibe"; // exibe a imagem  
    imgClube.alt = "Símbolo do " + clube; // modifica atributo alt  
  
    // salva no navegador a escolha do cliente  
    localStorage.setItem("clube", clube);  
}  
  
// captura os elementos do radiobutton  
var rbBrasil = document.getElementById("rbBrasil");  
var rbPelotas = document.getElementById("rbPelotas");  
var rbFarroupilha = document.getElementById("rbFarroupilha");  
  
// associa ao evento change a function trocarClube
```

```
rbBrasil.addEventListener("change", trocarClube);
rbPelotas.addEventListener("change", trocarClube);
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 `ex8_1.js`):

```
function verificarClube() {
    // se já estiver salvo algum clube
    if (localStorage.getItem("clube")) {

        var clube = localStorage.getItem("clube"); // obtém o nome do
        clube

        // conforme o clube, marca um dos elementos do input type radio
        if (clube == "Brasil") {
            rbBrasil.checked = true;
        } else if (clube == "Pelotas") {
            rbPelotas.checked = true;
        } else {
            rbFarroupilha.checked = true;
        }

        trocarClube(); // chama a function que troca a imagem e as cores
    }
}
// chama function que verifica se cliente já selecionou clube
anteriormente
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.

8.3 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` 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 8.1 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 `ex8_1.html` e acrescente apenas mais essa opção nas linhas do `input type="radio"`. Salve o arquivo com o nome `ex8_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). Há ainda

outros atributos importantes no `input type="radio"`, como o `value`, que define o valor do campo a ser recuperado em aplicações desenvolvidas no lado servidor (em PHP, por exemplo). Contudo, para o exemplo aqui demonstrado, esses outros atributos não são necessários.

Se na página HTML a modificação foi de apenas uma linha, no nosso programa JavaScript, as modificações serão várias. Como discutimos nos capítulos iniciais deste livro, sempre há diversas maneiras de resolver um problema em termos de programação. Vamos demonstrar isso aqui. O programa `ex8_2.js` vai explorar um novo método para recuperar os elementos `input type="radio"` e utilizar comandos de repetição que economizam algumas linhas de código. Começamos discutindo sobre o método `getElementsByName()`.

8.4 Uso do `getElementsByName()`

Em todos os nossos programas, utilizamos o método `getElementById()` 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 `getElementsByName()`.

Como o nome sugere, esse método captura os elementos da página com uma determinada `TagName`. Ele permite, por exemplo, acessar os parágrafos, as linhas de cabeçalho ou os campos de formulário do documento. 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 8 </p>
<script>
var p = document.getElementsByTagName("p");
p[0].style.color = "blue";
p[1].style.color = "red";
</script>
```


Esse documento HTML contém dois parágrafos. No script, o método `getElementsByTagName()` 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 dos `TagNames` 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:

```
var divResposta = document.getElementById("divResposta");
var p = divResposta.getElementsByTagName("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 `ex8_2.js`. A captura dos elementos, bem como a associação do evento `change` à função `trocarClube`, pode ser feita da seguinte forma:

```
// captura as tags input da página
var inputsRadio = document.getElementsByTagName("input");
// percorre os elementos para associar function ao evento change
for (var i = 0; i < inputsRadio.length; i++) {
    inputsRadio[i].addEventListener("change", trocarClube);
}
```

Observe que utilizamos o `getElementsByTagName()` 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 `function trocarClube()`, quando houver uma mudança de clube por parte do usuário.

Mas vamos modificar também o conteúdo da `function 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/ex8_2.js)

```
function trocarClube() {
    // cria referência aos elementos da página
    var imgClube = document.getElementById("imgClube");
    var divTitulo = document.getElementById("divTitulo");

    // armazena em um vetor a lista de clubes
    var clubes = ["Brasil", "Pelotas", "Farroupilha"];

    // percorre os radiobuttons para verificar qual está selecionado
    for (var i = 0; i < 4; i++) {
        if (inputsRadio[i].checked) { // se selecionado...
            var selecao = i; // armazena índice do radio selecionado
            break; // e sai da repetição
        }
    }
    if (selecao <= 2) { // se selecao <= 2, então torce por algum clube
        divTitulo.className = "row cores"+clubes[selecao]; // modifica
        cores (divTitulo)

        // muda a propriedade src com a imagem do clube selecionado
        imgClube.src = "img/" + clubes[selecao].toLowerCase() + ".png";
        imgClube.className = "exibe"; // exibe imagem
        imgClube.alt = "Símbolo do " + clubes[selecao]; // texto
        alternativo
        localStorage.setItem("clube", clubes[selecao]); // salva nome do
        clube
    } else { // else (selecionou "Nenhum")
        divTitulo.className = "row"; // tira a classe de cores da
        divTitulo
        imgClube.className = "oculta"; // oculta a imagem
        imgClube.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 `divTítulo` apenas para `row` (sem a classe das cores do clube), ocultar a imagem e remover o nome do clube da `localStorage`.

Já a function `verificarClube`, acionada quando a página é carregada, também foi ajustada para alterar o item selecionado de acordo com o nome dado ao vetor que contém os elementos `input type="radio"` da página.

```
function verificarClube() {  
  // se já estiver salvo algum clube  
  if (localStorage.getItem("clube")) {  
  
    var clube = localStorage.getItem("clube"); // obtém o nome do  
    clube  
  
    // conforme o clube, marca um dos elementos do input type radio  
    if (clube == "Brasil") {  
      inputsRadio[0].checked = true;  
    } else if (clube == "Pelotas") {  
      inputsRadio[1].checked = true;  
    } else {  
      inputsRadio[2].checked = true;  
    }  
  
    trocarClube(); // chama a function que troca a imagem e cores  
  }  
}  
// chama function que verifica se cliente já selecionou clube  
anteriormente  
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 8.5 exibe a página com essa guia selecionada.



Figura 8.5 – Ao trocar de clube, o conteúdo da localStorage é alterado, e em “Nenhum”, removido.

8.5 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 acidentalmente o navegador.

Exemplo 8.3 – Código HTML da página “Qual é o peso da Melancia?” (ex8_3.html)

```

<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 8.3</title>
</head>

<body>
  <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>
        <p><label for="inNome">Nome do Apostador: </label>
          <input type="text" id="inNome" class="form-control"
autofocus>
        </p>
        <p><label for="inPeso">Peso (em gramas): </label>
          <input type="text" id="inPeso" class="form-control">
        </p>
        <p><input type="button" value="Apostar" id="btApostar"
          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>
        <pre id="outApostas"></pre>
      </div>
    </div>
  </div>
</div>

```

```

<script src="js/ex8_3.js"></script>
</body>
</html>

```

A Figura 8.6 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 8.6 – Página inicial do jogo “Qual é o peso da Melancia?”

Vamos discutir a programação de cada botão separadamente. Começamos pela programação do botão **Apostar**, que deve ser inserida no arquivo `ex8_3.js`, na pasta `js`.

Programa JavaScript para cadastrar as apostas em localStorage (js/ex8_3.js)

```

function incluirAposta() {
    // cria referência aos elementos de entrada de dados da página
    var inNome = document.getElementById("inNome");
    var inPeso = document.getElementById("inPeso");

    var nome = inNome.value; // conteúdo do campo nome
    var peso = Number(inPeso.value); // conteúdo do campo peso (em
    número)

```

```

// se vazios ou Not-a-Number
if (nome == "" || peso == 0 || isNaN(peso)) {
    alert("Informe nome e peso da aposta"); // exibe alerta
    inNome.focus(); // joga cursor em nome
    return; // abandona execução da function
}
// chama function que verifica se peso já foi apostado
if (verApostaExiste(peso)) {
    alert("Alguém já apostou este peso, informe outro...");
    inPeso.focus();
    return;
}

// se houver dados salvos no localStorage
if (localStorage.getItem("melanciaNome")) {
    // obtém o conteúdo já salvo e acrescenta ";" + dados da aposta
    var melanciaNome = localStorage.getItem("melanciaNome") + ";" +
nome;
    var melanciaPeso = localStorage.getItem("melanciaPeso") + ";" +
peso;

    // salva os dados em localStorage
    localStorage.setItem("melanciaNome", melanciaNome);
    localStorage.setItem("melanciaPeso", melanciaPeso);
} else { // senão, é a primeira aposta (não tem ";")
    localStorage.setItem("melanciaNome", nome); // salva os dados
    localStorage.setItem("melanciaPeso", peso);
}

mostrarApostas(); // chama function que mostra as apostas já salvas

inNome.value = ""; // limpa os campos de formulário
inPeso.value = "";
inNome.focus(); // joga o foco (cursor) no campo inNome
}
// cria referência ao botão e associa a ocorrência do evento click à
function
var btApostar = document.getElementById("btApostar");
btApostar.addEventListener("click", incluirAposta);

```

A `function incluirAposta` inicia da mesma forma que as demais. Obtêm-se os elementos de entrada de dados e verifica-se se os campos foram preenchidos pelo usuário. 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 `function verApostaExiste()`, passando o peso como argumento. A programação da `function` é listada a seguir:

```
function verApostaExiste(peso) {  
  
    var existe = false; // valor inicial é false  
    // se existe algum dado salvo em localStorage...  
    if (localStorage.getItem("melanciaPeso")) {  
        // obtém seu conteúdo. A string é dividida em itens de vetor a  
        // cada ";"  
        var pesos = localStorage.getItem("melanciaPeso").split(";");  
  
        // verifica se existe com indexOf()  
        // 0 peso deve ser convertido em string, pois o vetor contém  
        // strings  
        if (pesos.indexOf(peso.toString()) >= 0) {  
            existe = true; // apenas neste caso troca o valor do "flag"  
        }  
    }  
  
    return existe; // retorna true ou false  
}
```

Para verificar se o peso já consta no `localStorage`, a função utiliza a variável `existe` que começa com o valor `false`. Se existirem dados salvos no `localStorage`, 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 `indexOf()`, que retorna a posição do item caso ele exista no vetor ou -1, caso ele não exista. 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()`. Dessa forma, se o retorno do método `indexOf()` for uma posição válida do vetor, significa que ele já foi apostado e troca-se o valor da variável `existe`

para `true`. No final do método, essa variável `existe` é então retornada para o programa que chamou a função.

Para finalizar a discussão sobre a `function incluirAposta()`, após a chamada da `function verApostaExiste()`, 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 `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. A `function mostrarApostas()` é apresentada na sequência.

```
function mostrarApostas() {
    // cria referência ao elemento que exibe as apostas
    var outApostas = document.getElementById("outApostas");

    // se não há apostas armazenadas em localStorage
    if (!localStorage.getItem("melanciaNome")) {
        // limpa o espaço de exibição das apostas (para quando "Limpar
        Apostas")
        outApostas.textContent = "";
        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 ";"
    var nomes = localStorage.getItem("melanciaNome").split(";");
    var pesos = localStorage.getItem("melanciaPeso").split(";");

    var linhas = ""; // irá acumular as linhas a serem exibidas

    // repetição para percorrer todos os elementos do vetor
    for (var i = 0; i < nomes.length; i++) {
        // concatena em linhas os nomes dos apostadores e suas apostas
```

```

    linhas += nomes[i] + " - " + pesos[i] + "gr \n";
}

// exibe as linhas (altera o conteúdo do elemento outApostas)
outApostas.textContent = linhas;
}
// chama a function quando a página é carregada, para mostrar
apostas salvas
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 4 e 5, 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 `ex8_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 `function`.

```

function verificarVencedor() {

    // 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
    var 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
var nomes = localStorage.getItem("melanciaNome").split(";");
var pesos = localStorage.getItem("melanciaPeso").split(";");

// valor inicial para vencedor é o da primeira aposta
var vencedorNome = nomes[0];
var vencedorPeso = Number(pesos[0]);

// percorre as apostas
for (var i = 1; i < nomes.length; i++) {

    // calcula a diferença de peso do "vencedor" e da aposta atual
    difVencedor = Math.abs(vencedorPeso - pesoCorreto);
    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
var mensagem = "Resultado - Peso Correto: " + pesoCorreto + "gr";
mensagem += "\n-----";
mensagem += "\nVencedor: " + vencedorNome;
mensagem += "\nAposta: " + vencedorPeso + "gr";
alert(mensagem);
}

var btVencedor = document.getElementById("btVencedor");
btVencedor.addEventListener("click", verificarVencedor);

```

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 8.7.

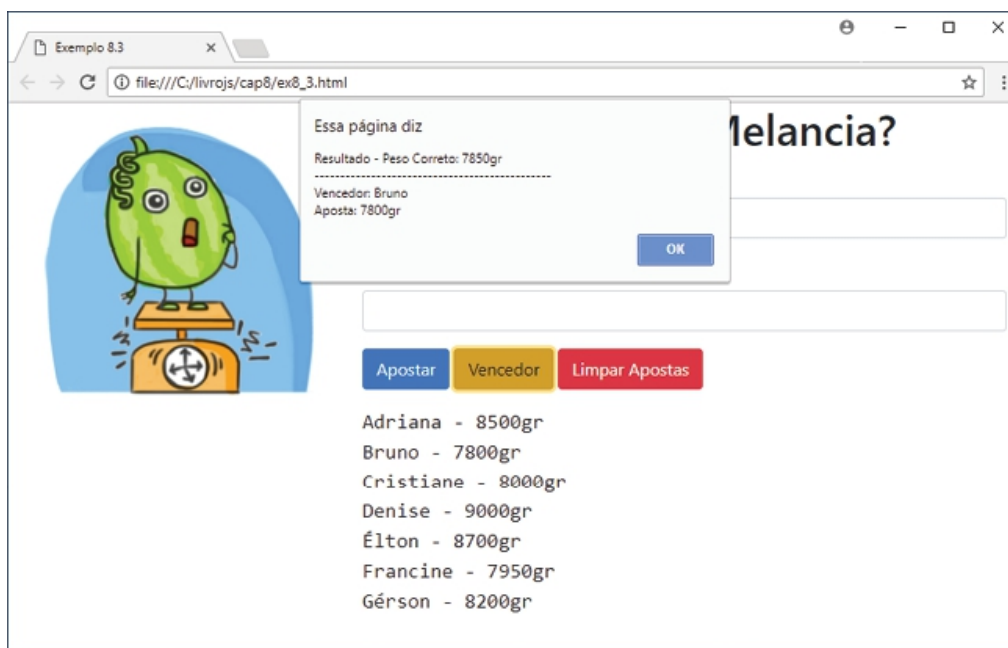


Figura 8.7 – Vencedor é quem acertou ou chegou mais perto do peso correto.

E, para finalizar, a função `limparApostas()` remove o conteúdo do `localStorage` e atualiza a listagem das apostas.

```
function limparApostas() {  
    // 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
}
}
var btLimpar = document.getElementById("btLimpar");
btLimpar.addEventListener("click", limparApostas);

```

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.

8.6 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 8.8 exibe a ilustração da página com alguns dados inseridos.



Figura 8.8 – 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 8.9 ilustra a página do sistema.



Figura 8.9 – Os serviços pendentes devem ser salvos no navegador e removidos a cada execução.

8.7 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.

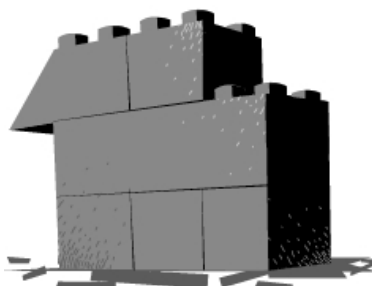
A linguagem JavaScript dispõe dos métodos `localStorage.setItem("chave", valor)` para gravar um conteúdo no Local Storage e `localStorage.getItem("chave")` para recuperar esse conteúdo. Também é possível excluir os dados salvos pelo cliente em seu navegador, a partir dos métodos `localStorage.removeItem("chave")` 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 9

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 é amplo e foge do escopo do livro. Apenas para exemplificar o momento em que os dados podem ser enviados para o servidor, 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 final deste livro, 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 9</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 9.1 ilustra a estruturação hierárquica da árvore DOM das tags da página HTML anterior.

Observe, a partir da Figura 9.1, 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 `div`s. 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 `getElementById()` 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.

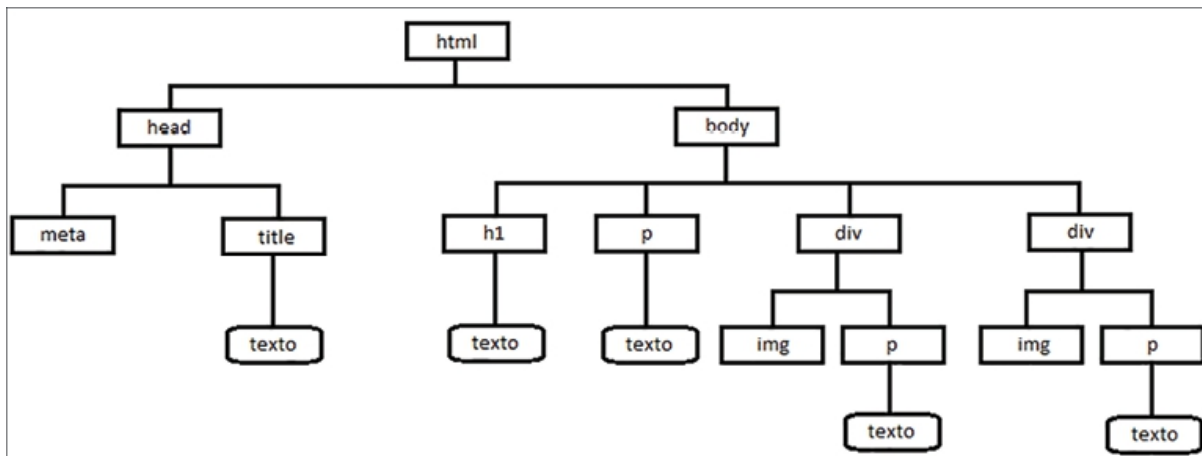


Figura 9.1 – Estrutura da árvore DOM do documento HTML anterior.

9.1 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 **cap9** 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 9.2 ilustra a página web do programa “Tarefas do Dia”, já com algumas tarefas adicionadas.

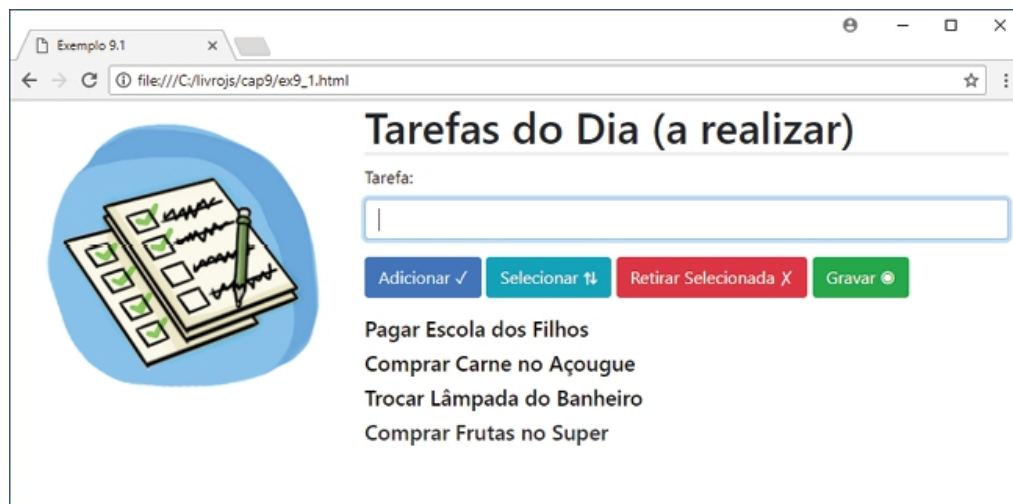


Figura 9.2 – As tarefas são tags *h5* inseridas no documento pelo programa JavaScript.

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; }
img.imgProg { max-width: 100%; }
.tarefaSeleccionada { color: red; font-style: italic; }
.tarefaNormal { 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; }  
.aDireita { text-align: right; }
```

A pasta `css` deve também conter o arquivo `bootstrap.min.css`. Você pode copiá-lo dessa mesma pasta do `cap8`. 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 `ex9_1.html`.

Exemplo 9.1 – Página HTML do programa Tarefas do Dia (`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">  
  <meta http-equiv="X-UA-Compatible" content="ie=edge">  
  <link rel="stylesheet" href="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">  
      <div class="col-sm-4">  
          
      </div>  
  
      <div class="col-sm-8" id="divQuadro">  
        <h1>Tarefas do Dia (a realizar)</h1>  
        <p>  
          <label for="inTarefa">Tarefa:</label>  
          <input type="text" class="form-control" id="inTarefa"  
autofocus>  
        </p>  
        <p>  
          <input type="button" class="btn btn-primary"  
            value="Adicionar &#10003;" id="btAdicionar">  
          <input type="button" class="btn btn-info"  
            value="Selecionar &#8645;" id="btSelecionar">  
        </p>  
      </div>  
    </div>  
  </div>
```

```

        <input type="button" class="btn btn-danger"
            value="Retirar Seleccionada &#10007;" id="btRetirar">
        <input type="button" class="btn btn-success"
            value="Gravar &#9673;" id="btGravar">
    </p>
</div>
</div>
</div>
<script src="js/ex9_1.js"></script>
</body>
</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 do botão **Adicionar** que contém os métodos destacados nesta seção. Crie o arquivo `ex9_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/ex9_1.js`)

```

function adicionarTarefa() {
    // cria referência ao campo de entrada de dados
    var inTarefa = document.getElementById("inTarefa");

    var tarefa = inTarefa.value; // obtém o conteúdo digitado

    // se não informou ...
    if (tarefa == "") {
        alert("Informe a tarefa"); // exibe alerta
        inTarefa.focus(); // posiciona no campo inTarefa
        return; // retorna
    }

    // cria referência ao elemento divQuadro (local onde tag h5 será
    inserida)
    var divQuadro = document.getElementById("divQuadro");

```

```

var h5 = document.createElement("h5"); // cria o elemento HTML h5
var texto = document.createTextNode(tarefa); // cria um texto
h5.appendChild(texto); // define que o texto será filho de h5
divQuadro.appendChild(h5); // ...e que h5 será filho de divQuadro

inTarefa.value = ""; // limpa o campo de edição
inTarefa.focus(); // joga o cursor neste campo
}
// cria referência ao btAdicionar e após associa evento à função
var btAdicionar = document.getElementById("btAdicionar");
btAdicionar.addEventListener("click", adicionarTarefa);

```

A função inicia pela validação do preenchimento do campo de texto. Na sequência, aparecem os comandos que inserem 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. Começamos pela captura do elemento da página em que as novas tags serão inseridas.

```
var divQuadro = document.getElementById("divQuadro");
```

Em seguida, cria-se a tag `h5` e o texto a ser exibido na página.

```
var h5 = document.createElement("h5"); // cria o elemento HTML h5
var 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 o texto será filho de h5
divQuadro.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 9.2.

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 `getElementsByTagName()`, discutido no capítulo anterior. Um clique no botão **Selecionar** vai destacar uma tarefa, alterando a classe da tag `h5` para `tarefaSelecionada`. 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 9.3 exemplifica a execução da `function selecionarTarefa()`, apresentada na sequência.



Figura 9.3 – Um clique no botão *Selecionar* destaca uma tarefa, que pode, então, ser removida.

Acrescente ao arquivo `ex9_1.js` a programação a seguir, que ficará vinculada ao evento click do botão `btSelecionar`.

```
function selecionarTarefa() {
    var h5 = document.getElementsByTagName("h5"); // obtém tags h5 da
    página
    var numH5 = h5.length; // obtém número de tags h5

    if (numH5 == 0) { // se zero...
        alert("Não há tarefas para selecionar"); // exibe alerta
        return; // retorna
    }

    var aux = -1; // variável auxiliar para indicar linha selecionada

    // percorre a lista de elementos h5 inseridos na página
    for (var i = 0; i < numH5; i++) {
        // se tag é da class tarefaSeleccionada (está selecionada)
        if (h5[i].className == "tarefaSeleccionada") {
            h5[i].className = "tarefaNormal"; // 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 == numH5 - 1) {
        aux = -1;
    }

    h5[aux + 1].className = "tarefaSelecionada"; // muda estilo da
    próxima tag h5
}
var btSelecionar = document.getElementById("btSelecionar");
btSelecionar.addEventListener("click", selecionarTarefa);

```

Observe que a `function` inicia pela captura das tags `h5` inseridas na página. Como o método `getElementsByTagName()` 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 `tarefaSelecionada`, 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 `function` é listada a seguir.

```

function retirarSelecionada() {
    // cria referência ao elemento que irá "perder" um filho
    var divQuadro = document.getElementById("divQuadro");
    var h5 = document.getElementsByTagName("h5"); // obtém tags h5 da
    página
    var numH5 = h5.length; // obtém quantidade de h5

```



```

var aux = -1; // variável auxiliar para indicar linha selecionada

// percorre a lista de elementos h5 inseridos na página
for (var i = 0; i < numH5; i++) {
    // verifica className da tag h5
    if (h5[i].className == "tarefaSelecionada") { // se selecionada
        aux = i; // muda valor da variável aux
        break; // sai da repetição
    }
}

// se não há tarefa selecionada (ou se lista vazia...)
if (aux == -1) {
    alert("Selecione uma tarefa para removê-la...");
    return;
}

// solicita confirmação (exibindo o conteúdo da tag h5 selecionada)
if (confirm("Confirma Exclusão de '" + h5[aux].textContent + "'?"))
{
    divQuadro.removeChild(h5[aux]); // remove um dos filhos de
divQuadro
}
}

var btRetirar = document.getElementById("btRetirar");
btRetirar.addEventListener("click", retirarSelecionada);

```

Novamente, nessa *function*, é preciso percorrer as tags h5 da página para verificar qual delas está selecionada, ou seja, a propriedade `className` igual a `tarefaSelecionada`. 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 `textContent`. A Figura 9.4 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 94 – O conteúdo da tag selecionada é recuperado pela function *removerSeleccionada*.

O quarto e último botão desse programa é o **Gravar**. Ele aciona a function `gravarTarefas()` que recupera o conteúdo das tags `h5` exibidas na página e as armazena na `localStorage`. Acrescenta-se entre as tarefas o delimitador “;”. Adicione as seguintes linhas ao arquivo `ex9_1.js` a fim de que o programa execute também essa ação.

```
function gravarTarefas() {
    var h5 = document.getElementsByTagName("h5"); // obtém tags h5 da
    página
    var numH5 = h5.length; // obtém quantidade de h5

    if (numH5 == 0) { // se zero...
        alert("Não há tarefas para serem salvas"); // exibe alerta
        return; // retorna
    }

    var tarefas = ""; // irá "acumular" as tarefas

    // percorre a lista de elementos h5 inseridos na página
    for (var i = 0; i < numH5; i++) {
        tarefas += h5[i].textContent + ";"; // acumula conteúdo de cada h5
    }
    // grava as tarefas em localStorage, removendo o último ";"
    localStorage.setItem("tarefasDia", tarefas.substr(0, tarefas.length
- 1));
    // confere se dados foram armazenados em localStorage
```

```

    if (localStorage.getItem("tarefasDia")) {
        alert("Ok! Tarefas Salvas");
    }
}
var btGravar = document.getElementById("btGravar");
btGravar.addEventListener("click", gravarTarefas);

```

Para o nosso programa ficar completo, falta acrescentar no arquivo `ex9_1.js` a function `recuperarTarefas()` que será executada quando a página for carregada. Ela verifica 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, na function `adicionarTarefa()`.

```

function recuperarTarefasSalvas() {
    // verifica se há tarefas salvas no navegador do usuário
    if (localStorage.getItem("tarefasDia")) {
        // cria um vetor com a lista de tarefas (separadas pelo
split(";"))
        var tarefas = localStorage.getItem("tarefasDia").split(";");

        // cria referência ao divQuadro (local onde as tags h5 serão
inseridas)
        var divQuadro = document.getElementById("divQuadro");

        // percorre todas as tarefas
        for (var i = 0; i < tarefas.length; i++) {
            var h5 = document.createElement("h5"); // cria o elemento HTML
h5
            var texto = document.createTextNode(tarefas[i]); // cria um
texto
            h5.appendChild(texto); // define que o texto será filho de h5
            divQuadro.appendChild(h5); // ...e que o h5 será filho de
divQuadro
        }
    }
}
recuperarTarefasSalvas();

```

9.2 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:

```
var body = document.getElementsByTagName("body")[0];
```

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 9.5 exibe a página inicial do programa.

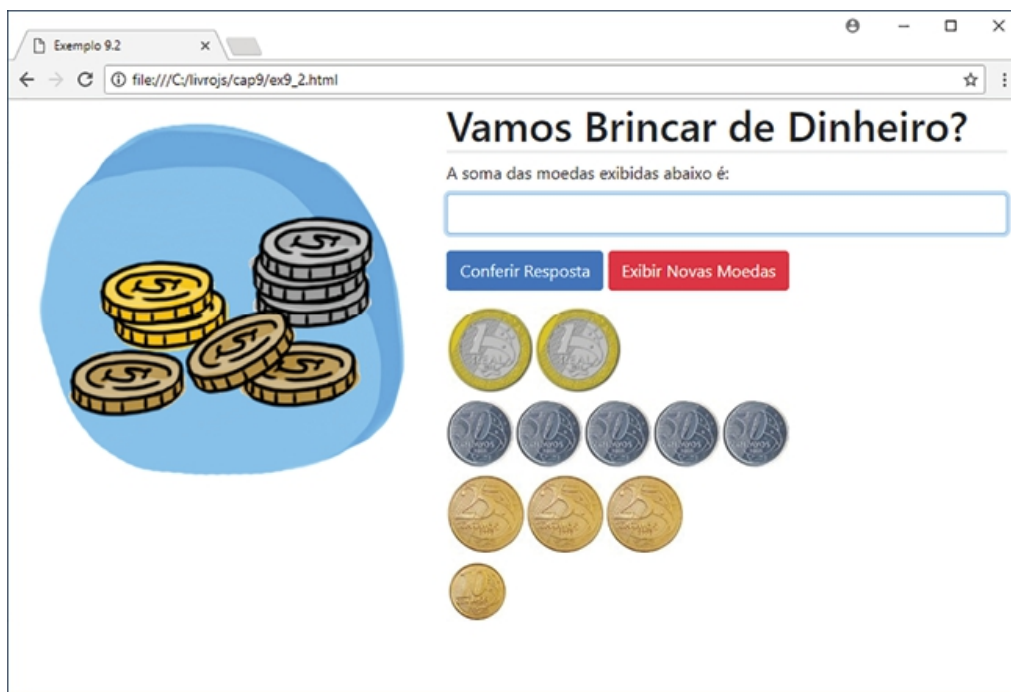


Figura 9.5 – As imagens das moedas são inseridas em número aleatório pelo programa

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 `ex9_2.html`.

Exemplo 9.2 – Página HTML “Vamos Brincar de Dinheiro?” (`ex9_2.html`)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 9.2</title>
</head>

<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-sm-4">
        
      </div>
      <div class="col-sm-8" id="divMoedas">
        <h1>Vamos Brincar de Dinheiro?</h1>
        <p>
          <label for="inSoma">A soma das moedas exibidas abaixo é:
        </label>
          <input type="text" class="form-control" id="inSoma" autofocus>
        </p>
        <p>
          <input type="button" class="btn btn-primary"
            value="Conferir Resposta" id="btConferir">
          <input type="button" class="btn btn-danger"
            value="Exibir Novas Moedas" id="btExibir">
        </p>
      </div>
    </div>
  </div>
</body>
</html>
```

```

        </p>
    </div>
</div>
</div>
<script src="js/ex9_2.js"></script>
</body>
</html>

```

Para inserir as moedas na página, crie o arquivo `ex9_2.js`. Ele contém algumas `functions`. Vamos apresentar, inicialmente, a `function` `exibirMoedas()`.

Programa JavaScript que insere imagens na página (`js/ex9_2.js`)

```

function exibirMoedas() {
    // gera números aleatórios, entre 1 e 5, para cada moeda
    var num1_00 = Math.ceil(Math.random() * 5);
    var num0_50 = Math.ceil(Math.random() * 5);
    var num0_25 = Math.ceil(Math.random() * 5);
    var num0_10 = Math.ceil(Math.random() * 5);
    // cria referência ao local onde as imagens serão inseridas
    var divMoedas = document.getElementById("divMoedas");

    // texto alternativo das imagens (para questões de acessibilidade)
    var alt1_00 = "Moedas de um real";
    var alt0_50 = "Moedas de Cinquenta Centavos";
    var alt0_25 = "Moedas de Vinte e Cinco Centavos";
    var alt0_10 = "Moedas de Dez Centavos";

    // chama o método criarMoedas passando os argumentos
    criarMoedas(num1_00, divMoedas, "1_00.jpg", alt1_00, "moeda1_00");
    criarMoedas(num0_50, divMoedas, "0_50.jpg", alt0_50, "moeda0_50");
    criarMoedas(num0_25, divMoedas, "0_25.jpg", alt0_25, "moeda0_25");
    criarMoedas(num0_10, divMoedas, "0_10.jpg", alt0_10, "moeda0_10");
}

exibirMoedas(); // chama o método exibirMoedas

```

A `function` `exibirMoedas()` começa realizando 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 que devem ser inseridas na página. 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.

Em seguida, cria-se uma referência a `div` identificada na página pelo `id=divMoedas`. É nesse container, ou melhor, no final dele, que as imagens serão inseridas. 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 a `function criarMoedas()` passando os argumentos relativos a cada moeda que deve ser criada. Essa `function` é destacada a seguir:

```
function criarMoedas(num, pai, moeda, textoAlt, classe) {  
    // cria laço de repetição para inserir várias imagens de moedas na  
    página  
    for (var i = 1; i <= num; i++) {  
        var novaMoeda = document.createElement("img"); // cria elemento  
        img  
        novaMoeda.src = "img/" + moeda; // atributo src  
        novaMoeda.textoAlt = textoAlt; // texto alternativo  
        novaMoeda.className = classe; // classe da moeda(css)  
        pai.appendChild(novaMoeda); // hierarquia DOM  
    }  
    var br = document.createElement("br"); // cria uma quebra de linha  
    (br)  
    pai.appendChild(br);  
}
```

Observe que a `function` possui cinco 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 9.5.

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 `ex9_2.js`.

```
function conferirSoma() {
    // cria referência ao campo de entrada e obtém o seu conteúdo
    var inSoma = document.getElementById("inSoma");
    var soma = Number(inSoma.value);

    // valida o preenchimento do campo
    if (soma == 0 || isNaN(soma)) {
        alert("Informe o valor da soma (use ponto para separar
decimais)");
        inSoma.focus();
        return;
    }

    // captura divMoedas que é o local onde as moedas foram inseridas
    var divMoedas = document.getElementById("divMoedas");
    // captura as tags img filhas de divMoedas
    var moedas = divMoedas.getElementsByTagName("img");

    var totalMoedas = 0; // declara e inicializa acumulador

    // percorre as tags img e verifica propriedade className
    for (var i = 0; i < moedas.length; i++) {
        if (moedas[i].className == "moeda1_00") {
            totalMoedas += 1; // acumula 1 (para moedas de 1)
        } else if (moedas[i].className == "moeda0_50") {
            totalMoedas += 0.50; // acumula 0.50 (para moedas de 0.50)
        } else if (moedas[i].className == "moeda0_25") {
            totalMoedas += 0.25; // acumula 0.25 (para moedas de 0.25)
        } else {
            totalMoedas += 0.10; // acumula 0.10 (para moedas de 0.10)
        }
    }

    var div = document.createElement("div"); // cria elemento div
    var h3 = document.createElement("h3"); // cria elemento h3
```



```

// 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
    var mensagem = "Parabéns!! Você acertou!"; // e mensagem a ser
exibida
} else {
    div.className = "alert alert-danger";
    var mensagem = "Ops... A resposta correta é " +
totalMoedas.toFixed(2);
}
var 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
divMoedas.appendChild(div); // e a div com alerta é filha de
divMoedas

    btConferir.disabled = true; // desabilita botão (resposta já foi
exibida)
}
var btConferir = document.getElementById("btConferir");
btConferir.addEventListener("click", conferirSoma);

```

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 na `function` `exibirMoedas()` e comparar esse valor na `function` `conferirSoma()`. Como o objetivo deste capítulo é manipular os elementos da página, optamos por fazer de uma forma diferente, sem utilizar a variável global.

Na `function` `conferirSoma()`, obtemos todas as imagens filhas de `divMoedas`, com o uso do método `getElementsByTagName()`. 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 9.6 exibe a caixa de mensagem, com um exemplo de anúncio de resposta incorreta e exibição do valor total das moedas.

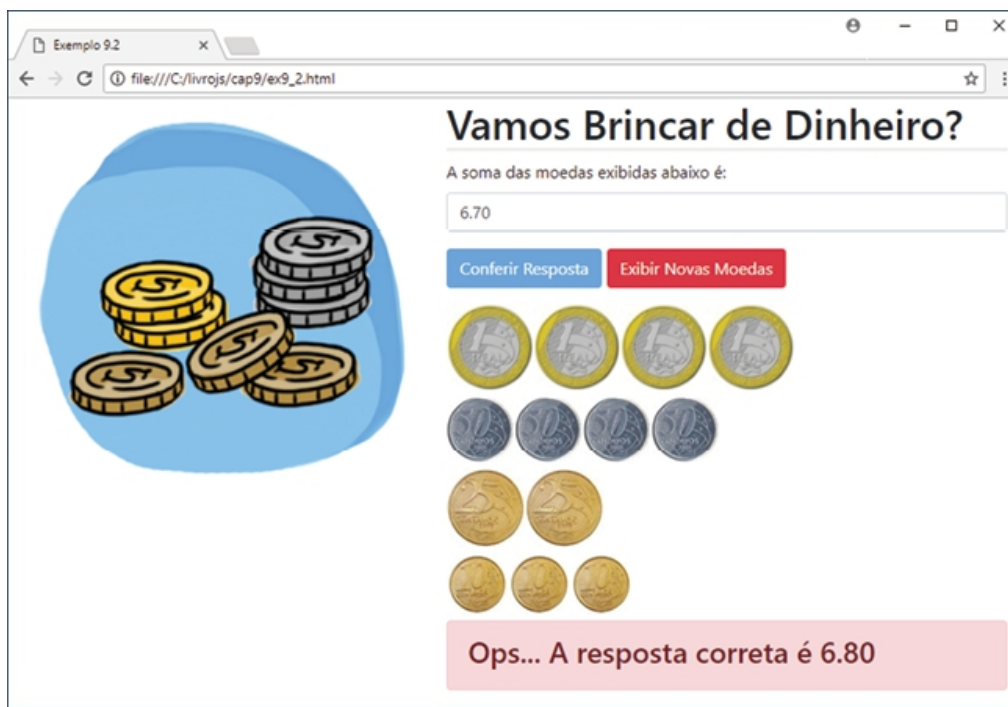


Figura 9.6 – Caixa de alerta do Bootstrap é criada na página para exibir mensagem. A cor da caixa muda de acordo com a resposta (correta ou incorreta).

Para concluir esse exemplo, falta digitar a programação associada ao evento click do botão **Exibir Novas Moedas**. Ele pode ser criado com uma função anônima que realiza um `reload` da página.

```
var btExibir = document.getElementById("btExibir");
btExibir.addEventListener("click", function() {
    location.reload();
});
```

9.3 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 (`num`) 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. O terceiro programa deste capítulo deve cadastrar filmes na `localStorage` e exibir uma tabela contendo título, gênero e uma coluna que permita ao usuário selecionar o filme para exclusão. A Figura 9.7 exibe a página desse programa com alguns filmes cadastrados.

Crie um novo arquivo HTML e salve-o com o nome de `ex9_3.html`. O código a ser inserido nesse arquivo é descrito no Exemplo 9.3.

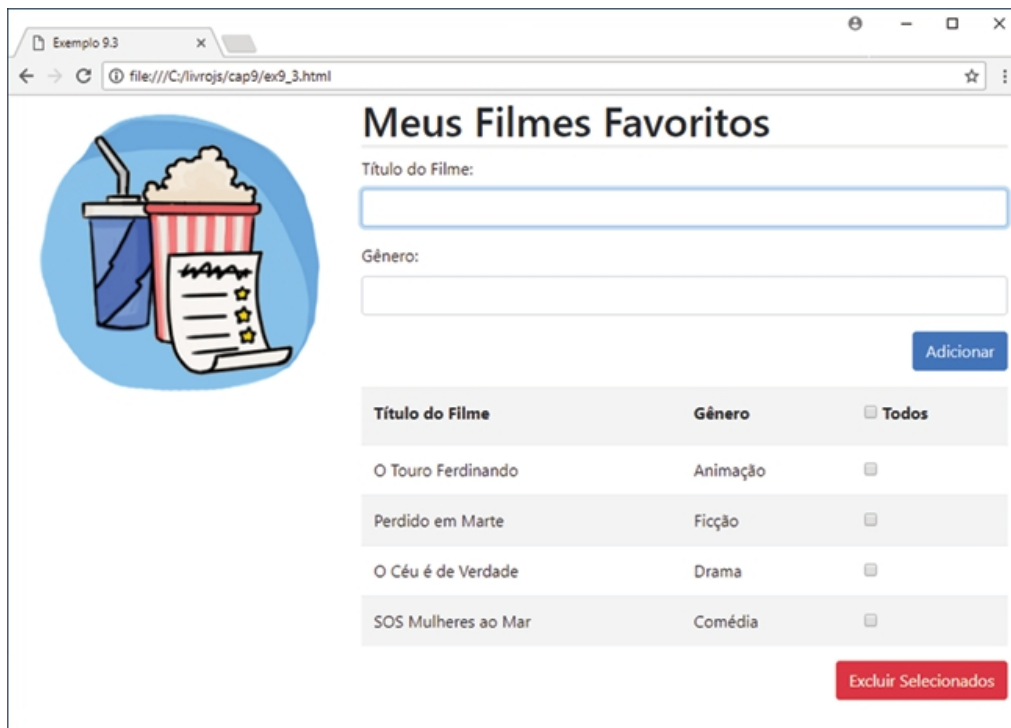


Figura 9.7 – Os filmes salvos em localStorage são exibidos em uma tabela na página.

Exemplo 9.3 – Página HTML que vai exibir uma tabela de filmes (ex9_3.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 9.3</title>
</head>

<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-sm-4">
        
      </div>
      <div class="col-sm-8">
```

```

<h1>Meus Filmes Favoritos</h1>
<p>
  <label for="inTitulo">Título do Filme:</label>
  <input type="text" class="form-control" id="inTitulo">
</p>
<p>
  <label for="inGenero">Gênero:</label>
  <input type="text" class="form-control" id="inGenero">
</p>
<p class="aDireita">
  <input type="button" class="btn btn-primary"
    value="Adicionar" id="btAdicionar">
</p>

<table class="table table-striped" id="tbFilmes">
  <tr>
    <th>Título do Filme</th>
    <th>Gênero</th>
    <th>
      <label><input type="checkbox" id="ckTodos"> Todos </label>
    </th>
  </tr>
</table>

<p class="aDireita">
  <input type="button" class="btn btn-danger"
    value="Excluir Seleccionados" id="btExcluir">
</p>
</div>
</div>
</div>
<script src="js/ex9_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 `ex9_3.js`. Como o programa é grande, vamos apresentar suas funções em partes. Começamos pela function `adicionarFilme()`.

Programa JavaScript que insere linhas e colunas em uma tabela (`js/ex9_3.js`)

```
function adicionarFilme() {  
    // cria referência aos campos de entrada  
    var inTitulo = document.getElementById("inTitulo");  
    var inGenero = document.getElementById("inGenero");  
  
    var titulo = inTitulo.value; // obtém conteúdo dos campos  
    var genero = inGenero.value;  
  
    // valida preenchimento  
    if (titulo == "" || genero == "") {  
        alert("Informe corretamente os dados");  
        inTitulo.focus();  
        return;  
    }  
  
    // cria referência ao elemento tbFilmes  
    var tbFilmes = document.getElementById("tbFilmes");  
  
    // chama function que irá inserir filme na tabela  
    inserirLinha(tbFilmes, titulo, genero);  
  
    // chama function que irá gravar dados em localStorage  
    gravarFilme(titulo, genero);  
  
    inTitulo.value = ""; // limpa campos de entrada  
    inGenero.value = "";  
    inTitulo.focus(); // posiciona o cursor em inTitulo  
}  
var btAdicionar = document.getElementById("btAdicionar");  
btAdicionar.addEventListener("click", adicionarFilme);
```

Essa função cria referência aos campos de entrada, valida o seu preenchimento e delega as tarefas de inserir uma nova linha na tabela e um novo filme na localStorage às funções `inserirLinha()` e `gravarFilme()`. A function `inserirLinha()` contém as principais novidades dessa seção.

Acrescente esse código no arquivo `ex9_3.js`.

```
function inserirLinha(tabela, titulo, genero) {  
  
    var linha = tabela.insertRow(-1); // adiciona uma linha na tabela  
  
    var col1 = linha.insertCell(0); // cria colunas na linha inserida  
    var col2 = linha.insertCell(1);  
    var col3 = linha.insertCell(2);  
  
    col1.textContent = titulo; // joga um conteúdo em cada célula  
    col2.textContent = genero;  
    col3.innerHTML = "<input type='checkbox'>"; // innerHTML renderiza  
    código  
}
```

Observe que a `function` recebe os parâmetros `tabela`, `titulo` e `genero`, sendo `tabela` uma referência à `tbFilmes` que é a `table` existente no código HTML. Nela, acrescentamos uma linha com o método `insertRow()`, em que `-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 o campo tipo `checkbox` seja renderizado na terceira coluna da tabela, utilizamos a propriedade `innerHTML` discutida na seção 2.5. 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 `textContent` e `innerHTML`, faça o seguinte teste: substitua a `innerHTML` por `textContent` e rode o programa. Você vai observar que com `textContent` o navegador exibe o texto `<input type=...>`, e não uma caixa de seleção para o usuário selecionar.

Vamos então à segunda `function` chamada pelo `adicionarFilme()`. Ela recebe os parâmetros `titulo` e `genero` e os grava em `localStorage`. Para separar os filmes, é utilizado o delimitador “;”.

```
function gravarFilme(titulo, genero) {  
  
    // se há filmes salvos em localStorage ...
```

```

    if (localStorage.getItem("filmesTitulo")) {
        // ... obtém os dados e acrescenta ";" e o título/gênero informado
        var filmesTitulo = localStorage.getItem("filmesTitulo") + ";" +
        titulo;
        var 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.

```

function recuperarFilmes() {

    // se houver dados salvos em localStorage
    if (localStorage.getItem("filmesTitulo")) {
        // obtém conteúdo e converte em elementos de vetor (na ocorrência
        ";")
        var titulos = localStorage.getItem("filmesTitulo").split(";");
        var generos = localStorage.getItem("filmesGenero").split(";");

        // cria referência ao elemento tbFilmes
        var tbFilmes = document.getElementById("tbFilmes");

        // percorre elementos do vetor e os insere na tabela
        for (var i = 0; i < titulos.length; i++) {
            inserirLinha(tbFilmes, titulos[i], generos[i]);
        }
    }
}

```



```

    }
}
recuperarFilmes();

```

A function `recuperarFilmes()` verifica se há dados salvos em `localStorage` e, se houver, joga o conteúdo de cada campo em elementos de vetor. Em seguida, cria referência à tabela `tbFilmes` para que na repetição possa também chamar a function `inserirLinha()` que fará a exibição dos dados na página. 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.

As funções discutidas a seguir permitem remover os filmes da tabela e também da `localStorage`. Começamos pela discussão da function anônima associada ao evento `change` do checkbox `ckTodos`. Ao clicar sobre ele, todos os filmes são marcados ou desmarcados, conforme ilustra a Figura 9.8.

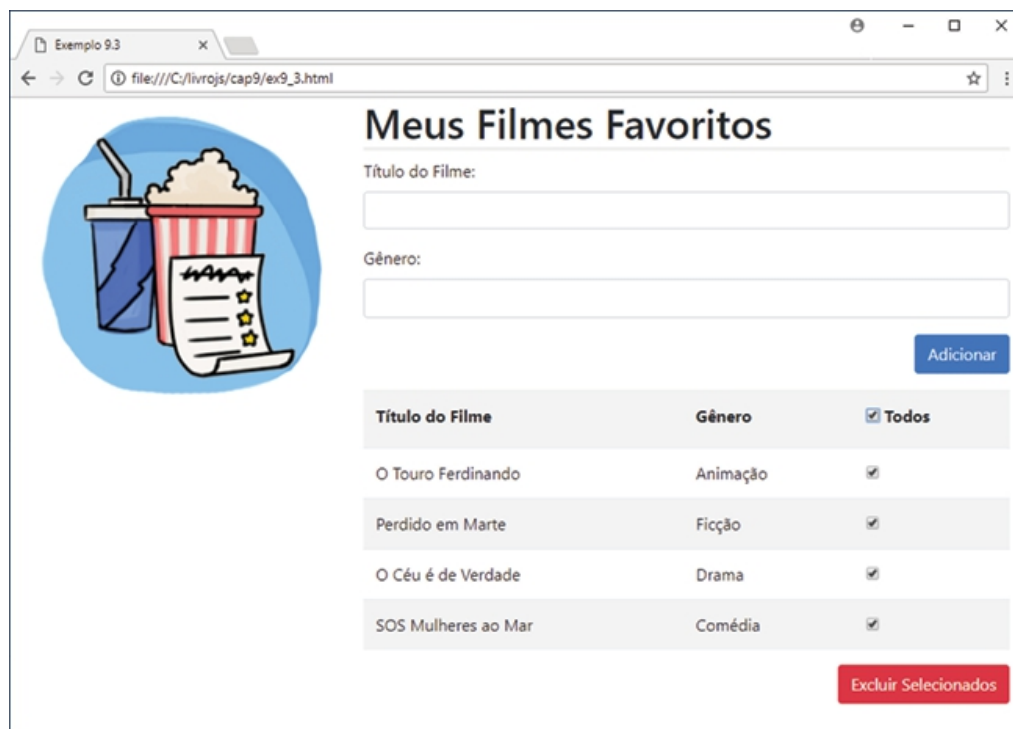


Figura 9.8 – Ao clicar em “Todos”, os filmes exibidos na tabela são marcados ou desmarcados.

Acrescente as seguintes linhas ao programa `ex9_3.js`. Então, faça o teste e observe que, quando clicado, o status de `ckTodos` é atribuído a todos os filmes da tabela.

```

// cria referência ao checkbox ckTodos (na linha de título da
tabela)
var ckTodos = document.getElementById("ckTodos");
// executa função anônima quando houver uma troca de status
ckTodos.addEventListener("change", function () {
    // cria referência à tabela e aos campos input (filhos da tabela)
    var tbFilmes = document.getElementById("tbFilmes");
    var ckExcluir = tbFilmes.getElementsByTagName("input");

    var status = ckTodos.checked; // obtém status de ckTodos ...

    // ... e percorre os demais checkbox para aplicar este status
    for (var i = 1; i < ckExcluir.length; i++) {
        ckExcluir[i].checked = status;
    }
});

```

Naturalmente, esse código poderia ficar em uma função nomeada como as demais desse programa. Vamos retomando alguns recursos discutidos ao longo do livro para lembrá-los. Sobre a `function`, observe que é criada uma referência à tag `table` `tbFilmes` e, em seguida, todos os campos `input` filhos de `tbFilmes` são recuperados com o método `getElementsByTagName()`. Na sequência do programa, obtém-se o status de `ckTodos` (marcado ou desmarcado) para aplicá-lo a todos os `inputs` da tabela, a partir de uma estrutura de repetição. Perceba que no `for` a variável `i` inicia com o valor 1, pois o `input` de índice 0 é o próprio `ckTodos`.

A `function` `removerFilme()` é um pouco mais complexa, pois envolve as ações de remover o filme da `localStorage` e também retirar uma linha da tabela. Acrescente essa última `function` no arquivo `ex9_3.js` para concluir o nosso programa de cadastro de filmes.

```

function removerFilmes() {
    // cria referência à tabela e aos campos input (filhos da tabela)
    var tbFilmes = document.getElementById("tbFilmes");
    var ckExcluir = tbFilmes.getElementsByTagName("input");

    var temSelecionado = false; // para verificar se há filmes
    selecionados

```

```

// percorre campos input type checkbox da tabela (exceto "Todos" no
título)
for (var i = 1; i < ckExcluir.length; i++) {
    if (ckExcluir[i].checked) { // se está selecionado
        temSelecionado = true; // muda valor da "flag"
        break; // sai da repetição
    }
}

// se não temSelecionado (se valor da variável é false)
if (!temSelecionado) {
    alert("Não há filmes selecionados para exclusão");
    return;
}

// solicita confirmação de exclusão dos filmes selecionados
if (confirm("Confirma Exclusão dos Filmes Selecionados?")) {

    // exclui conteúdo armazenado em localStorage
    localStorage.removeItem("filmesTitulo");
    localStorage.removeItem("filmesGenero");

    // primeiro irá gravar em localStorage os filmes não selecionados
    for (i = 1; i < ckExcluir.length; i++) {
        // se não está selecionado (para exclusão)
        if (!ckExcluir[i].checked) {
            // obtém o conteúdo da tabela (coluna 0: título; coluna 1:
gênero)
            var titulo = tbFilmes.rows[i].cells[0].textContent;
            var genero = tbFilmes.rows[i].cells[1].textContent;
            gravarFilme(titulo, genero); // chama gravarFilme com dados da
tabela
        }
    }

    // agora irá remover as linhas selecionadas (do fim para o início)
    for (i = ckExcluir.length - 1; i > 0; i--) {
        if (ckExcluir[i].checked) {
            tbFilmes.deleteRow(i); // remove a linha

```

```

    }
}

    ckExcluir[0].checked = false; // desmarca ckTodos (que é o input
0)
}
}
var btExcluir = document.getElementById("btExcluir");
btExcluir.addEventListener("click", removerFilmes);

```

As duas primeiras linhas de código da `function removerFilmes()` são idênticas às da função anônima discutida anteriormente, pois aqui também é necessário recuperar os campos `input` filhos de `tbFilmes`. Em seguida, temos um conjunto de comandos utilizados para verificar se há algum filme selecionado para exclusão, a fim de alertar o usuário e evitar a execução desnecessária de comandos.

O processo de exclusão ocorre se o usuário confirmar a execução dessa tarefa no `if` contendo o método `confirm()`. Para isso, primeiro se removem as variáveis armazenadas em `localStorage` (o usuário poderia ter selecionado todos os filmes para exclusão). Na sequência, obtém-se o conteúdo das células da tabela das linhas não marcadas 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.

O terceiro e último `for` dessa `function` visa retirar os filmes exibidos na tabela. Um detalhe importante nessa etapa é que a repetição deve ocorrer do final para o início, pois os índices dos elementos são alterados dinamicamente quando se remove um deles. Assim, se excluirmos primeiro o filme de índice 0, todos os demais sobem uma posição e a exclusão não funcionaria corretamente se mais de um filme estivesse selecionado.

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.

Estamos nos aproximando do final do livro. 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.

9.4 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 9.9 exibe a página com um exemplo de velas inseridas pelo programa.



Figura 9.9 – 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 9.10 contém um exemplo de execução desse programa.

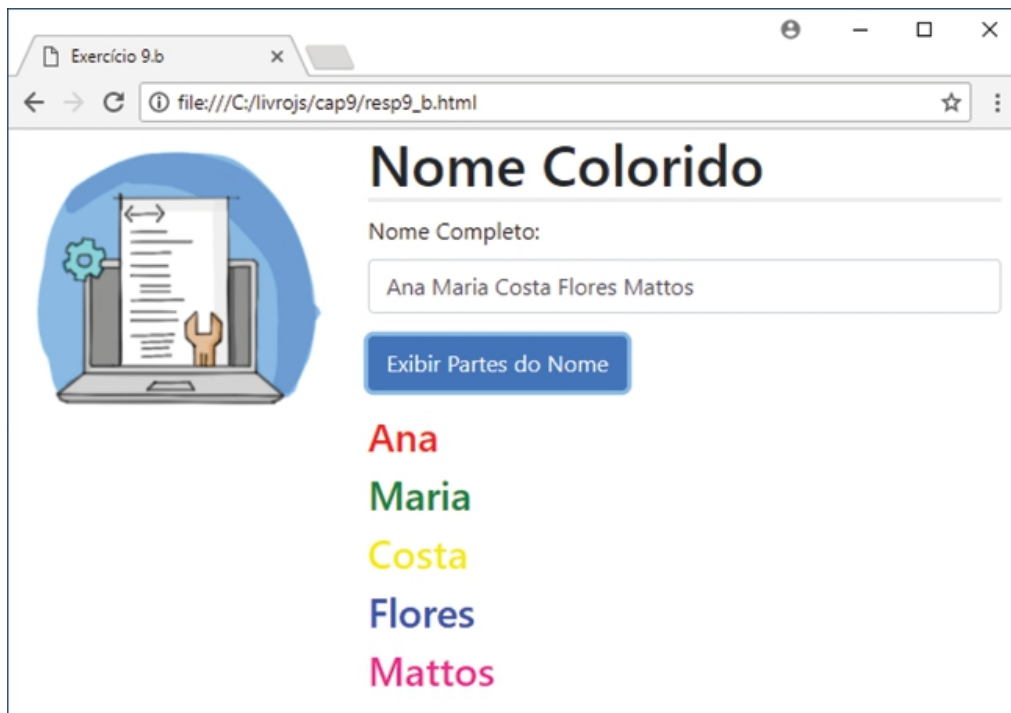


Figura 9.10 – As partes do nome são tags `h3` inseridas na página com cores aleatórias.

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 9.11 ilustra uma execução desse programa.



Figura 9.11 – O programa deve inserir os clubes em tags *h5* e, em seguida, exibir os jogos em uma tabela.

9.5 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.

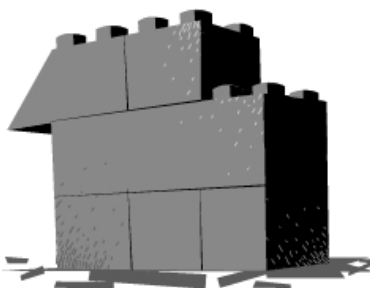
Os métodos JavaScript `createElement()` e `appendChild()` são os responsáveis para criar uma nova tag no documento e acrescentar um “filho” a um determinado elemento HTML da página, respectivamente. Para tags de texto, como parágrafos ou linhas de cabeçalho, é preciso utilizar o método `createTextNode()` 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 10

É muito bom programar... Programe!



Chegamos ao Capítulo 10, 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 pizzeria, 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 4, 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.

10.1 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 10.1 apresenta a página inicial do programa. Como nos demais capítulos, crie a pasta **cap10** e dentro dela adicione as pastas **css**, **img** e **js**. Lembre-se de inserir o arquivo **bootstrap.min.css** na pasta **css**. Além dele, essa pasta deve conter o arquivo **estilos.css** com o conteúdo descrito a seguir:

```
h1 { border-bottom-style: inset; }
h2 { margin-top: 12px; }
h5 { font-style: italic; margin-top: 40px;}
pre { font-style: italic; }
figure { display: inline-block; font-size: 0.7em; margin: 5px; }
img.imgProg { max-width: 100%; }
img.poltrona { width: 24; height: 16px; }
.aDireita { text-align: right; }
.margemSup { margin-top: 30px; }
.margemEsq { margin-left: 15px; }
.oculta { display: none; }
.exibe { display: inline; }
.margemJogo { margin-top: 15px; }
.corFundo { background-color: lightskyblue; }
.entreLetras { letter-spacing: 0.5em; }
.margemDica { padding: 12px; }
.fonteAzul { color: blue; }
.fonteVermelho { color: red; }
```



Figura 10.1 – Página inicial do programa Jockey Club.

O código HTML para exibir a página ilustrada na Figura 10.1 contém

apenas uma novidade em relação aos exemplos anteriores. Crie o arquivo `ex10_1.html`, insira esse código HTML no arquivo e, em seguida, vamos discutir sobre as tags dessa página.

Exemplo 10.1 – Página para controle de apostas de um Jockey Club (`ex10_1.html`)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 10.1</title>
</head>

<body>
  <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>

        <div class="row">
          <div class="col-sm-6">
            <p>
              <label for="inCavalo">Nº Cavalo:</label>
              <input type="text" class="form-control aDireita"
                id="inCavalo" autofocus>
            </p>
          </div>
          <div class="col-sm-6">
            <h5 id="outCavalo"></h5>
          </div>
        </div>
      </div>
    </div>
  </div>
```

```

<div class="row">
  <div class="col-sm-6">
    <p>
      <label for="inAposta">Aposta R$:</label>
      <input type="text" class="form-control aDireita"
id="inValor">
    </p>
  </div>
  <div class="col-sm-6">
    <input type="button" class="btn btn-primary margemSup"
      value="Apostar" id="btApostar">
    <input type="button" class="btn btn-success margemSup"
      value="Ganhador" id="btGanhador">
    <input type="button" class="btn btn-danger margemSup"
      value="Novo Páreo" id="btNovo">
  </div>
</div>

  <pre id="outApostas"></pre>
</div>
</div>
<script src="js/ex10_1.js"></script>
</body>

</html>

```

A novidade é que dentro da segunda coluna (`<div class="col-sm-8">`) foram criadas duas novas `div 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. A constante com os nomes

dos cavalos, o vetor global que vai armazenar as apostas e a function `adicionarAposta()` são as primeiras linhas que devem ser adicionadas ao arquivo `ex10_1.js` que deve ser salvo dentro da pasta `js`. Crie esse arquivo e insira o seguinte código:

Programa JavaScript para controle de apostas do Jockey Club (js/ex10_1.js)

```
// constante com nomes 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)
var apostas = [];

function adicionarAposta() {
    // cria referência aos elementos inValor e outApostas
    // (inCavalo é referenciado em variável global na sequência do
programa)
    var inValor = document.getElementById("inValor");
    var outApostas = document.getElementById("outApostas");
    // obtém o nº do cavalo e valor da aposta
    var cavalo = Number(inCavalo.value);
    var valor = Number(inValor.value);
    // valida os dados
    if (isNaN(cavalo) || isNaN(valor) || valor == 0 ||
!validarCavalo(cavalo)) {
        alert("Aposta Inválida");
        inCavalo.focus();
        return;
    }

    // adiciona ao vetor de objetos (atributos cavalo e valor)
    apostas.push({ cavalo: cavalo, valor: valor });
    // variável para exibir a lista das apostas realizadas
    var lista = "Apostas Realizadas\n-----
---\n";

    // percorre o vetor e concatena em lista as apostas realizadas
    for (var i = 0; i < apostas.length; i++) {
```

```

        lista += "Nº " + apostas[i].cavalo + " " +
obterCavalo(apostas[i].cavalo);
        lista += " - R$: " + apostas[i].valor.toFixed(2) + "\n";
    }
    outApostas.textContent = lista; // exibe a lista das apostas

    inCavalo.value = ""; // limpa campos de entrada
    inValor.value = "";
    inCavalo.focus(); // posiciona o cursor em inCavalo
}
// cria referência ao botão e associa ao evento click a função
indicada
var btApostar = document.getElementById("btApostar");
btApostar.addEventListener("click", adicionarAposta);

```

A function `adicionarAposta()` cria referência aos elementos da página e, em seguida, obtém-se o conteúdo dos campos de entrada. Na sequência, os dados são validados, inseridos no vetor de objetos e exibidos na página.

Observe que na validação foi acrescentada uma condição que chama a function `validarCavalo()` passando como argumento o número digitado pelo usuário. Acrescente, portanto, essa function no arquivo `ex10_1.js`.

```

function validarCavalo(num) {
    var tam = CAVALOS.length; // obtém nº de cavalos da const CAVALOS
    // verifica se o número do cavalo (passado como argumento) é válido
    if (num >= 1 && num <= tam) {
        return true;
    } else {
        return false;
    }
}

```

A function obtém o tamanho do vetor de constantes com os nomes dos cavalos e verifica se o número informado está no intervalo válido (entre 1 e o número total de elementos do vetor). O retorno da function é `true` ou `false`.

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 10.2 ilustra o funcionamento dessa rotina no programa.



Figura 10.2 – Ao sair do campo de edição, o programa exibe dados do cavalo e de apostas.

Para isso foi criada uma **function** associada ao evento **blur** do elemento **inCavalo**. **blur** é o evento que ocorre quando um campo perde o foco. Acrescente essa rotina em seu programa.

```
function mostrarCavalo() {  
    // cria referência ao elemento outCavalo (para exibir dados do  
    cavalo)  
    var outCavalo = document.getElementById("outCavalo");  
  
    // se não preencheu o campo, limpa outCavalo e retorna  
    // (não exibe mensagem de alerta, pois pode sair por um clique em  
    Ganhador)  
    if (inCavalo.value == "") {  
        outCavalo.textContent = "";  
        return;  
    }  
    // obtém o conteúdo do campo de entrada  
    var cavalo = Number(inCavalo.value);  
  
    // valida (se número do cavalo for preenchido)  
    if (isNaN(cavalo) || !validarCavalo(cavalo)) {  
        outCavalo.textContent = "Número do Cavalo Inválido";  
        return;  
    }  
}
```



```

}

// cria variáveis com retorno dos métodos a serem exibidos em
outCavalo
var nomeCavalo = obterCavalo(cavalo);
var numApostas = contarApostas(cavalo);
var total = totalizarApostas(cavalo);

// exibe nome, nº de apostas e total apostado no cavalo
outCavalo.textContent = nomeCavalo + " (Apostas: " + numApostas;
outCavalo.textContent += " - R$: " + total.toFixed(2) + ")";
}
// cria referência ao botão e associa ao evento blur a função
indicada
var inCavalo = document.getElementById("inCavalo");
inCavalo.addEventListener("blur", mostrarCavalo);

```

Na programação associada ao evento `blur` de um campo de edição, 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 `function`.

Caso ele informe um número inválido, a `function` exibe essa indicação modificando a propriedade `textContent` do elemento `outCavalo`. Ainda nesse caso, o programa retorna o controle ao usuário, que deverá executar outra operação válida.

Na sequência, a `function` `mostrarCavalo()` realiza chamada às funções `obterCavalo()`, `contarApostas()` e `totalizarApostas()`. Todas elas recebem como parâmetro o número do cavalo. Acrescente na sequência essas funções ao programa Jockey Club.

```

function obterCavalo(num) {
    var posicao = num - 1; // posição no vetor (subtrai 1, pois inicia
em 0)
    return CAVALOS[posicao]; // nome do cavalo (da const CAVALOS)
}

```

```

function contarApostas(numCavalo) {
    var contador = 0;
    // percorre o vetor apostas
    for (var i = 0; i < apostas.length; i++) {
        // verifica se a aposta é no cavalo passado como argumento
        if (apostas[i].cavalo == numCavalo) {
            contador++; // conta +1 quando a aposta for no cavalo do
            argumento
        }
    }
    return contador; // retorna o nº de apostas no cavalo numCavalo
}

function totalizarApostas(numCavalo) {
    var total = 0;
    for (var i = 0; i < apostas.length; i++) {
        if (apostas[i].cavalo == numCavalo) {
            total = total + apostas[i].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
inCavalo.addEventListener("focus", function () {
    inCavalo.value = "";
    document.getElementById("outCavalo").textContent = "";
});

```

A partir da inclusão dessas `functions` ao arquivo `ex10_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 10.2.

Falta-nos a programação a ser associada aos botões **Ganhador** e **Novo Páreo**. Começamos pela apresentação da function ganhadorPareo().

```
function ganhadorPareo() {
    // solicita o número do cavalo ganhador (já converte para número)
    var ganhador = Number(prompt("Nº Cavalo Ganhador: "));

    // para validar o preenchimento do prompt anterior
    if (isNaN(ganhador) || !validarCavalo(ganhador)) {
        alert("Cavalo Inválido");
        return;
    }

    // cria referência ao elemento outApostas (onde será exibido o
    // resumo)
    var outApostas = document.getElementById("outApostas");

    // concatena em resumo o resultado a ser exibido no elemento
    // outApostas
    var resumo = "Resultado Final do Páreo\n"
    resumo += "-----\n"
    resumo += "Nº Total de Apostas: " + apostas.length + "\n";
    resumo += "Total Geral R$: " + totalizarGeral().toFixed(2) +
    "\n\n";
    resumo += "Ganhador Nº " + ganhador + " - " + obterCavalo(ganhador)
    + "\n";
    resumo += "-----\n"
    resumo += "Nº de Apostas: " + contarApostas(ganhador) + "\n";
    resumo += "Total Apostado R$: " +
    totalizarApostas(ganhador).toFixed(2);

    outApostas.textContent = resumo; // exhibe o resultado

    btApostar.disabled = true; // desabilita os botões apostar e
    ganhador
    btGanhador.disabled = true;
    btNovo.focus(); // joga o foco no botão "Novo Páreo"
}
// cria referência ao botão e associa ao evento click a função
// indicada
```

```
var btGanhador = document.getElementById("btGanhador");
btGanhador.addEventListener("click", ganhadorPareo);
```

Ao clicar sobre o botão **Ganhador**, o programa aciona a **function ganhadorPareo()** que inicialmente solicita e valida o número do cavalo ganhador. A **function** atribui à variável **resumo** os totais do páreo e do cavalo vencedor. 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. Há ainda uma chamada a **function totalizarGeral()** que obtém a soma dos valores apostados, a partir da execução de um comando **for**. Ela está descrita a seguir.

```
function totalizarGeral() {
    var total = 0;
    // percorre o vetor para somar o total das apostas
    for (var i = 0; i < apostas.length; i++) {
        total = total + apostas[i].valor;
    }
    return total; // retorna o total das apostas
}
```

A Figura 10.3 ilustra a exibição dos dados de resumo de um páreo.



Figura 10.3 – 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 recorreremos, novamente, ao método

location.reload() que recarrega a página para uma nova execução do programa.

```
// cria referência ao btNovo e cria função anônima associada ao
evento click
var btNovo = document.getElementById("btNovo");
btNovo.addEventListener("click", function() {
    location.reload(); // recarrega a página
});
```

10.2 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 10.4 apresenta a página desse programa, já com algumas poltronas ocupadas.

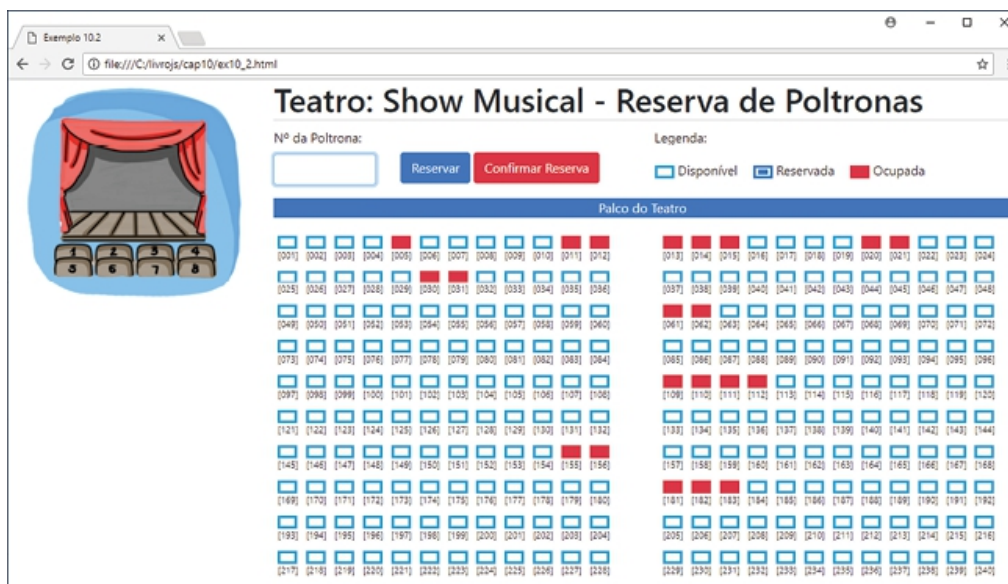


Figura 10.4 – A montagem das poltronas do Teatro é realizada pelo programa JavaScript.

Crie a página ex10_2.html e insira nela as seguintes tags HTML.

Exemplo 10.2 – Página para a reserva de poltronas de um Teatro (ex10_2.html)

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 10.2</title>
</head>

<body>
  <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>

        <div class="row">
          <div class="col-sm-2">
            <p>
              <label for="inPoltrona">Nº da Poltrona:</label>
              <input type="text" class="form-control aDireita"
                id="inPoltrona" autofocus>
            </p>
          </div>
          <div class="col-sm-4">
            <input type="button" class="btn btn-primary margenSup"
              value="Reservar" id="btReservar">
            <input type="button" class="btn btn-danger margenSup"
              value="Confirmar Reserva" id="btConfirmar">
          </div>
          <div class="col-sm-6">
            <p>Legenda:</p>
            <p>

```

```

         Disponível
         Reservada
         Ocupada
    </p>
</div>
</div>

<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>
</div>
<script src="js/ex10_2.js"></script>
</body>

</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 `ex10_2.js`. Insira inicialmente a `function` que realiza a montagem do palco, descrita a seguir:

Programa JavaScript para exibição e controle de reservas de poltronas (js/ex10_2.js)

```

const POLTRONAS = 240; // constante com o número de poltronas do
teatro

var reservadas = []; // vetor com as poltronas reservadas pelo
cliente

function montarPalco() {

    var ocupadas = []; // declara vetor para receber as poltronas
ocupadas

```

```

// verifica se há poltronas salvas em localStorage
if (localStorage.getItem("teatroOcupadas")) {
    // preenche o vetor com as poltronas ocupadas do teatro (salvas em
localStorage)
    ocupadas = localStorage.getItem("teatroOcupadas").split(";");
}

// captura divPalco, que é o local onde as imagens são inseridas
var divPalco = document.getElementById("divPalco");

// repetição para montar o nº total de poltronas (definida na
constante)
for (var i = 1; i <= POLTRONAS; i++) {

    var figure = document.createElement("figure"); // cria tag figure
    var imgStatus = document.createElement("img"); // cria tag img

    if (ocupadas.indexOf(i.toString()) >= 0) { // se consta em
ocupadas
        imgStatus.src = "img/ocupada.jpg"; // imagem ocupada.jpg
    } else { // se não constar ...
        imgStatus.src = "img/disponivel.jpg"; // imagem disponivel.jpg
    }
    imgStatus.className = "poltrona"; // classe com dimensão da img

    var figureCap = document.createElement("figcaption"); // cria
figcaption

    var zeros = "";
    if (i < 10) { // condições para exibir zeros nas poltronas...
        zeros = "00"; // menores que 10
    } else if (i < 100) {
        zeros = "0"; // menores que 100
    }

    var num = document.createTextNode "[" + zeros + i + "]"); // cria
texto

    figureCap.appendChild(num); // define os pais de cada tag criada
    figure.appendChild(imgStatus);
    figure.appendChild(figureCap);
}

```



```

        if (i % 24 == 12) { // se módulo 24, restar 12 (é o corredor)
            figure.style.marginRight = "60px"; // define margem direita de
60px
        }

        divPalco.appendChild(figure); // indica que figure é filha de
divPalco

        if (i % 24 == 0) { // se módulo 24, restar 0
            var br = document.createElement("br"); // cria tag br (quebra de
linha)
            divPalco.appendChild(br); // indica que br é filha de divPalco
        }
    }
}
montarPalco(); // chama function montarPalco()

```

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 **disponível.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 10.4. 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.

Para realizar a reserva de uma poltrona, o programa primeiro destaca as poltronas escolhidas pelo cliente com uma imagem diferente, como exibido na Figura 10.5. A programação da **function reservarPoltrona()** é descrita na sequência.

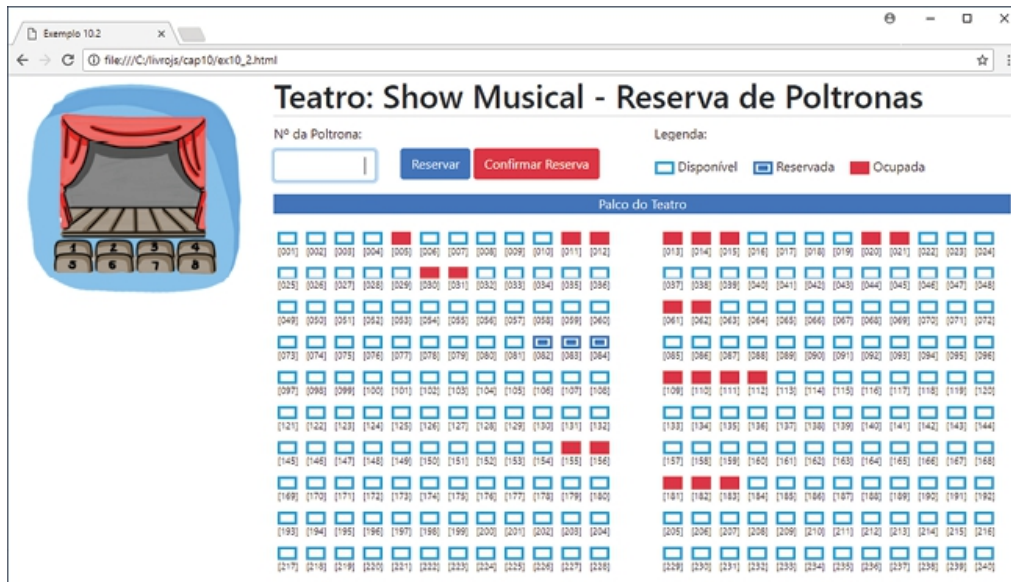


Figura 10.5 – A reserva da poltrona é realizada em duas etapas, primeiro as poltronas são destacadas com uma figura diferente.

```
function reservarPoltrona() {
    var poltrona = Number(inPoltrona.value); // obtém conteúdo de
    inPoltrona

    // valida o preenchimento do campo de entrada... não pode ser maior
    que a const
    if (poltrona <= 0 || isNaN(poltrona) || poltrona > POLTRONAS) {
        alert("Informe um número de poltrona válido");
        inPoltrona.focus();
        return;
    }

    var ocupadas = []; // declara vetor para receber as poltronas
    ocupadas

    // se há poltronas salvas em localStorage
    if (localStorage.getItem("teatroOcupadas")) {
        // preenche o vetor com as poltronas ocupadas do teatro (salvas em
        localStorage)
        ocupadas = localStorage.getItem("teatroOcupadas").split(";");
    }

    // se poltrona escolhida já está ocupada (existe em localStorage)
    if (ocupadas.indexOf(poltrona.toString()) >= 0) {
```

```

    alert("Poltrona " + poltrona + " já está ocupada...");
    inPoltrona.value = "";
    inPoltrona.focus();
    return;
}

// captura divPalco para obter a imagem
var divPalco = document.getElementById("divPalco");

// captura imagem da poltrona, filha de divPalco. É -1 pois começa
em 0
var imgPoltrona = divPalco.getElementsByTagName("img")[poltrona -
1];

imgPoltrona.src = "img/reservada.jpg"; // modifica atributo da
imagem

reservadas.push(poltrona); // adiciona poltrona ao vetor reservadas

inPoltrona.value = ""; // limpa campo
inPoltrona.focus(); // joga o foco em inPoltrona
}
var btReservar = document.getElementById("btReservar");
btReservar.addEventListener("click", reservarPoltrona);

// cria referência ao campo de entrada inPoltrona
var inPoltrona = document.getElementById("inPoltrona");
inPoltrona.addEventListener("keypress", function (tecla) {
    if (tecla.keyCode == 13) {
        reservarPoltrona();
    }
});
});

```

Observe que a `function reservarPoltrona()` 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 `getElementsByTagName()` indicando o índice do elemento a ser referenciado. Na sequência do código, o atributo `src` da imagem é alterado. A `function` anônima exibida no final do trecho de código anterior permite a execução da rotina que reserva uma poltrona para o

cliente a partir do pressionamento da tecla **Enter** (código 13) no campo `inPoltrona`.

Para finalizar esse programa, acrescente o código da function `confirmarReserva()` 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 10.4 destacada anteriormente).

```
function confirmarReserva() {
  if (reservadas.length == 0) {
    alert("Não há poltronas reservadas");
    inPoltrona.focus(); // elemento já foi referenciado de forma
global
    return;
  }
  // captura divPalco para obter as imagens
  var divPalco = document.getElementById("divPalco");

  var ocupadas = "";

  if (localStorage.getItem("teatroOcupadas")) {
    ocupadas = localStorage.getItem("teatroOcupadas") + ";";
  }

  for (var i = 0; i < reservadas.length; i++) {
    ocupadas += reservadas[i] + ";";

    // captura imagem da poltrona, filha de divPalco. É -1 pois começa
em 0
    var imgPoltrona = divPalco.getElementsByTagName("img")
[reservadas[i] - 1];

    imgPoltrona.src = "img/ocupada.jpg"; // modifica atributo da
imagem
  }

  reservadas = []; // limpa vetor (pois as reservas já foram salvas
em localStorage)

  // .length-1 é para retirar o último ";"
```

```
localStorage.setItem("teatroOcupadas", ocupadas.substr(0,
ocupadas.length - 1));
}
var btConfirmar = document.getElementById("btConfirmar");
btConfirmar.addEventListener("click", confirmarReserva);
```

10.3 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.

10.3.1 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 10.6 exibe a página que realiza o cadastro de palavras do jogo. O código HTML descrito na sequência deve ser inserido no arquivo `ex10_3.html` e contém uma novidade, a inserção de links na página. Para melhor organizar o código das funções que compõem o programa, as quais são várias, dividimos o programa igualmente 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 `<a href="nomearq.html"> texto de link </a>`. Observe, no código a seguir, os exemplos de uso dessas tags.



Figura 10.6 – O jogador pode cadastrar várias palavras. Uma delas é sorteada ao iniciar o jogo.

Exemplo 10.3 – Página de cadastro de palavras do jogo (ex10_3.html)

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos3.css">
  <title>Exemplo 10.3 - Cadastro</title>
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-sm-12">
        <h1>Jogo "Descubra a Palavra"</h1>

        <a href="ex10_3.html" class="btn btn-primary">Cadastrar
Palavras</a>
        <a href="ex10_3lista.html" class="btn btn-success">Listar
Palavras</a>
        <a href="ex10_3jogo.html" class="btn btn-danger">Iniciar
Jogo</a>
```

```

    <h2>Cadastrar Palavras</h2>
    <p>
        <label for="inPalavra">Palavra:</label>
        <input type="text" class="form-control" id="inPalavra"
autofocus>
    </p>
    <p>
        <label for="inDica">Dica da Palavra:</label>
        <input type="text" class="form-control" id="inDica">
    </p>
    <input type="button" class="btn btn-primary"
        value="Cadastrar" id="btCadastrar">

    </div>
</div>
</div>
<script src="js/ex10_3.js"></script>
</body>
</html>

```

Já o programa JavaScript que realiza o cadastro de palavras deve validar o preenchimento dos campos e salvar os dados em Local Storage. Essas ações já foram realizadas em outros programas. Crie o arquivo `ex10_3.js` (na pasta `js`) e insira o código descrito a seguir.

Programa JavaScript que realiza o cadastro das palavras (`js/ex10_3.js`)

```

function cadastrarPalavra() {
    // cria referência aos elementos de entrada de dados
    var inPalavra = document.getElementById("inPalavra");
    var inDica = document.getElementById("inDica");

    var palavra = inPalavra.value; // obtém conteúdo dos campos
    var dica = inDica.value;

    // valida preenchimento (palavra não deve possuir espaço em branco)
    if (palavra == "" || dica == "" || palavra.indexOf(" ") >= 0) {
        alert("Informe uma palavra válida (sem espaços) e sua dica
correspondente");
        inPalavra.focus();
        return;
    }
}

```

```

}
// se já existe...
if (localStorage.getItem("jogoPalavra")) {
    // grava conteúdo anterior + ";" e a palavra / dica
    localStorage.setItem("jogoPalavra",
        localStorage.getItem("jogoPalavra")+";"+palavra);
    localStorage.setItem("jogoDica",
        localStorage.getItem("jogoDica")+";"+dica);
} else {
    // senão, é a primeira inclusão: grava apenas a palavra e dica
    localStorage.setItem("jogoPalavra", palavra);
    localStorage.setItem("jogoDica", dica);
}
// verifica se salvou
if (localStorage.getItem("jogoPalavra")) {
    alert("Ok! Palavra " + palavra + " Cadastrada com Sucesso");
}

inPalavra.value = ""; // limpa campos de edição
inDica.value = "";
inPalavra.focus(); // joga foco em inPalavra
}
// cria referência ao elemento btCadastrar e associa function ao
evento click
var btCadastrar = document.getElementById("btCadastrar");
btCadastrar.addEventListener("click", cadastrarPalavra);

```

Como pode ser observado no código `ex10_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.

10.3.2 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 10.7 ilustra

a tela exibida pelo programa quando se clica no link **Listar Palavras**.



Figura 10.7 – 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 **checkbox** que permite selecionar a palavra para exclusão. A Figura 10.8 apresenta uma lista de palavras cadastradas no programa.

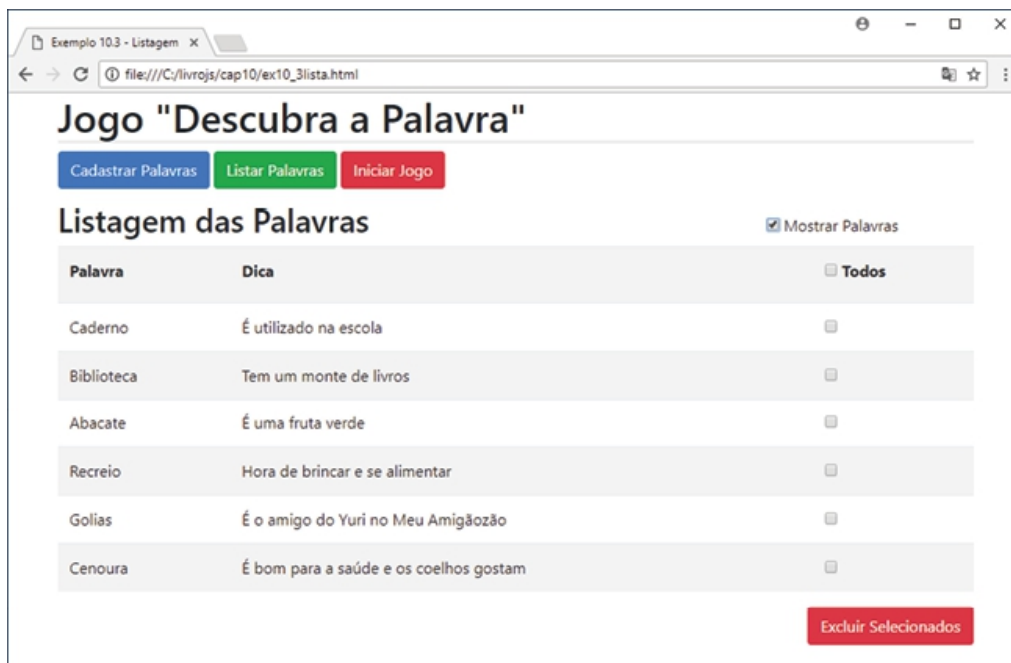


Figura 10.8 – As palavras cadastradas no jogo podem ser excluídas.

O código HTML com as tags que exibem a estrutura inicial dessa página é descrito a seguir. Crie um novo arquivo, chamado `ex10_3lista.html`,

para acrescentar os comandos. Observe que a parte inicial, até os links, é semelhante ao código anterior.

Exemplo 10.3 lista – Página da listagem de palavras cadastradas (ex10_3lista.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 10.3 - Listagem</title>
</head>

<body>
  <div class="container">

    <div class="row">
      <div class="col-sm-12">
        <h1>Jogo "Descubra a Palavra"</h1>

        <a href="ex10_3.html" class="btn btn-primary">Cadastrar
Palavras</a>
        <a href="ex10_3lista.html" class="btn btn-success">Listar
Palavras</a>
        <a href="ex10_3jogo.html" class="btn btn-danger">Iniciar
Jogo</a>

        <div class="row">
          <div class="col-sm-9">
            <h2>Listagem das Palavras</h2>
          </div>
          <div class="col-sm-3">
            <input type="checkbox" class="margemSup"
id="ckMostrar">Mostrar Palavras
          </div>
        </div>
      </div>
    </div>
  </div>
</body>
</html>
```

```

<table class="table table-striped" id="tbPalavras">
  <tr>
    <th>Palavra</th>
    <th>Dica</th>
    <th>
      <label>
        <input type="checkbox" id="ckTodos"> Todos </label>
      </th>
    </tr>
  </table>

  <p class="aDireita">
    <input type="button" class="btn btn-danger oculta"
      value="Excluir Seleccionados" id="btExcluir">
  </p>

</div>
</div>
</div>
<script src="js/ex10_3lista.js"></script>
</body>

</html>

```

As palavras serão adicionadas pelo programa JavaScript na tabela identificada pelo `id="tbPalavras"`. Acrescente agora o arquivo `ex10_3lista.js`, responsável por gerenciar a exibição das palavras e permitir a exclusão de algumas delas.

Programa Javascript que controla a listagem e exclusão de palavras (js/ex10_3lista.js)

```

function montarTabela() {
  // se houver dados salvos em localStorage
  if (localStorage.getItem("jogoPalavra")) {
    // obtém conteúdo e converte em elementos de vetor (na ocorrência
    ";")
    var palavras = localStorage.getItem("jogoPalavra").split(";");
    var dicas = localStorage.getItem("jogoDica").split(";");

    // cria referência ao elemento tbPalavras

```

```

var tbPalavras = document.getElementById("tbPalavras");

var linha, col1, col2, col3; // declara as variáveis

// percorre elementos do vetor e os insere na tabela
for (var i = 0; i < palavras.length; i++) {

    linha = tbPalavras.insertRow(-1); // adiciona uma linha na
tabela

    col1 = linha.insertCell(0); // cria colunas na linha inserida
    col2 = linha.insertCell(1);
    col3 = linha.insertCell(2);

    col1.textContent = palavras[i]; // joga um conteúdo em cada
célula
    col2.textContent = dicas[i];
    col3.innerHTML = "<input type='checkbox'>"; // exibe campo
checkbox
}
}
}

// cria referência ao ckMostrar (se marcado, deve exibir tabela e
botão excluir)
var ckMostrar = document.getElementById("ckMostrar");
// cria function anônima, associada ao evento change do ckMostrar
ckMostrar.addEventListener("change", function () {
    if (ckMostrar.checked) { // se marcado ...
        montarTabela(); // exibe tabela (palavras e dicas)
        btExcluir.className = "btn btn-danger exibe"; // exibe o botão
Excluir
    } else { // senão ...
        location.reload(); // recarrega a página
    }
});

// cria referência ao checkbox ckTodos (na linha de título da
tabela)
var ckTodos = document.getElementById("ckTodos");
// executa função anônima quando houver uma troca de status

```

```

ckTodos.addEventListener("change", function () {
    // cria referência à tabela e aos campos input (filhos da tabela)
    var tbPalavras = document.getElementById("tbPalavras");
    var ckExcluir = tbPalavras.getElementsByTagName("input");

    var status = ckTodos.checked; // obtém status de ckTodos ...

    // ... e percorre os demais checkbox para aplicar este status
    for (var i = 1; i < ckExcluir.length; i++) {
        ckExcluir[i].checked = status;
    }
});

function removerPalavras() {
    // cria referência à tabela e aos campos input (filhos da tabela)
    var tbPalavras = document.getElementById("tbPalavras");
    var ckExcluir = tbPalavras.getElementsByTagName("input");

    var temSelecionado = false; // para verificar se há palavras
    selecionadas

    // percorre campos input type checkbox da tabela (exceto "Todos" no
    título)
    for (var i = 1; i < ckExcluir.length; i++) {
        if (ckExcluir[i].checked) { // se está selecionado
            temSelecionado = true; // muda valor da "flag"
            break; // sai da repetição
        }
    }
    // se não temSelecionado (se valor da variável é false)
    if (!temSelecionado) {
        alert("Não há palavras selecionadas para exclusão");
        return;
    }
    // solicita confirmação de exclusão das palavras selecionadas
    if (confirm("Confirma Exclusão das Palavras Selecionadas?")) {

        // exclui conteúdo armazenado em localStorage
        localStorage.removeItem("jogoPalavra");
        localStorage.removeItem("jogoDica");
    }
}

```

```

    palavras = ""; // para acumular as palavras da tabela
    dicas = "";

    // primeiro irá gravar em localStorage as palavras não
    selecionadas
    for (i = 1; i < ckExcluir.length; i++) {
        // se não está selecionado (para exclusão)
        if (!ckExcluir[i].checked) {
            // obtém o conteúdo da tabela (coluna 0: palavra; coluna 1:
            dica)
            palavras += tbPalavras.rows[i].cells[0].textContent + ";";
            dicas += tbPalavras.rows[i].cells[1].textContent + ";";
        }
    }
    // se vazio, significa que marcou todos checkbox (não salva em
    localStorage)
    if (palavras != "") {
        // .length-1 (para retirar o último ";")
        localStorage.setItem("jogoPalavra", palavras.substr(0,
        palavras.length - 1));
        localStorage.setItem("jogoDica", dicas.substr(0, dicas.length -
        1));
    }

    // agora irá remover as linhas selecionadas (do fim para o início)
    for (i = ckExcluir.length - 1; i > 0; i--) {
        if (ckExcluir[i].checked) {
            tbPalavras.deleteRow(i); // remove a linha
        }
    }
    ckExcluir[0].checked = false; // desmarca ckTodos (que é o input
    0)
}
}

var btExcluir = document.getElementById("btExcluir");
btExcluir.addEventListener("click", removerPalavras);

```

Como destacado anteriormente, 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 botão ckMostrar.

10.3.3 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 10.3jogo – Página HTML que exhibe o layout inicial do jogo (ex10_3jogo.html)

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/estilos.css">
  <title>Exemplo 10.3 - Jogo</title>
</head>

<body>
  <div class="container">

    <div class="row">
      <div class="col-sm-12">
        <h1>Jogo "Descubra a Palavra"</h1>

        <a href="ex10_3.html" class="btn btn-primary">Cadastrar
Palavras</a>
        <a href="ex10_3lista.html" class="btn btn-success">Listar
Palavras</a>
        <a href="ex10_3jogo.html" class="btn btn-danger">Iniciar
Jogo</a>

        <div class="row margemJogo">
```

```

<div class="col-sm-4 corFundo">
  <h2 class="aDireita">
    <label for="inLetra">Informe uma letra:</label>
  </h2>
</div>
<div class="col-sm-2 corFundo">
  <input type="text" class="form-control margemJogo"
    id="inLetra" autofocus>
</div>
<div class="col-sm-6 corFundo">
  <input type="button" class="btn btn-primary margemJogo"
    value="Jogar" id="btJogar">
</div>
</div>

<div class="row">
  <div class="col-sm-8">
    <h2>Palavra:
      <span id="outPalavra" class="entreLetras fonteAzul"></span>
    </h2>
    <h2>Nº Chances:
      <span id="outChances" class="fonteVermelho">4</span>
    </h2>
    <h2>Erros:
      <span id="outErros" class="entreLetras fonteVermelho">
</span>
    </h2>
    <h2 id="outMensagemFinal"></h2>
  </div>
  <div class="col-sm-4 bordaQuadro">
    <h3> Situação do Jogador </h3>
    
    </div>
</div>

<div class="row">
  <div class="col-sm-12 corFundo margemDica">

```



```

        <input type="button" value="Ver Dica"
            class="btn btn-primary" id="btVerDica">
        <span id="outDica">* Custo: 1 chance</span>
    </div>
</div>
</div>
</div>
</div>
<script src="js/ex10_3jogo.js"></script>
</body>

</html>

```

O código do arquivo `ex10_3jogo.html` é responsável por criar o layout inicial da página, conforme ilustra a Figura 10.9.

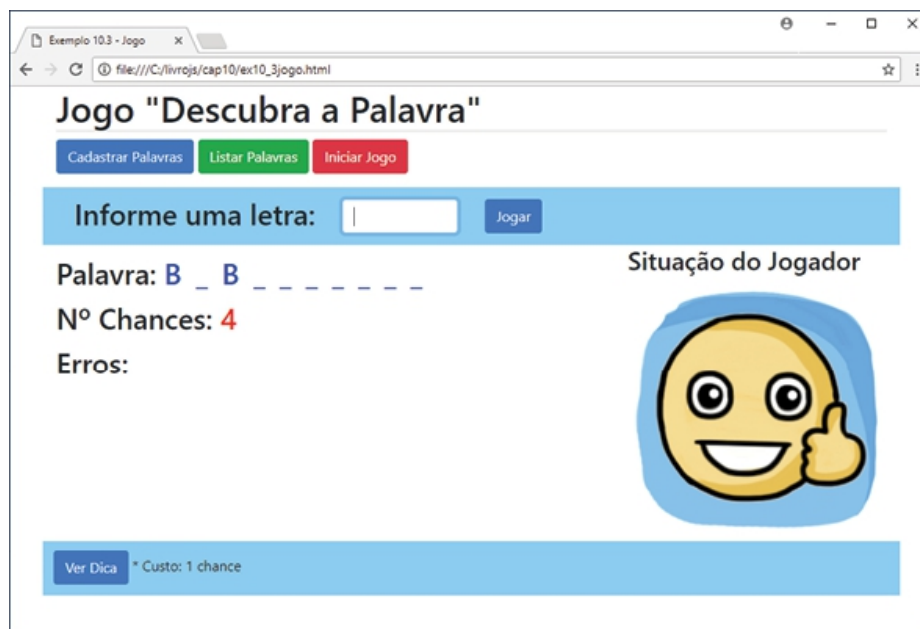


Figura 10.9 – A cada execução, uma palavra aleatória do cadastro é selecionada.

As tarefas do programa JavaScript para gerenciar a execução do jogo serão discutidas em partes. Crie o arquivo `ex10_3jogo.js` (na pasta `js`) e insira as seguintes linhas.

Programa JavaScript do Jogo "Descubra a Palavra" (`js/ex10_3jogo.js`)

```

var palavraSorteada; // declara variáveis globais
var dicaSorteada;

// cria referência aos elementos que irão conter eventos associados

```

```

a functions
var inLetra = document.getElementById("inLetra");
var btJogar = document.getElementById("btJogar");
var btVerDica = document.getElementById("btVerDica");

function montarJogo() {
    // cria referência ao local onde a palavra sorteada (letra inicial
    e _) é exibida
    var outPalavra = document.getElementById("outPalavra");

    // obtém conteúdo do localStorage e separa em elementos de vetor
    var palavras = localStorage.getItem("jogoPalavra").split(";");
    var dicas = localStorage.getItem("jogoDica").split(";");

    var tam = palavras.length; // número de palavras cadastradas
    // gera um número entre 0 e tam-1 (pois arredonda para baixo)
    var 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];
    var novaPalavra = ""; // para montar palavra exibida (com letra
    inicial e "_")

    // for para exibir a letra inicial e as demais ocorrências desta
    letra na palavra
    for (var i = 0; i < palavraSorteada.length; i++) {
        // se igual a letra inicial, acrescenta esta letra na exibição
        if (palavraSorteada.charAt(0) == palavraSorteada.charAt(i)) {
            novaPalavra += palavraSorteada.charAt(0);
        } else { // senão...
            novaPalavra += "_"; // acrescenta "_"
        }
    }
    outPalavra.textContent = novaPalavra; // exibe a novaPalavra
}

if (localStorage.getItem("jogoPalavra")) { // se houver palavras
    cadastradas
    montarJogo(); // sorteia e "monta" palavra do jogo
}

```

```
} else { // senão...  
  alert("Cadastre palavras para jogar"); // exibe alerta  
  inLetra.disabled = true; // desabilita inLetra e botões  
  btJogar.disabled = true; // (por isso, eles são referenciados  
  btVerDica.disabled = true; // no início do programa)  
}
```

O primeiro cuidado que devemos ter nesse programa 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 após a declaração das variáveis globais, criando referência aos elementos da página. E, no final dessa primeira parte do código, desabilitamos esses elementos, caso o retorno do `if` que verifica a existência de `jogoPalavra` em `localStorage` seja falso.

Observe na `function montarJogo()` 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. O resultado da execução dessa rotina é o ilustrado na Figura 10.9.

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 10.10 apresenta a página quando o jogador clicou no botão logo no início do programa e perdeu uma chance.

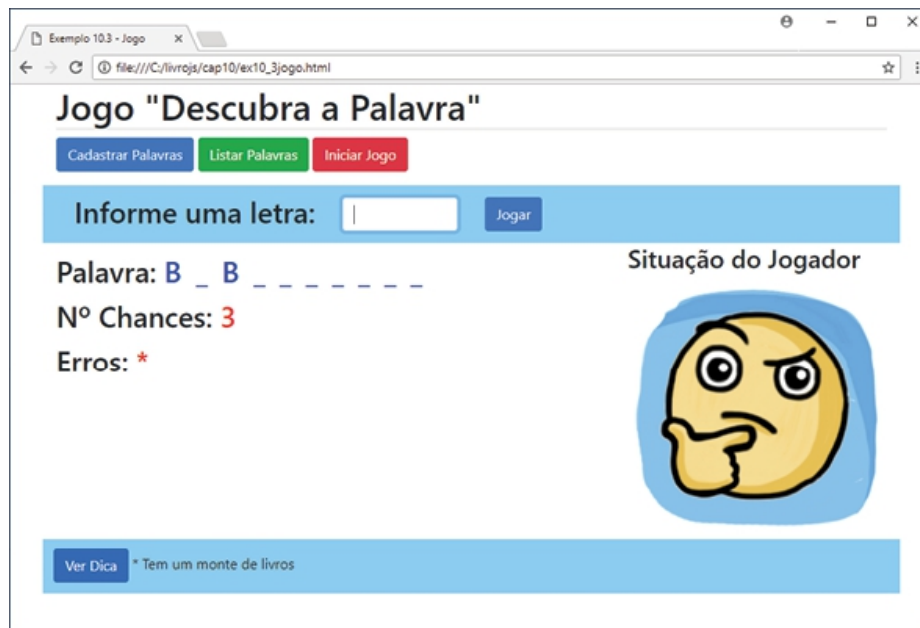


Figura 10.10 – Ao clicar em “Ver Dica”, o jogador perde uma chance e a imagem é modificada.

Acrescente ao programa `ex10_3jogo.cs` a programação associada ao evento `click` do botão `btVerDica` descrita a seguir.

```
function mostrarDica() {
    // cria referência aos elementos da página a serem alterados nesta
    function
    var outErros = document.getElementById("outErros");
    var outDica = document.getElementById("outDica");
    var outChances = document.getElementById("outChances");

    var erros = outErros.textContent; // obtém o conteúdo do elemento
    outErros

    // verifica se o jogador já clicou anteriormente no botão
    if (erros.indexOf("*") >= 0) {
        alert("Você já solicitou a dica...");
        inLetra.focus();
        return;
    }

    outDica.textContent = " * " + dicaSorteada; // exhibe a dica
    outErros.textContent = erros + "*"; // acrescenta "*" nos erros
    var chances = Number(outChances.textContent) - 1; // diminui 1 em
```

```

chances
  outChances.textContent = chances; // mostra o nº de chances

  trocarStatus(chances); // troca a imagem

  verificarFim(); // verifica se atingiu limite de chances

  inLetra.focus(); // joga o foco em inLetra (já referenciado no
  início do prog)
}
// associa ocorrência do evento click deste elemento à function
mostrarDica()
btVerDica.addEventListener("click", mostrarDica);

```

A function `mostrarDica()` cria referência aos elementos de exibição de dados da página. Em seguida, obtém o conteúdo apresentado em `outErros`, para, então, verificar se o jogador já clicou em **Ver Dica**. 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()`, chamada ao final dessa function.

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`.

```

function trocarStatus(num) {
  if (num > 0) {
    var imgStatus = document.getElementById("imgStatus");
    imgStatus.src = "img/status" + num + ".jpg";
  }
}

```

Vamos destacar a seguir a programação associada ao evento `click` do botão **Jogar**. Essa function também é acionada quando o usuário

pressionar a tecla Enter no campo de entrada inLetra.

```
function jogarLetra() {
    var outPalavra = document.getElementById("outPalavra");
    var outErros = document.getElementById("outErros");
    var outChances = document.getElementById("outChances");

    // obtém o conteúdo do campo inLetra e converte-o para maiúscula
    var letra = inLetra.value.toUpperCase();

    // valida o preenchimento de uma única letra
    if (letra == "" || letra.length != 1) {
        alert("Informe uma letra");
        inLetra.focus();
        return;
    }

    var erros = outErros.textContent; // obtém o conteúdo dos elementos
da página
    var palavra = outPalavra.textContent;

    // se a letra apostada já consta em erros, significa que ele já
apostou esta letra
    if (erros.indexOf(letra) >= 0 || palavra.indexOf(letra) >= 0) {
        alert("Você já apostou esta letra");
        inLetra.focus();
        return;
    }

    if (palavraSorteada.indexOf(letra) >= 0) { // se letra consta em
palavraSorteada

        var novaPalavra = ""; // para compor novaPalavra
        // for para montar palavra a ser exibida
        for (var i = 0; i < palavraSorteada.length; i++) {
            // se igual a letra apostada, acrescenta esta letra na exibição
            if (palavraSorteada.charAt(i) == letra) {
                novaPalavra += letra;
            } else { // senão...
                novaPalavra += palavra.charAt(i); // acrescenta a letra ou ""_
existente
            }
        }
    }
}
```

```

    }
  }
  outPalavra.textContent = novaPalavra; // exhibe a novaPalavra

} else { // se letra não consta em palavraSorteada
  erros += letra; // acrescenta letra em erros
  outErros.textContent = erros; // exhibe os erros
  var chances = Number(outChances.textContent) - 1; // diminui nº de
chances
  outChances.textContent = chances; // exhibe novo nº de chances

  trocarStatus(chances); // troca imagem
}

verificarFim(); // verifica se já ganhou ou perdeu

inLetra.value = "";
inLetra.focus();
}
btJogar.addEventListener("click", jogarLetra);

// associa evento keypress à function anônima que verifica se
pressionou enter (13)
inLetra.addEventListener("keypress", function (tecla) {
  if (tecla.keyCode == 13) {
    jogarLetra(); // ... e chama jogarLetra
  }
});

```

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 7. A function `jogarLetra()` cria referência aos elementos da página, obtém o conteúdo do campo `inLetra` e valida o preenchimento do campo, tarefas comuns a outros programas. Na sequência, a function 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 10.11 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...

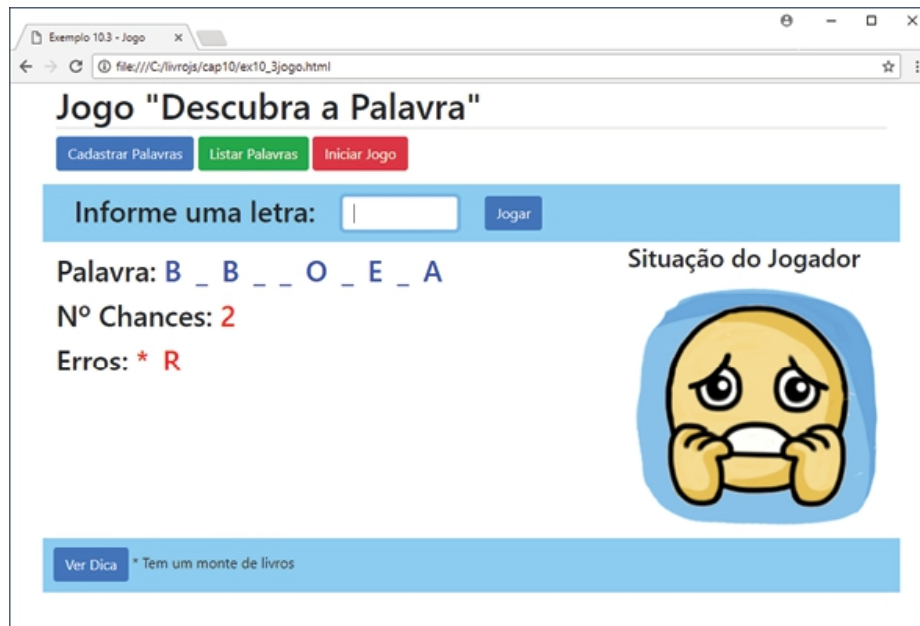


Figura 10.11 – As letras corretas são acrescentadas à palavra. Já os erros diminuem as chances e complicam a situação do jogador.

Vamos destacar a seguir a programação da `function verificarFim()` e `concluirJogo()` que finalizam esse programa. A Figura 10.12 ilustra uma situação em que o jogador perdeu o jogo. Já a Figura 10.13 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.

```
function verificarFim() {  
    var outChances = document.getElementById("outChances");  
    var outMensagemFinal = document.getElementById("outMensagemFinal");  
  
    var chances = Number(outChances.textContent); // obtém número de chances  
  
    if (chances == 0) { // se 0, perdeu  
        // display-3 é um estilo do Bootstrap
```



```

        outMensagemFinal.className = "display-3 fonteVermelho";
        outMensagemFinal.textContent = "Ah... é " + palavraSorteada + ".
Você Perdeu!";
        concluirJogo();
        // se não é 0 e a palavra exibida em outPalavra é igual a palavra
        sorteada, ganhou
    } else if (outPalavra.textContent == palavraSorteada) {
        outMensagemFinal.className = "display-3 fonteAzul";
        outMensagemFinal.textContent = "Parabéns!! Você Ganhou.";
        concluirJogo();
    }
}
// function concluirJogo, modifica o texto da dica e desabilita os
botões de jogar
function concluirJogo() {
    var outDica = document.getElementById("outDica");
    outDica.textContent = "* Clique no botão 'Iniciar Jogo' para jogar
novamente";
    inLetra.disabled = true;
    btJogar.disabled = true;
    btVerDica.disabled = true;
}

```

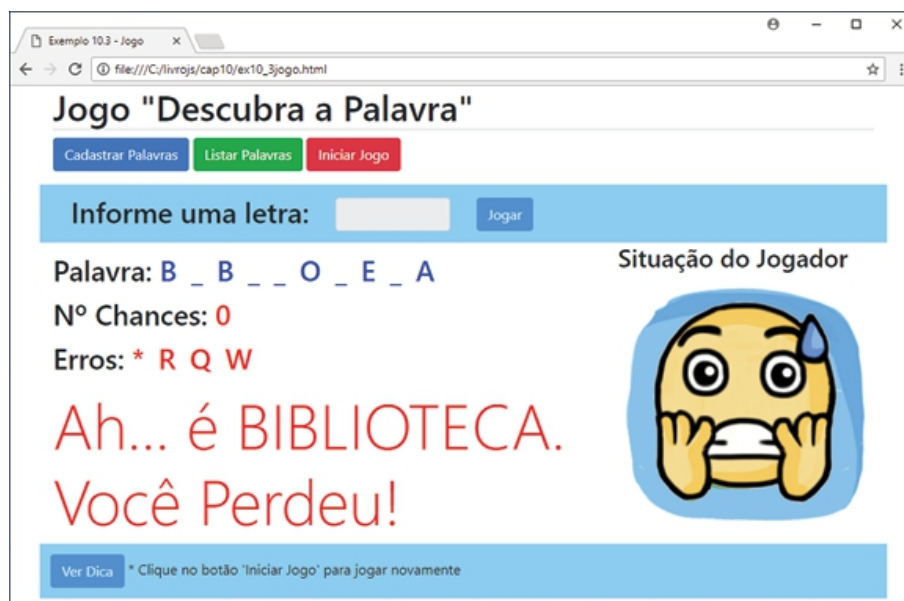


Figura 10.12 – Quando o jogador atingir o limite de chances, uma mensagem é exibida com a palavra correta.

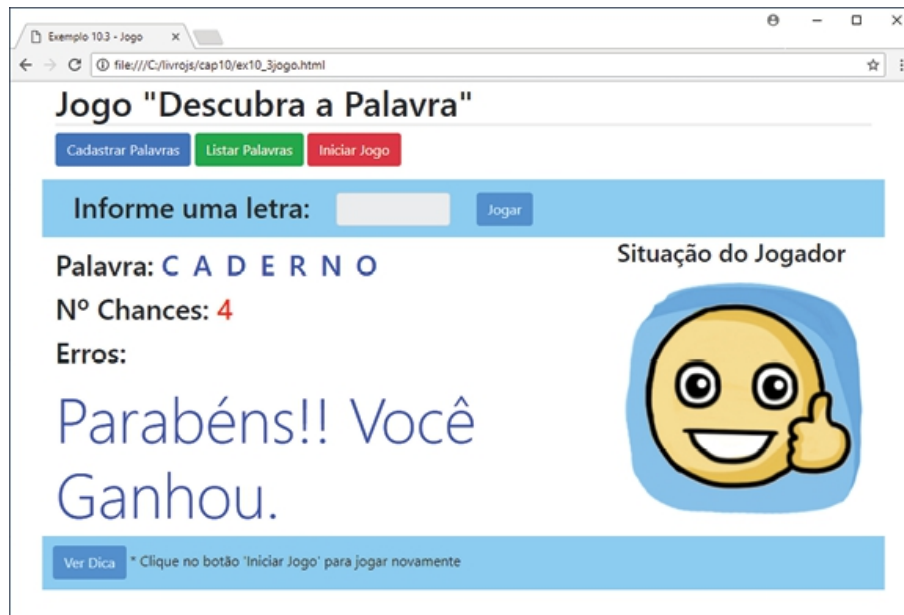


Figura 10.13 – Página exibida quando o jogador acertar a palavra.

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.

10.4 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 4.

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.

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. *Lógica de Programação – A construção de Algoritmos e Estruturas de Dados*. São Paulo: Makron Books, 1993.

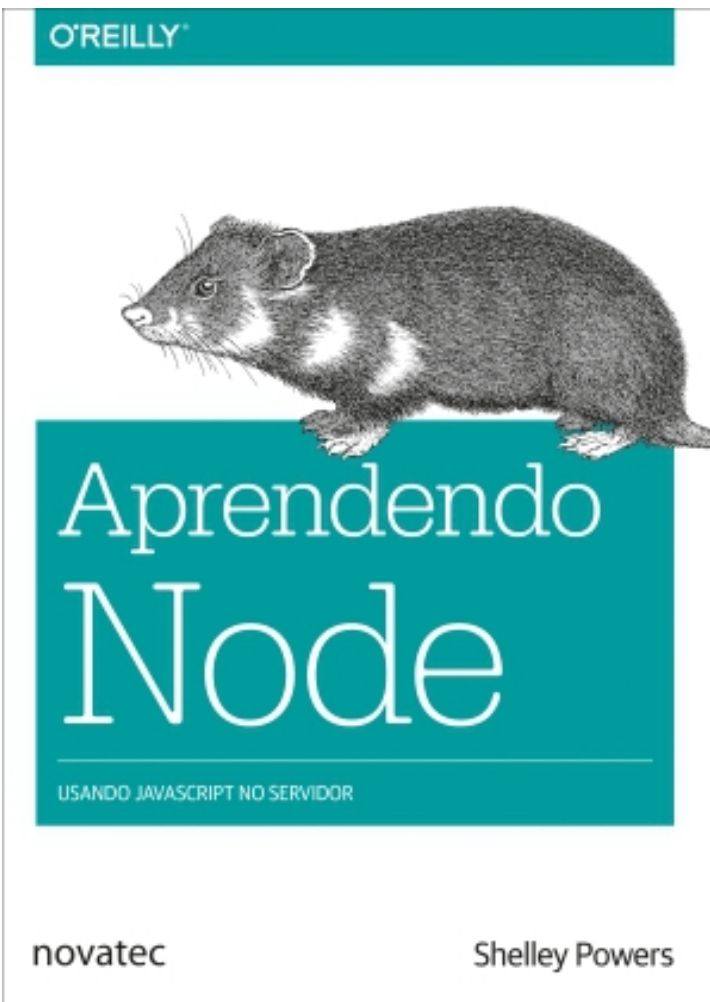
Mozilla Developer Network. Disponível em: <<https://developer.mozilla.org/pt-BR>>.

POWERS, Shelley. *Aprendendo JavaScript*. São Paulo: Novatec; Califórnia, EUA: O'Reilly, 2010.

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>>.



Aprendendo Node

Powers, Shelley

9788575227695

312 páginas

[Compre agora e leia](#)

Reúna o conhecimento de programação que você já usa no navegador e empregue no servidor com Node! Aprenda a criar aplicações em rede de alto desempenho e plenamente escalonáveis nesta plataforma baseada em JavaScript. O conteúdo deste livro é

bastante prático, "mão na massa", e fará com que o programador de JavaScript, acostumado a lidar apenas com processamento no navegador, domine com maestria os fundamentos do Node no servidor. Serão abordados inúmeros módulos nativos e também de terceiros. Esta edição foi atualizada para abranger tanto a mais recente versão tradicional (Long Term Support - LTS) do Node quanto a versão estável mais atual. Aprenda rapidamente sobre o modelo de I/O assíncrono e baseado em eventos do Node e, com ele, desenvolva aplicações que usam dados intensivamente, que são acessados com frequência e, ainda assim, são computacionalmente muito simples. Se você já domina o JavaScript, este livro oferece muitos exemplos de programas e implementações que se beneficiam do desenvolvimento centralizado no servidor proporcionado pelo Node. Explore a funcionalidade do Node e de seus frameworks para desenvolver soluções em full-stack. Mergulhe no sistema de módulos e pacotes do Node. Teste imediatamente o código de sua aplicação ou módulo com o console REPL do Node. Use os módulos nativos do Node para criar aplicações web que tenham seu próprio servidor HTTP. Aprenda todos os recursos do Node para networks, segurança e sockets. Acesse funcionalidades do sistema operacional com processos-filho. Aprenda as melhores ferramentas e técnicas para desenvolver e colocar em produção seus sistemas em Node. Use o Node em microcontroladores, microcomputadores e na Internet das Coisas (IoT - Internet of Things)

[Compre agora e leia](#)



Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

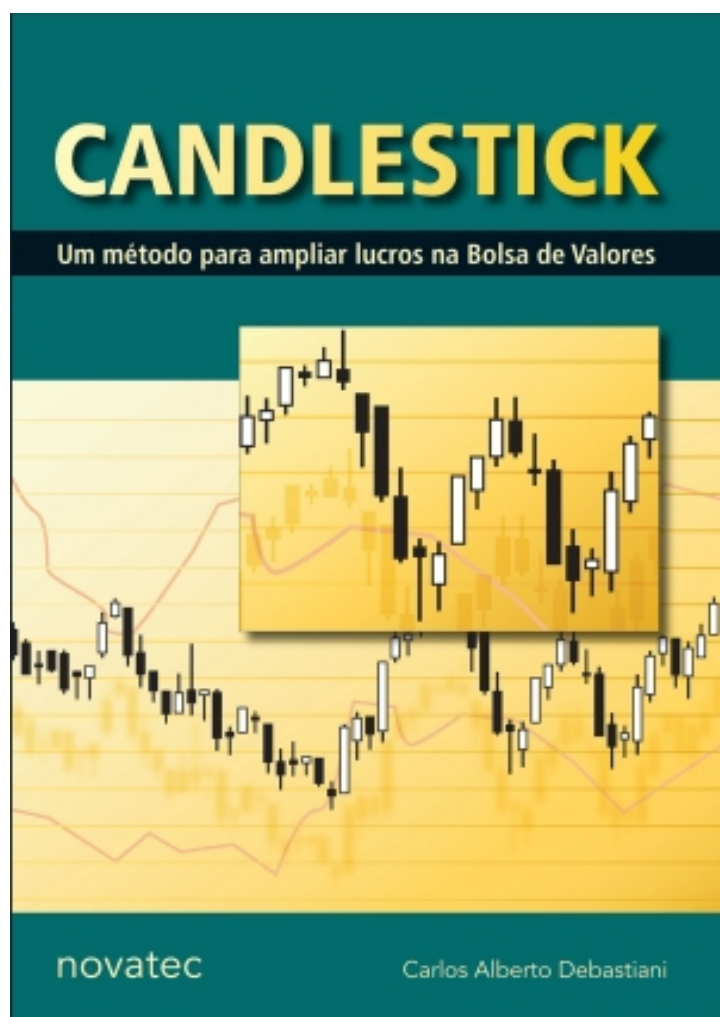
[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto

diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma.
- Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos.
- Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes.
- Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)



Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em

tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento

profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)



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 o leitor que, para ganhar dinheiro, não importa se o

mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Como 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](#)