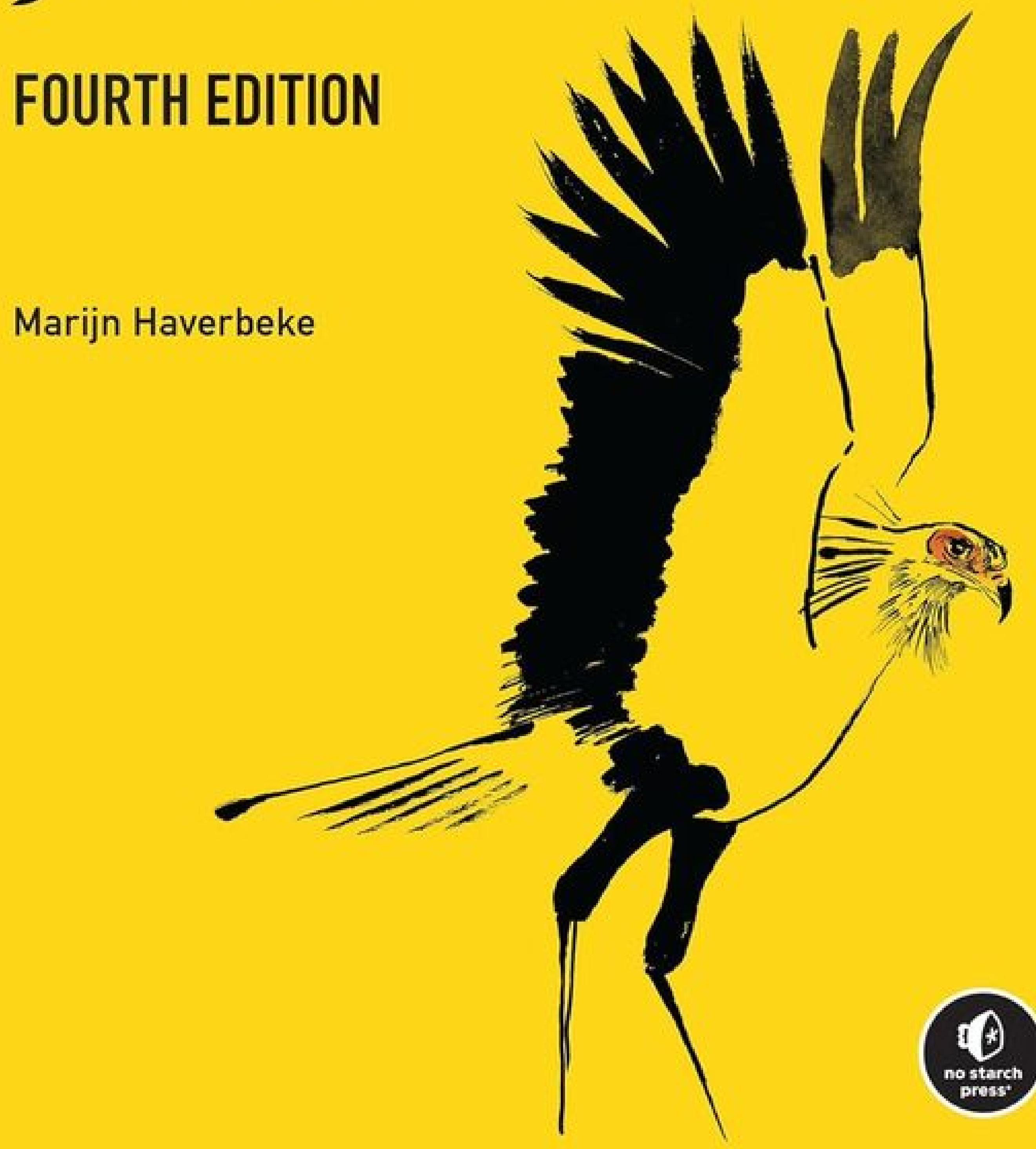
Eloquent JavaScript

Fourth Edition

Marijn Haverbeke



Eloquent JavaScript

4th edition

Marijn Haverbeke

Copyright © 2024 by Marijn Haverbeke

This work is licensed under a Creative Commons attribution-noncommercial license (http://creativecommons.org/licenses/by-nc/3.0/). All code in the book may also be considered licensed under an MIT license (https://eloquentjavascript.net/code/LICENSE).

The illustrations are contributed by various artists: Cover by Péchane Sumi-e. Chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with regexper.com by Jeff Avallone. Game concept for Chapter 16 by Thomas Palef.

You can buy a print version of this book, with an extra bonus chapter included, printed by No Starch Press at http://a-fwd.com/com=marijhaver-20&asin-com=1593279507.

Contents

Introdução 1
- Sobre programação 2
- Por que a linguagem importa 3
- O que é JavaScript? 6
- Código, e o que fazer com ele 7
- Visão geral deste livro 8
- Convenções tipográficas 9

1 Valores, Tipos e Operadores 10
- Valores 10
- Números 11
- Strings 14
- Operadores unários 16
- Valores booleanos 16
- Valores vazios 18
- Conversão automática de tipos 19
- Resumo 21

2 Estrutura do Programa 23
- Expressões e instruções 23
- Bindings 24
- Nomes de bindings 26
- O ambiente 27
- Funções 27
- A função console.log 28
- Valores de retorno 28
- Fluxo de controle 29
- Execução condicional 29
- Loops while e do 31
- Indentando Código 33
- Loops for 34
- Saindo de um Loop 35

Atualizando bindings sucintamente . . . 36
Despachando com um valor usando switch . . . 36
Capitalização . . . 37
Comentários . . . 38
Resumo . . . 38
Exercícios . . . 39

3 Funções 41
Definindo uma função . . . 41
Bindings e escopos . . . 42
Escopo aninhado . . . 43
Funções como valores . . . 44
Notação de declaração . . . 45
Arrow functions . . . 45
A pilha de chamadas . . . 46
Argumentos Opcionais . . . 48
Closure . . . 49
Recursão . . . 50
Crescendo funções . . . 53
Funções e efeitos colaterais . . . 56
Resumo . . . 57
Exercícios . . . 57

4 Estruturas de Dados: Objetos e Arrays 59
O lobisomem-esquilo . . . 59
Conjuntos de dados . . . 60
Propriedades . . . 61
Métodos . . . 62
Objetos . . . 63
Mutabilidade . . . 65
O registro do licantropo . . . 67
Calculando correlação . . . 69
Loops de array . . . 70
A análise final . . . 71
Mais sobre arrays . . . 73
Strings e suas propriedades . . . 74
Parâmetros rest . . . 76
O objeto Math . . . 77
Desestruturação . . . 79
Acesso opcional a propriedades . . . 79

JSON 80
Resumo 81
Exercícios 82

5 Funções de Ordem Superior 85
Abstração 86
Abstraindo repetição 87
Funções de ordem superior 88
Conjunto de dados de scripts 89
Filtrando arrays 90
Transformando com map 91
Resumindo com reduce 92
Composabilidade 93
Strings e códigos de caracteres 95
Reconhecendo texto 97
Resumo 98
Exercícios 98

6 A Vida Secreta dos Objetos 100
Tipos Abstratos de Dados 100
Métodos 101
Protótipos 102
Classes 104
Propriedades Privadas 106
Sobrescrevendo propriedades derivadas 107
Maps 108
Polimorfismo 110
Getters, setters e estáticos 111
Symbols 112
A interface de iteração 114
Herança 116
O operador instanceof 117
Resumo 118
Exercícios 119

7 Projeto: Um Robô 120
Meadowfield 120
A tarefa 122
Dados persistentes 124
Simulação 125

A rota do caminhão de correspondência 126
Busca de caminho 127
Exercícios 129

8 Bugs e Erros 131
Linguagem 131
Modo estrito 132
Tipos 133
Testes 134
Depuração 135
Propagação de erros 137
Exceções 138
Limpando depois de exceções 139
Captura seletiva 141
Asserções 143
Resumo 144
Exercícios 145

9 Expressões Regulares 146
Criando uma expressão regular 146
Testando correspondências 147
Conjuntos de caracteres 147
Caracteres internacionais 149
Repetindo partes de um padrão 150
Agrupando subexpressões 151
Correspondências e grupos 151
A classe Date 152
Limites e look-ahead 154
Padrões de escolha 155
A mecânica da correspondência 155
Retrocesso 156
O método replace 157
Ganância 159
Criando objetos RegExp dinamicamente 160
O método search 161
A propriedade lastIndex 161
Analisando um arquivo INI 163
Unidades de código e caracteres 165
Resumo 166
Exercícios 167

10 Módulos 169
Programas modulares . . . 169
Módulos ES . . . 170
Pacotes . . . 172
Módulos CommonJS . . . 173
Construção e empacotamento . . . 176
Design de módulos . . . 177
Resumo . . . 179
Exercícios . . . 180

11 Programação Assíncrona 182
Assincronicidade . . . 182
Callbacks . . . 184
Promises . . . 185
Falha . . . 187
Carla . . . 189
Invadindo . . . 190
Funções async . . . 192
Geradores . . . 193
Um Projeto de Arte Corvídea . . . 194
O loop de eventos . . . 197
Bugs assíncronos . . . 199
Resumo . . . 200
Exercícios . . . 200

12 Projeto: Uma Linguagem de Programação 203
Análise Sintática . . . 203
O avaliador . . . 207
Formas especiais . . . 209
O ambiente . . . 210
Funções . . . 212
Compilação . . . 213
Trapaceando . . . 214
Exercícios . . . 214

13 JavaScript e o Navegador 217
Redes e a Internet . . . 217
A Web . . . 219
HTML . . . 219
HTML e JavaScript . . . 222

Na sandbox 223
Compatibilidade e as guerras dos navegadores 223

14 O Modelo de Objeto do Documento 225
Estrutura do documento 225
Árvores 226
O padrão 227
Navegando pela árvore 228
Encontrando elementos 229
Alterando o documento 230
Criando nós 231
Atributos 233
Layout 233
Estilos 235
Estilos em cascata 237
Seletores de consulta 238
Posicionamento e animação 239
Resumo 242
Exercícios 242

15 Manipulando Eventos 244
Manipuladores de eventos 244
Eventos e nós DOM 245
Objetos de evento 246
Propagação 246
Ações padrão 248
Eventos de tecla 249
Eventos de ponteiro 250
Eventos de rolagem 254
Eventos de foco 255
Evento de carregamento 256
Eventos e o loop de eventos 257
Temporizadores 258
Debouncing 258
Resumo 260
Exercícios 260

16 Projeto: Um Jogo de Plataforma 261
O jogo 261
A tecnologia 261

Níveis . . . 261
Lendo um nível . . . 262
Atores . . . 263
Desenhando . . . 265
Movimento e colisão . . . 268
Atualizações dos atores . . . 269
Rastreando teclas . . . 271
Executando o jogo . . . 271
Exercícios . . . 272

17 Desenhando no Canvas 274
SVG . . . 274
O elemento canvas . . . 274
Linhas e superfícies . . . 275
Caminhos . . . 275
Curvas . . . 276
Desenhando um gráfico de pizza . . . 277
Texto . . . 278
Imagens . . . 279
Transformação . . . 280
Armazenando e limpando transformações . . . 281
De volta ao jogo . . . 282
Escolhendo uma interface gráfica . . . 285
Resumo . . . 285
Exercícios . . . 286

18 HTTP e Formulários 287
O protocolo . . . 287
Navegadores e HTTP . . . 288
Fetch . . . 288
Sandboxing HTTP . . . 289
Apreciando o HTTP . . . 289
Segurança e HTTPS . . . 289
Campos de formulário . . . 289
Foco . . . 290
Campos desabilitados . . . 291
O formulário como um todo . . . 291
Campos de texto . . . 292
Checkboxes e botões de rádio . . . 292
Campos de seleção . . . 293

Campos de arquivo 294
Armazenando dados no lado do cliente 295
Resumo 296
Exercícios 296

19 Projeto: Um Editor de Pixel Art 298
Componentes 298
O estado 298
Construção de DOM 299
O canvas 299
A aplicação 301
Ferramentas de desenho 303
Salvando e carregando 305
Histórico de desfazer 306
Vamos desenhar 307
Por que isso é tão difícil? 308
Exercícios 308

20 Node.js 310
Contexto 310
O comando node 310
Módulos 310
Instalando com NPM 311
O módulo de sistema de arquivos 312
O módulo HTTP 313
Streams 313
Um servidor de arquivos 314
Resumo 316
Exercícios 317

21 Projeto: Website de Compartilhamento de Habilidades 318
Design 318
Long polling 318
Interface HTTP 319
O servidor 320
O cliente 324
Exercícios 328

Exercise Hints 329
Estrutura do Programa 329

Funções 329
Estruturas de Dados: Objetos e Arrays 329
Funções de Ordem Superior 330
A Vida Secreta dos Objetos 330
Projeto: Um Robô 330
Bugs e Erros 330
Expressões Regulares 331
Módulos 331
Programação Assíncrona 331
Projeto: Uma Linguagem de Programação 332
O Modelo de Objeto do Documento 332
Manipulando Eventos 332
Projeto: Um Jogo de Plataforma 332
Desenhando no Canvas 333
HTTP e Formulários 333
Projeto: Um Editor de Pixel Art 334
Node.js 334
Projeto: Website de Compartilhamento de Habilidades 334

"We think we are creating the system for our own purposes. We believe we are making it in our own image... But the computer is not really like us. It is a projection of a very slim part of ourselves: that portion devoted to logic, order, rule, and clarity."

—Ellen Ullman, Close to the Machine: Technophilia and Its Discontents

Introdução

Este é um livro sobre instruir computadores. Computadores são tão comuns quanto chaves de fenda hoje em dia, mas são consideravelmente mais complexos, e fazer com que façam o que você quer nem sempre é fácil.

Se a tarefa que você tem para o seu computador é uma tarefa comum e bem compreendida, como mostrar seus e-mails ou funcionar como uma calculadora, você pode abrir o aplicativo apropriado e começar a trabalhar. Mas para tarefas únicas ou abertas, geralmente não existe um aplicativo apropriado.

É aí que a programação pode entrar. *Programar* é o ato de construir um *programa* — um conjunto de instruções precisas que dizem ao computador o que fazer. Como computadores são bestas burras e pedantes, programar é fundamentalmente tedioso e frustrante.

Felizmente, se você consegue superar esse fato — e talvez até gostar do rigor de pensar em termos que máquinas burras conseguem lidar — programar pode ser gratificante. Permite que você faça coisas em segundos que levariam uma *eternidade* manualmente. É uma forma de fazer sua ferramenta computacional realizar coisas que não conseguia antes. Além disso, é um maravilhoso jogo de resolução de quebra-cabeças e pensamento abstrato.

A maioria da programação é feita com linguagens de programação. Uma *linguagem de programação* é uma linguagem artificialmente construída usada para instruir computadores. É interessante que a forma mais eficaz que encontramos para nos comunicar com um computador tome tanto emprestado da forma como nos comunicamos uns com os outros. Assim como as linguagens humanas, linguagens de computador permitem que palavras e frases sejam combinadas de novas formas, tornando possível expressar conceitos sempre novos.

Em determinado momento, interfaces baseadas em linguagem, como os prompts do BASIC e do DOS dos anos 1980 e 1990, eram o principal método de interação com computadores. Para uso rotineiro do computador, essas foram amplamente substituídas por interfaces visuais, que são mais fáceis de aprender mas oferecem menos liberdade. Mas se você souber onde procurar, as linguagens ainda estão lá. Uma delas, *JavaScript*, está embutida em todos os navegadores web modernos — e, portanto, está disponível em praticamente todos os dispositivos.

Este livro tentará familiarizá-lo o suficiente com essa linguagem para que você possa fazer coisas úteis e divertidas com ela.

Sobre programação

Além de explicar JavaScript, vou apresentar os princípios básicos da programação. Programar, acontece que, é difícil. As regras fundamentais são simples e claras, mas programas construídos sobre essas regras tendem a se tornar complexos o suficiente para introduzir suas próprias regras e complexidade. Você está construindo seu próprio labirinto, de certa forma, e pode facilmente se perder nele.

Haverá momentos em que a leitura deste livro parecerá terrivelmente frustrante. Se você é novo em programação, haverá muito material novo para digerir. Muito desse material será então *combinado* de formas que exigem que você faça conexões adicionais.

Cabe a você fazer o esforço necessário. Quando estiver com dificuldades para acompanhar o livro, não tire conclusões precipitadas sobre suas próprias capacidades. Você está bem — só precisa continuar. Faça uma pausa, releia algum material e certifique-se de ler e compreender os programas de exemplo e os exercícios. Aprender é trabalho árduo, mas tudo que você aprende é seu e tornará o aprendizado futuro mais fácil.

> When action grows unprofitable, gather information; when information grows unprofitable, sleep.
>
> —Ursula K. Le Guin, The Left Hand of Darkness

Um programa é muitas coisas. É um pedaço de texto digitado por um programador, é a força diretriz que faz o computador fazer o que faz, são dados na memória do computador e, ao mesmo tempo, controla as ações realizadas nessa memória. Analogias que tentam comparar programas a objetos familiares tendem a ficar aquém. Uma que se encaixa superficialmente é comparar um programa a uma máquina — muitas partes separadas tendem a estar envolvidas, e para fazer a coisa toda funcionar, precisamos considerar as formas como essas partes se interconectam e contribuem para o funcionamento do todo.

Um computador é uma máquina física que atua como hospedeiro dessas máquinas imateriais. Computadores em si só conseguem fazer coisas estupidamente diretas. A razão pela qual são tão úteis é que fazem essas coisas a uma velocidade incrivelmente alta. Um programa pode combinar engenhosamente um enorme número dessas ações simples para fazer coisas muito complicadas.

Um programa é uma construção do pensamento. Não custa nada construí-lo, não tem peso e cresce facilmente sob nossos dedos que digitam. Mas conforme um programa cresce, também cresce sua complexidade. A habilidade de programar é a habilidade de construir programas que não confundam o programador. Os melhores programas são aqueles que conseguem fazer algo interessante enquanto ainda são fáceis de entender.

Alguns programadores acreditam que essa complexidade é melhor gerenciada usando apenas um pequeno conjunto de técnicas bem compreendidas em seus programas. Eles compuseram regras estritas ("melhores práticas") prescrevendo a forma que os programas devem ter e cuidadosamente permanecem dentro de sua pequena zona segura.

Isso não é apenas chato — é ineficaz. Novos problemas frequentemente requerem novas soluções. O campo da programação é jovem e ainda está se desenvolvendo rapidamente, e é variado o suficiente para ter espaço para abordagens completamente diferentes. Há muitos erros terríveis a se cometer em design de programas, e você deveria ir em frente e cometê-los pelo menos uma vez para que os compreenda. A noção do que faz um bom programa se desenvolve com prática, não é aprendida a partir de uma lista de regras.

Por que a linguagem importa

No início, no nascimento da computação, não havia linguagens de programação. Programas tinham esta aparência:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Esse é um programa para somar os números de 1 a 10 e imprimir o resultado: `1 + 2 + ... + 10 = 55`. Ele poderia rodar em uma máquina hipotética simples. Para programar os primeiros computadores, era necessário posicionar grandes conjuntos de interruptores na posição correta ou fazer furos em tiras de papelão e alimentá-las no computador. Você pode imaginar quão tedioso e propenso a erros esse procedimento era. Mesmo escrever programas simples exigia muita astúcia e disciplina. Os complexos eram quase inconcebíveis.

É claro que inserir manualmente esses padrões arcanos de bits (os zeros e uns) dava ao programador uma profunda sensação de ser um mago poderoso. E isso tem que valer algo em termos de satisfação no trabalho.

Cada linha do programa anterior contém uma única instrução. Poderia ser escrito em português assim:

1. Armazene o número 0 na posição de memória 0.
2. Armazene o número 1 na posição de memória 1.
3. Armazene o valor da posição de memória 1 na posição de memória 2.
4. Subtraia o número 11 do valor na posição de memória 2.
5. Se o valor na posição de memória 2 for o número 0, continue na instrução 9.
6. Some o valor da posição de memória 1 à posição de memória 0.
7. Some o número 1 ao valor da posição de memória 1.
8. Continue na instrução 3.
9. Exiba o valor da posição de memória 0.

Embora isso já seja mais legível que a sopa de bits, ainda é bastante obscuro. Usar nomes em vez de números para as instruções e posições de memória ajuda.

```
  Set "total" to 0.
  Set "count" to 1.
[loop]
  Set "compare" to "count".
  Subtract 11 from "compare".
  If "compare" is 0, continue at [end].
  Add "count" to "total".
  Add 1 to "count".
  Continue at [loop].
[end]
  Output "total".
```

Você consegue ver como o programa funciona neste ponto? As duas primeiras linhas dão a duas posições de memória seus valores iniciais: `total` será usado para construir o resultado da computação, e `count` acompanhará o número que estamos olhando no momento. As linhas usando `compare` são provavelmente as mais confusas. O programa quer verificar se `count` é igual a 11 para decidir se

pode parar de executar. Como nossa máquina hipotética é bastante primitiva, ela só consegue testar se um número é zero e tomar uma decisão com base nisso. Portanto, usa a posição de memória rotulada `compare` para calcular o valor de `count - 11` e toma uma decisão com base nesse valor. As duas linhas seguintes adicionam o valor de `count` ao resultado e incrementam `count` em 1 toda vez que o programa decide que `count` ainda não é 11.

Aqui está o mesmo programa em JavaScript:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

Esta versão nos dá algumas melhorias a mais. Mais importante, não há mais necessidade de especificar a forma como queremos que o programa salte para frente e para trás — a construção `while` cuida disso. Ela continua executando o bloco (envolvido em chaves) abaixo dela enquanto a condição fornecida se mantiver verdadeira. Essa condição é `count <= 10`, que significa "count é menor ou igual a 10". Não precisamos mais criar um valor temporário e compará-lo com zero, o que era apenas um detalhe desinteressante. Parte do poder das linguagens de programação é que elas podem cuidar de detalhes desinteressantes para nós.

No final do programa, após a construção `while` ter terminado, a operação `console.log` é usada para escrever o resultado.

Finalmente, aqui está como o programa poderia ficar se tivéssemos as operações convenientes `range` e `sum` disponíveis, que respectivamente criam uma coleção de números dentro de um intervalo e calculam a soma de uma coleção de números:

```
console.log(sum(range(1, 10)));
// → 55
```

A moral desta história é que o mesmo programa pode ser expresso de formas longas e curtas, ilegíveis e legíveis. A primeira versão do programa era extremamente obscura, enquanto esta última é quase inglês: `log` a `sum` do `range` de números de 1 a 10. (Veremos em capítulos posteriores como definir operações como `sum` e `range`.)

Uma boa linguagem de programação ajuda o programador ao permitir que fale sobre as ações que o computador deve realizar em um nível mais alto.

Ajuda a omitir detalhes, fornece blocos de construção convenientes (como `while` e `console.log`), permite que você defina seus próprios blocos de construção (como `sum` e `range`) e torna esses blocos fáceis de compor.

O que é JavaScript?

JavaScript foi introduzido em 1995 como uma forma de adicionar programas a páginas web no navegador Netscape Navigator. A linguagem foi desde então adotada por todos os outros principais navegadores web gráficos. Tornou possíveis as aplicações web modernas — ou seja, aplicações com as quais você pode interagir diretamente sem precisar recarregar a página a cada ação. JavaScript também é usado em sites mais tradicionais para fornecer diversas formas de interatividade e engenhosidade.

É importante notar que JavaScript não tem quase nada a ver com a linguagem de programação chamada Java. O nome similar foi inspirado por considerações de marketing em vez de bom senso. Quando o JavaScript estava sendo introduzido, a linguagem Java estava sendo fortemente divulgada e ganhando popularidade. Alguém achou que seria uma boa ideia tentar surfar nesse sucesso. Agora estamos presos ao nome.

Após sua adoção fora da Netscape, um documento de padrão foi escrito para descrever a forma como a linguagem JavaScript deveria funcionar, para que os diversos softwares que afirmavam suportar JavaScript pudessem ter certeza de que realmente forneciam a mesma linguagem. Isso é chamado de padrão ECMAScript, em homenagem à organização Ecma International que conduziu a padronização. Na prática, os termos ECMAScript e JavaScript podem ser usados de forma intercambiável — são dois nomes para a mesma linguagem.

Há quem diga coisas *terríveis* sobre JavaScript. Muitas dessas coisas são verdade. Quando fui obrigado a escrever algo em JavaScript pela primeira vez, rapidamente passei a desprezá-lo. Ele aceitava quase qualquer coisa que eu digitava, mas interpretava de uma forma completamente diferente do que eu queria dizer. Isso tinha muito a ver com o fato de que eu não fazia ideia do que estava fazendo, é claro, mas há um problema real aqui: JavaScript é ridiculamente liberal no que permite. A ideia por trás desse design era que tornaria a programação em JavaScript mais fácil para iniciantes. Na realidade, isso geralmente torna mais difícil encontrar problemas em seus programas, porque o sistema não os apontará para você.

Essa flexibilidade também tem suas vantagens, no entanto. Ela deixa espaço para técnicas que são impossíveis em linguagens mais rígidas e proporciona um estilo de programação agradável e informal. Depois de aprender a linguagem

adequadamente e trabalhar com ela por um tempo, passei a realmente *gostar* de JavaScript.

Houve várias versões de JavaScript. A versão 3 do ECMAScript era a versão amplamente suportada durante a ascensão do JavaScript ao domínio, aproximadamente entre 2000 e 2010. Durante esse período, estava em andamento o trabalho em uma ambiciosa versão 4, que planejava uma série de melhorias e extensões radicais à linguagem. Mudar uma linguagem viva e amplamente usada de forma tão radical acabou sendo politicamente difícil, e o trabalho na versão 4 foi abandonado em 2008. Uma versão 5 muito menos ambiciosa, que fez apenas algumas melhorias incontroverssas, saiu em 2009. Em 2015, saiu a versão 6, uma atualização importante que incluiu algumas das ideias planejadas para a versão 4. Desde então, temos tido atualizações novas e pequenas a cada ano.

O fato de JavaScript estar evoluindo significa que os navegadores precisam acompanhar constantemente. Se você está usando um navegador mais antigo, ele pode não suportar todos os recursos. Os designers da linguagem são cuidadosos para não fazer nenhuma mudança que possa quebrar programas existentes, então novos navegadores ainda podem executar programas antigos. Neste livro, estou usando a versão 2024 do JavaScript.

Navegadores web não são as únicas plataformas em que JavaScript é usado. Alguns bancos de dados, como MongoDB e CouchDB, usam JavaScript como sua linguagem de scripting e consulta. Várias plataformas para programação desktop e servidor, mais notavelmente o projeto Node.js (assunto do Capítulo 20), fornecem um ambiente para programar JavaScript fora do navegador.

Código, e o que fazer com ele

Código é o texto que compõe programas. A maioria dos capítulos neste livro contém bastante código. Acredito que ler código e escrever código são partes indispensáveis de aprender a programar. Tente não apenas passar os olhos pelos exemplos — leia-os atentamente e entenda-os. Isso pode ser lento e confuso no início, mas prometo que você rapidamente pegará o jeito. O mesmo vale para os exercícios. Não presuma que os entendeu até ter realmente escrito uma solução funcional.

Recomendo que você tente suas soluções para os exercícios em um interpretador JavaScript real. Dessa forma, você terá feedback imediato sobre se o que está fazendo está funcionando e, espero, ficará tentado a experimentar e ir além dos exercícios.

A forma mais fácil de executar o código de exemplo do livro — e de experi-

mentar com ele — é procurá-lo na versão online do livro em *https://eloquentjavascript.net*. Lá, você pode clicar em qualquer exemplo de código para editá-lo e executá-lo e ver a saída que ele produz. Para trabalhar nos exercícios, vá para *https://eloquentjavascript.net/code*, que fornece código inicial para cada exercício de codificação e permite que você veja as soluções.

Executar os programas definidos neste livro fora do site do livro requer algum cuidado. Muitos exemplos são autossuficientes e devem funcionar em qualquer ambiente JavaScript. Mas código em capítulos posteriores é frequentemente escrito para um ambiente específico (o navegador ou Node.js) e só pode ser executado lá. Além disso, muitos capítulos definem programas maiores, e os trechos de código que aparecem neles dependem uns dos outros ou de arquivos externos. O sandbox no site fornece links para arquivos ZIP contendo todos os scripts e arquivos de dados necessários para executar o código de um determinado capítulo.

Visão geral deste livro

Este livro contém aproximadamente três partes. Os primeiros 12 capítulos discutem a linguagem JavaScript. Os sete capítulos seguintes são sobre navegadores web e a forma como JavaScript é usado para programá-los. Finalmente, dois capítulos são dedicados ao Node.js, outro ambiente para programar JavaScript. Há cinco *capítulos de projeto* no livro que descrevem programas de exemplo maiores para dar a você uma amostra de programação real.

A parte sobre a linguagem do livro começa com quatro capítulos que introduzem a estrutura básica da linguagem JavaScript. Eles discutem estruturas de controle (como a palavra `while` que você viu nesta introdução), funções (escrever seus próprios blocos de construção) e estruturas de dados. Depois desses, você será capaz de escrever programas básicos. Em seguida, os Capítulos 5 e 6 introduzem técnicas para usar funções e objetos para escrever código mais *abstrato* e manter a complexidade sob controle.

Após um primeiro capítulo de projeto que constrói um robô de entregas rudimentar, a parte sobre a linguagem do livro continua com capítulos sobre tratamento de erros e correção de bugs, expressões regulares (uma ferramenta importante para trabalhar com texto), modularidade (outra defesa contra a complexidade) e programação assíncrona (lidando com eventos que levam tempo). O segundo capítulo de projeto, onde implementamos uma linguagem de programação, conclui a primeira parte do livro.

A segunda parte do livro, Capítulos 13 a 19, descreve as ferramentas que o JavaScript no navegador tem acesso. Você aprenderá a exibir coisas na tela

(Capítulos 14 e 17), responder a entrada do usuário (Capítulo 15) e se comunicar pela rede (Capítulo 18). Há novamente dois capítulos de projeto nesta parte: construir um jogo de plataforma e um programa de pintura de pixels.

O Capítulo 20 descreve Node.js, e o Capítulo 21 constrói um pequeno site usando essa ferramenta.

Convenções tipográficas

Neste livro, texto escrito em uma fonte `monoespaçada` representará elementos de programas. Às vezes são fragmentos autossuficientes, e às vezes apenas se referem a parte de um programa próximo. Programas (dos quais você já viu alguns) são escritos da seguinte forma:

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

Às vezes, para mostrar a saída que um programa produz, a saída esperada é escrita depois dele, com duas barras e uma seta na frente.

```
console.log(factorial(8));
// → 40320
```

Boa sorte!

> *"Below the surface of the machine, the program moves. Without effort, it expands and contracts. In great harmony, electrons scatter and regroup. The forms on the monitor are but ripples on the water. The essence stays invisibly below."*
>
> —Master Yuan-Ma, The Book of Programming

Chapter 1

Valores, Tipos e Operadores

No mundo do computador, existem apenas dados. Você pode ler dados, modificar dados, criar novos dados — mas aquilo que não são dados não pode ser mencionado. Todos esses dados são armazenados como longas sequências de bits e, portanto, são fundamentalmente semelhantes.

Bits são qualquer tipo de coisa com dois valores, geralmente descritos como zeros e uns. Dentro do computador, eles assumem formas como uma carga elétrica alta ou baixa, um sinal forte ou fraco, ou um ponto brilhante ou opaco na superfície de um CD. Qualquer pedaço de informação discreta pode ser reduzido a uma sequência de zeros e uns e, assim, representado em bits.

Por exemplo, podemos expressar o número 13 em bits. Isso funciona da mesma forma que um número decimal, mas em vez de 10 dígitos diferentes, temos apenas 2, e o peso de cada um aumenta por um fator de 2 da direita para a esquerda. Aqui estão os bits que compõem o número 13, com os pesos dos dígitos mostrados abaixo deles:

```
   0   0   0   0   1   1   0   1
 128  64  32  16   8   4   2   1
```

Esse é o número binário 00001101. Seus dígitos diferentes de zero representam 8, 4 e 1, e somam 13.

Valores

Imagine um mar de bits — um oceano deles. Um computador moderno típico tem mais de 100 bilhões de bits em seu armazenamento volátil de dados (memória de trabalho). O armazenamento não volátil (o disco rígido ou equivalente) tende a ter ainda algumas ordens de magnitude a mais.

Para poder trabalhar com tais quantidades de bits sem se perder, nós os separamos em pedaços que representam partes de informação. Em um ambiente JavaScript, esses pedaços são chamados de *valores*. Embora todos os valores sejam feitos de bits, eles desempenham papéis diferentes. Cada valor tem um

tipo que determina seu papel. Alguns valores são números, alguns valores são pedaços de texto, alguns valores são funções, e assim por diante.

Para criar um valor, você precisa simplesmente invocar seu nome. Isso é conveniente. Você não precisa juntar material de construção para seus valores ou pagar por eles. Você apenas chama por um, e *whoosh*, você o tem. Claro, valores não são realmente criados do nada. Cada um precisa ser armazenado em algum lugar, e se você quiser usar um número gigantesco deles ao mesmo tempo, pode ficar sem memória no computador. Felizmente, isso é um problema apenas se você precisar de todos simultaneamente. Assim que você não usar mais um valor, ele se dissipará, deixando para trás seus bits para serem reciclados como material de construção para a próxima geração de valores.

O restante deste capítulo introduz os elementos atômicos dos programas JavaScript, isto é, os tipos simples de valores e os operadores que podem agir sobre esses valores.

Números

Valores do tipo *number* são, sem surpresa, valores numéricos. Em um programa JavaScript, eles são escritos da seguinte forma:

```
13
```

Usar isso em um programa fará com que o padrão de bits para o número 13 passe a existir dentro da memória do computador.

JavaScript usa um número fixo de bits, 64 deles, para armazenar um único valor numérico. Existem apenas tantos padrões que você pode fazer com 64 bits, o que limita a quantidade de números diferentes que podem ser representados. Com N dígitos decimais, você pode representar 10^{N} números. De forma similar, dados 64 dígitos binários, você pode representar 2^{64} números diferentes, o que é cerca de 18 quintilhões (um 18 com 18 zeros depois). Isso é muito.

A memória dos computadores costumava ser muito menor, e as pessoas tendiam a usar grupos de 8 ou 16 bits para representar seus números. Era fácil acidentalmente causar *overflow* em números tão pequenos — acabar com um número que não cabia no número dado de bits. Hoje, até computadores que cabem no seu bolso têm memória de sobra, então você é livre para usar blocos de 64 bits e só precisa se preocupar com overflow ao lidar com números verdadeiramente astronômicos.

Nem todos os números inteiros menores que 18 quintilhões cabem em um número JavaScript, no entanto. Esses bits também armazenam números nega-

tivos, então um bit indica o sinal do número. Uma questão maior é representar números não inteiros. Para fazer isso, alguns dos bits são usados para armazenar a posição do ponto decimal. O número inteiro máximo real que pode ser armazenado está mais na faixa de 9 quadrilhões (15 zeros) — o que ainda é agradavelmente enorme.

Números fracionários são escritos usando um ponto:

```
9.81
```

Para números muito grandes ou muito pequenos, você também pode usar notação científica adicionando um *e* (de *expoente*), seguido do expoente do número.

```
2.998e8
```

Isso é $2{,}998 \times 10^8 = 299.800.000$.

Cálculos com números inteiros (também chamados de *inteiros*) que são menores que os 9 quadrilhões mencionados são garantidamente sempre precisos. Infelizmente, cálculos com números fracionários geralmente não são. Assim como π (pi) não pode ser expresso com precisão por um número finito de dígitos decimais, muitos números perdem alguma precisão quando apenas 64 bits estão disponíveis para armazená-los. Isso é uma pena, mas causa problemas práticos apenas em situações específicas. O importante é estar ciente disso e tratar números digitais fracionários como aproximações, não como valores precisos.

Aritmética

A principal coisa a se fazer com números é aritmética. Operações aritméticas como adição ou multiplicação pegam dois valores numéricos e produzem um novo número a partir deles. Aqui está como elas se parecem em JavaScript:

```
100 + 4 * 11
```

Os símbolos `+` e `*` são chamados de *operadores*. O primeiro representa adição e o segundo representa multiplicação. Colocar um operador entre dois valores o aplicará a esses valores e produzirá um novo valor.

Este exemplo significa "some 4 e 100, e multiplique o resultado por 11", ou a multiplicação é feita antes da adição? Como você deve ter adivinhado, a multiplicação acontece primeiro. Como na matemática, você pode mudar isso envolvendo a adição em parênteses.

```
(100 + 4) * 11
```

Para subtração, existe o operador `-`. Divisão pode ser feita com o operador `/`.

Quando operadores aparecem juntos sem parênteses, a ordem em que são aplicados é determinada pela *precedência* dos operadores. O exemplo mostra que a multiplicação vem antes da adição. O operador `/` tem a mesma precedência que `*`. Da mesma forma, `+` e `-` têm a mesma precedência. Quando múltiplos operadores com a mesma precedência aparecem lado a lado, como em `1 - 2 + 1`, eles são aplicados da esquerda para a direita: `(1 - 2)+ 1`.

Não se preocupe demais com essas regras de precedência. Quando tiver dúvida, apenas adicione parênteses.

Existe mais um operador aritmético, que você pode não reconhecer imediatamente. O símbolo `%` é usado para representar a operação de *resto*. `X % Y` é o resto da divisão de `X` por `Y`. Por exemplo, `314 % 100` produz `14`, e `144 % 12` dá `0`. A precedência do operador resto é a mesma da multiplicação e divisão. Você também verá frequentemente esse operador referido como *módulo*.

Números especiais

Existem três valores especiais em JavaScript que são considerados números mas não se comportam como números normais. Os dois primeiros são `Infinity` e `-Infinity`, que representam os infinitos positivo e negativo. `Infinity - 1` ainda é `Infinity`, e assim por diante. Não confie demais em computação baseada em infinito, porém. Ela não é matematicamente sólida e rapidamente levará ao próximo número especial: `NaN`.

`NaN` significa "not a number" (não é um número), embora *seja* um valor do tipo number. Você obterá esse resultado quando, por exemplo, tentar calcular `0 / 0` (zero dividido por zero), `Infinity - Infinity`, ou qualquer outra operação numérica que não produza um resultado significativo.

Strings

O próximo tipo básico de dados é a *string*. Strings são usadas para representar texto. Elas são escritas envolvendo seu conteúdo em aspas.

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

Você pode usar aspas simples, aspas duplas ou crases para marcar strings, desde que as aspas no início e no final da string correspondam.

Você pode colocar quase qualquer coisa entre aspas para fazer JavaScript criar um valor de string a partir disso. Mas alguns caracteres são mais difíceis. Você pode imaginar como colocar aspas dentro de aspas pode ser complicado,

já que elas parecerão o final da string. *Novas linhas* (os caracteres que você obtém quando pressiona ENTER) podem ser incluídas apenas quando a string está entre crases (`).

Para tornar possível incluir tais caracteres em uma string, a seguinte notação é usada: uma barra invertida (\) dentro de texto entre aspas indica que o caractere depois dela tem um significado especial. Isso é chamado de *escapar* o caractere. Uma aspas precedida por uma barra invertida não encerrará a string, mas fará parte dela. Quando um caractere `n` aparece após uma barra invertida, ele é interpretado como uma nova linha. Da mesma forma, um `t` após uma barra invertida significa um caractere de tabulação. Considere a seguinte string:

```
"This is the first line\nAnd this is the second"
```

Este é o texto real nessa string:

```
This is the first line
And this is the second
```

Existem, é claro, situações em que você quer que uma barra invertida em uma string seja apenas uma barra invertida, não um código especial. Se duas barras invertidas se seguem, elas se colapsam juntas, e apenas uma ficará no valor da string resultante. É assim que a string "*Um caractere de nova linha é escrito como "\n".*" pode ser expressa:

```
"A newline character is written like \"\\n\"."
```

Strings também precisam ser modeladas como uma série de bits para poder existir dentro do computador. A forma como JavaScript faz isso é baseada no padrão *Unicode*. Esse padrão atribui um número a virtualmente todo caractere que você possa precisar, incluindo caracteres do grego, árabe, japonês, armênio, e assim por diante. Se temos um número para cada caractere, uma string pode ser descrita por uma sequência de números. E é isso que o JavaScript faz.

Há uma complicação, porém: a representação do JavaScript usa 16 bits por elemento de string, o que pode descrever até 2^{16} caracteres diferentes. No entanto, o Unicode define mais caracteres que isso — cerca de duas vezes mais, neste momento. Então alguns caracteres, como muitos emoji, ocupam duas "posições de caractere" em strings JavaScript. Voltaremos a isso no Capítulo 5.

Strings não podem ser divididas, multiplicadas ou subtraídas. O operador `+` *pode* ser usado nelas, não para somar, mas para *concatenar* — colar duas strings juntas. A linha a seguir produzirá a string `"concatenate"`:

```
"con" + "cat" + "e" + "nate"
```

Valores de string têm uma série de funções associadas (*métodos*) que podem ser usadas para realizar outras operações nelas. Falarei mais sobre isso no Capítulo 4.

Strings escritas com aspas simples ou duplas se comportam de forma muito semelhante — a única diferença está em qual tipo de aspas você precisa escapar dentro delas. Strings entre crases, geralmente chamadas de *template literals*, podem fazer mais algumas coisas. Além de poderem abranger várias linhas, elas também podem incorporar outros valores.

```
`half of 100 is ${100 / 2}`
```

Quando você escreve algo dentro de `${}` em um template literal, seu resultado será calculado, convertido em uma string e incluído naquela posição. Este exemplo produz a string `"half of 100 is 50"`.

Operadores unários

Nem todos os operadores são símbolos. Alguns são escritos como palavras. Um exemplo é o operador `typeof`, que produz um valor de string nomeando o tipo do valor que você lhe dá.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

Usaremos `console.log` no código de exemplo para indicar que queremos ver o resultado de avaliar algo. (Mais sobre isso no próximo capítulo.)

Os outros operadores mostrados até agora neste capítulo operavam todos sobre dois valores, mas `typeof` recebe apenas um. Operadores que usam dois valores são chamados de operadores *binários*, enquanto aqueles que recebem um são chamados de operadores *unários*. O operador menos (`-`) pode ser usado tanto como operador binário quanto como operador unário.

```
console.log(- (10 - 2))
// → -8
```

Valores booleanos

Muitas vezes é útil ter um valor que distingue entre apenas duas possibilidades, como “sim” e “não” ou “ligado” e “desligado”. Para esse propósito, JavaScript tem um tipo *Boolean*, que possui apenas dois valores, true e false, escritos como essas palavras.

Comparação

Aqui está uma forma de produzir valores booleanos:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

Os sinais `>` e `<` são os símbolos tradicionais para "é maior que” e "é menor que”, respectivamente. Eles são operadores binários. Aplicá-los resulta em um valor booleano que indica se eles são verdadeiros neste caso.

Strings podem ser comparadas da mesma forma.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

A forma como strings são ordenadas é aproximadamente alfabética, mas não realmente o que você esperaria ver em um dicionário: letras maiúsculas são sempre “menores” que minúsculas, então `"Z" < "a"`, e caracteres não-alfabéticos (!, -, e assim por diante) também estão incluídos na ordenação. Ao comparar strings, JavaScript percorre os caracteres da esquerda para a direita, comparando os códigos Unicode um por um.

Outros operadores similares são `>=` (maior ou igual a), `<=` (menor ou igual a), `==` (igual a) e `!=` (diferente de).

```
console.log("Garnet" != "Ruby")
// → true
console.log("Pearl" == "Amethyst")
// → false
```

Existe apenas um valor em JavaScript que não é igual a si mesmo, e esse é `NaN` (“not a number”).

```
console.log(NaN == NaN)
// → false
```

`NaN` supostamente denota o resultado de uma computação sem sentido e, como tal, não é igual ao resultado de nenhuma *outra* computação sem sentido.

Operadores lógicos

Existem também algumas operações que podem ser aplicadas a valores booleanos em si. JavaScript suporta três operadores lógicos: *e*, *ou* e *não*. Eles podem ser usados para "raciocinar" sobre booleanos.

O operador `&&` representa o *e* lógico. É um operador binário, e seu resultado é verdadeiro apenas se ambos os valores dados a ele forem verdadeiros.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

O operador `||` denota o *ou* lógico. Ele produz verdadeiro se qualquer um dos valores dados a ele for verdadeiro.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

Não é escrito como um ponto de exclamação (`!`). É um operador unário que inverte o valor dado a ele — `!true` produz `false` e `!false` produz `true`.

Ao misturar esses operadores booleanos com aritmética e outros operadores, nem sempre é óbvio quando parênteses são necessários. Na prática, você geralmente consegue se virar sabendo que dos operadores que vimos até agora, `||` tem a menor precedência, depois vem `&&`, depois os operadores de comparação (`>`, `==`, e assim por diante), e depois o resto. Essa ordem foi escolhida de modo que, em expressões típicas como a seguinte, o mínimo de parênteses possível seja necessário:

```
1 + 1 == 2 && 10 * 10 > 50
```

O último operador lógico que veremos não é unário, nem binário, mas *ternário*, operando sobre três valores. É escrito com um ponto de interrogação e dois-pontos, assim:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Este é chamado de operador *condicional* (ou às vezes apenas *o operador ternário* já que é o único operador desse tipo na linguagem). O operador usa o valor à esquerda do ponto de interrogação para decidir qual dos dois outros valores "escolher". Se você escrever `a ? b : c`, o resultado será `b` quando `a` for verdadeiro e `c` caso contrário.

Valores vazios

Existem dois valores especiais, escritos `null` e `undefined`, que são usados para denotar a ausência de um valor *significativo*. Eles são valores em si, mas não carregam nenhuma informação.

Muitas operações na linguagem que não produzem um valor significativo produzem `undefined` simplesmente porque precisam produzir *algum* valor.

A diferença de significado entre `undefined` e `null` é um acidente do design do JavaScript, e na maioria das vezes não importa. Nos casos em que você realmente precisa se preocupar com esses valores, recomendo tratá-los como praticamente intercambiáveis.

Conversão automática de tipos

Na introdução, mencionei que JavaScript se esforça para aceitar quase qualquer programa que você lhe dá, mesmo programas que fazem coisas estranhas. Isso é bem demonstrado pelas seguintes expressões:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

Quando um operador é aplicado ao tipo "errado" de valor, JavaScript converterá silenciosamente esse valor para o tipo que precisa, usando um conjunto de regras que frequentemente não são o que você quer ou espera. Isso é chamado de *coerção de tipos*. O `null` na primeira expressão se torna `0` e o `"5"` na segunda expressão se torna `5` (de string para número). Porém, na terceira expressão, `+` tenta concatenação de string antes de adição numérica, então o `1` é convertido

para `"1"` (de número para string).

Quando algo que não mapeia para um número de forma óbvia (como `"five"` ou `undefined`) é convertido para um número, você obtém o valor `NaN`. Operações aritméticas subsequentes sobre `NaN` continuam produzindo `NaN`, então se você encontrar um desses em um lugar inesperado, procure por conversões de tipo acidentais.

Ao comparar valores do mesmo tipo usando o operador `==`, o resultado é fácil de prever: você deve obter verdadeiro quando ambos os valores são iguais, exceto no caso de `NaN`. Mas quando os tipos diferem, JavaScript usa um conjunto complicado e confuso de regras para determinar o que fazer. Na maioria dos casos, ele apenas tenta converter um dos valores para o tipo do outro valor. No entanto, quando `null` ou `undefined` aparece em qualquer lado do operador, ele produz verdadeiro apenas se ambos os lados forem `null` ou `undefined`.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

Esse comportamento é frequentemente útil. Quando você quer testar se um valor tem um valor real em vez de `null` ou `undefined`, você pode compará-lo com `null` usando o operador `==` ou `!=`.

E se você quiser testar se algo se refere ao valor preciso `false`? Expressões como `0 == false` e `"" == false` também são verdadeiras por causa da conversão automática de tipos. Quando você *não* quer que nenhuma conversão de tipo aconteça, existem dois operadores adicionais: `===` e `!==`. O primeiro testa se um valor é *precisamente* igual ao outro, e o segundo testa se não é precisamente igual. Assim, `"" === false` é falso, como esperado.

Recomendo usar os operadores de comparação de três caracteres defensivamente para prevenir conversões de tipo inesperadas que possam lhe causar problemas. Mas quando você tem certeza de que os tipos em ambos os lados serão os mesmos, não há problema em usar os operadores mais curtos.

Curto-circuito de operadores lógicos

Os operadores lógicos `&&` e `||` lidam com valores de diferentes tipos de uma forma peculiar. Eles converterão o valor do seu lado esquerdo para o tipo booleano para decidir o que fazer, mas dependendo do operador e do resultado dessa conversão, eles retornarão o valor *original* do lado esquerdo ou o valor do lado direito.

O operador `||`, por exemplo, retornará o valor à sua esquerda quando esse

valor puder ser convertido para verdadeiro e retornará o valor à sua direita caso contrário. Isso tem o efeito esperado quando os valores são booleanos e faz algo análogo para valores de outros tipos.

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

Podemos usar essa funcionalidade como uma forma de recorrer a um valor padrão. Se você tem um valor que pode estar vazio, pode colocar `||` depois dele com um valor de substituição. Se o valor inicial puder ser convertido para falso, você obterá a substituição em vez dele. As regras para converter strings e números para valores booleanos estabelecem que `0`, `NaN` e a string vazia (`""`) contam como falso, enquanto todos os outros valores contam como verdadeiro. Isso significa que `0 || -1` produz `-1`, e `"" || "!?"` produz `"!?"`.

O operador `??` se assemelha a `||` mas retorna o valor da direita apenas se o da esquerda for `null` ou `undefined`, não se for algum outro valor que possa ser convertido para `false`. Frequentemente, isso é preferível ao comportamento de `||`.

```
console.log(0 || 100);
// → 100
console.log(0 ?? 100);
// → 0
console.log(null ?? 100);
// → 100
```

O operador `&&` funciona de forma similar, mas ao contrário. Quando o valor à sua esquerda é algo que se converte para falso, ele retorna esse valor; caso contrário, retorna o valor à sua direita.

Outra propriedade importante desses dois operadores é que a parte à sua direita é avaliada apenas quando necessário. No caso de `true || X`, não importa o que `X` seja — mesmo que seja um pedaço de programa que faz algo *terrível* — o resultado será verdadeiro, e `X` nunca é avaliado. O mesmo vale para `false && X`, que é falso e ignorará `X`. Isso é chamado de *avaliação de curto-circuito*.

O operador condicional funciona de forma similar. Do segundo e terceiro valores, apenas o que é selecionado é avaliado.

Resumo

Vimos quatro tipos de valores JavaScript neste capítulo: números, strings, booleanos e valores indefinidos. Tais valores são criados digitando seu nome (`true`, `null`) ou valor (`13`, `"abc"`).

Você pode combinar e transformar valores com operadores. Vimos operadores binários para aritmética (`+`, `-`, `*`, `/` e `%`), concatenação de strings (`+`), comparação (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) e lógica (`&&`, `||`, `??`), bem como vários operadores unários (`-` para negar um número, `!` para negar logicamente e `typeof` para encontrar o tipo de um valor) e um operador ternário (`?:`) para escolher um de dois valores com base em um terceiro valor.

Isso lhe dá informação suficiente para usar JavaScript como uma calculadora de bolso, mas não muito mais. O próximo capítulo começará a unir essas expressões em programas básicos.

> *"And my heart glows bright red under my filmy, translucent skin and they have to administer 10cc of JavaScript to get me to come back. (I respond well to toxins in the blood.) Man, that stuff will kick the peaches right out your gills!"*
>
> —_why, Why's (Poignant) Guide to Ruby

Chapter 2

Estrutura do Programa

Neste capítulo, começaremos a fazer coisas que podem realmente ser chamadas de *programação*. Expandiremos nosso domínio da linguagem JavaScript além dos substantivos e fragmentos de frases que vimos até agora, até o ponto em que podemos expressar prosa significativa.

Expressões e instruções

No Capítulo 1, criamos valores e aplicamos operadores a eles para obter novos valores. Criar valores dessa forma é a substância principal de qualquer programa JavaScript. Mas essa substância precisa ser enquadrada em uma estrutura maior para ser útil. É isso que abordaremos neste capítulo.

Um fragmento de código que produz um valor é chamado de *expressão*. Todo valor escrito literalmente (como `22` ou `"psychoanalysis"`) é uma expressão. Uma expressão entre parênteses também é uma expressão, assim como um operador binário aplicado a duas expressões ou um operador unário aplicado a uma.

Isso mostra parte da beleza de uma interface baseada em linguagem. Expressões podem conter outras expressões de forma similar a como subfrases em linguagens humanas são aninhadas — uma subfrase pode conter suas próprias subfrases, e assim por diante. Isso nos permite construir expressões que descrevem computações arbitrariamente complexas.

Se uma expressão corresponde a um fragmento de frase, uma *instrução* JavaScript corresponde a uma frase completa. Um programa é uma lista de instruções.

O tipo mais simples de instrução é uma expressão com um ponto e vírgula depois dela. Este é um programa:

```
1;
!false;
```

É um programa inútil, porém. Uma expressão pode se contentar em apenas

produzir um valor, que pode então ser usado pelo código que a envolve. No entanto, uma instrução existe por si só, então se ela não afetar o mundo, é inútil. Ela pode exibir algo na tela, como com `console.log`, ou mudar o estado da máquina de uma forma que afetará as instruções que vêm depois dela. Essas mudanças são chamadas de *efeito colateral*. As instruções no exemplo anterior apenas produzem os valores `1` e `true` e depois os jogam fora imediatamente. Isso não deixa nenhuma impressão no mundo. Quando você executa esse programa, nada observável acontece.

Em alguns casos, JavaScript permite que você omita o ponto e vírgula no final de uma instrução. Em outros casos, ele precisa estar lá, ou a próxima linha será tratada como parte da mesma instrução. As regras para quando ele pode ser omitido com segurança são um tanto complexas e propensas a erros. Então, neste livro, toda instrução que precisa de um ponto e vírgula sempre terá um. Recomendo que você faça o mesmo, pelo menos até ter aprendido mais sobre as sutilezas dos pontos e vírgulas ausentes.

Bindings

Como um programa mantém um estado interno? Como ele lembra coisas? Vimos como produzir novos valores a partir de valores antigos, mas isso não muda os valores antigos, e o novo valor precisa ser usado imediatamente ou se dissipará novamente. Para capturar e manter valores, JavaScript fornece uma coisa chamada *binding*, ou *variável*.

```
let caught = 5 * 5;
```

Isso nos dá um segundo tipo de instrução. A palavra especial (*palavra-chave*) `let` indica que esta frase vai definir um binding. Ela é seguida pelo nome do binding e, se quisermos imediatamente lhe dar um valor, por um operador `=` e uma expressão.

O exemplo cria um binding chamado `caught` e o usa para agarrar o número que é produzido pela multiplicação de 5 por 5.

Após um binding ter sido definido, seu nome pode ser usado como uma expressão. O valor de tal expressão é o valor que o binding mantém no momento. Aqui está um exemplo:

```
let ten = 10;
console.log(ten * ten);
// → 100
```

Quando um binding aponta para um valor, isso não significa que está amarrado

a esse valor para sempre. O operador = pode ser usado a qualquer momento em bindings existentes para desconectá-los de seu valor atual e fazê-los apontar para um novo:

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```

Você deve imaginar bindings como tentáculos em vez de caixas. Eles não *contêm* valores; eles os *agarram* — dois bindings podem se referir ao mesmo valor. Um programa pode acessar apenas os valores aos quais ainda tem uma referência. Quando você precisa lembrar de algo, você ou faz crescer um tentáculo para segurá-lo ou reacopla um dos seus tentáculos existentes a ele.

Vejamos outro exemplo. Para lembrar o número de dólares que Luigi ainda lhe deve, você cria um binding. Quando ele paga $35 de volta, você dá a esse binding um novo valor.

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

Quando você define um binding sem lhe dar um valor, o tentáculo não tem nada para agarrar, então termina no ar. Se você pedir o valor de um binding vazio, obterá o valor `undefined`.

Uma única instrução `let` pode definir múltiplos bindings. As definições devem ser separadas por vírgulas:

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

As palavras `var` e `const` também podem ser usadas para criar bindings, de forma similar a `let`.

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

A primeira delas, `var` (abreviação de “variable”), é a forma como bindings eram declarados no JavaScript pré-2015, quando `let` ainda não existia. Voltarei à

forma precisa como difere de `let` no próximo capítulo. Por enquanto, lembre-se de que ela faz basicamente a mesma coisa, mas raramente a usaremos neste livro porque se comporta de forma estranha em algumas situações.

A palavra `const` significa *constante*. Ela define um binding constante, que aponta para o mesmo valor enquanto existir. Isso é útil para bindings que apenas dão um nome a um valor para que você possa facilmente se referir a ele depois.

Nomes de bindings

Nomes de binding podem ser qualquer sequência de uma ou mais letras. Dígitos podem fazer parte de nomes de binding — `catch22` é um nome válido, por exemplo — mas o nome não pode começar com um dígito. Um nome de binding pode incluir cifrões (`$`) ou sublinhados (`_`), mas nenhuma outra pontuação ou caractere especial.

Palavras com significado especial, como `let`, são *palavra-chaves* e não podem ser usadas como nomes de binding. Há também uma série de palavras que são "reservadas para uso" em versões futuros do JavaScript, que também não podem ser usadas como nomes de binding. A lista completa de palavras-chave e palavras reservadas é bastante longa:

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

Não se preocupe em memorizar essa lista. Quando criar um binding produzir um erro de sintaxe inesperado, verifique se você está tentando definir uma palavra reservada.

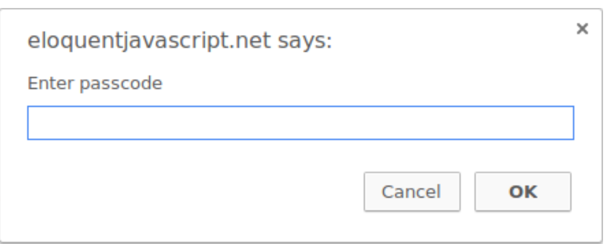
O ambiente

A coleção de bindings e seus valores que existem em um dado momento é chamada de *ambiente*. Quando um programa inicia, esse ambiente não está vazio. Ele sempre contém bindings que fazem parte do padrão da linguagem e, na maioria das vezes, também possui bindings que fornecem formas de interagir com o sistema ao redor. Por exemplo, em um navegador, existem funções para interagir com o site atualmente carregado e para ler entrada de mouse e teclado.

Funções

Muitos dos valores fornecidos no ambiente padrão têm o tipo *função*. Uma função é um pedaço de programa envolvido em um valor. Tais valores podem ser *aplicados* para executar o programa envolvido. Por exemplo, em um ambiente de navegador, o binding `prompt` contém uma função que mostra um pequeno diálogo pedindo entrada do usuário. Ela é usada assim:

```
prompt("Enter passcode");
```



Executar uma função é chamado de *invocar*, *chamar* ou *aplicar* a função. Você pode chamar uma função colocando parênteses após uma expressão que produz um valor de função. Geralmente você usará diretamente o nome do binding que contém a função. Os valores entre os parênteses são dados ao programa dentro da função. No exemplo, a função `prompt` usa a string que lhe damos como o texto a mostrar na caixa de diálogo. Valores dados a funções são chamados de *argumentos*. Funções diferentes podem precisar de um número ou tipos diferentes de argumentos.

A função `prompt` não é muito usada na programação web moderna, principalmente porque você não tem controle sobre a aparência do diálogo resultante, mas pode ser útil em programas de brinquedo e experimentos.

A função console.log

Nos exemplos, usei `console.log` para exibir valores. A maioria dos sistemas JavaScript (incluindo todos os navegadores web modernos e o Node.js) fornece uma função `console.log` que escreve seus argumentos em *algum* dispositivo de saída de texto. Nos navegadores, a saída vai para o console JavaScript. Essa parte da interface do navegador está oculta por padrão, mas a maioria dos navegadores a abre quando você pressiona F12 ou, em um Mac, COMMAND-OPTION-I. Se isso não funcionar, procure nos menus por um item chamado Developer Tools ou similar.

Embora nomes de binding não possam conter caractere ponto, `console.log`

tem um. Isso porque `console.log` não é um binding simples, mas uma expressão que recupera a propriedade `log` do valor mantido pelo binding `console`. Descobriremos exatamente o que isso significa no Capítulo 4.

Valores de retorno

Mostrar uma caixa de diálogo ou escrever texto na tela é um *efeito colateral*. Muitas funções são úteis por causa dos efeitos colaterais que produzem. Funções também podem produzir valores, caso em que não precisam ter um efeito colateral para ser úteis. Por exemplo, a função `Math.max` recebe qualquer quantidade de argumentos numéricos e retorna o maior.

```
console.log(Math.max(2, 4));
// → 4
```

Quando uma função produz um valor, diz-se que ela *retorna* esse valor. Qualquer coisa que produz um valor é uma expressão em JavaScript, o que significa que chamadas de função podem ser usadas dentro de expressões maiores. No código a seguir, uma chamada a `Math.min`, que é o oposto de `Math.max`, é usada como parte de uma expressão de soma:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

O Capítulo 3 explicará como escrever suas próprias funções.

Fluxo de controle

Quando seu programa contém mais de uma instrução, as instruções são executadas como se fossem uma história, de cima para baixo. Por exemplo, o programa a seguir tem duas instruções. A primeira pede ao usuário um número, e a segunda, que é executada após a primeira, mostra o quadrado desse número:

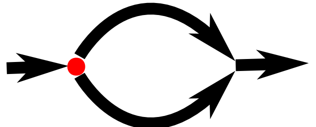
```
let theNumber = Number(prompt("Pick a number"));
console.log("Your number is the square root of " +
            theNumber * theNumber);
```

A função `Number` converte um valor para um número. Precisamos dessa conversão porque o resultado de `prompt` é um valor de string, e queremos um número. Existem funções similares chamadas `String` e `Boolean` que convertem valores para esses tipos.

Aqui está a representação esquemática bastante trivial do fluxo de controle em linha reta:

Execução condicional

Nem todos os programas são estradas retas. Podemos, por exemplo, querer criar uma estrada ramificada onde o programa toma o caminho adequado com base na situação em questão. Isso é chamado de *execução condicional*.



A execução condicional é criada com a palavra-chave `if` em JavaScript. No caso simples, queremos que algum código seja executado se, e somente se, uma certa condição for verdadeira. Podemos, por exemplo, querer mostrar o quadrado da entrada apenas se a entrada for realmente um número:

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

Com essa modificação, se você digitar "parrot", nenhuma saída é mostrada.

A palavra-chave `if` executa ou pula uma instrução dependendo do valor de uma expressão booleana. A expressão decisória é escrita após a palavra-chave, entre parênteses, seguida pela instrução a executar.

A função `Number.isNaN` é uma função padrão do JavaScript que retorna `true` apenas se o argumento que recebe for `NaN`. A função `Number` retorna `NaN` quando você lhe dá uma string que não representa um número válido. Assim, a condição se traduz como "a menos que `theNumber` não seja um número, faça isso".

A instrução após o `if` está envolvida em chaves (`{` e `}`) neste exemplo. As chaves podem ser usadas para agrupar qualquer número de instruções em uma única instrução, chamada de *bloco*. Você também poderia tê-las omitido neste caso, já que contêm apenas uma única instrução, mas para evitar ter que pensar se são necessárias, a maioria dos programadores JavaScript as usa em toda instrução envolvida como esta. Seguiremos essa convenção na maior parte deste livro, exceto pela ocasional instrução de uma linha.

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

Frequentemente você não terá apenas código que executa quando uma condição é verdadeira, mas também código que lida com o outro caso. Esse caminho alternativo é representado pela segunda seta no diagrama. Você pode usar a palavra-chave `else`, junto com `if`, para criar dois caminhos de execução separados e alternativos:

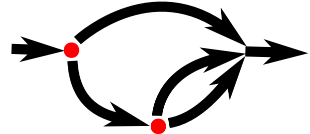
```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
  console.log("Hey. Why didn't you give me a number?");
}
```

Se você tiver mais de dois caminhos para escolher, pode "encadear" múltiplos pares `if`/`else` juntos. Aqui está um exemplo:

```
let num = Number(prompt("Pick a number"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

O programa primeiro verificará se `num` é menor que 10. Se for, ele escolhe esse caminho, mostra `"Small"` e termina. Se não for, ele toma o caminho `else`, que por sua vez contém um segundo `if`. Se a segunda condição (`< 100`) for verdadeira, isso significa que o número é pelo menos 10 mas menor que 100, e `"Medium"` é mostrado. Se não for, o segundo e último caminho `else` é escolhido.

O esquema para esse programa se parece com algo assim:

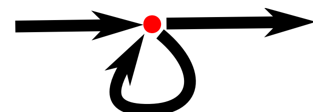


Loops while e do

Considere um programa que exibe todos os números pares de 0 a 12. Uma forma de escrever isso é a seguinte:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

Isso funciona, mas a ideia de escrever um programa é fazer *menos* trabalho, não mais. Se precisássemos de todos os números pares menores que 1.000, essa abordagem seria inviável. O que precisamos é de uma forma de executar um trecho de código múltiplas vezes. Essa forma de fluxo de controle é chamada de *loop*.



O fluxo de controle de loop nos permite voltar a algum ponto no programa onde estávamos antes e repeti-lo com nosso estado atual do programa. Se combinarmos isso com um binding que conta, podemos fazer algo assim:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
// …   etcetera
```

Uma instrução começando com a palavra-chave `while` cria um loop. A palavra `while` é seguida por uma expressão entre parênteses e depois uma instrução, assim como `if`. O loop continua entrando nessa instrução enquanto a expressão produzir um valor que resulte em `true` quando convertido para booleano.

O binding `number` demonstra a forma como um binding pode acompanhar o progresso de um programa. Toda vez que o loop se repete, `number` recebe um valor que é 2 a mais que seu valor anterior. No início de cada repetição, ele é comparado com o número 12 para decidir se o trabalho do programa está

terminado.

Como um exemplo que realmente faz algo útil, agora podemos escrever um programa que calcula e mostra o valor de 2^{10} (2 elevado à 10ª potência). Usamos dois bindings: um para acompanhar nosso resultado e um para contar quantas vezes multiplicamos esse resultado por 2. O loop testa se o segundo binding já atingiu 10 e, se não, atualiza ambos os bindings.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

O contador também poderia ter começado em `1` e verificado `<= 10`, mas por razões que ficarão aparentes no Capítulo 4, é uma boa ideia se acostumar a contar a partir de 0.

Note que JavaScript também tem um operador para exponenciação (`2 ** 10`), que você usaria para calcular isso em código real — mas isso teria arruinado o exemplo.

Um loop `do` é uma estrutura de controle similar ao loop `while`. Difere apenas em um ponto: um loop `do` sempre executa seu corpo pelo menos uma vez, e começa a testar se deve parar apenas após essa primeira execução. Para refletir isso, o teste aparece após o corpo do loop:

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log("Hello " + yourName);
```

Este programa forçará você a inserir um nome. Ele perguntará repetidamente até obter algo que não seja uma string vazia. Aplicar o operador `!` converterá um valor para o tipo booleano antes de negá-lo, e todas as strings exceto `""` se convertem para `true`. Isso significa que o loop continua rodando até que você forneça um nome não vazio.

Indentando Código

Nos exemplos, tenho adicionado espaços na frente de instruções que fazem parte de alguma instrução maior. Esses espaços não são necessários — o computador

aceitará o programa perfeitamente sem eles. Na verdade, até as quebras de linha em programas são opcionais. Você poderia escrever um programa como uma única linha longa se quisesse.

O papel dessa indentação dentro de blocos é fazer a estrutura do código se destacar para leitores humanos. Em código onde novos blocos são abertos dentro de outros blocos, pode se tornar difícil ver onde um bloco termina e outro começa. Com indentação adequada, a forma visual de um programa corresponde à forma dos blocos dentro dele. Gosto de usar dois espaços para cada bloco aberto, mas gostos variam — algumas pessoas usam quatro espaços, e algumas usam caractere de tabulação. O importante é que cada novo bloco adicione a mesma quantidade de espaço.

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

A maioria dos editores de código ajudará indentando automaticamente novas linhas na quantidade adequada.

Loops for

Muitos loops seguem o padrão mostrado nos exemplos de `while`. Primeiro um binding "contador" é criado para acompanhar o progresso do loop. Depois vem um loop `while`, geralmente com uma expressão de teste que verifica se o contador atingiu seu valor final. No final do corpo do loop, o contador é atualizado para acompanhar o progresso.

Como esse padrão é tão comum, JavaScript e linguagens similares fornecem uma forma ligeiramente mais curta e mais abrangente, o loop `for`:

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
// …   etcetera
```

Este programa é exatamente equivalente ao exemplo anterior de impressão de números pares. A única mudança é que todas as instruções que são relacionadas ao "estado" do loop estão agrupadas juntas após `for`.

Os parênteses após uma palavra-chave `for` devem conter dois ponto e vírgulas. A parte antes do primeiro ponto e vírgula *inicializa* o loop, geralmente definindo um binding. A segunda parte é a expressão que *verifica* se o loop deve continuar. A parte final *atualiza* o estado do loop após cada iteração. Na maioria dos casos, isso é mais curto e claro que uma construção `while`.

Este é o código que calcula 2^{10} usando `for` em vez de `while`:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
  result = result * 2;
}
console.log(result);
// → 1024
```

Saindo de um Loop

Fazer a condição do loop produzir `false` não é a única forma de um loop terminar. A instrução `break` tem o efeito de imediatamente saltar para fora do loop que a envolve. Seu uso é demonstrado no programa a seguir, que encontra o primeiro número que é tanto maior ou igual a 20 quanto divisível por 7:

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

Usar o operador resto (`%`) é uma forma fácil de testar se um número é divisível por outro número. Se for, o resto da divisão é zero.

A construção `for` no exemplo não tem uma parte que verifica o final do loop. Isso significa que o loop nunca parará a menos que a instrução `break` dentro dele seja executada.

Se você removesse essa instrução `break` ou acidentalmente escrevesse uma condição final que sempre produz `true`, seu programa ficaria preso em um *loop infinito*. Um programa preso em um loop infinito nunca terminará de executar, o que geralmente é uma coisa ruim.

A palavra-chave `continue` é similar a `break` no sentido de que influencia o progresso de um loop. Quando `continue` é encontrado em um corpo de loop, o controle salta para fora do corpo e continua com a próxima iteração do loop.

Atualizando bindings sucintamente

Especialmente ao fazer loops, um programa frequentemente precisa “atualizar” um binding para manter um valor baseado no valor anterior desse binding.

```
counter = counter + 1;
```

JavaScript fornece um atalho para isso:

```
counter += 1;
```

Atalhos similares funcionam para muitos outros operadores, como `result *= 2` para dobrar `result` ou `counter -= 1` para contar para baixo.

Isso nos permite encurtar ainda mais nosso exemplo de contagem:

```
for (let number = 0; number <= 12; number += 2) {
  console.log(number);
}
```

Para `counter += 1` e `counter -= 1`, existem equivalentes ainda mais curtos: `counter++` e `counter--`.

Despachando com um valor usando switch

Não é incomum que código tenha esta aparência:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

Existe uma construção chamada `switch` que se destina a expressar tal “despacho” de forma mais direta. Infelizmente, a sintaxe que JavaScript usa para isso (que herdou da linhagem de linguagens C/Java) é um tanto desajeitada — uma cadeia de instruções `if` pode parecer melhor. Aqui está um exemplo:

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
```

```
    console.log("Unknown weather type!");
    break;
}
```

Você pode colocar quantas etiquetas `case` quiser dentro do bloco aberto por `switch`. O programa começará a executar na etiqueta que corresponde ao valor que `switch` recebeu, ou em `default` se nenhum valor correspondente for encontrado. Ele continuará executando, mesmo através de outras etiquetas, até encontrar uma instrução `break`. Em alguns casos, como o caso `"sunny"` no exemplo, isso pode ser usado para compartilhar código entre casos (ele recomenda ir para fora tanto para tempo ensolarado quanto nublado). Mas tenha cuidado — é fácil esquecer tal `break`, o que fará o programa executar código que você não quer que seja executado.

Capitalização

Nomes de binding não podem conter espaços, mas muitas vezes é útil usar múltiplas palavras para descrever claramente o que o binding representa. Essas são basicamente suas opções para escrever um nome de binding com várias palavras:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

O primeiro estilo pode ser difícil de ler. Gosto bastante da aparência dos sublinhados, embora esse estilo seja um pouco doloroso de digitar. As funções padrão do JavaScript, e a maioria dos programadores JavaScript, seguem o último estilo — capitalizam cada palavra exceto a primeira. Não é difícil se acostumar com coisas pequenas assim, e código com estilos de nomeação mistos pode ser irritante de ler, então seguimos essa convenção.

Em alguns casos, como na função `Number`, a primeira letra de um binding também é capitalizada. Isso foi feito para marcar essa função como um construtor. Ficará claro o que é um construtor no Capítulo 6. Por enquanto, o importante é não se incomodar com essa aparente falta de consistência.

Comentários

Frequentemente, código bruto não transmite toda a informação que você quer que um programa transmita para leitores humanos, ou o transmite de forma tão

críptica que as pessoas podem não entendê-lo. Em outros momentos, você pode querer incluir alguns pensamentos relacionados como parte do seu programa. É para isso que servem os *comentários*.

Um comentário é um pedaço de texto que faz parte de um programa mas é completamente ignorado pelo computador. JavaScript tem duas formas de escrever comentários. Para escrever um comentário de uma linha, você pode usar dois caracteres de barra (`//`) e depois o texto do comentário após eles:

```
let accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
let report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

Um comentário `//` vai apenas até o final da linha. Uma seção de texto entre `/*` e `*/` será ignorada em sua totalidade, independentemente de conter quebras de linha. Isso é útil para adicionar blocos de informação sobre um arquivo ou trecho de programa:

```
/*
  I first found this number scrawled on the back of an old
  notebook. Since then, it has often dropped by, showing up in
  phone numbers and the serial numbers of products that I've
  bought. It obviously likes me, so I've decided to keep it.
*/
const myNumber = 11213;
```

Resumo

Agora você sabe que um programa é construído a partir de instruções, que por sua vez às vezes contêm mais instruções. Instruções tendem a conter expressões, que por sua vez podem ser construídas a partir de expressões menores.

Colocar instruções uma após a outra lhe dá um programa que é executado de cima para baixo. Você pode introduzir perturbações no fluxo de controle usando instruções condicionais (`if`, `else` e `switch`) e de loop (`while`, `do` e `for`).

Bindings podem ser usados para arquivar pedaços de dados sob um nome, e são úteis para acompanhar estado em seu programa. O ambiente é o conjunto de bindings que são definidos. Sistemas JavaScript sempre colocam uma série de bindings padrão úteis em seu ambiente.

Funções são valores especiais que encapsulam um pedaço de programa. Você pode invocá-las escrevendo `functionName(argument1, argument2)`. Tal chamada de função é uma expressão e pode produzir um valor.

Exercícios

Se você não tem certeza de como testar suas soluções para os exercícios, consulte a introdução.

Cada exercício começa com uma descrição do problema. Leia essa descrição e tente resolver o exercício. Se tiver problemas, considere ler as dicas no final do livro. Você pode encontrar soluções completas para os exercícios online em *https://eloquentjavascript.net/code*. Se quiser aprender algo com os exercícios, recomendo olhar as soluções apenas depois de ter resolvido o exercício, ou pelo menos depois de tê-lo atacado tempo e esforço suficientes para ter uma leve dor de cabeça.

Fazendo um triângulo com loop

Escreva um loop que faça sete chamadas a `console.log` para exibir o seguinte triângulo:

```
#
##
###
####
#####
######
#######
```

Pode ser útil saber que você pode encontrar o comprimento de uma string escrevendo `.length` após ela.

```
let abc = "abc";
console.log(abc.length);
// → 3
```

FizzBuzz

Escreva um programa que use `console.log` para imprimir todos os números de 1 a 100, com duas exceções. Para números divisíveis por 3, imprima `"Fizz"` em vez do número, e para números divisíveis por 5 (e não por 3), imprima `"Buzz"` em vez do número.

Quando isso estiver funcionando, modifique seu programa para imprimir "`FizzBuzz`" para números que são divisíveis por ambos 3 e 5 (e ainda imprimir "`Fizz`" ou "`Buzz`" para números divisíveis por apenas um deles).

(Isso é na verdade uma pergunta de entrevista que supostamente elimina uma porcentagem significativa de candidatos a programador. Então se você resolveu, seu valor no mercado de trabalho acabou de subir.)

Tabuleiro de xadrez

Escreva um programa que cria uma string que representa uma grade 8×8, usando caracteres de nova linha para separar linhas. Em cada posição da grade há um espaço ou um caractere "#". Os caracteres devem formar um tabuleiro de xadrez.

Passar essa string para `console.log` deve mostrar algo assim:

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
```

Quando tiver um programa que gera esse padrão, defina um binding `size = 8` e mude o programa para que funcione para qualquer `size`, produzindo uma grade da largura e altura fornecidas.

> *"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."*
>
> —Donald Knuth

Chapter 3

Funções

Funções são uma das ferramentas mais centrais na programação JavaScript. O conceito de envolver um pedaço de programa em um valor tem muitos usos. Nos dá uma forma de estruturar programas maiores, reduzir repetição, associar nomes a subprogramas e isolar esses subprogramas uns dos outros.

A aplicação mais óbvia de funções é definir novo vocabulário. Criar novas palavras em prosa é geralmente mau estilo, mas em programação é indispensável.

Falantes adultos típicos de inglês têm cerca de 20.000 palavras em seu vocabulário. Poucas linguagens de programação vêm com 20.000 comandos embutidos. E o vocabulário que *está* disponível tende a ser definido de forma mais precisa, e portanto menos flexível, que na linguagem humana. Portanto, *temos* que introduzir novas palavras para evitar verbosidade excessiva.

Definindo uma função

Uma definição de função é um binding regular onde o valor do binding é uma função. Por exemplo, este código define `square` para se referir a uma função que produz o quadrado de um dado número:

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

Uma função é criada com uma expressão que começa com a palavra-chave `function`. Funções têm um conjunto de *parâmetros* (neste caso, apenas `x`) e um *corpo*, que contém as instruções que devem ser executadas quando a função é chamada. O corpo de uma função criada dessa forma deve sempre estar envolvido em chaves, mesmo quando consiste em apenas uma única instrução.

Uma função pode ter múltiplos parâmetros ou nenhum parâmetro. No exemplo a seguir, `makeNoise` não lista nenhum nome de parâmetro, enquanto `roundTo` (que arredonda `n` para o múltiplo mais próximo de `step`) lista dois:

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const roundTo = function(n, step) {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};

console.log(roundTo(23, 10));
// → 20
```

Algumas funções, como `roundTo` e `square`, produzem um valor, e outras não, como `makeNoise`, cujo único resultado é um efeito colateral. Uma instrução `return` determina o valor que a função retorna. Quando o controle encontra tal instrução, ele imediatamente salta para fora da função atual e dá o valor retornado ao código que chamou a função. Uma palavra-chave `return` sem uma expressão após ela fará a função retornar `undefined`. Funções que não têm uma instrução `return`, como `makeNoise`, similarmente retornam `undefined`.

Parâmetros de uma função se comportam como bindings regulares, mas seus valores iniciais são dados pelo *chamador* da função, não pelo código na própria função.

Bindings e escopos

Cada binding tem um *escopo*, que é a parte do programa na qual o binding é visível. Para bindings definidos fora de qualquer função, bloco ou módulo (veja Capítulo 10), o escopo é o programa inteiro — você pode se referir a tais bindings onde quiser. Estes são chamados de *globais*.

Bindings criados para parâmetros de funções ou declarados dentro de uma função podem ser referenciados apenas naquela função, então são conhecidos como bindings *locais*. Toda vez que a função é chamada, novas instâncias desses bindings são criadas. Isso fornece algum isolamento entre funções — cada chamada de função age em seu próprio pequeno mundo (seu ambiente local) e frequentemente pode ser entendida sem saber muito sobre o que está

acontecendo no ambiente global.

Bindings declarados com `let` e `const` são de fato locais ao *bloco* em que são declarados, então se você criar um desses dentro de um loop, o código antes e depois do loop não pode "vê-lo". No JavaScript pré-2015, apenas funções criavam novos escopos, então bindings no estilo antigo, criados com a palavra-chave `var`, são visíveis em toda a função em que aparecem — ou em todo o escopo global, se não estiverem em uma função.

```
let x = 10;   // global
if (true) {
  let y = 20; // local ao bloco
  var z = 30; // também global
}
```

Cada escopo pode "olhar para fora" no escopo ao seu redor, então `x` é visível dentro do bloco no exemplo. A exceção é quando múltiplos bindings têm o mesmo nome — nesse caso, o código pode ver apenas o mais interno. Por exemplo, quando o código dentro da função `halve` se refere a `n`, ele está vendo seu *próprio* `n`, não o `n` global.

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

Escopo aninhado

JavaScript distingue não apenas bindings globais e locais. Blocos e funções podem ser criados dentro de outros blocos e funções, produzindo múltiplos graus de localidade.

Por exemplo, esta função — que exibe os ingredientes necessários para fazer uma porção de homus — tem outra função dentro dela:

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
```

```
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

O código dentro da função `ingredient` pode ver o binding `factor` da função externa, mas seus bindings locais, como `unit` ou `ingredientAmount`, não são visíveis na função externa.

O conjunto de bindings visíveis dentro de um bloco é determinado pelo lugar daquele bloco no texto do programa. Cada escopo local também pode ver todos os escopos locais que o contêm, e todos os escopos podem ver o escopo global. Essa abordagem à visibilidade de bindings é chamada de *escopo léxico*.

Funções como valores

Um binding de função geralmente atua simplesmente como um nome para um pedaço específico do programa. Tal binding é definido uma vez e nunca alterado. Isso torna fácil confundir a função e seu nome.

Mas os dois são diferentes. Um valor de função pode fazer todas as coisas que outros valores podem — você pode usá-lo em expressões arbitrárias, não apenas chamá-lo. É possível armazenar um valor de função em um novo binding, passá-lo como argumento para uma função, e assim por diante. Similarmente, um binding que contém uma função ainda é apenas um binding regular e pode, se não for constante, receber um novo valor, assim:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* não faz nada */};
}
```

No Capítulo 5, discutiremos as coisas interessantes que podemos fazer passando valores de função para outras funções.

Notação de declaração

Existe uma forma ligeiramente mais curta de criar um binding de função. Quando a palavra-chave `function` é usada no início de uma instrução, funciona de forma diferente:

```
function square(x) {
  return x * x;
}
```

Isso é uma *declaração* de função. A instrução define o binding `square` e o aponta para a função dada. É ligeiramente mais fácil de escrever e não requer um ponto e vírgula após a função.

Há uma sutileza com essa forma de definição de função.

```
console.log("The future says:", future());

function future() {
  return "You'll never have flying cars";
}
```

O código anterior funciona, embora a função seja definida *abaixo* do código que a usa. Declarações de função não fazem parte do fluxo de controle regular de cima para baixo. Elas são conceitualmente movidas para o topo de seu escopo e podem ser usadas por todo o código nesse escopo. Isso às vezes é útil porque oferece a liberdade de ordenar o código de uma forma que pareça mais clara, sem se preocupar em ter que definir todas as funções antes que sejam usadas.

Arrow functions

Existe uma terceira notação para funções, que parece muito diferente das outras. Em vez da palavra-chave `function`, ela usa uma seta (`=>`) composta por um sinal de igual e um caractere de maior-que (não confundir com o operador maior-ou-igual, que é escrito `>=`):

```
const roundTo = (n, step) => {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};
```

A seta vem *após* a lista de parâmetros e é seguida pelo corpo da função. Expressa algo como "esta entrada (os parâmetros) produz este resultado (o corpo)".

Quando há apenas um nome de parâmetro, você pode omitir os parênteses ao redor da lista de parâmetros. Se o corpo for uma única expressão em vez de um bloco entre chaves, essa expressão será retornada pela função. Então, estas duas definições de `square` fazem a mesma coisa:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

Quando uma arrow function não tem parâmetros, sua lista de parâmetros é apenas um conjunto vazio de parênteses.

```
const horn = () => {
  console.log("Toot");
};
```

Não há razão profunda para ter tanto arrow functions quanto expressões `function` na linguagem. Além de um detalhe menor, que discutiremos no Capítulo 6, elas fazem a mesma coisa. Arrow functions foram adicionadas em 2015, principalmente para tornar possível escrever pequenas expressões de função de forma menos verbosa. Nós as usaremos frequentemente no Capítulo 5.

A pilha de chamadas

A forma como o controle flui através de funções é um pouco envolvente. Vamos dar uma olhada mais de perto. Aqui está um programa simples que faz algumas chamadas de função:

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

Uma execução desse programa vai aproximadamente assim: a chamada a `greet` faz o controle saltar para o início dessa função (linha 2). A função chama `console.log`, que toma o controle, faz seu trabalho, e então retorna o controle para a linha 2. Lá, ele alcança o final da função `greet`, então retorna ao lugar que a chamou — linha 4. A linha seguinte chama `console.log` novamente. Após isso retornar, o programa alcança seu fim.

Poderíamos mostrar o fluxo de controle esquematicamente assim:

```
fora de função
  em greet
    em console.log
```

```
  em greet
fora de função
  em console.log
fora de função
```

Como uma função tem que saltar de volta ao lugar que a chamou quando retorna, o computador deve lembrar o contexto de onde a chamada aconteceu. Em um caso, `console.log` tem que retornar à função `greet` quando termina. No outro caso, retorna ao final do programa.

O lugar onde o computador armazena esse contexto é a *pilha de chamadas*. Toda vez que uma função é chamada, o contexto atual é armazenado no topo dessa pilha. Quando uma função retorna, ela remove o contexto do topo da pilha e usa esse contexto para continuar a execução.

Armazenar essa pilha requer espaço na memória do computador. Quando a pilha cresce demais, o computador falhará com uma mensagem como "out of stack space" ou "too much recursion". O código a seguir ilustra isso fazendo ao computador uma pergunta realmente difícil que causa um vaivém infinito entre duas funções. Ou melhor, *seria* infinito se o computador tivesse uma pilha infinita. Como está, ficaremos sem espaço, ou "estouraremos a pilha".

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

Argumentos Opcionais

O código a seguir é permitido e executa sem nenhum problema:

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog"));
// → 16
```

Definimos `square` com apenas um parâmetro. No entanto, quando a chamamos com três, a linguagem não reclama. Ela ignora os argumentos extras e calcula o quadrado do primeiro.

JavaScript é extremamente tolerante quanto ao número de argumentos que você pode passar para uma função. Se você passar muitos, os extras são igno-

rados. Se passar poucos, os parâmetros faltantes recebem o valor `undefined`.

A desvantagem disso é que é possível — provável, até — que você acidentalmente passe o número errado de argumentos para funções. E ninguém lhe dirá sobre isso. A vantagem é que você pode usar esse comportamento para permitir que uma função seja chamada com diferentes números de argumentos. Por exemplo, esta função `minus` tenta imitar o operador `-` agindo sobre um ou dois argumentos:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

Se você escrever um operador `=` após um parâmetro, seguido de uma expressão, o valor dessa expressão substituirá o argumento quando ele não for fornecido. Por exemplo, esta versão de `roundTo` torna seu segundo argumento opcional. Se você não o fornecer ou passar o valor `undefined`, ele terá como padrão o valor um:

```
function roundTo(n, step = 1) {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};

console.log(roundTo(4.5));
// → 5
console.log(roundTo(4.5, 2));
// → 4
```

O próximo capítulo apresentará uma forma pela qual o corpo de uma função pode obter a lista completa de argumentos que lhe foi passada. Isso é útil porque permite que uma função aceite qualquer número de argumentos. Por exemplo, `console.log` faz isso, exibindo todos os valores que lhe são dados:

```
console.log("C", "O", 2);
// → C O 2
```

Closure

A capacidade de tratar funções como valores, combinada com o fato de que bindings locais são recriados toda vez que uma função é chamada, traz uma pergunta interessante: o que acontece com bindings locais quando a chamada de função que os criou não está mais ativa?

O código a seguir mostra um exemplo disso. Ele define uma função, `wrapValue`, que cria um binding local. Então retorna uma função que acessa e retorna esse binding local.

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

Isso é permitido e funciona como você esperaria — ambas as instâncias do binding ainda podem ser acessadas. Essa situação é uma boa demonstração do fato de que bindings locais são criados novamente para cada chamada, e chamadas diferentes não afetam os bindings locais uma da outra.

Esse recurso — poder referenciar uma instância específica de um binding local em um escopo envolvente — é chamado de *closure*. Uma função que referencia bindings de escopos locais ao seu redor é chamada de *uma* closure. Esse comportamento não apenas o libera de ter que se preocupar com o tempo de vida dos bindings, mas também torna possível usar valores de função de formas criativas.

Com uma pequena mudança, podemos transformar o exemplo anterior em uma forma de criar funções que multiplicam por um valor arbitrário.

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

O binding explícito `local` do exemplo `wrapValue` não é realmente necessário, já

que um parâmetro é em si um binding local.

Pensar sobre programas assim requer alguma prática. Um bom modelo mental é pensar em valores de função como contendo tanto o código em seu corpo quanto o ambiente em que são criados. Quando chamada, o corpo da função vê o ambiente em que foi criada, não o ambiente em que é chamada.

No exemplo anterior, `multiplier` é chamada e cria um ambiente em que seu parâmetro `factor` está vinculado a 2. O valor de função que ela retorna, que é armazenado em `twice`, lembra esse ambiente para que, quando chamado, multiplique seu argumento por 2.

Recursão

É perfeitamente aceitável que uma função chame a si mesma, desde que não faça isso com tanta frequência que estoure a pilha. Uma função que chama a si mesma é chamada de *recursiva*. Recursão permite que algumas funções sejam escritas em um estilo diferente. Tome, por exemplo, esta função `power`, que faz o mesmo que o operador `**` (exponenciação):

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

Isso é bastante próximo da forma como matemáticos definem exponenciação e descreve o conceito de forma indiscutivelmente mais clara que o loop que usamos no Capítulo 2. A função chama a si mesma múltiplas vezes com expoentes cada vez menores para alcançar a multiplicação repetida.

No entanto, essa implementação tem um problema: em implementações típicas de JavaScript, ela é cerca de três vezes mais lenta que uma versão usando um loop `for`. Percorrer um loop simples é geralmente mais barato que chamar uma função múltiplas vezes.

O dilema de velocidade versus elegância é interessante. Você pode vê-lo como uma espécie de contínuo entre amigabilidade para humanos e amigabilidade para máquinas. Quase qualquer programa pode ser tornado mais rápido tornando-o maior e mais complicado. O programador tem que encontrar um

equilíbrio apropriado.

No caso da função `power`, uma versão inelegante (com loop) ainda é bastante simples e fácil de ler. Não faz muito sentido substituí-la por uma função recursiva. Frequentemente, porém, um programa lida com conceitos tão complexos que abrir mão de alguma eficiência para tornar o programa mais direto é útil.

Preocupar-se com eficiência pode ser uma distração. É mais um fator que complica o design do programa, e quando você está fazendo algo que já é difícil, essa coisa extra para se preocupar pode ser paralisante.

Portanto, você geralmente deve começar escrevendo algo que é correto e fácil de entender. Se estiver preocupado que é lento demais — o que geralmente não é, já que a maioria do código simplesmente não é executado com frequência suficiente para levar uma quantidade significativa de tempo — você pode medir depois e melhorar se necessário.

Recursão nem sempre é apenas uma alternativa ineficiente ao loop. Alguns problemas são realmente mais fáceis de resolver com recursão do que com loops. Na maioria das vezes, são problemas que requerem explorar ou processar várias "ramificações", cada uma das quais pode se ramificar novamente em ainda mais ramificações.

Considere este quebra-cabeça: começando do número 1 e repetidamente somando 5 ou multiplicando por 3, um conjunto infinito de números pode ser produzido. Como você escreveria uma função que, dado um número, tenta encontrar uma sequência de tais adições e multiplicações que produz esse número? Por exemplo, o número 13 pode ser alcançado primeiro multiplicando por 3 e depois somando 5 duas vezes, enquanto o número 15 não pode ser alcançado de forma alguma.

Aqui está uma solução recursiva:

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ??
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
```

```
// → (((1 * 3) + 5) * 3)
```

Note que este programa não necessariamente encontra a sequência *mais curta* de operações. Ele se satisfaz quando encontra qualquer sequência.

Tudo bem se você não ver como este código funciona imediatamente. Vamos percorrê-lo, já que é um ótimo exercício de pensamento recursivo.

A função interna `find` faz a recursão real. Ela recebe dois argumentos: o número atual e uma string que registra como chegamos a esse número. Se encontra uma solução, retorna uma string que mostra como chegar ao alvo. Se não pode encontrar nenhuma solução a partir deste número, retorna `null`.

Para fazer isso, a função realiza uma de três ações. Se o número atual é o número alvo, o histórico atual é uma forma de alcançar esse alvo, então ele é retornado. Se o número atual é maior que o alvo, não faz sentido explorar mais esse caminho porque tanto somar quanto multiplicar só tornarão o número maior, então retorna `null`. Finalmente, se ainda estamos abaixo do número alvo, a função tenta ambos os caminhos possíveis que começam do número atual, chamando a si mesma duas vezes, uma para adição e uma para multiplicação. Se a primeira chamada retorna algo que não é `null`, ele é retornado. Caso contrário, a segunda chamada é retornada, independentemente de produzir uma string ou `null`.

Para entender melhor como essa função produz o efeito que estamos procurando, vejamos todas as chamadas a `find` que são feitas ao procurar uma solução para o número 13:

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        grande demais
      find(33, "(((1 + 5) + 5) * 3)")
        grande demais
    find(18, "((1 + 5) * 3)")
      grande demais
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        encontrado!
```

A indentação indica a profundidade da pilha de chamadas. A primeira vez que `find` é chamada, a função começa chamando a si mesma para explorar a solução que começa com `(1 + 5)`. Essa chamada recursará ainda mais para explorar *toda* solução continuada que produza um número menor ou igual ao número alvo. Como não encontra uma que atinja o alvo, retorna `null` de volta

à primeira chamada. Lá o operador `??` faz com que a chamada que explora `(1 * 3)` aconteça. Essa busca tem mais sorte — sua primeira chamada recursiva, através de *outra* chamada recursiva, encontra o número alvo. Essa chamada mais interna retorna uma string, e cada um dos operadores `??` nas chamadas intermediárias passa essa string adiante, finalmente retornando a solução.

Crescendo funções

Existem duas formas mais ou menos naturais de funções serem introduzidas em programas.

A primeira ocorre quando você se encontra escrevendo código similar múltiplas vezes. Você preferiria não fazer isso, já que ter mais código significa mais espaço para erros se esconderem e mais material para ler para pessoas tentando entender o programa. Então você pega a funcionalidade repetida, encontra um bom nome para ela e a coloca em uma função.

A segunda forma é que você descobre que precisa de alguma funcionalidade que ainda não escreveu e que parece merecer sua própria função. Você começa nomeando a função e depois escreve seu corpo. Pode até começar a escrever código que usa a função antes de realmente definir a função em si.

Quão difícil é encontrar um bom nome para uma função é uma boa indicação de quão claro é o conceito que você está tentando envolver. Vamos percorrer um exemplo.

Queremos escrever um programa que imprime dois números: os números de vacas e galinhas em uma fazenda, com as palavras `Cows` e `Chickens` depois deles e zeros preenchidos antes de ambos os números para que sejam sempre três dígitos:

```
007 Cows
011 Chickens
```

Isso pede uma função de dois argumentos — o número de vacas e o número de galinhas. Vamos codificar.

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
    cowString = "0" + cowString;
  }
  console.log(`${cowString} Cows`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
```

```
  }
  console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

Escrever `.length` após uma expressão de string nos dará o comprimento dessa string. Assim, os loops `while` continuam adicionando zeros na frente das strings de números até que tenham pelo menos três caracteres de comprimento.

Missão cumprida! Mas assim que estamos prestes a enviar o código ao fazendeiro (junto com uma fatura salgada), ela liga e nos diz que também começou a criar porcos, e se não poderíamos por favor estender o software para também imprimir porcos?

Claro que podemos. Mas bem quando estamos no processo de copiar e colar essas quatro linhas mais uma vez, paramos e reconsideramos. Tem que haver uma forma melhor. Aqui está uma primeira tentativa:

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "Cows");
  printZeroPaddedWithLabel(chickens, "Chickens");
  printZeroPaddedWithLabel(pigs, "Pigs");
}

printFarmInventory(7, 11, 3);
```

Funciona! Mas esse nome, `printZeroPaddedWithLabel`, é um pouco desajeitado. Ele conflita três coisas — imprimir, preencher com zeros e adicionar um rótulo — em uma única função.

Em vez de extrair a parte repetida do nosso programa por inteiro, vamos tentar selecionar um único *conceito*:

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}
```

```
function printFarmInventory(cows, chickens, pigs) {
  console.log(`${zeroPad(cows, 3)} Cows`);
  console.log(`${zeroPad(chickens, 3)} Chickens`);
  console.log(`${zeroPad(pigs, 3)} Pigs`);
}

printFarmInventory(7, 16, 3);
```

Uma função com um nome bonito e óbvio como `zeroPad` torna mais fácil para alguém que lê o código descobrir o que ela faz. Tal função também é útil em mais situações do que apenas esse programa específico. Por exemplo, você poderia usá-la para ajudar a imprimir tabelas de números bem alinhadas.

Quão inteligente e versátil *deve* ser nossa função? Poderíamos escrever qualquer coisa, desde uma função terrivelmente simples que só pode preencher um número com três caracteres de largura até um sistema generalizado e complicado de formatação de números que lida com números fracionários, números negativos, alinhamento de pontos decimais, preenchimento com diferentes caracteres, e assim por diante.

Um princípio útil é se abster de adicionar esperteza a menos que você tenha certeza absoluta de que vai precisar. Pode ser tentador escrever "frameworks" gerais para cada pedaço de funcionalidade que encontrar. Resista a esse impulso. Você não fará nenhum trabalho real — estará muito ocupado escrevendo código que nunca usa.

Funções e efeitos colaterais

Funções podem ser grosseiramente divididas naquelas que são chamadas por seus efeitos colaterais e naquelas que são chamadas por seu valor de retorno (embora também seja possível ter efeitos colaterais e retornar um valor).

A primeira função auxiliar no exemplo fazenda, `printZeroPaddedWithLabel`, é chamada por seu efeito colateral: ela imprime uma linha. A segunda versão, `zeroPad`, é chamada por seu valor de retorno. Não é coincidência que a segunda é útil em mais situações que a primeira. Funções que criam valores são mais fáceis de combinar de novas formas do que funções que diretamente realizam efeitos colaterais.

Uma função *pura* é um tipo específico de função produtora de valor que não apenas não tem efeitos colaterais mas também não depende de efeitos colaterais de outro código — por exemplo, não lê bindings globais cujo valor pode mudar. Uma função pura tem a propriedade agradável de que, quando chamada com os mesmos argumentos, sempre produz o mesmo valor (e não faz mais nada).

Uma chamada a tal função pode ser substituída por seu valor de retorno sem mudar o significado do código. Quando você não tem certeza de que uma função pura está funcionando corretamente, pode testá-la simplesmente chamando-a e saber que se funcionar naquele contexto, funcionará em qualquer contexto. Funções não puras tendem a exigir mais estrutura para serem testadas.

Ainda assim, não é preciso se sentir mal ao escrever funções que não são puras. Efeitos colaterais são frequentemente úteis. Não há como escrever uma versão pura de `console.log`, por exemplo, e `console.log` é bom de ter. Algumas operações também são mais fáceis de expressar de forma eficiente quando usamos efeitos colaterais.

Resumo

Este capítulo ensinou como escrever suas próprias funções. A palavra-chave `function`, quando usada como expressão, pode criar um valor de função. Quando usada como instrução, pode ser usada para declarar um binding e dar a ele uma função como valor. Arrow functions são mais uma forma de criar funções.

```
// Definir f para conter um valor de função
const f = function(a) {
  console.log(a + 2);
};

// Declarar g como uma função
function g(a, b) {
  return a * b * 3.5;
}

// Um valor de função menos verboso
let h = a => a % 3;
```

Uma parte fundamental de entender funções é entender escopos. Cada bloco cria um novo escopo. Parâmetros e bindings declarados em um dado escopo são locais e não visíveis de fora. Bindings declarados com `var` se comportam de forma diferente — acabam no escopo da função mais próxima ou no escopo global.

Separar as tarefas que seu programa executa em diferentes funções é útil. Você não terá que se repetir tanto, e funções podem ajudar a organizar um programa agrupando código em partes que fazem coisas específicas.

Exercícios

Mínimo

O capítulo anterior introduziu a função padrão `Math.min` que retorna seu menor argumento. Podemos escrever uma função como essa nós mesmos agora. Defina a função `min` que recebe dois argumentos e retorna o menor deles.

Recursão

Vimos que podemos usar `%` (o operador resto) para testar se um número é par ou ímpar usando `% 2` para ver se é divisível por dois. Aqui está outra forma de definir se um número inteiro positivo é par ou ímpar:

- Zero é par.
- Um é ímpar.
- Para qualquer outro número *N*, sua paridade é a mesma de *N* - 2.

Defina uma função recursiva `isEven` correspondendo a essa descrição. A função deve aceitar um único parâmetro (um número inteiro positivo) e retornar um booleano.

Teste-a com 50 e 75. Veja como ela se comporta com -1. Por quê? Você consegue pensar em uma forma de corrigir isso?

Contando feijões

Você pode obter o *N*-ésimo caractere, ou letra, de uma string escrevendo `[N]` após a string (por exemplo, `string[2]`). O valor resultante será uma string contendo apenas um caractere (por exemplo, `"b"`). O primeiro caractere está na posição 0, o que faz com que o último seja encontrado na posição `string.length - 1`. Em outras palavras, uma string de dois caracteres tem comprimento 2, e seus caracteres estão nas posições 0 e 1.

Escreva uma função chamada `countBs` que recebe uma string como seu único argumento e retorna um número que indica quantos caracteres B maiúsculos existem na string.

Em seguida, escreva uma função chamada `countChar` que se comporta como `countBs`, exceto que recebe um segundo argumento que indica o caractere que deve ser contado (em vez de contar apenas caracteres B maiúsculos). Reescreva `countBs` para fazer uso dessa nova função.

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

—Charles Babbage, Passages from the Life of a Philosopher (1864)

Chapter 4

Estruturas de Dados: Objetos e Arrays

Números, booleanos e strings são os átomos a partir dos quais estruturas de dados são construídas. Muitos tipos de informação requerem mais de um átomo, porém. *Objetos* nos permitem agrupar valores — incluindo outros objetos — para construir estruturas mais complexas.

Os programas que construímos até agora foram limitados pelo fato de que operavam apenas sobre tipos de dados simples. Após aprender o básico de estruturas de dados neste capítulo, você saberá o suficiente para começar a escrever programas úteis.

O capítulo trabalhará com um exemplo de programação mais ou menos realista, introduzindo conceitos conforme se aplicam ao problema em questão. O código de exemplo frequentemente se apoiará em funções e bindings introduzidos anteriormente no livro.

O sandbox de codificação online para o livro (*https://eloquentjavascript.net/code*) fornece uma forma de executar código no contexto de um capítulo específico. Se decidir trabalhar nos exemplos em outro ambiente, certifique-se de primeiro baixar o código completo deste capítulo da página do sandbox.

O lobisomem-esquilo

De vez em quando, geralmente entre 20h e 22h, Jacques se vê transformando em um pequeno roedor peludo com uma cauda espessa.

Por um lado, Jacques está bastante contente por não ter licantropia clássica. Transformar-se em um esquilo causa menos problemas do que transformar-se em um lobo. Em vez de ter que se preocupar em acidentalmente comer o vizinho (*isso* seria constrangedor), ele se preocupa em ser comido pelo gato do vizinho. Depois de duas ocasiões de acordar em um galho precariamente fino na copa de um carvalho, nu e desorientado, ele passou a trancar as portas e janelas do seu quarto à noite e colocar algumas nozes no chão para se manter ocupado.

Mas Jacques preferiria se livrar totalmente de sua condição. As ocorrências

irregulares da transformação o fazem suspeitar que podem ser desencadeadas por algo. Por um tempo, ele acreditou que acontecia apenas em dias em que estivera perto de carvalhos. Porém, evitar carvalhos não resolveu o problema.

Mudando para uma abordagem mais científica, Jacques começou a manter um registro diário de tudo que faz em um dado dia e se mudou de forma. Com esses dados, ele espera restringir as condições que desencadeiam as transformações.

A primeira coisa de que ele precisa é uma estrutura de dados para armazenar essa informação.

Conjuntos de dados

Para trabalhar com um pedaço de dados digitais, primeiro precisamos encontrar uma forma de representá-lo na memória da nossa máquina. Digamos, por exemplo, que queremos representar uma coleção dos números 2, 3, 5, 7 e 11.

Poderíamos ser criativos com strings — afinal, strings podem ter qualquer comprimento, então podemos colocar muitos dados nelas — e usar `"2 3 5 7 11"` como nossa representação. Mas isso é desajeitado. Teríamos que de alguma forma extrair os dígitos e convertê-los de volta para números para acessá-los.

Felizmente, JavaScript fornece um tipo de dado especificamente para armazenar sequências de valores. É chamado de *array* e é escrito como uma lista de valores entre colchetes, separados por vírgulas.

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

A notação para acessar os elementos dentro de um array também usa colchetes. Um par de colchetes imediatamente após uma expressão, com outra expressão dentro deles, procurará o elemento na expressão à esquerda que corresponde ao *índice* dado pela expressão entre colchetes.

O primeiro índice de um array é zero, não um, então o primeiro elemento é obtido com `listOfNumbers[0]`. A contagem baseada em zero tem uma longa tradição em tecnologia e de certas formas faz muito sentido, mas leva algum tempo para se acostumar. Pense no índice como o número de itens a pular, contando a partir do início do array.

Propriedades

Vimos algumas expressões como `myString.length` (para obter o comprimento de uma string) e `Math.max` (a função de máximo) em capítulos anteriores. Essas expressões acessam uma *propriedade* de algum valor. No primeiro caso, acessamos a propriedade `length` do valor em `myString`. No segundo, acessamos a propriedade chamada `max` no objeto `Math` (que é uma coleção de constantes e funções relacionadas à matemática).

Quase todos os valores JavaScript têm propriedades. As exceções são `null` e `undefined`. Se você tentar acessar uma propriedade em um desses não-valores, obterá um erro:

```
null.length;
// → TypeError: null has no properties
```

As duas principais formas de acessar propriedades em JavaScript são com um ponto e com colchetes. Tanto `value.x` quanto `value[x]` acessam uma propriedade em `value` — mas não necessariamente a mesma propriedade. A diferença está em como `x` é interpretado. Ao usar um ponto, a palavra após o ponto é o nome literal da propriedade. Ao usar colchetes, a expressão entre os colchetes é *avaliada* para obter o nome da propriedade. Enquanto `value.x` busca a propriedade de `value` chamada "x", `value[x]` pega o valor da variável chamada `x` e o usa, convertido para string, como nome da propriedade.

Se você sabe que a propriedade em que está interessado se chama *color*, você diz `value.color`. Se quer extrair a propriedade nomeada pelo valor mantido no binding `i`, você diz `value[i]`. Nomes de propriedades são strings. Podem ser qualquer string, mas a notação de ponto funciona apenas com nomes que parecem nomes válidos de binding — começando com uma letra ou sublinhado, e contendo apenas letras, números e sublinhados. Se quiser acessar uma propriedade chamada *2* ou *John Doe*, deve usar colchetes: `value[2]` ou `value["John Doe"]`.

Os elementos em um array são armazenados como propriedades do array, usando números como nomes de propriedades. Como você não pode usar a notação de ponto com números e geralmente quer usar um binding que contém o índice de qualquer forma, precisa usar a notação de colchetes para acessá-los.

Assim como strings, arrays têm uma propriedade `length` que nos diz quantos elementos o array tem.

Métodos

Tanto valores de string quanto de array contêm, além da propriedade `length`, uma série de propriedades que contêm valores de função.

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

Toda string tem uma propriedade `toUpperCase`. Quando chamada, retornará uma cópia da string em que todas as letras foram convertidas para maiúsculas. Existe também `toLowerCase`, que faz o inverso.

Curiosamente, embora a chamada a `toUpperCase` não passe nenhum argumento, a função de alguma forma tem acesso à string `"Doh"`, o valor cuja propriedade chamamos. Você descobrirá como isso funciona no Capítulo 6.

Propriedades que contêm funções são geralmente chamadas de *métodos* do valor a que pertencem, como em "`toUpperCase` é um método de uma string".

Este exemplo demonstra dois métodos que você pode usar para manipular arrays.

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

O método `push` adiciona valores ao final de um array. O método `pop` faz o oposto, removendo o último valor do array e retornando-o.

Esses nomes um tanto bobos são os termos tradicionais para operações em uma *pilha*. Uma pilha, em programação, é uma estrutura de dados que permite empurrar valores para dentro dela e retirá-los na ordem inversa, de modo que o que foi adicionado por último é removido primeiro. Pilhas são comuns em programação — você pode lembrar da pilha de chamadas de funções do capítulo anterior, que é um exemplo da mesma ideia.

Objetos

De volta ao lobisomem-esquilo. Um conjunto de entradas de registro diário pode ser representado como um array, mas as entradas não consistem apenas de um número ou uma string — cada entrada precisa armazenar uma lista de atividades e um valor booleano que indica se Jacques se transformou em esquilo ou não. Idealmente, gostaríamos de agrupar estes juntos em um único valor e então colocar esses valores agrupados em um array de entradas de registro.

Valores do tipo *objeto* são coleções arbitrárias de propriedades. Uma forma de criar um objeto é usando chaves como uma expressão.

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

Dentro das chaves, você escreve uma lista de propriedades separadas por vírgulas. Cada propriedade tem um nome seguido de dois-pontos e um valor. Quando um objeto é escrito em múltiplas linhas, indentá-lo como mostrado neste exemplo ajuda na legibilidade. Propriedades cujos nomes não são nomes válidos de binding ou números válidos devem ser colocados entre aspas:

```
let descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

Isso significa que chaves têm *dois* significados em JavaScript. No início de uma instrução, elas iniciam um bloco de instruções. Em qualquer outra posição, elas descrevem um objeto. Felizmente, raramente é útil iniciar uma instrução com um objeto em chaves, então a ambiguidade entre esses dois não é um grande problema. O único caso em que isso surge é quando você quer retornar um objeto de uma arrow function abreviada — você não pode escrever `n => {prop : n}` pois as chaves serão interpretadas como um corpo de função. Em vez disso, você precisa colocar parênteses ao redor do objeto para deixar claro que é uma expressão.

Ler uma propriedade que não existe lhe dará o valor `undefined`.

É possível atribuir um valor a uma expressão de propriedade com o operador `=`. Isso substituirá o valor da propriedade se ela já existia ou criará uma nova propriedade no objeto se não existia.

Para retornar brevemente ao nosso modelo de tentáculos de bindings — bindings de propriedades são similares. Eles *agarram* valores, mas outros bindings e propriedades podem estar segurando esses mesmos valores. Você pode pensar em objetos como polvos com qualquer número de tentáculos, cada um com um nome escrito nele.

O operador `delete` corta um tentáculo de tal polvo. É um operador unário que, quando aplicado a uma propriedade de objeto, removerá a propriedade nomeada do objeto. Isso não é algo comum de se fazer, mas é possível.

```
let anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

O operador binário `in`, quando aplicado a uma string e um objeto, lhe diz se aquele objeto tem uma propriedade com aquele nome. A diferença entre definir uma propriedade como `undefined` e realmente deletá-la é que, no primeiro caso, o objeto ainda *tem* a propriedade (ela simplesmente não tem um valor muito interessante), enquanto no segundo caso, a propriedade não está mais presente e `in` retornará `false`.

Para descobrir quais propriedades um objeto tem, você pode usar a função `Object.keys`. Dê a ela um objeto e ela retornará um array de strings — os nomes das propriedades do objeto:

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

Há uma função `Object.assign` que copia todas as propriedades de um objeto para outro:

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

Arrays, então, são apenas um tipo de objeto especializado para armazenar sequências de coisas. Se você avaliar `typeof []`, ele produz `"object"`. Você pode visualizar arrays como polvos longos e achatados com todos os seus tentáculos em uma fileira organizada, rotulados com números.

Jacques representará o diário que ele mantém como um array de objetos:

```
let journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break", "peanuts",
            "beer"],
   squirrel: true},
  /* E assim por diante... */
];
```

Mutabilidade

Em breve chegaremos à programação de verdade, mas primeiro, há mais um pedaço de teoria para entender.

Vimos que valores de objetos podem ser modificados. Os tipos de valores discutidos em capítulos anteriores, como números, strings e booleanos, são todos *imutáveis* — é impossível mudar valores desses tipos. Você pode combiná-los e derivar novos valores deles, mas quando você pega um valor de string específico, esse valor sempre permanecerá o mesmo. O texto dentro dele não pode ser alterado. Se você tem uma string que contém `"cat"`, não é possível que outro código mude um caractere na sua string para fazê-la dizer `"rat"`.

Objetos funcionam de forma diferente. Você *pode* mudar suas propriedades, fazendo com que um único valor de objeto tenha conteúdo diferente em momentos diferentes.

Quando temos dois números, 120 e 120, podemos considerá-los precisamente o mesmo número, quer se refiram ou não aos mesmos bits físicos. Com objetos, há uma diferença entre ter duas referências ao mesmo objeto e ter dois objetos diferentes que contêm as mesmas propriedades. Considere o seguinte código:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};
```

```
console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

Os bindings `object1` e `object2` agarram o *mesmo* objeto, razão pela qual mudar `object1` também muda o valor de `object2`. Diz-se que eles têm a mesma *identidade*. O binding `object3` aponta para um objeto diferente, que inicialmente contém as mesmas propriedades que `object1` mas vive uma vida separada.

Bindings também podem ser mutáveis ou constantes, mas isso é separado da forma como seus valores se comportam. Embora valores numéricos não mudem, você pode usar um binding `let` para acompanhar um número que muda, mudando o valor para o qual o binding aponta. Similarmente, embora um binding `const` para um objeto não possa ser mudado e continuará a apontar para o mesmo objeto, o *conteúdo* desse objeto pode mudar.

```
const score = {visitors: 0, home: 0};
// Isso é ok
score.visitors = 1;
// Isso não é permitido
score = {visitors: 1, home: 1};
```

Quando você compara objetos com o operador `==` do JavaScript, ele compara por identidade: produzirá `true` apenas se ambos os objetos forem precisamente o mesmo valor. Comparar objetos diferentes retornará `false`, mesmo que tenham propriedades idênticas. Não há operação de comparação "profunda" embutida no JavaScript que compare objetos por conteúdo, mas é possível escrevê-la você mesmo (que é um dos exercícios no final deste capítulo).

O registro do licantropo

Jacques inicia seu interpretador JavaScript e configura o ambiente necessário para manter seu diário:

```
let journal = [];

function addEntry(events, squirrel) {
```

```
  journal.push({events, squirrel});
}
```

Note que o objeto adicionado ao diário parece um pouco estranho. Em vez de declarar propriedades como `events: events`, ele apenas dá um nome de propriedade: `events`. Isso é uma abreviação que significa a mesma coisa — se um nome de propriedade em notação de chaves não é seguido por um valor, seu valor é tomado do binding com o mesmo nome.

Toda noite às 22h — ou às vezes na manhã seguinte, após descer da prateleira mais alta de sua estante — Jacques registra o dia:

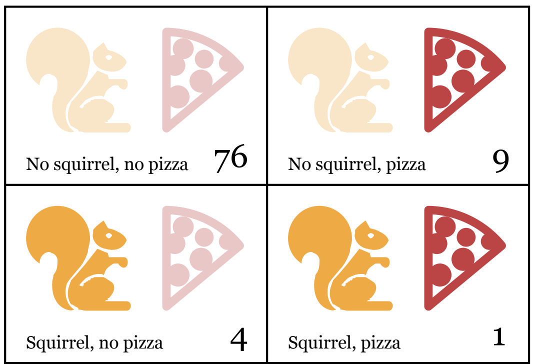
```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "beer"], true);
```

Uma vez que tenha pontos de dados suficientes, pretende usar estatística para descobrir quais desses eventos podem estar relacionados às esquilificações.

Correlação é uma medida de dependência entre variáveis estatísticas. Uma variável estatística não é exatamente a mesma coisa que uma variável de programação. Em estatística, você tipicamente tem um conjunto de *medições*, e cada variável é medida para cada medição. Correlação entre variáveis é geralmente expressa como um valor que varia de -1 a 1. Correlação zero significa que as variáveis não estão relacionadas. Uma correlação de 1 indica que as duas são perfeitamente relacionadas — se você sabe uma, também sabe a outra. Menos 1 também significa que as variáveis são perfeitamente relacionadas mas são opostas — quando uma é verdadeira, a outra é falsa.

Para calcular a medida de correlação entre duas variáveis booleanas, podemos usar o *coeficiente phi* (φ). Esta é uma fórmula cuja entrada é uma tabela de frequência contendo o número de vezes que as diferentes combinações das variáveis foram observadas. A saída da fórmula é um número entre -1 e 1 que descreve a correlação.

Poderíamos pegar o evento de comer pizza e colocá-lo em uma tabela de frequência como esta, onde cada número indica o número de vezes que essa combinação ocorreu em nossas medições.



Se chamarmos essa tabela de n, podemos calcular φ usando a seguinte fórmula:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}} \tag{4.1}$$

(Se neste ponto você está largando o livro para focar em um terrível flashback de aulas de matemática do ensino médio — espere! Não tenho a intenção de torturá-lo com páginas infinitas de notação críptica — é apenas essa fórmula por enquanto. E mesmo com essa, tudo que fazemos é transformá-la em JavaScript.)

A notação n_{01} indica o número de medições onde a primeira variável (esquilidade) é falsa (0) e a segunda variável (pizza) é verdadeira (1). Na tabela de pizza, n_{01} é 9.

O valor $n_{1\bullet}$ se refere à soma de todas as medições onde a primeira variável é verdadeira, que é 5 na tabela de exemplo. Da mesma forma, $n_{\bullet 0}$ se refere à soma das medições onde a segunda variável é falsa.

Então para a tabela de pizza, a parte acima da linha de divisão (o dividendo) seria $1 \times 76 - 4 \times 9 = 40$, e a parte abaixo (o divisor) seria a raiz quadrada de $5 \times 85 \times 10 \times 80$, ou $\sqrt{340,000}$. Isso resulta em $\varphi \approx 0{,}069$, que é minúsculo. Comer pizza não parece ter influência nas transformações.

Calculando correlação

Podemos representar uma tabela dois-por-dois em JavaScript com um array de quatro elementos (`[76, 9, 4, 1]`). Poderíamos também usar outras representações, como um array contendo dois arrays de dois elementos (`[[76, 9], [4, 1]]`) ou um objeto com nomes de propriedade como `"11"` e `"01"`, mas o array plano é simples e torna as expressões que acessam a tabela agradavelmente curtas. Interpretaremos os índices do array como números binários de dois bits, onde o dígito mais à esquerda (mais significativo) se refere à variável esquilo e

o dígito mais à direita (menos significativo) se refere à variável do evento. Por exemplo, o número binário `10` se refere ao caso em que Jacques se transformou em esquilo, mas o evento (digamos, "pizza") não ocorreu. Isso aconteceu quatro vezes. E como o binário `10` é 2 em notação decimal, armazenaremos esse número no índice 2 do array.

Esta é a função que calcula o coeficiente φ a partir de tal array:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

Esta é uma tradução direta da fórmula φ para JavaScript. `Math.sqrt` é a função de raiz quadrada, conforme fornecida pelo objeto `Math` em um ambiente JavaScript padrão. Temos que adicionar dois campos da tabela para obter campos como $n_{1\bullet}$ porque as somas de linhas ou colunas não são armazenadas diretamente em nossa estrutura de dados.

Jacques mantém seu diário por três meses. O conjunto de dados resultante está disponível no sandbox de codificação para este capítulo (*https://eloquentjavascript.net/code#4*), onde é armazenado no binding `JOURNAL`, e em um arquivo para download.

Para extrair uma tabela dois-por-dois para um evento específico do diário, devemos percorrer todas as entradas e contabilizar quantas vezes o evento ocorre em relação às transformações em esquilo:

```
function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Arrays têm um método `includes` que verifica se um dado valor existe no array. A função usa isso para determinar se o nome do evento em que está interessada faz parte da lista de eventos para um dado dia.

O corpo do loop em `tableFor` descobre em qual caixa da tabela cada entrada do diário se encaixa verificando se a entrada contém o evento específico em que está interessada e se o evento acontece junto com um incidente de esquilo. O loop então adiciona um à caixa correta na tabela.

Agora temos as ferramentas necessárias para calcular correlações individuais. O único passo restante é encontrar uma correlação para cada tipo de evento que foi registrado e ver se algo se destaca.

Loops de array

Na função `tableFor`, há um loop assim:

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // Fazer algo com entry
}
```

Esse tipo de loop é comum no JavaScript clássico — percorrer arrays um elemento de cada vez é algo que surge muito, e para fazer isso você executa um contador pelo comprimento do array e seleciona cada elemento por vez.

Há uma forma mais simples de escrever tais loops no JavaScript moderno:

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

Quando um loop `for` usa a palavra `of` após sua definição de variável, ele percorrerá os elementos do valor dado após `of`. Isso funciona não apenas para arrays mas também para strings e algumas outras estruturas de dados. Discutiremos *como* funciona no Capítulo 6.

A análise final

Precisamos calcular uma correlação para cada tipo de evento que ocorre no conjunto de dados. Para fazer isso, primeiro precisamos *encontrar* cada tipo de evento.

```
function journalEvents(journal) {
  let events = [];
```

```
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", …]
```

Adicionando quaisquer nomes de eventos que ainda não estão nele ao array `events`, a função coleta cada tipo de evento.

Usando essa função, podemos ver todas as correlações:

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot:   0.0140970969
// → exercise: 0.0685994341
// → weekend:  0.1371988681
// → bread:   -0.0757554019
// → pudding: -0.0648203724
// E assim por diante...
```

A maioria das correlações parece estar perto de zero. Comer cenouras, pão ou pudim aparentemente não desencadeia a licantropia-esquilo. As transformações *parecem* ocorrer um pouco mais nos finais de semana. Vamos filtrar os resultados para mostrar apenas correlações maiores que 0,1 ou menores que -0,1:

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}
// → weekend:        0.1371988681
// → brushed teeth: -0.3805211953
// → candy:          0.1296407447
// → work:          -0.1371988681
// → spaghetti:      0.2425356250
// → reading:        0.1106828054
// → peanuts:        0.5902679812
```

Ahá! Há dois fatores com uma correlação claramente mais forte que os outros. Comer amendoins tem um forte efeito positivo na chance de se transformar em esquilo, enquanto escovar os dentes tem um significativo efeito negativo.

Interessante. Vamos tentar algo.

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
     !entry.events.includes("brushed teeth")) {
    entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

É um resultado forte. O fenômeno ocorre precisamente quando Jacques come amendoins e não escova os dentes. Se pelo menos ele não fosse tão desleixado com a higiene dental, nem teria notado sua aflição.

Sabendo disso, Jacques para de comer amendoins completamente e descobre que suas transformações param.

Mas são necessários apenas alguns meses para ele notar que algo está faltando nessa forma inteiramente humana de viver. Sem suas aventuras selvagens, Jacques quase não se sente vivo. Ele decide que prefere ser um animal selvagem em tempo integral. Depois de construir uma bela casinha na árvore na floresta e equipá-la com um dispensador de pasta de amendoim e um suprimento de pasta de amendoim para dez anos, ele se transforma uma última vez, e vive a curta e energética vida de um esquilo.

Mais sobre arrays

Antes de terminar o capítulo, quero apresentar mais alguns conceitos relacionados a objetos. Começarei com alguns métodos de array geralmente úteis.

Vimos `push` e `pop`, que adicionam e removem elementos no final de um array, mais cedo neste capítulo. Os métodos correspondentes para adicionar e remover coisas no início de um array são chamados `unshift` e `shift`.

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
```

```
  todoList.unshift(task);
}
```

Esse programa gerencia uma fila de tarefas. Você adiciona tarefas ao final da fila chamando `remember("groceries")`, e quando está pronto para fazer algo, chama `getTask()` para obter (e remover) o item da frente da fila. A função `rememberUrgently` também adiciona uma tarefa, mas a adiciona à frente em vez do final da fila.

Para buscar um valor específico, arrays fornecem um método `indexOf`. O método busca pelo array do início ao fim e retorna o índice em que o valor solicitado foi encontrado — ou -1 se não foi encontrado. Para buscar do final em vez do início, há um método similar chamado `lastIndexOf`:

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

Tanto `indexOf` quanto `lastIndexOf` aceitam um segundo argumento opcional que indica de onde começar a busca.

Outro método fundamental de array é `slice`, que recebe índices de início e fim e retorna um array que contém apenas os elementos entre eles. O índice de início é inclusivo e o índice de fim é exclusivo.

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

Quando o índice de fim não é dado, `slice` pegará todos os elementos após o índice de início. Você também pode omitir o índice de início para copiar o array inteiro.

O método `concat` pode ser usado para juntar arrays e criar um novo array, similar ao que o operador `+` faz para strings.

O exemplo a seguir mostra tanto `concat` quanto `slice` em ação. Ele recebe um array e um índice e retorna um novo array que é uma cópia do array original com o elemento no índice dado removido:

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

Se você passar a `concat` um argumento que não é um array, esse valor será adicionado ao novo array como se fosse um array de um elemento.

Strings e suas propriedades

Podemos ler propriedades como `length` e `toUpperCase` de valores de string. Mas se tentarmos adicionar uma nova propriedade, ela não persiste.

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

Valores do tipo string, number e Boolean não são objetos, e embora a linguagem não reclame se você tentar definir novas propriedades neles, na verdade não armazena essas propriedades. Como mencionado antes, tais valores são imutáveis e não podem ser alterados.

Mas esses tipos têm propriedades embutidas. Toda string tem uma série de métodos. Alguns muito úteis são `slice` e `indexOf`, que se assemelham aos métodos de array com o mesmo nome:

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```

Uma diferença é que o `indexOf` de uma string pode buscar uma string contendo mais de um caractere, enquanto o método de array correspondente busca apenas um único elemento:

```
console.log("one two three".indexOf("ee"));
// → 11
```

O método `trim` remove espaços em branco (espaços, novas linhas, tabulações e caracteres similares) do início e do final de uma string:

```
console.log("  okay \n ".trim());
// → okay
```

A função `zeroPad` do capítulo anterior também existe como método. É chamada `padStart` e recebe o comprimento desejado e o caractere de preenchimento como argumentos:

```
console.log(String(6).padStart(3, "0"));
// → 006
```

Você pode dividir uma string em cada ocorrência de outra string com `split` e juntá-la novamente com `join`:

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

Uma string pode ser repetida com o método `repeat`, que cria uma nova string contendo múltiplas cópias da string original, coladas juntas:

```
console.log("LA".repeat(3));
// → LALALA
```

Já vimos a propriedade `length` do tipo string. Acessar os caracteres individuais em uma string se parece com acessar elementos de um array (com uma complicação que discutiremos no Capítulo 5).

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

Parâmetros rest

Pode ser útil para uma função aceitar qualquer número de argumentos. Por exemplo, `Math.max` calcula o máximo de *todos* os argumentos que recebe. Para escrever tal função, você coloca três pontos antes do último parâmetro da função, assim:

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

Quando tal função é chamada, o *parâmetro rest* é vinculado a um array con-

tendo todos os argumentos subsequentes. Se houver outros parâmetros antes dele, seus valores não fazem parte desse array. Quando, como em `max`, é o único parâmetro, ele conterá todos os argumentos.

Você pode usar uma notação similar de três pontos para *chamar* uma função com um array de argumentos.

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

Isso "espalha" o array na chamada de função, passando seus elementos como argumentos separados. É possível incluir um array assim junto com outros argumentos, como em `max(9, ...numbers, 2)`.

A notação de colchetes de array similarmente permite que o operador de três pontos espalhe outro array no novo array:

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

Isso funciona até em objetos com chaves, onde adiciona todas as propriedades de outro objeto. Se uma propriedade é adicionada múltiplas vezes, o último valor a ser adicionado vence:

```
let coordinates = {x: 10, y: 0};
console.log({...coordinates, y: 5, z: 1});
// → {x: 10, y: 5, z: 1}
```

O objeto Math

Como já vimos, `Math` é um saco de utilidades numéricas, como `Math.max` (máximo), `Math.min` (mínimo) e `Math.sqrt` (raiz quadrada).

O objeto `Math` é usado como contêiner para agrupar funcionalidades relacionadas. Há apenas um objeto `Math`, e ele quase nunca é útil como valor. Em vez disso, ele fornece um *namespace* para que todas essas funções e valores não precisem ser bindings globais.

Ter muitos bindings globais "polui" o namespace. Quanto mais nomes forem tomados, mais provável é que você acidentalmente sobrescreva o valor de algum binding existente. Por exemplo, não é improvável que você queira nomear algo como `max` em um de seus programas. Como a função `max` embutida do JavaScript está guardada com segurança dentro do objeto `Math`, você não precisa se preocupar em sobrescrevê-la.

Muitas linguagens irão pará-lo, ou pelo menos avisá-lo, quando você estiver definindo um binding com um nome que já está tomado. JavaScript faz isso para bindings que você declarou com `let` ou `const` mas — perversamente — não para bindings padrão nem para bindings declarados com `var` ou `function`.

De volta ao objeto `Math`. Se precisar fazer trigonometria, `Math` pode ajudar. Ele contém `cos` (cosseno), `sin` (seno) e `tan` (tangente), bem como suas funções inversas, `acos`, `asin` e `atan`, respectivamente. O número π (pi) — ou pelo menos a aproximação mais próxima que cabe em um número JavaScript — está disponível como `Math.PI`. Há uma velha tradição de programação de escrever os nomes de valores constantes em letras maiúsculas.

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

Se você não está familiarizado com senos e cossenos, não se preocupe. Vou explicá-los quando forem usados no Capítulo 14.

O exemplo anterior usou `Math.random`. Esta é uma função que retorna um novo número pseudoaleatório entre 0 (inclusivo) e 1 (exclusivo) toda vez que você a chama:

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Embora computadores sejam máquinas determinísticas — sempre reagem da mesma forma dado o mesmo input — é possível fazê-los produzir números que parecem aleatórios. Para fazer isso, a máquina mantém algum valor oculto, e sempre que você pede um novo número aleatório, ela realiza computações complicadas nesse valor oculto para criar um novo valor. Ela armazena um novo valor e retorna algum número derivado dele. Dessa forma, pode produzir números sempre novos e difíceis de prever de uma forma que *parece* aleatória.

Se quisermos um número inteiro aleatório em vez de um fracionário, podemos usar `Math.floor` (que arredonda para baixo até o número inteiro mais próximo) no resultado de `Math.random`:

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Multiplicar o número aleatório por 10 nos dá um número maior ou igual a 0 e menor que 10. Como `Math.floor` arredonda para baixo, essa expressão produzirá, com chance igual, qualquer número de 0 a 9.

Há também as funções `Math.ceil` (de "ceiling" ou "teto", que arredonda para cima até um número inteiro), `Math.round` (para o número inteiro mais próximo) e `Math.abs`, que pega o valor absoluto de um número, significando que nega valores negativos mas deixa os positivos como estão.

Desestruturação

Vamos retornar à função `phi` por um momento.

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

Uma razão pela qual esta função é desajeitada de ler é que temos um binding apontando para nosso array, mas preferiríamos muito ter bindings para os *elementos* do array — ou seja, `let n00 = table[0]` e assim por diante. Felizmente, há uma forma sucinta de fazer isso em JavaScript:

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

Isso também funciona para bindings criados com `let`, `var` ou `const`. Se você sabe que o valor que está vinculando é um array, pode usar colchetes para "olhar dentro" do valor, vinculando seu conteúdo.

Um truque similar funciona para objetos, usando chaves em vez de colchetes.

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

Note que se você tentar desestruturar `null` ou `undefined`, obterá um erro, assim

como obteria se tentasse acessar diretamente uma propriedade desses valores.

Acesso opcional a propriedades

Quando você não tem certeza se um dado valor produz um objeto, mas ainda quer ler uma propriedade dele quando produz, pode usar uma variante da notação de ponto: `object?.property`.

```
function city(object) {
  return object.address?.city;
}
console.log(city({address: {city: "Toronto"}}));
// → Toronto
console.log(city({name: "Vera"}));
// → undefined
```

A expressão `a?.b` significa o mesmo que `a.b` quando `a` não é null nem undefined. Quando é, ela avalia para `undefined`. Isso pode ser conveniente quando, como no exemplo, você não tem certeza de que uma dada propriedade existe ou quando uma variável pode conter um valor indefinido.

Uma notação similar pode ser usada com acesso por colchetes, e até com chamadas de função, colocando `?.` na frente dos parênteses ou colchetes:

```
console.log("string".notAMethod?.());
// → undefined
console.log({}.arrayProp?.[0]);
// → undefined
```

JSON

Como propriedades agarram seu valor em vez de contê-lo, objetos e arrays são armazenados na memória do computador como sequências de bits contendo os *endereços* — o lugar na memória — de seu conteúdo. Um array com outro array dentro dele consiste em (pelo menos) uma região de memória para o array interno e outra para o array externo, contendo (entre outras coisas) um número que representa o endereço do array interno.

Se você quer salvar dados em um arquivo para uso posterior ou enviá-los para outro computador pela rede, precisa de alguma forma converter esses emaranhados de endereços de memória em uma descrição que possa ser armazenada ou enviada. Você *poderia* enviar toda a memória do seu computador junto com o endereço do valor em que está interessado, suponho, mas essa não parece ser

a melhor abordagem.

O que podemos fazer é *serializar* os dados. Isso significa convertê-los em uma descrição plana. Um formato de serialização popular é chamado *JSON* (pronuncia-se "Jason"), que significa JavaScript Object Notation. É amplamente usado como formato de armazenamento e comunicação de dados na web, mesmo com linguagens diferentes de JavaScript.

JSON se parece com a forma do JavaScript de escrever arrays e objetos, com algumas restrições. Todos os nomes de propriedades devem estar entre aspas duplas, e apenas expressões de dados simples são permitidas — sem chamadas de função, bindings ou qualquer coisa que envolva computação real. Comentários não são permitidos em JSON.

Uma entrada de diário pode parecer assim quando representada como dados JSON:

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript nos dá as funções `JSON.stringify` e `JSON.parse` para converter dados de e para esse formato. A primeira recebe um valor JavaScript e retorna uma string codificada em JSON. A segunda recebe tal string e a converte no valor que ela codifica:

```
let string = JSON.stringify({squirrel: false,
                             events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

Resumo

Objetos e arrays fornecem formas de agrupar vários valores em um único valor. Isso nos permite colocar um monte de coisas relacionadas em um saco e correr com o saco em vez de envolver nossos braços ao redor de todas as coisas individuais e tentar segurá-las separadamente.

A maioria dos valores em JavaScript tem propriedades, com as exceções sendo `null` e `undefined`. Propriedades são acessadas usando `value.prop` ou `value["prop"]`. Objetos tendem a usar nomes para suas propriedades e armazenar um conjunto mais ou menos fixo delas. Arrays, por outro lado, geralmente contêm

quantidades variáveis de valores conceitualmente idênticos e usam números (começando de 0) como nomes de suas propriedades.

Existem algumas propriedades nomeadas em arrays, como `length` e uma série de métodos. Métodos são funções que vivem em propriedades e (geralmente) agem sobre o valor do qual são propriedade.

Você pode iterar sobre arrays usando um tipo especial de loop `for`: `for (let element of array)`.

Exercícios

A soma de um intervalo

A introdução deste livro aludiu ao seguinte como uma forma elegante de calcular a soma de um intervalo de números:

```
console.log(sum(range(1, 10)));
```

Escreva uma função `range` que recebe dois argumentos, `start` e `end`, e retorna um array contendo todos os números de `start` até e incluindo `end`.

Em seguida, escreva uma função `sum` que recebe um array de números e retorna a soma desses números. Execute o programa de exemplo e veja se ele de fato retorna 55.

Como tarefa bônus, modifique sua função `range` para receber um terceiro argumento opcional que indica o valor de "passo" usado ao construir o array. Se nenhum passo for dado, os elementos devem subir em incrementos de um, correspondendo ao comportamento antigo. A chamada de função `range(1, 10, 2)` deve retornar `[1, 3, 5, 7, 9]`. Certifique-se de que isso também funciona com valores de passo negativos para que `range(5, 2, -1)` produza `[5, 4, 3, 2]`.

Invertendo um array

Arrays têm um método `reverse` que muda o array invertendo a ordem em que seus elementos aparecem. Para este exercício, escreva duas funções, `reverseArray` e `reverseArrayInPlace`. A primeira, `reverseArray`, deve receber um array como argumento e produzir um *novo* array que tem os mesmos elementos na ordem inversa. A segunda, `reverseArrayInPlace`, deve fazer o que o método `reverse` faz: *modificar* o array dado como argumento invertendo seus elementos. Nenhuma das duas pode usar o método `reverse` padrão.

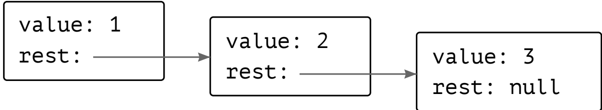
Pensando nas notas sobre efeitos colaterais e funções puras no capítulo anterior, qual variante você espera ser útil em mais situações? Qual é mais rápida?

Uma lista

Como blobs genéricos de valores, objetos podem ser usados para construir todo tipo de estruturas de dados. Uma estrutura de dados comum é a *lista* (não confundir com arrays). Uma lista é um conjunto aninhado de objetos, com o primeiro objeto contendo uma referência ao segundo, o segundo ao terceiro, e assim por diante:

```
let list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
  }
};
```

Os objetos resultantes formam uma cadeia, como mostrado no diagrama a seguir:



Uma coisa legal sobre listas é que elas podem compartilhar partes de sua estrutura. Por exemplo, se eu criar dois novos valores `{value: 0, rest: list}` e `{value: -1, rest: list}` (com `list` se referindo ao binding definido antes), ambos são listas independentes, mas compartilham a estrutura que compõe seus últimos três elementos. A lista original também continua sendo uma lista válida de três elementos.

Escreva uma função `arrayToList` que constrói uma estrutura de lista como a mostrada quando recebe `[1, 2, 3]` como argumento. Também escreva uma função `listToArray` que produz um array a partir de uma lista. Adicione as funções auxiliares `prepend`, que recebe um elemento e uma lista e cria uma nova lista que adiciona o elemento à frente da lista de entrada, e `nth`, que recebe uma lista e um número e retorna o elemento na posição dada na lista (com zero se referindo ao primeiro elemento) ou `undefined` quando não há tal elemento.

Se ainda não o fez, também escreva uma versão recursiva de `nth`.

Comparação profunda

O operador `==` compara objetos por identidade, mas às vezes você prefere comparar os valores de suas propriedades reais.

Escreva uma função `deepEqual` que recebe dois valores e retorna `true` apenas se eles forem o mesmo valor ou forem objetos com as mesmas propriedades, onde os valores das propriedades são iguais quando comparados com uma chamada recursiva a `deepEqual`.

Para descobrir se valores devem ser comparados diretamente (usando o operador `===` para isso) ou ter suas propriedades comparadas, você pode usar o operador `typeof`. Se ele produzir `"object"` para ambos os valores, você deve fazer uma comparação profunda. Mas precisa considerar uma exceção boba: por um acidente histórico, `typeof null` também produz `"object"`.

A função `Object.keys` será útil quando você precisar percorrer as propriedades dos objetos para compará-los.

“

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

—C.A.R. Hoare, 1980 ACM Turing Award Lecture

Chapter 5

Funções de Ordem Superior

Um programa grande é um programa custoso, e não apenas pelo tempo que leva para construir. Tamanho quase sempre envolve complexidade, e complexidade confunde programadores. Programadores confusos, por sua vez, introduzem erros (*bugs*) em programas. Um programa grande então fornece muito espaço para esses bugs se esconderem, tornando-os difíceis de encontrar.

Vamos voltar brevemente aos dois últimos programas de exemplo na introdução. O primeiro é autocontido e tem seis linhas.

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

O segundo depende de duas funções externas e tem uma linha.

```
console.log(sum(range(1, 10)));
```

Qual tem mais chance de conter um bug?

Se contarmos o tamanho das definições de `sum` e `range`, o segundo programa também é grande — até maior que o primeiro. Mas ainda assim, eu argumentaria que é mais provável que esteja correto.

Isso porque a solução é expressa em um vocabulário que corresponde ao problema sendo resolvido. Somar um intervalo de números não é sobre loops e contadores. É sobre intervalos e somas.

As definições desse vocabulário (as funções `sum` e `range`) ainda envolverão loops, contadores e outros detalhes incidentais. Mas porque estão expressando conceitos mais simples que o programa como um todo, são mais fáceis de acertar.

Abstração

No contexto da programação, esses tipos de vocabulários são geralmente chamados de *abstrações*. Abstrações nos dão a capacidade de falar sobre problemas em um nível mais alto (ou mais abstrato), sem nos desviar com detalhes desinteressantes.

Como analogia, compare essas duas receitas de sopa de ervilha. A primeira vai assim:

> Coloque 1 xícara de ervilhas secas por pessoa em um recipiente. Adicione água até as ervilhas estarem bem cobertas. Deixe as ervilhas de molho na água por pelo menos 12 horas. Retire as ervilhas da água e coloque-as em uma panela. Adicione 4 xícaras de água por pessoa. Cubra a panela e mantenha as ervilhas cozinhando em fogo brando por duas horas. Pegue meia cebola por pessoa. Corte-a em pedaços com uma faca. Adicione às ervilhas. Pegue um talo de salsão por pessoa. Corte-o em pedaços com uma faca. Adicione às ervilhas. Pegue uma cenoura por pessoa. Corte-a em pedaços. Com uma faca! Adicione às ervilhas. Cozinhe por mais 10 minutos.

E esta é a segunda receita:

> Por pessoa: 1 xícara de ervilhas secas partidas, 4 xícaras de água, meia cebola picada, um talo de salsão e uma cenoura.
>
> Deixe as ervilhas de molho por 12 horas. Cozinhe em fogo brando por 2 horas. Pique e adicione os vegetais. Cozinhe por mais 10 minutos.

A segunda é mais curta e mais fácil de interpretar. Mas você precisa entender mais algumas palavras relacionadas à culinária, como *de molho*, *fogo brando*, *picar*, e, suponho, *vegetal*.

Ao programar, não podemos contar que todas as palavras de que precisamos estarão nos esperando no dicionário. Assim, podemos cair no padrão da primeira receita — detalhar os passos precisos que o computador deve realizar, um por um, cegos aos conceitos de nível mais alto que eles expressam.

É uma habilidade útil, em programação, perceber quando você está trabalhando em um nível de abstração muito baixo.

Abstraindo repetição

Funções simples, como as vimos até agora, são uma boa forma de construir abstrações. Mas às vezes ficam aquém.

É comum um programa fazer algo um certo número de vezes. Você pode escrever um loop `for` para isso, assim:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Podemos abstrair "fazer algo *N* vezes" como uma função? Bem, é fácil escrever uma função que chama `console.log` *N* vezes.

```
function repeatLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

Mas e se quisermos fazer algo diferente de registrar os números? Como "fazer algo" pode ser representado como uma função e funções são apenas valores, podemos passar nossa ação como um valor de função.

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

Não precisamos passar uma função predefinida para `repeat`. Frequentemente, é mais fácil criar um valor de função na hora.

```
let labels = [];
repeat(5, i => {
  labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

Isso é estruturado um pouco como um loop `for` — primeiro descreve o tipo de

loop e depois fornece um corpo. No entanto, o corpo agora é escrito como um valor de função, que está envolvido nos parênteses da chamada a `repeat`. É por isso que precisa ser fechado com a chave de fechamento *e* o parêntese de fechamento. Em casos como este exemplo, onde o corpo é uma única expressão pequena, você também pode omitir as chaves e escrever o loop em uma única linha.

Funções de ordem superior

Funções que operam sobre outras funções, seja recebendo-as como argumentos ou retornando-as, são chamadas de *funções de ordem superior*. Como já vimos que funções são valores regulares, não há nada particularmente notável no fato de que tais funções existem. O termo vem da matemática, onde a distinção entre funções e outros valores é levada mais a sério.

Funções de ordem superior nos permitem abstrair sobre *ações*, não apenas valores. Elas vêm em várias formas. Por exemplo, podemos ter funções que criam novas funções.

```
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

Também podemos ter funções que mudam outras funções.

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

Podemos até escrever funções que fornecem novos tipos de fluxo de controle.

```
function unless(test, then) {
  if (!test) then();
}
```

```
repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, "is even");
  });
});
// → 0 is even
// → 2 is even
```

Existe um método de array embutido, `forEach`, que fornece algo como um loop `for`/`of` como uma função de ordem superior.

```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

Conjunto de dados de scripts

Uma área onde funções de ordem superior brilham é processamento de dados. Para processar dados, precisamos de alguns dados de exemplo reais. Este capítulo usará um conjunto de dados sobre scripts — sistemas de escrita como Latim, Cirílico ou Árabe.

Lembra do Unicode, o sistema que atribui um número a cada caractere em linguagem escrita, do Capítulo 1? A maioria desses caracteres está associada a um script específico. O padrão contém 140 scripts diferentes, dos quais 81 ainda estão em uso hoje e 59 são históricos.

Embora eu só consiga ler fluentemente caracteres latinos, aprecio o fato de que pessoas estão escrevendo textos em pelo menos 80 outros sistemas de escrita, muitos dos quais eu nem reconheceria. Por exemplo, aqui está uma amostra de escrita manual em tâmil:

O conjunto de dados de exemplo contém algumas informações sobre os 140 scripts definidos no Unicode. Está disponível no sandbox de codificação para este capítulo (*https://eloquentjavascript.net/code#5*) como o binding `SCRIPTS`. O binding contém um array de objetos, cada um dos quais descreve um script.

```
{
  name: "Coptic",
```

```
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Tal objeto nos diz o nome do script, os intervalos Unicode atribuídos a ele, a direção em que é escrito, o tempo de origem (aproximado), se ainda está em uso e um link para mais informações. A direção pode ser `"ltr"` para esquerda para direita, `"rtl"` para direita para esquerda (a forma como texto árabe e hebraico são escritos) ou `"ttb"` para cima para baixo (como na escrita mongol).

A propriedade `ranges` contém um array de intervalos de caracteres Unicode, cada um dos quais é um array de dois elementos contendo um limite inferior e um limite superior. Quaisquer códigos de caracteres dentro desses intervalos são atribuídos ao script. O limite inferior é inclusivo (o código 994 é um caractere copta) e o limite superior é não inclusivo (o código 1008 não é).

Filtrando arrays

Se quisermos encontrar os scripts no conjunto de dados que ainda estão em uso, a seguinte função pode ser útil. Ela filtra os elementos de um array que não passam em um teste.

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}

console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", …}, …]
```

A função usa o argumento chamado `test`, um valor de função, para preencher uma "lacuna" na computação — o processo de decidir quais elementos coletar.

Note como a função `filter`, em vez de deletar elementos do array existente, constrói um novo array com apenas os elementos que passam no teste. Esta função é *pura*. Ela não modifica o array que lhe é dado.

Como `forEach`, `filter` é um método de array padrão. O exemplo definiu a

função apenas para mostrar o que ela faz internamente. De agora em diante, a usaremos assim:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", …}, …]
```

Transformando com map

Digamos que temos um array de objetos representando scripts, produzido ao filtrar o array `SCRIPTS` de alguma forma. Queremos um array de nomes, que é mais fácil de inspecionar.

O método `map` transforma um array aplicando uma função a todos os seus elementos e construindo um novo array a partir dos valores retornados. O novo array terá o mesmo comprimento que o array de entrada, mas seu conteúdo terá sido *mapeado* para uma nova forma pela função.

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", …]
```

Como `forEach` e `filter`, `map` é um método de array padrão.

Resumindo com reduce

Outra coisa comum de se fazer com arrays é calcular um único valor a partir deles. Nosso exemplo recorrente, somar uma coleção de números, é um exemplo disso. Outro exemplo é encontrar o script com mais caracteres.

A operação de ordem superior que representa esse padrão é chamada de *reduce* (às vezes também chamada de *fold*). Ela constrói um valor tomando repetidamente um único elemento do array e combinando-o com o valor atual. Ao somar números, você começaria com o número zero e, para cada elemento, adicionaria ao total.

Os parâmetros de `reduce` são, além do array, uma função de combinação e

um valor inicial. Esta função é um pouco menos direta que `filter` e `map`, então observe-a de perto:

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

O método de array padrão `reduce`, que naturalmente corresponde a esta função, tem uma conveniência adicional. Se seu array contiver pelo menos um elemento, você pode omitir o argumento `start`. O método pegará o primeiro elemento do array como seu valor inicial e começará a reduzir a partir do segundo elemento.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

Para usar `reduce` (duas vezes) para encontrar o script com mais caracteres, podemos escrever algo assim:

```
function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", …}
```

A função `characterCount` reduz os intervalos atribuídos a um script somando seus tamanhos. Note o uso de desestruturação na lista de parâmetros da função redutora. A segunda chamada a `reduce` então usa isso para encontrar o maior script comparando repetidamente dois scripts e retornando o maior.

O script Han tem mais de 89.000 caracteres atribuídos a ele no padrão Unicode, tornando-o de longe o maior sistema de escrita no conjunto de dados. Han é um script às vezes usado para texto chinês, japonês e coreano. Essas línguas compartilham muitos caracteres, embora tendam a escrevê-los de forma diferente. O Consórcio Unicode (baseado nos EUA) decidiu tratá-los como um

único sistema de escrita para economizar códigos de caracteres. Isso é chamado de *unificação Han* e ainda deixa algumas pessoas muito irritadas.

Composabilidade

Considere como teríamos escrito o exemplo anterior (encontrar o maior script) sem funções de ordem superior. O código não é muito pior.

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", …}
```

Há mais alguns bindings, e o programa é quatro linhas mais longo, mas ainda é muito legível.

As abstrações que essas funções fornecem realmente brilham quando você precisa *compor* operações. Como exemplo, vamos escrever código que encontra o ano médio de origem para scripts vivos e mortos no conjunto de dados.

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1165
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 204
```

Como você pode ver, os scripts mortos no Unicode são, em média, mais antigos que os vivos. Essa não é uma estatística terrivelmente significativa ou surpreendente. Mas espero que você concorde que o código usado para calculá-la não é difícil de ler. Você pode vê-lo como um pipeline: começamos com todos os scripts, filtramos os vivos (ou mortos), pegamos os anos desses, calculamos a média e arredondamos o resultado.

Você definitivamente também poderia escrever essa computação como um grande loop.

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1165
```

No entanto, é mais difícil ver o que estava sendo computado e como. E como resultados intermediários não são representados como valores coerentes, seria muito mais trabalhoso extrair algo como `average` em uma função separada.

Em termos do que o computador realmente está fazendo, essas duas abordagens também são bastante diferentes. A primeira construirá novos arrays ao executar `filter` e `map`, enquanto a segunda computa apenas alguns números, fazendo menos trabalho. Geralmente você pode se dar ao luxo da abordagem legível, mas se estiver processando arrays enormes e fazendo isso muitas vezes, o estilo menos abstrato pode valer a velocidade extra.

Strings e códigos de caracteres

Um uso interessante desse conjunto de dados seria descobrir qual script um trecho de texto está usando. Vamos percorrer um programa que faz isso.

Lembre-se de que cada script tem um array de intervalos de códigos de caracteres associados a ele. Dado um código de caractere, poderíamos usar uma função como esta para encontrar o script correspondente (se houver):

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
  return null;
}

console.log(characterScript(121));
// → {name: "Latin", …}
```

O método `some` é outra função de ordem superior. Ele recebe uma função de

teste e lhe diz se essa função retorna verdadeiro para qualquer um dos elementos no array.

Mas como obtemos os códigos de caracteres em uma string?

No Capítulo 1 mencionei que strings JavaScript são codificadas como uma sequência de números de 16 bits. Estes são chamados de *unidades de código*. Um código de caractere Unicode supostamente caberia em tal unidade (o que lhe dá um pouco mais de 65.000 caracteres). Quando ficou claro que isso não seria suficiente, muitas pessoas relutaram em usar mais memória por caractere. Para resolver essas preocupações, UTF-16, o formato também usado pelas strings JavaScript, foi inventado. Ele descreve a maioria dos caracteres comuns usando uma única unidade de código de 16 bits, mas usa um par de duas unidades para outros.

UTF-16 é geralmente considerado uma má ideia hoje. Parece quase intencionalmente projetado para provocar erros. É fácil escrever programas que fingem que unidades de código e caracteres são a mesma coisa. E se sua linguagem não usa caracteres de duas unidades, isso parecerá funcionar perfeitamente. Mas assim que alguém tentar usar tal programa com alguns caracteres chineses menos comuns, ele quebra. Felizmente, com o advento dos emoji, todos começaram a usar caracteres de duas unidades, e o fardo de lidar com tais problemas é mais justamente distribuído.

Infelizmente, operações óbvias em strings JavaScript, como obter seu comprimento através da propriedade `length` e acessar seu conteúdo usando colchetes, lidam apenas com unidades de código.

```
// Dois caracteres emoji, cavalo e sapato
let horseShoe = "";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Meio-caractere inválido)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Código do meio-caractere)
console.log(horseShoe.codePointAt(0));
// → 128052 (Código real do emoji de cavalo)
```

O método `charCodeAt` do JavaScript lhe dá uma unidade de código, não um código de caractere completo. O método `codePointAt`, adicionado depois, dá um caractere Unicode completo, então poderíamos usá-lo para obter caracteres de uma string. Mas o argumento passado a `codePointAt` ainda é um índice na sequência de unidades de código. Para percorrer todos os caracteres em uma string, ainda precisaríamos lidar com a questão de se um caractere ocupa uma ou duas unidades de código.

No capítulo anterior, mencionei que um loop `for/of` também pode ser usado em strings. Assim como `codePointAt`, esse tipo de loop foi introduzido em um momento em que as pessoas estavam agudamente cientes dos problemas com UTF-16. Quando você o usa para percorrer uma string, ele lhe dá caracteres reais, não unidades de código.

```
let roseDragon = "";
for (let char of roseDragon) {
  console.log(char);
}
// →
// →
```

Se você tem um caractere (que será uma string de uma ou duas unidades de código), pode usar `codePointAt(0)` para obter seu código.

Reconhecendo texto

Temos uma função `characterScript` e uma forma de percorrer caracteres corretamente. O próximo passo é contar os caracteres que pertencem a cada script. A seguinte abstração de contagem será útil aqui:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.find(c => c.name == name);
    if (!known) {
      counts.push({name, count: 1});
    } else {
      known.count++;
    }
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

A função `countBy` espera uma coleção (qualquer coisa sobre a qual possamos iterar com `for/of`) e uma função que computa um nome de grupo para um dado elemento. Ela retorna um array de objetos, cada um dos quais nomeia um grupo e lhe diz o número de elementos que foram encontrados naquele grupo.

Ela usa outro método de array, `find`, que percorre os elementos do array e retorna o primeiro para o qual uma função retorna verdadeiro. Retorna `undefined` quando não encontra tal elemento.

Usando `countBy`, podemos escrever a função que nos diz quais scripts são usados em um trecho de texto.

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

A função primeiro conta os caracteres por nome, usando `characterScript` para atribuir-lhes um nome e recorrendo à string `"none"` para caracteres que não fazem parte de nenhum script. A chamada `filter` descarta a entrada para `"none"` do array resultante, já que não estamos interessados nesses caracteres.

Para poder calcular porcentagemns, primeiro precisamos do número total de caracteres que pertencem a um script, que podemos calcular com `reduce`. Se não encontrarmos tais caracteres, a função retorna uma string específica. Caso contrário, ela transforma as entradas de contagem em strings legíveis com `map` e então as combina com `join`.

Resumo

Poder passar valores de função para outras funções é um aspecto profundamente útil de JavaScript. Nos permite escrever funções que modelam computações com "lacunas" nelas. O código que chama essas funções pode preencher as lacunas fornecendo valores de função.

Arrays fornecem uma série de métodos de ordem superior úteis. Você pode usar `forEach` para percorrer os elementos de um array. O método `filter` retorna um novo array contendo apenas os elementos que passam na função

predicado. Você pode transformar um array passando cada elemento por uma função usando `map`. Pode usar `reduce` para combinar todos os elementos de um array em um único valor. O método `some` testa se algum elemento corresponde a uma dada função predicado, enquanto `find` encontra o primeiro elemento que corresponde a um predicado.

Exercícios

Achatamento

Use o método `reduce` em combinação com o método `concat` para "achatar" um array de arrays em um único array que tem todos os elementos dos arrays originais.

Seu próprio loop

Escreva uma função de ordem superior `loop` que forneça algo como uma instrução de loop `for`. Ela deve receber um valor, uma função de teste, uma função de atualização e uma função de corpo. A cada iteração, ela deve primeiro executar a função de teste no valor atual do loop e parar se retornar `false`. Então deve chamar a função de corpo, dando a ela o valor atual, e finalmente chamar a função de atualização para criar um novo valor e recomeçar do início.

Ao definir a função, você pode usar um loop regular para fazer o loop de fato.

Everything

Arrays também têm um método `every` análogo ao método `some`. Este método retorna `true` quando a função dada retorna `true` para *todos* os elementos no array. De certa forma, `some` é uma versão do operador `||` que age sobre arrays, e `every` é como o operador `&&`.

Implemente `every` como uma função que recebe um array e uma função predicado como parâmetros. Escreva duas versões, uma usando um loop e uma usando o método `some`.

Direção de escrita dominante

Escreva uma função que calcule a direção de escrita dominante em uma string de texto. Lembre-se de que cada objeto de script tem uma propriedade `direction` que pode ser `"ltr"` (esquerda para direita), `"rtl"` (direita para esquerda) ou `"ttb"` (cima para baixo).

A direção dominante é a direção da maioria dos caracteres que têm um script associado a eles. As funções `characterScript` e `countBy` definidas anteriormente no capítulo provavelmente são úteis aqui.

"Um tipo abstrato de dados é realizado escrevendo-se um tipo especial de programa [...] que define o tipo em termos das operações que podem ser executadas sobre ele."

—Barbara Liskov, Programming with Abstract Data Types

Chapter 6

A Vida Secreta dos Objetos

O Capítulo 4 introduziu os objetos do JavaScript como contêineres que armazenam outros dados. Na cultura da programação, a *programação orientada a objetos* é um conjunto de técnicas que usa objetos como princípio central de organização de programas. Embora ninguém realmente concorde sobre sua definição precisa, a programação orientada a objetos moldou o design de muitas linguagens de programação, incluindo o JavaScript. Este capítulo descreve como essas ideias podem ser aplicadas em JavaScript.

Tipos Abstratos de Dados

A ideia principal na programação orientada a objetos é usar objetos, ou melhor, *tipos* de objetos, como unidade de organização de programas. Configurar um programa como um conjunto de tipos de objetos estritamente separados fornece uma maneira de pensar sobre sua estrutura e, assim, impor algum tipo de disciplina, evitando que tudo fique emaranhado.

A maneira de fazer isso é pensar em objetos de forma semelhante a como você pensaria em um liquidificador elétrico ou outro eletrodoméstico. As pessoas que projetam e montam um liquidificador precisam fazer um trabalho especializado que requer ciência dos materiais e conhecimento de eletricidade. Elas cobrem tudo isso com uma carcaça plástica lisa para que as pessoas que só querem misturar massa de panqueca não precisem se preocupar com tudo aquilo — elas só precisam entender os poucos botões com os quais o liquidificador pode ser operado.

De forma semelhante, um *tipo abstrato de dados*, ou *classe de objetos*, é um subprograma que pode conter código arbitrariamente complicado, mas expõe um conjunto limitado de métodos e propriedades que as pessoas que trabalham com ele devem usar. Isso permite que programas grandes sejam construídos a partir de vários tipos de eletrodomésticos, limitando o grau de emaranhamento entre essas diferentes partes ao exigir que interajam apenas de maneiras específicas.

Se um problema é encontrado em uma dessas classes de objetos, frequentemente ele pode ser reparado ou até completamente reescrito sem impactar o restante do programa. Melhor ainda, pode ser possível usar classes de objetos em vários programas diferentes, evitando a necessidade de recriar sua funcionalidade do zero. Você pode pensar nas estruturas de dados embutidas do JavaScript, como *arrays* e *strings*, como esses tipos abstratos de dados reutilizáveis.

Cada tipo abstrato de dados tem uma *interface*, a coleção de operações que o código externo pode executar sobre ele. Quaisquer detalhes além dessa interface são *encapsulados*, tratados como internos ao tipo e sem importância para o restante do programa.

Até coisas básicas como números podem ser pensadas como um tipo abstrato de dados cuja interface nos permite somá-los, multiplicá-los, compará-los e assim por diante. Na verdade, a fixação em *objetos* individuais como a unidade principal de organização na programação orientada a objetos clássica é um tanto infeliz, pois funcionalidades úteis frequentemente envolvem um grupo de diferentes classes de objetos trabalhando juntas.

Métodos

Em JavaScript, métodos são nada mais que propriedades que armazenam valores de função. Este é um método simples:

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};

whiteRabbit.speak("Oh my fur and whiskers");
// → The white rabbit says 'Oh my fur and whiskers'
hungryRabbit.speak("Got any carrots?");
// → The hungry rabbit says 'Got any carrots?'
```

Normalmente, um método precisa fazer algo com o objeto no qual foi chamado. Quando uma função é chamada como método — pesquisada como propriedade e chamada imediatamente, como em `object.method()` — a *binding* chamada `this` em seu corpo aponta automaticamente para o objeto no qual foi chamada.

Você pode pensar em `this` como um parâmetro extra que é passado para a função de uma maneira diferente dos parâmetros regulares. Se quiser fornecê-lo explicitamente, pode usar o método `call` de uma função, que recebe o valor

de `this` como seu primeiro argumento e trata os argumentos adicionais como parâmetros normais.

```
speak.call(whiteRabbit, "Hurry");
// → The white rabbit says 'Hurry'
```

Como cada função tem sua própria *binding* `this`, cujo valor depende da maneira como é chamada, você não pode se referir ao `this` do escopo envolvente em uma função regular definida com a palavra-chave `function`.

Arrow functions são diferentes — elas não vinculam seu próprio `this`, mas podem ver a *binding* `this` do escopo ao redor delas. Assim, você pode fazer algo como o código a seguir, que referencia `this` de dentro de uma função local:

```
let finder = {
  find(array) {
    return array.some(v => v == this.value);
  },
  value: 5
};
console.log(finder.find([4, 5]));
// → true
```

Uma propriedade como `find(array)` em uma expressão de objeto é uma forma abreviada de definir um método. Ela cria uma propriedade chamada `find` e dá a ela uma função como valor.

Se eu tivesse escrito o argumento de `some` usando a palavra-chave `function`, este código não funcionaria.

Protótipos

Uma maneira de criar um tipo de objeto coelho com um método `speak` seria criar uma função auxiliar que recebe o tipo do coelho como parâmetro e retorna um objeto contendo isso como sua propriedade `type` e nossa função speak em sua propriedade `speak`.

Todos os coelhos compartilham esse mesmo método. Especialmente para tipos com muitos métodos, seria bom se houvesse uma maneira de manter os métodos de um tipo em um único lugar, em vez de adicioná-los a cada objeto individualmente.

Em JavaScript, *protótipos* são a maneira de fazer isso. Objetos podem ser vinculados a outros objetos, para magicamente obter todas as propriedades que o outro objeto tem. Objetos comuns criados com a notação `{}` são vinculados a um objeto chamado `Object.prototype`.

```
let empty = {};
console.log(empty.toString);
// → function toString()…{}
console.log(empty.toString());
// → [object Object]
```

Parece que acabamos de extrair uma propriedade de um objeto vazio. Mas na verdade, `toString` é um método armazenado em `Object.prototype`, o que significa que ele está disponível na maioria dos objetos.

Quando um objeto recebe uma solicitação por uma propriedade que não possui, seu protótipo será pesquisado pela propriedade. Se esse não a tiver, o protótipo *do protótipo* é pesquisado, e assim por diante até que um objeto sem protótipo seja alcançado (`Object.prototype` é um desses objetos).

```
console.log(Object.getPrototypeOf({}) == Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Como você pode imaginar, `Object.getPrototypeOf` retorna o protótipo de um objeto.

Muitos objetos não têm `Object.prototype` diretamente como seu protótipo, mas sim outro objeto que fornece um conjunto diferente de propriedades padrão. Funções derivam de `Function.prototype` e *arrays* derivam de `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) == Array.prototype);
// → true
```

Esse objeto protótipo terá, ele próprio, um protótipo, frequentemente `Object.prototype`, de modo que ainda fornece indiretamente métodos como `toString`.

Você pode usar `Object.create` para criar um objeto com um protótipo específico.

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};
let blackRabbit = Object.create(protoRabbit);
blackRabbit.type = "black";
blackRabbit.speak("I am fear and darkness");
// → The black rabbit says 'I am fear and darkness'
```

O coelho "proto" atua como um contêiner para as propriedades compartilhadas por todos os coelhos. Um objeto coelho individual, como o coelho preto, contém propriedades que se aplicam apenas a ele mesmo — neste caso, seu tipo — e deriva propriedades compartilhadas de seu protótipo.

Classes

O sistema de protótipos do JavaScript pode ser interpretado como uma abordagem um tanto livre de tipos abstratos de dados ou classes. Uma *classe* define a forma de um tipo de objeto — quais métodos e propriedades ele tem. Tal objeto é chamado de *instância* da classe.

Protótipos são úteis para definir propriedades cujo valor é o mesmo para todas as instâncias de uma classe. Propriedades que diferem por instância, como a propriedade `type` dos nossos coelhos, precisam ser armazenadas diretamente nos próprios objetos.

Para criar uma instância de uma dada classe, você precisa criar um objeto que derive do protótipo adequado, mas *também* precisa garantir que ele próprio tenha as propriedades que instâncias dessa classe devem ter. É isso que uma função *construtora* faz.

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

A notação de classe do JavaScript facilita a definição desse tipo de função, juntamente com um objeto protótipo.

```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
}
```

A palavra-chave `class` inicia uma declaração de classe, que nos permite definir um construtor e um conjunto de métodos juntos. Qualquer número de métodos pode ser escrito dentro das chaves da declaração. Este código tem o efeito de definir uma *binding* chamada `Rabbit`, que armazena uma função que executa o código em `constructor` e tem uma propriedade `prototype` que contém o método

`speak`.

Essa função não pode ser chamada como uma função normal. Construtores, em JavaScript, são chamados colocando a palavra-chave `new` na frente deles. Fazer isso cria um novo objeto instância cujo protótipo é o objeto encontrado na propriedade `prototype` da função, depois executa a função com `this` vinculado ao novo objeto e, finalmente, retorna o objeto.

```
let killerRabbit = new Rabbit("killer");
```

Na verdade, `class` foi introduzido apenas na edição de 2015 do JavaScript. Qualquer função pode ser usada como construtor, e antes de 2015, a maneira de definir uma classe era escrever uma função regular e depois manipular sua propriedade `prototype`.

```
function ArchaicRabbit(type) {
  this.type = type;
}
ArchaicRabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};
let oldSchoolRabbit = new ArchaicRabbit("old school");
```

Por essa razão, todas as funções não-arrow começam com uma propriedade `prototype` contendo um objeto vazio.

Por convenção, os nomes dos construtores são capitalizados para que possam ser facilmente distinguidos de outras funções.

É importante entender a distinção entre a maneira como um protótipo é associado a um construtor (através de sua propriedade `prototype`) e a maneira como objetos *têm* um protótipo (que pode ser encontrado com `Object.getPrototypeOf`). O protótipo real de um construtor é `Function.prototype`, pois construtores são funções. A *propriedade* `prototype` da função construtora contém o protótipo usado pelas instâncias criadas através dela.

```
console.log(Object.getPrototypeOf(Rabbit) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(killerRabbit) ==
            Rabbit.prototype);
// → true
```

Construtores normalmente adicionam algumas propriedades por instância a `this`. Também é possível declarar propriedades diretamente na declaração de classe. Diferentemente dos métodos, essas propriedades são adicionadas aos objetos instância e não ao protótipo.

```
class Particle {
  speed = 0;
  constructor(position) {
    this.position = position;
  }
}
```

Assim como `function`, `class` pode ser usado tanto em declarações quanto em expressões. Quando usado como expressão, não define uma *binding*, mas apenas produz o construtor como valor. Você pode omitir o nome da classe em uma expressão de classe.

```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

Propriedades Privadas

É comum que classes definam algumas propriedades e métodos para uso interno que não fazem parte de sua interface. Estas são chamadas de propriedades *privadas*, em oposição às *públicas*, que fazem parte da interface externa do objeto.

Para declarar um método privado, coloque um sinal `#` na frente de seu nome. Esses métodos podem ser chamados apenas de dentro da declaração `class` que os define.

```
class SecretiveObject {
  #getSecret() {
    return "I ate all the plums";
  }
  interrogate() {
    let shallISayIt = this.#getSecret();
    return "never";
  }
}
```

Quando uma classe não declara um construtor, ela receberá automaticamente um vazio.

Se você tentar chamar `#getSecret` de fora da classe, receberá um erro. Sua existência é inteiramente oculta dentro da declaração da classe.

Para usar propriedades de instância privadas, você deve declará-las. Propriedades regulares podem ser criadas simplesmente atribuindo a elas, mas propriedades privadas *devem* ser declaradas na declaração da classe para estarem

disponíveis.

Esta classe implementa um dispositivo para obter um número inteiro aleatório abaixo de um número máximo dado. Ela tem apenas uma propriedade pública: `getNumber`.

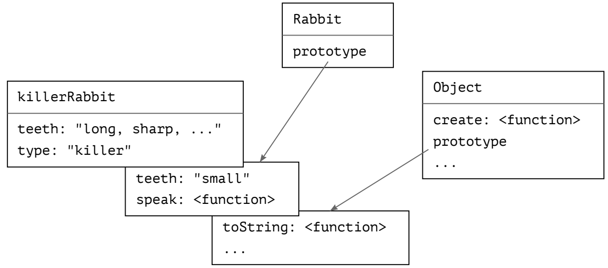
```
class RandomSource {
  #max;
  constructor(max) {
    this.#max = max;
  }
  getNumber() {
    return Math.floor(Math.random() * this.#max);
  }
}
```

Sobrescrevendo propriedades derivadas

Quando você adiciona uma propriedade a um objeto, esteja ela presente no protótipo ou não, a propriedade é adicionada ao objeto *em si*. Se já existia uma propriedade com o mesmo nome no protótipo, essa propriedade não afetará mais o objeto, pois agora está oculta atrás da própria propriedade do objeto.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log((new Rabbit("basic")).teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

O diagrama a seguir esboça a situação depois que esse código foi executado. Os protótipos `Rabbit` e `Object` ficam atrás de `killerRabbit` como uma espécie de pano de fundo, onde propriedades que não são encontradas no objeto em si podem ser consultadas.



Sobrescrever propriedades que existem em um protótipo pode ser algo útil. Como o exemplo dos dentes de coelho mostra, a sobrescrita pode ser usada para expressar propriedades excepcionais em instâncias de uma classe mais genérica de objetos, enquanto permite que os objetos não-excepcionais obtenham um valor padrão de seu protótipo.

A sobrescrita também é usada para dar aos protótipos padrão de funções e *arrays* um método `toString` diferente daquele do protótipo básico de objeto.

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Chamar `toString` em um *array* produz um resultado semelhante a chamar `.join(",")` nele — coloca vírgulas entre os valores no *array*. Chamar diretamente `Object.prototype.toString` com um *array* produz uma *string* diferente. Essa função não sabe sobre *arrays*, então simplesmente coloca a palavra *object* e o nome do tipo entre colchetes.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Maps

Vimos a palavra *map* usada no capítulo anterior para uma operação que transforma uma estrutura de dados aplicando uma função a seus elementos. Por mais confuso que seja, na programação a mesma palavra é usada para algo relacionado, mas bastante diferente.

Um *map* (substantivo) é uma estrutura de dados que associa valores (as chaves) a outros valores. Por exemplo, você pode querer mapear nomes para idades. É possível usar objetos para isso.

```
let ages = {
```

```
  Boris: 39,
  Liang: 22,
  Júlia: 62
};

console.log(`Júlia is ${ages["Júlia"]}`);
// → Júlia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

Aqui, os nomes das propriedades do objeto são os nomes das pessoas e os valores das propriedades são suas idades. Mas certamente não listamos ninguém chamado toString em nosso map. No entanto, como objetos comuns derivam de `Object.prototype`, parece que a propriedade está lá.

Por essa razão, usar objetos comuns como maps é perigoso. Existem várias maneiras possíveis de evitar esse problema. Primeiro, você pode criar objetos com *nenhum* protótipo. Se passar `null` para `Object.create`, o objeto resultante não derivará de `Object.prototype` e poderá ser usado com segurança como um map.

```
console.log("toString" in Object.create(null));
// → false
```

Nomes de propriedades de objetos devem ser *strings*. Se você precisar de um map cujas chaves não possam ser facilmente convertidas em *strings* — como objetos — não pode usar um objeto como seu map.

Felizmente, JavaScript vem com uma classe chamada `Map` que é escrita exatamente para esse propósito. Ela armazena um mapeamento e permite qualquer tipo de chave.

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);

console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

Os métodos `set`, `get` e `has` fazem parte da interface do objeto `Map`. Escrever

uma estrutura de dados que possa rapidamente atualizar e pesquisar em um grande conjunto de valores não é fácil, mas não precisamos nos preocupar com isso. Alguém já fez isso por nós, e podemos usar essa interface simples para utilizar o trabalho dessa pessoa.

Se você tiver um objeto comum que precise tratar como map por algum motivo, é útil saber que `Object.keys` retorna apenas as chaves *próprias* de um objeto, não aquelas do protótipo. Como alternativa ao operador `in`, você pode usar a função `Object.hasOwn`, que ignora o protótipo do objeto.

```
console.log(Object.hasOwn({x: 1}, "x"));
// → true
console.log(Object.hasOwn({x: 1}, "toString"));
// → false
```

Polimorfismo

Quando você chama a função `String` (que converte um valor em *string*) em um objeto, ela chama o método `toString` nesse objeto para tentar criar uma *string* significativa a partir dele. Mencionei que alguns dos protótipos padrão definem sua própria versão de `toString` para que possam criar uma *string* que contenha informações mais úteis do que `"[object Object]"`. Você também pode fazer isso.

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};

console.log(String(killerRabbit));
// → a killer rabbit
```

Este é um exemplo simples de uma ideia poderosa. Quando um trecho de código é escrito para trabalhar com objetos que possuem uma certa interface — neste caso, um método `toString` — qualquer tipo de objeto que suporte essa interface pode ser encaixado no código e funcionará com ele.

Essa técnica é chamada de *polimorfismo*. Código polimórfico pode trabalhar com valores de diferentes formas, desde que suportem a interface que ele espera.

Um exemplo de uma interface amplamente usada é a de objetos semelhantes a arrays que possuem uma propriedade `length` contendo um número e propriedades numeradas para cada um de seus elementos. Tanto *arrays* quanto *strings* suportam essa interface, assim como vários outros objetos, alguns dos quais veremos mais adiante nos capítulos sobre o *browser*. Nossa implemen-

tação de `forEach` do Capítulo 5 funciona em qualquer coisa que forneça essa interface. Na verdade, `Array.prototype.forEach` também funciona.

```
Array.prototype.forEach.call({
  length: 2,
  0: "A",
  1: "B"
}, elt => console.log(elt));
// → A
// → B
```

Getters, setters e estáticos

Interfaces frequentemente contêm propriedades simples, não apenas métodos. Por exemplo, objetos `Map` possuem uma propriedade `size` que informa quantas chaves estão armazenadas neles.

Não é necessário que tal objeto compute e armazene essa propriedade diretamente na instância. Até propriedades que são acessadas diretamente podem esconder uma chamada de método. Esses métodos são chamados de *getters* e são definidos escrevendo `get` na frente do nome do método em uma expressão de objeto ou declaração de classe.

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};

console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

Sempre que alguém lê a propriedade `size` deste objeto, o método associado é chamado. Você pode fazer algo semelhante quando uma propriedade é escrita, usando um *setter*.

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
```

```
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

A classe `Temperature` permite que você leia e escreva a temperatura em graus Celsius ou graus Fahrenheit, mas internamente armazena apenas Celsius e automaticamente converte de e para Celsius no *getter* e *setter* de `fahrenheit`.

Às vezes você quer anexar algumas propriedades diretamente à sua função construtora em vez de ao protótipo. Esses métodos não terão acesso a uma instância de classe, mas podem, por exemplo, ser usados para fornecer maneiras adicionais de criar instâncias.

Dentro de uma declaração de classe, métodos ou propriedades que têm `static` escrito antes de seu nome são armazenados no construtor. Por exemplo, a classe `Temperature` permite que você escreva `Temperature.fromFahrenheit(100)` para criar uma temperatura usando graus Fahrenheit.

```
let boil = Temperature.fromFahrenheit(212);
console.log(boil.celsius);
// → 100
```

Symbols

Mencionei no Capítulo 4 que um *loop* `for/of` pode iterar sobre vários tipos de estruturas de dados. Este é outro caso de polimorfismo — esses *loops* esperam que a estrutura de dados exponha uma interface específica, o que *arrays* e *strings* fazem. E podemos também adicionar essa interface aos nossos próprios objetos! Mas antes de podermos fazer isso, precisamos dar uma breve olhada no tipo *symbol*.

É possível que múltiplas interfaces usem o mesmo nome de propriedade para coisas diferentes. Por exemplo, em objetos semelhantes a *arrays*, `length` se

refere ao número de elementos na coleção. Mas uma interface de objeto descrevendo uma rota de caminhada poderia usar `length` para fornecer o comprimento da rota em metros. Não seria possível para um objeto se conformar a ambas as interfaces.

Um objeto tentando ser uma rota e semelhante a um *array* (talvez para enumerar seus pontos de passagem) é um tanto rebuscado, e esse tipo de problema não é tão comum na prática. Para coisas como o protocolo de iteração, porém, os designers da linguagem precisavam de um tipo de propriedade que *realmente* não conflitasse com nenhuma outra. Então, em 2015, *symbols* foram adicionados à linguagem.

A maioria das propriedades, incluindo todas as que vimos até agora, são nomeadas com *strings*. Mas também é possível usar *symbols* como nomes de propriedades. *Symbols* são valores criados com a função `Symbol`. Diferentemente de *strings*, *symbols* recém-criados são únicos — você não pode criar o mesmo *symbol* duas vezes.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(killerRabbit[sym]);
// → 55
```

A *string* que você passa para `Symbol` é incluída quando você o converte para *string* e pode facilitar o reconhecimento de um *symbol* quando, por exemplo, o mostra no console. Mas ela não tem significado além disso — múltiplos *symbols* podem ter o mesmo nome.

Ser tanto único quanto utilizável como nome de propriedade torna os *symbols* adequados para definir interfaces que podem coexistir pacificamente com outras propriedades, independentemente de seus nomes.

```
const length = Symbol("length");
Array.prototype[length] = 0;

console.log([1, 2].length);
// → 2
console.log([1, 2][length]);
// → 0
```

É possível incluir propriedades *symbol* em expressões de objetos e classes usando colchetes ao redor do nome da propriedade. Isso faz com que a expressão entre os colchetes seja avaliada para produzir o nome da propriedade, de forma análoga à notação de acesso a propriedade com colchetes.

```
let myTrip = {
  length: 2,
  0: "Lankwitz",
  1: "Babelsberg",
  [length]: 21500
};
console.log(myTrip[length], myTrip.length);
// → 21500 2
```

A interface de iteração

O objeto passado a um *loop* `for/of` deve ser *iterável*. Isso significa que ele tem um método nomeado com o *symbol* `Symbol.iterator` (um valor *symbol* definido pela linguagem, armazenado como uma propriedade da função `Symbol`).

Quando chamado, esse método deve retornar um objeto que fornece uma segunda interface, o *iterador*. Este é o que realmente itera. Ele tem um método `next` que retorna o próximo resultado. Esse resultado deve ser um objeto com uma propriedade `value` que fornece o próximo valor, se houver um, e uma propriedade `done`, que deve ser `true` quando não houver mais resultados e `false` caso contrário.

Note que os nomes das propriedades `next`, `value` e `done` são *strings* comuns, não *symbols*. Apenas `Symbol.iterator`, que provavelmente será adicionado a *muitos* objetos diferentes, é um *symbol* de verdade.

Podemos usar essa interface nós mesmos diretamente.

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

Vamos implementar uma estrutura de dados iterável semelhante à lista encadeada do exercício no Capítulo 4. Desta vez escreveremos a lista como uma classe.

```
class List {
  constructor(value, rest) {
    this.value = value;
    this.rest = rest;
  }
```

```
  get length() {
    return 1 + (this.rest ? this.rest.length : 0);
  }

  static fromArray(array) {
    let result = null;
    for (let i = array.length - 1; i >= 0; i--) {
      result = new this(array[i], result);
    }
    return result;
  }
}
```

Note que `this`, em um método estático, aponta para o construtor da classe, não para uma instância — não existe instância quando um método estático é chamado.

Iterar sobre uma lista deve retornar todos os elementos da lista do início ao fim. Escreveremos uma classe separada para o iterador.

```
class ListIterator {
  constructor(list) {
    this.list = list;
  }

  next() {
    if (this.list == null) {
      return {done: true};
    }
    let value = this.list.value;
    this.list = this.list.rest;
    return {value, done: false};
  }
}
```

A classe rastreia o progresso da iteração pela lista atualizando sua propriedade `list` para avançar ao próximo objeto da lista sempre que um valor é retornado, e indica que terminou quando essa lista está vazia (null).

Vamos configurar a classe `List` para ser iterável. Ao longo deste livro, ocasionalmente usarei manipulação de protótipo após o fato para adicionar métodos a classes, para que os trechos individuais de código permaneçam pequenos e autocontidos. Em um programa regular, onde não há necessidade de dividir o código em pequenos pedaços, você declararia esses métodos diretamente na classe.

```
List.prototype[Symbol.iterator] = function() {
```

```
  return new ListIterator(this);
};
```

Agora podemos iterar sobre uma lista com `for/of`.

```
let list = List.fromArray([1, 2, 3]);
for (let element of list) {
  console.log(element);
}
// → 1
// → 2
// → 3
```

A sintaxe `...` em notação de *array* e chamadas de função funciona de forma semelhante com qualquer objeto iterável. Por exemplo, você pode usar `[...value]` para criar um *array* contendo os elementos de um objeto iterável arbitrário.

```
console.log([..."PCI"]);
// → ["P", "C", "I"]
```

Herança

Imagine que precisamos de um tipo de lista muito parecido com a classe `List` que vimos antes, mas como pediremos seu comprimento o tempo todo, não queremos que ela tenha que percorrer seu `rest` a cada vez. Em vez disso, queremos armazenar o comprimento em cada instância para acesso eficiente.

O sistema de protótipos do JavaScript torna possível criar uma *nova* classe, muito parecida com a antiga, mas com novas definições para algumas de suas propriedades. O protótipo da nova classe deriva do antigo protótipo, mas adiciona uma nova definição para, digamos, o *getter* `length`.

Em termos de programação orientada a objetos, isso é chamado de *herança*. A nova classe herda propriedades e comportamento da antiga classe.

```
class LengthList extends List {
  #length;

  constructor(value, rest) {
    super(value, rest);
    this.#length = super.length;
  }

  get length() {
```

```
    return this.#length;
  }
}

console.log(LengthList.fromArray([1, 2, 3]).length);
// → 3
```

O uso da palavra `extends` indica que esta classe não deve ser baseada diretamente no protótipo padrão `Object`, mas em alguma outra classe. Esta é chamada de *superclasse*. A classe derivada é a *subclasse*.

Para inicializar uma instância de `LengthList`, o construtor chama o construtor de sua superclasse através da palavra-chave `super`. Isso é necessário porque, se esse novo objeto deve se comportar (aproximadamente) como uma `List`, ele vai precisar das propriedades de instância que listas possuem.

O construtor então armazena o comprimento da lista em uma propriedade privada. Se tivéssemos escrito `this.length` ali, o próprio *getter* da classe teria sido chamado, o que não funciona ainda, pois `#length` ainda não foi preenchido. Podemos usar `super.something` para chamar métodos e *getters* no protótipo da superclasse, o que frequentemente é útil.

A herança nos permite construir tipos de dados levemente diferentes a partir de tipos de dados existentes com relativamente pouco trabalho. Ela é uma parte fundamental da tradição orientada a objetos, junto com o encapsulamento e o polimorfismo. Mas enquanto os dois últimos são agora geralmente considerados ideias maravilhosas, a herança é mais controversa.

Enquanto o encapsulamento e o polimorfismo podem ser usados para *separar* partes do código umas das outras, reduzindo o emaranhamento geral do programa, a herança fundamentalmente amarra classes entre si, criando *mais* emaranhamento. Quando se herda de uma classe, geralmente é preciso saber mais sobre como ela funciona do que quando simplesmente a usa. A herança pode ser uma ferramenta útil para tornar alguns tipos de programas mais sucintos, mas não deveria ser a primeira ferramenta a que você recorre, e provavelmente não deveria procurar ativamente por oportunidades de construir hierarquias de classes (árvores genealógicas de classes).

O operador instanceof

Ocasionalmente é útil saber se um objeto foi derivado de uma classe específica. Para isso, JavaScript fornece um operador binário chamado `instanceof`.

```
console.log(
  new LengthList(1, null) instanceof LengthList);
```

```
// → true
console.log(new LengthList(2, null) instanceof List);
// → true
console.log(new List(3, null) instanceof LengthList);
// → false
console.log([1] instanceof Array);
// → true
```

O operador enxerga através dos tipos herdados, então uma `LengthList` é uma instância de `List`. O operador também pode ser aplicado a construtores padrão como `Array`. Quase todo objeto é uma instância de `Object`.

Resumo

Objetos fazem mais do que apenas armazenar suas próprias propriedades. Eles possuem protótipos, que são outros objetos. Eles agirão como se tivessem propriedades que não possuem, desde que seu protótipo tenha essa propriedade. Objetos simples têm `Object.prototype` como seu protótipo.

Construtores, que são funções cujos nomes geralmente começam com letra maiúscula, podem ser usados com o operador `new` para criar novos objetos. O protótipo do novo objeto será o objeto encontrado na propriedade `prototype` do construtor. Você pode fazer bom uso disso colocando as propriedades que todos os valores de um dado tipo compartilham em seu protótipo. Existe uma notação `class` que fornece uma maneira clara de definir um construtor e seu protótipo.

Você pode definir *getters* e *setters* para chamar métodos secretamente toda vez que uma propriedade de um objeto é acessada. Métodos estáticos são métodos armazenados no construtor de uma classe em vez de em seu protótipo.

O operador `instanceof` pode, dado um objeto e um construtor, dizer se aquele objeto é uma instância daquele construtor.

Uma coisa útil a fazer com objetos é especificar uma interface para eles e dizer a todos que devem se comunicar com seu objeto apenas através dessa interface. O restante dos detalhes que compõem seu objeto são agora *encapsulados*, ocultos atrás da interface. Você pode usar propriedades privadas para esconder uma parte de seu objeto do mundo exterior.

Mais de um tipo pode implementar a mesma interface. Código escrito para usar uma interface automaticamente sabe como trabalhar com qualquer número de objetos diferentes que forneçam a interface. Isso é chamado de *polimorfismo*.

Quando se implementam múltiplas classes que diferem apenas em alguns detalhes, pode ser útil escrever as novas classes como *subclasses* de uma classe existente, *herdando* parte de seu comportamento.

Exercícios

Um tipo vetor

Escreva uma classe `Vec` que represente um vetor em espaço bidimensional. Ela recebe parâmetros `x` e `y` (números), que salva em propriedades de mesmo nome.

Dê ao protótipo de `Vec` dois métodos, `plus` e `minus`, que recebem outro vetor como parâmetro e retornam um novo vetor que tem a soma ou diferença dos valores x e y dos dois vetores (`this` e o parâmetro).

Adicione uma propriedade getter `length` ao protótipo que calcula o comprimento do vetor — ou seja, a distância do ponto (x, y) até a origem (0, 0).

Grupos

O ambiente padrão do JavaScript fornece outra estrutura de dados chamada `Set`. Como uma instância de `Map`, um conjunto armazena uma coleção de valores. Diferentemente de `Map`, ele não associa outros valores a esses — ele apenas rastreia quais valores fazem parte do conjunto. Um valor pode fazer parte de um conjunto apenas uma vez — adicioná-lo novamente não tem efeito.

Escreva uma classe chamada `Group` (já que `Set` já está em uso). Como `Set`, ela tem métodos `add`, `delete` e `has`. Seu construtor cria um grupo vazio, `add` adiciona um valor ao grupo (mas apenas se ele já não for um membro), `delete` remove seu argumento do grupo (se era um membro) e `has` retorna um valor booleano indicando se seu argumento é um membro do grupo.

Use o operador `===`, ou algo equivalente como `indexOf`, para determinar se dois valores são iguais.

Dê à classe um método estático `from` que recebe um objeto iterável como argumento e cria um grupo que contém todos os valores produzidos pela iteração sobre ele.

Grupos iteráveis

Torne a classe `Group` do exercício anterior iterável. Consulte a seção sobre a interface de iteração mais cedo neste capítulo se não tiver clareza sobre a forma exata da interface.

Se você usou um *array* para representar os membros do grupo, não retorne simplesmente o iterador criado chamando o método `Symbol.iterator` no *array*. Isso funcionaria, mas frustra o propósito deste exercício.

Não há problema se o seu iterador se comportar de forma estranha quando o grupo é modificado durante a iteração.

“A questão de se Máquinas Podem Pensar [...] é tão relevante quanto a questão de se Submarinos Podem Nadar.”

—Edsger Dijkstra, The Threats to Computing Science

Chapter 7

Projeto: Um Robô

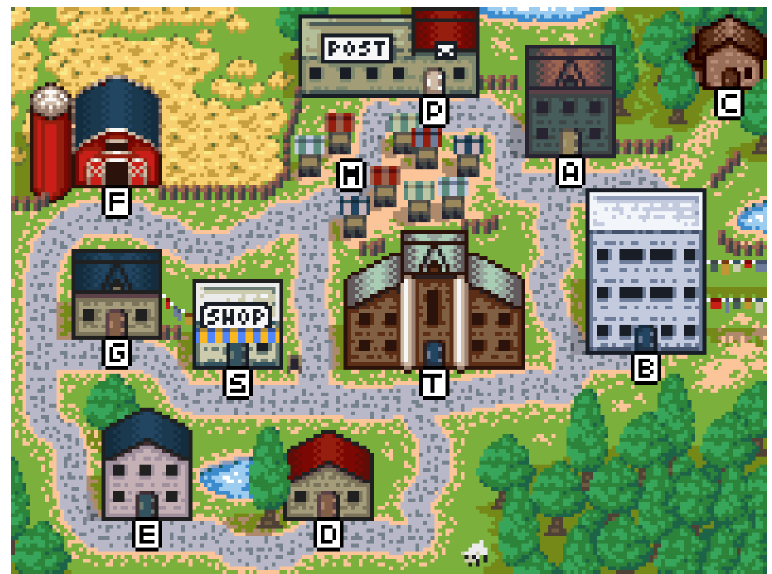
Nos capítulos de “projeto”, vou parar de bombardeá-lo com teoria nova por um breve momento e, em vez disso, trabalharemos em um programa juntos. A teoria é necessária para aprender a programar, mas ler e entender programas reais é igualmente importante.

Nosso projeto neste capítulo é construir um autômato, um pequeno programa que realiza uma tarefa em um mundo virtual. Nosso autômato será um robô de entrega de correspondências que pega e entrega encomendas.

Meadowfield

A vila de Meadowfield não é muito grande. Ela consiste em 11 lugares com 14 estradas entre eles. Pode ser descrita com este *array* de estradas:

```
const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```



A rede de estradas na vila forma um *grafo*. Um grafo é uma coleção de pontos (lugares na vila) com linhas entre eles (estradas). Esse grafo será o mundo por onde nosso robô se move.

O *array* de *strings* não é muito fácil de trabalhar. O que nos interessa são os destinos que podemos alcançar a partir de um dado lugar. Vamos converter a lista de estradas em uma estrutura de dados que, para cada lugar, nos diga o que pode ser alcançado a partir dali.

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (from in graph) {
      graph[from].push(to);
    } else {
      graph[from] = [to];
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

Dado um *array* de arestas, `buildGraph` cria um objeto map que, para cada nó, armazena um *array* de nós conectados. Ele usa o método `split` para ir das

strings de estrada — que possuem a forma `"Início-Fim"` — para *arrays* de dois elementos contendo o início e o fim como *strings* separadas.

A tarefa

Nosso robô estará se movendo pela vila. Há encomendas em vários lugares, cada uma endereçada a algum outro lugar. O robô pega encomendas quando as encontra e as entrega quando chega ao destino delas.

O autômato deve decidir, a cada ponto, para onde ir em seguida. Ele termina sua tarefa quando todas as encomendas foram entregues.

Para podermos simular esse processo, precisamos definir um mundo virtual que possa descrevê-lo. Esse modelo nos diz onde o robô está e onde estão as encomendas. Quando o robô decide se mover para algum lugar, precisamos atualizar o modelo para refletir a nova situação.

Se você está pensando em termos de programação orientada a objetos, seu primeiro impulso pode ser começar a definir objetos para os vários elementos do mundo: uma classe para o robô, uma para uma encomenda, talvez uma para lugares. Esses poderiam então armazenar propriedades que descrevem seu estado atual, como a pilha de encomendas em um local, que poderíamos alterar ao atualizar o mundo.

Isso está errado. Pelo menos, geralmente está. O fato de que algo parece um objeto não significa automaticamente que deve ser um objeto no seu programa. Escrever classes reflexivamente para cada conceito em sua aplicação tende a deixá-lo com uma coleção de objetos interconectados que possuem seu próprio estado interno mutável. Tais programas são frequentemente difíceis de entender e, portanto, fáceis de quebrar.

Em vez disso, vamos condensar o estado da vila no conjunto mínimo de valores que o define. Há a localização atual do robô e a coleção de encomendas não entregues, cada uma com uma localização atual e um endereço de destino. É isso.

E enquanto estamos nisso, vamos fazer de modo que não *alteremos* esse estado quando o robô se move, mas sim calculemos um *novo* estado para a situação após o movimento.

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
```

```
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

O método `move` é onde a ação acontece. Ele primeiro verifica se há uma estrada indo do lugar atual ao destino e, se não houver, retorna o estado antigo, pois esse não é um movimento válido.

Em seguida, o método cria um novo estado com o destino como o novo lugar do robô. Ele também precisa criar um novo conjunto de encomendas — encomendas que o robô está carregando (que estão no lugar atual do robô) precisam ser movidas para o novo lugar. E encomendas endereçadas ao novo lugar precisam ser entregues — ou seja, precisam ser removidas do conjunto de encomendas não entregues. A chamada a `map` cuida do movimento, e a chamada a `filter` faz a entrega.

Objetos de encomenda não são alterados quando movidos, mas sim recriados. O método `move` nos dá um novo estado de vila, mas deixa o antigo completamente intacto.

```
let first = new VillageState(
  "Post Office",
  [{place: "Post Office", address: "Alice's House"}]
);
let next = first.move("Alice's House");

console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

O movimento faz com que a encomenda seja entregue, o que se reflete no próximo estado. Mas o estado inicial ainda descreve a situação onde o robô está no correio e a encomenda não foi entregue.

Dados persistentes

Estruturas de dados que não mudam são chamadas de *imutáveis* ou *persistentes*. Elas se comportam de maneira semelhante a *strings* e números, no sentido de que são o que são e permanecem assim, em vez de conter coisas diferentes em momentos diferentes.

Em JavaScript, praticamente tudo *pode* ser alterado, então trabalhar com valores que devem ser persistentes requer alguma contenção. Existe uma função chamada `Object.freeze` que altera um objeto de modo que escrever em suas propriedades seja ignorado. Você poderia usá-la para garantir que seus objetos não sejam alterados, se quiser ser cuidadoso. O congelamento exige que o computador faça algum trabalho extra, e ter atualizações ignoradas é tão provável de confundir alguém quanto tê-las fazendo a coisa errada. Eu geralmente prefiro simplesmente dizer às pessoas que um dado objeto não deve ser mexido e esperar que elas se lembrem disso.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

Por que estou me esforçando tanto para não alterar objetos quando a linguagem está obviamente esperando que eu faça isso? Porque isso me ajuda a entender meus programas. Isto é sobre gerenciamento de complexidade novamente. Quando os objetos no meu sistema são coisas fixas e estáveis, posso considerar operações sobre eles isoladamente — mover para a casa de Alice a partir de um dado estado inicial sempre produz o mesmo novo estado. Quando objetos mudam ao longo do tempo, isso adiciona toda uma nova dimensão de complexidade a esse tipo de raciocínio.

Para um sistema pequeno como o que estamos construindo neste capítulo, poderíamos lidar com esse pouco de complexidade extra. Mas o limite mais importante sobre que tipo de sistemas podemos construir é quanto conseguimos entender. Qualquer coisa que torne seu código mais fácil de entender torna possível construir um sistema mais ambicioso.

Infelizmente, embora entender um sistema construído sobre estruturas de dados persistentes seja mais fácil, *projetar* um, especialmente quando sua linguagem de programação não está ajudando, pode ser um pouco mais difícil. Procuraremos oportunidades de usar estruturas de dados persistentes neste livro, mas também usaremos as mutáveis.

Simulação

Um robô de entrega olha para o mundo e decide em qual direção quer se mover. Portanto, poderíamos dizer que um robô é uma função que recebe um objeto `VillageState` e retorna o nome de um lugar próximo.

Como queremos que robôs possam lembrar coisas para fazer e executar planos, também passamos a eles sua memória e permitimos que retornem uma nova memória. Assim, o que um robô retorna é um objeto contendo tanto a direção para a qual quer se mover quanto um valor de memória que será devolvido a ele na próxima vez que for chamado.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

Considere o que um robô precisa fazer para "resolver" um dado estado. Ele deve pegar todas as encomendas visitando cada local que tem uma encomenda e entregá-las visitando cada local para o qual uma encomenda é endereçada, mas apenas depois de pegar a encomenda.

Qual é a estratégia mais burra que poderia funcionar? O robô poderia simplesmente andar em uma direção aleatória a cada turno. Isso significa que, com grande probabilidade, ele eventualmente encontrará todas as encomendas e, em algum momento, alcançará o lugar onde elas devem ser entregues.

Veja como isso poderia parecer:

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

Lembre-se de que `Math.random()` retorna um número entre 0 e 1 — mas sempre

abaixo de 1. Multiplicar esse número pelo comprimento de um *array* e depois aplicar `Math.floor` nos dá um índice aleatório para o *array*.

Como este robô não precisa lembrar de nada, ele ignora seu segundo argumento (lembre-se de que funções JavaScript podem ser chamadas com argumentos extras sem problemas) e omite a propriedade `memory` em seu objeto retornado.

Para colocar esse robô sofisticado para trabalhar, primeiro precisaremos de uma maneira de criar um novo estado com algumas encomendas. Um método estático (escrito aqui adicionando diretamente uma propriedade ao construtor) é um bom lugar para colocar essa funcionalidade.

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

Não queremos que nenhuma encomenda seja enviada do mesmo lugar para o qual é endereçada. Por essa razão, o *loop* `do` continua escolhendo novos lugares quando obtém um que é igual ao endereço.

Vamos iniciar um mundo virtual.

```
runRobot(VillageState.random(), randomRobot);
// → Moved to Marketplace
// → Moved to Town Hall
// →…
// → Done in 63 turns
```

O robô leva muitos turnos para entregar as encomendas porque não está planejando com antecedência. Vamos abordar isso em breve.

A rota do caminhão de correspondência

Deveríamos ser capazes de fazer muito melhor do que o robô aleatório. Uma melhoria fácil seria se inspirar na maneira como a entrega de correspondência do mundo real funciona. Se encontrarmos uma rota que passe por todos os

lugares da vila, o robô poderia percorrer essa rota duas vezes, e nesse ponto estaria garantido que terminou. Aqui está uma dessas rotas (começando pelo correio):

```
const mailRoute = [
  "Alice's House", "Cabin", "Alice's House", "Bob's House",
  "Town Hall", "Daria's House", "Ernie's House",
  "Grete's House", "Shop", "Grete's House", "Farm",
  "Marketplace", "Post Office"
];
```

Para implementar o robô seguidor de rota, precisaremos usar a memória do robô. O robô mantém o restante de sua rota na memória e descarta o primeiro elemento a cada turno.

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
  return {direction: memory[0], memory: memory.slice(1)};
}
```

Este robô já é muito mais rápido. Ele levará no máximo 26 turnos (duas vezes a rota de 13 passos), mas geralmente menos.

Busca de caminho

Ainda assim, eu não chamaria realmente de comportamento inteligente seguir cegamente uma rota fixa. O robô poderia trabalhar mais eficientemente se ajustasse seu comportamento ao trabalho real que precisa ser feito.

Para isso, ele precisa ser capaz de se mover deliberadamente em direção a uma dada encomenda ou em direção ao local onde uma encomenda precisa ser entregue. Fazer isso, mesmo quando o objetivo está a mais de um movimento de distância, exigirá algum tipo de função de busca de caminho.

O problema de encontrar uma rota através de um grafo é um típico *problema de busca*. Podemos dizer se uma dada solução (uma rota) é válida, mas não podemos calcular diretamente a solução como faríamos para 2 + 2. Em vez disso, temos que continuar criando soluções potenciais até encontrar uma que funcione.

O número de rotas possíveis através de um grafo é infinito. Mas ao procurar uma rota de A para B, estamos interessados apenas naquelas que começam em A. Também não nos importamos com rotas que visitam o mesmo lugar duas

vezes — essas definitivamente não são a rota mais eficiente para lugar nenhum. Isso reduz o número de rotas que o buscador precisa considerar.

Na verdade, como estamos interessados principalmente na rota *mais curta*, queremos ter certeza de que olhamos para rotas curtas antes de olhar para as mais longas. Uma boa abordagem seria "crescer" rotas a partir do ponto de partida, explorando cada lugar acessível que ainda não foi visitado, até que uma rota alcance o objetivo. Dessa forma, exploraremos apenas rotas potencialmente interessantes, e sabemos que a primeira rota que encontrarmos é a mais curta (ou uma das mais curtas, se houver mais de uma).

Aqui está uma função que faz isso:

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}
```

A exploração precisa ser feita na ordem correta — os lugares que foram alcançados primeiro precisam ser explorados primeiro. Não podemos explorar imediatamente um lugar assim que o alcançamos, porque isso significaria que lugares alcançados *a partir de lá* também seriam explorados imediatamente, e assim por diante, mesmo que possam existir outros caminhos mais curtos que ainda não foram explorados.

Portanto, a função mantém uma *lista de trabalho*. Este é um *array* de lugares que devem ser explorados em seguida, juntamente com a rota que nos levou até lá. Ela começa apenas com a posição inicial e uma rota vazia.

A busca então opera pegando o próximo item da lista e explorando-o, o que significa que examina todas as estradas saindo daquele lugar. Se uma delas é o objetivo, uma rota finalizada pode ser retornada. Caso contrário, se ainda não olhamos para esse lugar, um novo item é adicionado à lista. Se já olhamos para ele antes, como estamos olhando para rotas curtas primeiro, encontramos uma rota mais longa para aquele lugar ou uma exatamente tão longa quanto a existente, e não precisamos explorá-lo.

Você pode visualizar isso como uma teia de rotas conhecidas se espalhando a partir do local de início, crescendo uniformemente por todos os lados (mas

nunca se emaranhando consigo mesma). Assim que o primeiro fio alcança o local de destino, esse fio é rastreado de volta ao início, nos dando nossa rota.

Nosso código não lida com a situação em que não há mais itens de trabalho na lista de trabalho porque sabemos que nosso grafo é *conectado*, o que significa que cada local pode ser alcançado a partir de todos os outros locais. Sempre seremos capazes de encontrar uma rota entre dois pontos, e a busca não pode falhar.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

Este robô usa seu valor de memória como uma lista de direções para se mover, assim como o robô seguidor de rota. Sempre que essa lista está vazia, ele precisa descobrir o que fazer em seguida. Ele pega a primeira encomenda não entregue do conjunto e, se essa encomenda ainda não foi coletada, traça uma rota até ela. Se a encomenda *já* foi coletada, ela ainda precisa ser entregue, então o robô cria uma rota até o endereço de entrega.

Este robô geralmente termina a tarefa de entregar 5 encomendas em cerca de 16 turnos. Isso é um pouco melhor que `routeRobot`, mas ainda definitivamente não é ótimo. Continuaremos refinando-o nos exercícios.

Exercícios

Medindo um robô

É difícil comparar robôs objetivamente apenas deixando-os resolver alguns cenários. Talvez um robô tenha recebido tarefas mais fáceis ou o tipo de tarefa em que é bom, enquanto o outro não.

Escreva uma função `compareRobots` que recebe dois robôs (e suas memórias iniciais). Ela deve gerar 100 tarefas e deixar ambos os robôs resolverem cada uma dessas tarefas. Quando terminar, deve exibir o número médio de passos que cada robô levou por tarefa.

Para fins de justiça, certifique-se de dar cada tarefa a ambos os robôs, em

vez de gerar tarefas diferentes por robô.

Eficiência do robô

Você consegue escrever um robô que termine a tarefa de entrega mais rápido que `goalOrientedRobot`? Se observar o comportamento desse robô, quais coisas obviamente estúpidas ele faz? Como elas poderiam ser melhoradas?

Se você resolveu o exercício anterior, pode usar sua função `compareRobots` para verificar se melhorou o robô.

Grupo persistente

A maioria das estruturas de dados fornecidas no ambiente padrão do JavaScript não é muito adequada para uso persistente. *Arrays* têm os métodos `slice` e `concat`, que nos permitem criar novos *arrays* facilmente sem danificar o antigo. Mas `Set`, por exemplo, não tem métodos para criar um novo conjunto com um item adicionado ou removido.

Escreva uma nova classe `PGroup`, semelhante à classe `Group` do Capítulo 6, que armazena um conjunto de valores. Como `Group`, ela tem métodos `add`, `delete` e `has`. Seu método `add`, porém, deve retornar uma *nova* instância de `PGroup` com o membro dado adicionado e deixar a antiga inalterada. Da mesma forma, `delete` deve criar uma nova instância sem um dado membro.

A classe deve funcionar para valores de qualquer tipo, não apenas *strings*. Ela *não* precisa ser eficiente quando usada com grandes números de valores.

O construtor não deve fazer parte da interface da classe (embora você definitivamente vá querer usá-lo internamente). Em vez disso, existe uma instância vazia, `PGroup.empty`, que pode ser usada como valor inicial.

Por que você precisa de apenas um valor `PGroup.empty` em vez de ter uma função que cria um novo map vazio toda vez?

> *"Depurar é duas vezes mais difícil do que escrever o código. Portanto, se você escreve o código da maneira mais inteligente possível, você é, por definição, não inteligente o suficiente para depurá-lo."*
>
> —Brian Kernighan and P.J. Plauger, The Elements of Programming Style

Chapter 8

Bugs e Erros

Falhas em programas de computador são geralmente chamadas de *bugs*. Faz os programadores se sentirem bem imaginar que são coisinhas que simplesmente se esgueiram para dentro do nosso trabalho. Na realidade, é claro, somos nós que as colocamos lá.

Se um programa é pensamento cristalizado, podemos categorizar *bugs* grosseiramente entre aqueles causados por pensamentos confusos e aqueles causados por erros introduzidos ao converter um pensamento em código. O primeiro tipo é geralmente mais difícil de diagnosticar e corrigir do que o segundo.

Linguagem

Muitos erros poderiam ser apontados automaticamente pelo computador se ele soubesse o suficiente sobre o que estamos tentando fazer. Mas aqui, a flexibilidade do JavaScript é um empecilho. Seu conceito de *bindings* e propriedades é vago o suficiente para que raramente detecte erros de digitação antes de realmente executar o programa. Mesmo então, ele permite que você faça algumas coisas claramente absurdas sem reclamar, como calcular `true * "monkey"`.

Há algumas coisas sobre as quais o JavaScript reclama. Escrever um programa que não segue a gramática da linguagem fará o computador reclamar imediatamente. Outras coisas, como chamar algo que não é uma função ou procurar uma propriedade em um valor undefined, causarão um erro quando o programa tentar executar a ação.

Frequentemente, porém, seu cálculo sem sentido simplesmente produzirá `NaN` (não é um número) ou um valor undefined, enquanto o programa alegremente continua, convencido de que está fazendo algo significativo. O erro se manifestará apenas mais tarde, depois que o valor falso tiver viajado por várias funções. Ele pode não gerar nenhum erro, mas silenciosamente causar uma saída errada do programa. Encontrar a fonte desses problemas pode ser difícil.

O processo de encontrar erros — *bugs* — em programas é chamado de *depuração*.

Modo estrito

O JavaScript pode ser tornado um *pouco* mais estrito habilitando o *modo estrito*. Isso pode ser feito colocando a *string* `"use strict"` no topo de um arquivo ou corpo de função. Aqui está um exemplo:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Happy happy");
  }
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

Código dentro de classes e módulos (que discutiremos no Capítulo 10) é automaticamente estrito. O antigo comportamento não-estrito ainda existe apenas porque algum código antigo pode depender dele, e os designers da linguagem trabalham duro para evitar quebrar quaisquer programas existentes.

Normalmente, quando você esquece de colocar `let` na frente de sua *binding*, como com `counter` no exemplo, o JavaScript silenciosamente cria uma *binding* global e a usa. No modo estrito, um erro é reportado em vez disso. Isso é muito útil. Deve-se notar, porém, que isso não funciona quando a *binding* em questão já existe em algum lugar no escopo. Nesse caso, o *loop* ainda sobrescreverá silenciosamente o valor da *binding*.

Outra mudança no modo estrito é que a *binding* `this` mantém o valor `undefined` em funções que não são chamadas como métodos. Ao fazer tal chamada fora do modo estrito, `this` se refere ao objeto de escopo global, que é um objeto cujas propriedades são as *bindings* globais. Então, se você acidentalmente chamar um método ou construtor incorretamente no modo estrito, o JavaScript produzirá um erro assim que tentar ler algo de `this`, em vez de alegremente escrever no escopo global.

Por exemplo, considere o código a seguir, que chama uma função construtora sem a palavra-chave `new` de modo que seu `this` *não* se referirá a um objeto recém-construído:

```
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // ops
console.log(name);
// → Ferdinand
```

A chamada falsa a `Person` teve sucesso, mas retornou um valor undefined e

criou a *binding* global `name`. No modo estrito, o resultado é diferente.

```
"use strict";
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // esqueceu new
// → TypeError: Cannot set property 'name' of undefined
```

Somos imediatamente informados de que algo está errado. Isso é útil.

Felizmente, construtores criados com a notação `class` sempre reclamam se são chamados sem `new`, tornando isso menos problemático mesmo no modo não-estrito.

O modo estrito faz mais algumas coisas. Ele proíbe dar a uma função múltiplos parâmetros com o mesmo nome e remove certas funcionalidades problemáticas da linguagem inteiramente (como a declaração `with`, que é tão errada que não é discutida mais neste livro).

Em resumo, colocar `"use strict"` no topo do seu programa raramente prejudica e pode ajudá-lo a identificar um problema.

Tipos

Algumas linguagens querem saber os tipos de todas as suas *bindings* e expressões antes mesmo de executar o programa. Elas dirão imediatamente quando um tipo é usado de maneira inconsistente. O JavaScript considera tipos apenas ao realmente executar o programa e, mesmo assim, frequentemente tenta converter implicitamente valores para o tipo que espera, então não ajuda muito.

Ainda assim, tipos fornecem um *framework* útil para falar sobre programas. Muitos erros vêm de confusão sobre que tipo de valor entra ou sai de uma função. Se você tiver essa informação anotada, é menos provável que fique confuso.

Você poderia adicionar um comentário como o seguinte antes da função `findRoute` do capítulo anterior para descrever seu tipo:

```
// (graph: Object, from: string, to: string) => string[]
function findRoute(graph, from, to) {
  // ...
}
```

Há diversas convenções diferentes para anotar programas JavaScript com tipos.

Uma coisa sobre tipos é que eles precisam introduzir sua própria complexidade para serem capazes de descrever código suficiente para serem úteis. Qual você acha que seria o tipo da função `randomPick` que retorna um elemento

aleatório de um *array*? Você precisaria introduzir uma *variável de tipo*, T, que pode substituir qualquer tipo, para poder dar a `randomPick` um tipo como `(T[])→T` (função de um *array* de Ts para um T).

Quando os tipos de um programa são conhecidos, é possível para o computador *verificá-los* para você, apontando erros antes que o programa seja executado. Existem vários dialetos de JavaScript que adicionam tipos à linguagem e os verificam. O mais popular é chamado TypeScript. Se você tem interesse em adicionar mais rigor aos seus programas, recomendo que experimente.

Neste livro, continuaremos usando código JavaScript cru, perigoso e não-tipado.

Testes

Se a linguagem não vai fazer muito para nos ajudar a encontrar erros, teremos que encontrá-los da maneira difícil: executando o programa e vendo se ele faz a coisa certa.

Fazer isso manualmente, de novo e de novo, é uma péssima ideia. Além de ser irritante, também tende a ser ineficaz, já que leva muito tempo para testar tudo exaustivamente toda vez que você faz uma alteração.

Computadores são bons em tarefas repetitivas, e testar é a tarefa repetitiva ideal. Testes automatizados são o processo de escrever um programa que testa outro programa. Escrever testes dá um pouco mais de trabalho do que testar manualmente, mas uma vez que você o fez, ganha uma espécie de superpoder: leva apenas alguns segundos para verificar que seu programa ainda se comporta corretamente em todas as situações para as quais escreveu testes. Quando você quebra algo, perceberá imediatamente em vez de esbarrar nisso aleatoriamente em algum momento posterior.

Testes geralmente tomam a forma de pequenos programas rotulados que verificam algum aspecto do seu código. Por exemplo, um conjunto de testes para o método `toUpperCase` (padrão, provavelmente já testado por outra pessoa) poderia parecer com isto:

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΊΡΕΤΕ";
```

```
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

Escrever testes assim tende a produzir código bastante repetitivo e estranho. Felizmente, existem softwares que ajudam você a construir e executar coleções de testes (*suítes de teste*) fornecendo uma linguagem (na forma de funções e métodos) adequada para expressar testes e produzindo informações úteis quando um teste falha. Estes são geralmente chamados de *test runners*.

Algum código é mais fácil de testar do que outro. Geralmente, quanto mais objetos externos o código interage, mais difícil é configurar o contexto no qual testá-lo. O estilo de programação mostrado no capítulo anterior, que usa valores persistentes autocontidos em vez de objetos mutáveis, tende a ser fácil de testar.

Depuração

Uma vez que você percebe que há algo errado com seu programa porque ele se comporta mal ou produz erros, o próximo passo é descobrir *qual* é o problema.

Às vezes é óbvio. A mensagem de erro apontará para uma linha específica do seu programa e, se você olhar para a descrição do erro e aquela linha de código, frequentemente poderá ver o problema.

Mas nem sempre. Às vezes, a linha que disparou o problema é simplesmente o primeiro lugar onde um valor problemático produzido em outro lugar é usado de maneira inválida. Se você tem resolvido os exercícios dos capítulos anteriores, provavelmente já experimentou tais situações.

O programa de exemplo a seguir tenta converter um número inteiro em uma *string* em uma dada base (decimal, binário, e assim por diante) repetidamente extraindo o último dígito e depois dividindo o número para se livrar desse dígito. Mas a saída estranha que ele atualmente produz sugere que tem um bug.

```
function numberToString(n, base = 10) {
  let result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
```

```
  return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e…-3181.3
```

Mesmo se você já vê o problema, finja por um momento que não vê. Sabemos que nosso programa está funcionando mal e queremos descobrir por quê.

É aqui que você deve resistir à vontade de começar a fazer mudanças aleatórias no código para ver se isso melhora. Em vez disso, *pense*. Analise o que está acontecendo e elabore uma teoria de por que isso pode estar acontecendo. Então faça observações adicionais para testar essa teoria — ou, se ainda não tiver uma teoria, faça observações adicionais para ajudá-lo a elaborar uma.

Colocar algumas chamadas estratégicas a `console.log` no programa é uma boa maneira de obter informações adicionais sobre o que o programa está fazendo. Neste caso, queremos que `n` assuma os valores `13`, `1` e depois `0`. Vamos imprimir seu valor no início do *loop*.

```
13
1.3
0.13
0.013…

1.5e-323
```

Certo. Dividir 13 por 10 não produz um número inteiro. Em vez de `n /= base`, o que realmente queremos é `n = Math.floor(n / base)` para que o número seja adequadamente “deslocado” para a direita.

Uma alternativa ao uso de `console.log` para espiar o comportamento do programa é usar as capacidades do *depurador* do seu *browser*. *Browsers* vêm com a capacidade de definir um *breakpoint* em uma linha específica do seu código. Quando a execução do programa atinge uma linha com um *breakpoint*, ela é pausada e você pode inspecionar os valores das *bindings* naquele ponto. Não entrarei em detalhes, pois depuradores diferem de *browser* para *browser*, mas procure nas ferramentas de desenvolvedor do seu *browser* ou pesquise na web por instruções.

Outra maneira de definir um *breakpoint* é incluir uma declaração `debugger` (consistindo simplesmente dessa palavra-chave) no seu programa. Se as ferramentas de desenvolvedor do seu *browser* estiverem ativas, o programa será pausado sempre que atingir tal declaração.

Propagação de erros

Nem todos os problemas podem ser prevenidos pelo programador, infelizmente. Se seu programa se comunica com o mundo exterior de alguma forma, é possível receber entrada malformada, ficar sobrecarregado de trabalho ou ter a rede falhando.

Se você está programando apenas para si mesmo, pode se dar ao luxo de simplesmente ignorar tais problemas até que ocorram. Mas se você constrói algo que será usado por qualquer outra pessoa, geralmente quer que o programa faça melhor do que simplesmente travar. Às vezes, a coisa certa a fazer é aceitar a entrada ruim e continuar executando. Em outros casos, é melhor reportar ao usuário o que deu errado e então desistir. Em qualquer situação, o programa precisa ativamente fazer algo em resposta ao problema.

Digamos que você tenha uma função `promptNumber` que pede ao usuário um número e o retorna. O que ela deveria retornar se o usuário digitar "orange"?

Uma opção é fazer com que retorne um valor especial. Escolhas comuns para tais valores são `null`, `undefined` ou `-1`.

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"));
```

Agora qualquer código que chame `promptNumber` deve verificar se um número real foi lido e, caso contrário, deve de alguma forma se recuperar — talvez perguntando novamente ou preenchendo um valor padrão. Ou poderia novamente retornar um valor especial para *seu* chamador para indicar que falhou em fazer o que foi pedido.

Em muitas situações, principalmente quando erros são comuns e o chamador deve explicitamente levá-los em conta, retornar um valor especial é uma boa maneira de indicar um erro. Porém, isso tem suas desvantagens. Primeiro, e se a função já pode retornar todo tipo possível de valor? Em tal função, você terá que fazer algo como envolver o resultado em um objeto para poder distinguir sucesso de falha, como o método `next` na interface do iterador faz.

```
function lastElement(array) {
  if (array.length == 0) {
    return {failed: true};
  } else {
```

```
    return {value: array[array.length - 1]};
  }
}
```

O segundo problema com retornar valores especiais é que pode levar a código estranho. Se um trecho de código chama `promptNumber` 10 vezes, ele precisa verificar 10 vezes se `null` foi retornado. Se sua resposta ao encontrar `null` é simplesmente retornar `null` ele mesmo, os chamadores da função terão por sua vez que verificar, e assim por diante.

Exceções

Quando uma função não pode prosseguir normalmente, o que frequentemente *gostaríamos* de fazer é simplesmente parar o que estamos fazendo e imediatamente pular para um lugar que saiba como lidar com o problema. É isso que o *tratamento de exceções* faz.

Exceções são um mecanismo que torna possível para código que encontra um problema *lançar* (ou *throw*) uma exceção. Uma exceção pode ser qualquer valor. Lançar uma se assemelha a um retorno superpotente de uma função: ela salta para fora não apenas da função atual, mas também de seus chamadores, descendo até a primeira chamada que iniciou a execução atual. Isso é chamado de *desenrolar a pilha*. Você deve se lembrar da pilha de chamadas de função mencionada no Capítulo 3. Uma exceção desce essa pilha, descartando todos os contextos de chamada que encontra.

Se exceções sempre descessem direto até o fundo da pilha, elas não seriam muito úteis. Seriam apenas uma maneira nova de explodir seu programa. Seu poder está no fato de que você pode colocar "obstáculos" ao longo da pilha para *capturar* a exceção enquanto ela desce. Uma vez que você captura uma exceção, pode fazer algo com ela para resolver o problema e então continuar a executar o programa.

Aqui está um exemplo:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L") {
    return "a house";
```

```
  } else {
    return "two angry bears";
  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

A palavra-chave `throw` é usada para lançar uma exceção. A captura é feita envolvendo um trecho de código em um bloco `try`, seguido pela palavra-chave `catch`. Quando o código no bloco `try` causa uma exceção, o bloco `catch` é avaliado, com o nome entre parênteses vinculado ao valor da exceção. Após o bloco `catch` terminar — ou se o bloco `try` terminar sem problemas — o programa continua abaixo de toda a declaração `try/catch`.

Neste caso, usamos o construtor `Error` para criar nosso valor de exceção. Este é um construtor padrão do JavaScript que cria um objeto com uma propriedade `message`. Instâncias de `Error` também coletam informações sobre a pilha de chamadas que existia quando a exceção foi criada, um chamado *rastreamento de pilha*. Essa informação é armazenada na propriedade `stack` e pode ser útil ao tentar depurar um problema: ela nos diz em qual função o problema ocorreu e quais funções fizeram a chamada que falhou.

Note que a função `look` ignora completamente a possibilidade de que `promptDirection` possa dar errado. Esta é a grande vantagem das exceções: o código de tratamento de erros é necessário apenas no ponto onde o erro ocorre e no ponto onde ele é tratado. As funções intermediárias podem esquecer tudo sobre isso.

Bem, quase...

Limpando depois de exceções

O efeito de uma exceção é outro tipo de fluxo de controle. Cada ação que pode causar uma exceção, que é praticamente toda chamada de função e acesso a propriedade, pode fazer com que o controle deixe seu código repentinamente.

Isso significa que quando o código tem vários efeitos colaterais, mesmo que seu fluxo de controle "regular" pareça que todos sempre acontecerão, uma exceção pode impedir que alguns deles ocorram.

Aqui está um código bancário realmente ruim:

```
const accounts = {
  a: 100,
```

```
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!Object.hasOwn(accounts, accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

A função `transfer` transfere uma quantia de dinheiro de uma dada conta para outra, pedindo o nome da outra conta no processo. Se receber um nome de conta inválido, `getAccount` lança uma exceção.

Mas `transfer` *primeiro* remove o dinheiro da conta e *depois* chama `getAccount` antes de adicioná-lo a outra conta. Se for interrompido por uma exceção nesse ponto, o dinheiro simplesmente desaparecerá.

Esse código poderia ter sido escrito de forma um pouco mais inteligente, por exemplo chamando `getAccount` antes de começar a mover dinheiro. Mas frequentemente problemas assim ocorrem de formas mais sutis. Até funções que não parecem que vão lançar uma exceção podem fazê-lo em circunstâncias excepcionais ou quando contêm um erro do programador.

Uma maneira de abordar isso é usar menos efeitos colaterais. Novamente, um estilo de programação que computa novos valores em vez de alterar dados existentes ajuda. Se um trecho de código para de executar no meio da criação de um novo valor, nenhuma estrutura de dados existente foi danificada, facilitando a recuperação.

Como isso nem sempre é prático, declarações `try` têm outra funcionalidade: elas podem ser seguidas por um bloco `finally` em vez de ou em adição a um bloco `catch`. Um bloco `finally` diz "não importa *o que* aconteça, execute este código após tentar executar o código no bloco `try`."

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
```

```
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

Esta versão da função rastreia seu progresso e, se ao sair perceber que foi abortada em um ponto onde criou um estado inconsistente, repara o dano que fez.

Note que mesmo que o código `finally` seja executado quando uma exceção é lançada no bloco `try`, ele não interfere com a exceção. Após o bloco `finally` ser executado, a pilha continua se desenrolando.

Escrever programas que operam de forma confiável mesmo quando exceções surgem em lugares inesperados é difícil. Muitas pessoas simplesmente não se incomodam, e como exceções são tipicamente reservadas para circunstâncias excepcionais, o problema pode ocorrer tão raramente que nunca é sequer notado. Se isso é uma coisa boa ou realmente ruim depende de quanto dano o software causará quando falhar.

Captura seletiva

Quando uma exceção percorre todo o caminho até o fundo da pilha sem ser capturada, ela é tratada pelo ambiente. O que isso significa difere entre ambientes. Nos *browsers*, uma descrição do erro é tipicamente escrita no console JavaScript (acessível através do menu Ferramentas ou Desenvolvedor do *browser*). O Node.js, o ambiente JavaScript sem *browser* que discutiremos no Capítulo 20, é mais cuidadoso com a corrupção de dados. Ele aborta todo o processo quando uma exceção não tratada ocorre.

Para erros do programador, simplesmente deixar o erro passar é frequentemente o melhor que se pode fazer. Uma exceção não tratada é uma maneira razoável de sinalizar um programa quebrado, e o console JavaScript, em *browsers* modernos, fornecerá informações sobre quais chamadas de função estavam na pilha quando o problema ocorreu.

Para problemas que se *espera* que aconteçam durante o uso rotineiro, travar com uma exceção não tratada é uma estratégia terrível.

Usos inválidos da linguagem, como referenciar uma *binding* inexistente, procu-

rar uma propriedade em `null` ou chamar algo que não é uma função, também resultarão em exceções sendo lançadas. Tais exceções também podem ser capturadas.

Quando um corpo `catch` é atingido, tudo o que sabemos é que *algo* no nosso corpo `try` causou uma exceção. Mas não sabemos *o que* causou ou *qual* exceção causou.

O JavaScript (em uma omissão bastante flagrante) não fornece suporte direto para capturar exceções seletivamente: ou você captura todas ou não captura nenhuma. Isso torna tentador *supor* que a exceção que você obtém é aquela em que estava pensando quando escreveu o bloco `catch`.

Mas pode não ser. Alguma outra suposição pode ter sido violada, ou você pode ter introduzido um *bug* que está causando uma exceção. Aqui está um exemplo que *tenta* continuar chamando `promptDirection` até obter uma resposta válida:

```
for (;;) {
  try {
    let dir = promtDirection("Where?"); // ← erro de digitação!
    console.log("You chose ", dir);
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}
```

A construção `for (;;)` é uma maneira de criar intencionalmente um *loop* que não termina por conta própria. Saímos do *loop* apenas quando uma direção válida é dada. Infelizmente, escrevemos errado `promptDirection`, o que resultará em um erro de "variável indefinida". Como o bloco `catch` ignora completamente o valor de sua exceção (`e`), assumindo que sabe qual é o problema, ele erroneamente trata o erro de *binding* como indicando entrada ruim. Isso não só causa um *loop* infinito como também "enterra" a mensagem de erro útil sobre a *binding* mal escrita.

Como regra geral, não capture exceções indiscriminadamente a menos que seja com o propósito de "roteá-las" para algum lugar — por exemplo, através da rede para informar outro sistema que nosso programa travou. E mesmo assim, pense cuidadosamente sobre como pode estar escondendo informações.

Queremos capturar um tipo *específico* de exceção. Podemos fazer isso verificando no bloco `catch` se a exceção que obtivemos é aquela em que estamos interessados e, se não, relançá-la. Mas como reconhecemos uma exceção?

Poderíamos comparar sua propriedade `message` com a mensagem de erro que esperamos. Mas essa é uma maneira frágil de escrever código — estaríamos

usando informação destinada ao consumo humano (a mensagem) para tomar uma decisão programática. Assim que alguém muda (ou traduz) a mensagem, o código parará de funcionar.

Em vez disso, vamos definir um novo tipo de erro e usar `instanceof` para identificá-lo.

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

A nova classe de erro estende `Error`. Ela não define seu próprio construtor, o que significa que herda o construtor de `Error`, que espera uma mensagem *string* como argumento. Na verdade, ela não define nada — a classe está vazia. Objetos `InputError` se comportam como objetos `Error`, exceto que têm uma classe diferente pela qual podemos reconhecê-los.

Agora o *loop* pode capturá-los mais cuidadosamente.

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

Isso capturará apenas instâncias de `InputError` e deixará exceções não relacionadas passarem. Se você reintroduzir o erro de digitação, o erro de *binding* indefinida será adequadamente reportado.

Asserções

Asserções são verificações dentro de um programa que verificam que algo é como deveria ser. Elas são usadas não para lidar com situações que podem

surgir em operação normal, mas para encontrar erros do programador.

Se, por exemplo, `firstElement` é descrita como uma função que nunca deve ser chamada em *arrays* vazios, poderíamos escrevê-la assim:

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("firstElement called with []");
  }
  return array[0];
}
```

Agora, em vez de silenciosamente retornar undefined (que é o que você obtém ao ler uma propriedade de *array* que não existe), isso explodirá ruidosamente seu programa assim que você usá-lo indevidamente. Isso torna menos provável que tais erros passem despercebidos e mais fácil encontrar sua causa quando ocorrem.

Não recomendo tentar escrever asserções para todo tipo possível de entrada ruim. Isso daria muito trabalho e levaria a código muito ruidoso. Você vai querer reservá-las para erros que são fáceis de cometer (ou que você se pega cometendo).

Resumo

Uma parte importante da programação é encontrar, diagnosticar e corrigir *bugs*. Problemas podem se tornar mais fáceis de notar se você tiver uma suíte de testes automatizada ou adicionar asserções aos seus programas.

Problemas causados por fatores fora do controle do programa geralmente devem ser ativamente planejados. Às vezes, quando o problema pode ser tratado localmente, valores de retorno especiais são uma boa maneira de rastreá-los. Caso contrário, exceções podem ser preferíveis.

Lançar uma exceção faz com que a pilha de chamadas seja desenrolada até o próximo bloco `try/catch` envolvente ou até o fundo da pilha. O valor da exceção será dado ao bloco `catch` que a captura, que deve verificar que é realmente o tipo esperado de exceção e então fazer algo com ela. Para ajudar a lidar com o fluxo de controle imprevisível causado por exceções, blocos `finally` podem ser usados para garantir que um trecho de código *sempre* execute quando um bloco termina.

Exercícios

Repetir

Digamos que você tenha uma função `primitiveMultiply` que em 20% dos casos multiplica dois números e nos outros 80% dos casos lança uma exceção do tipo `MultiplicatorUnitFailure`. Escreva uma função que envolva essa função desajeitada e continue tentando até que uma chamada tenha sucesso, retornando então o resultado.

Certifique-se de tratar apenas as exceções que está tentando tratar.

A caixa trancada

Considere o seguinte objeto (um tanto artificial):

```
const box = new class {
  locked = true;
  #content = [];

  unlock() { this.locked = false; }
  lock() { this.locked = true;  }
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this.#content;
  }
};
```

É uma caixa com uma tranca. Há um *array* na caixa, mas você só pode acessá-lo quando a caixa estiver destrancada.

Escreva uma função chamada `withBoxUnlocked` que recebe um valor de função como argumento, destranca a caixa, executa a função e então garante que a caixa seja trancada novamente antes de retornar, independentemente de a função argumento ter retornado normalmente ou lançado uma exceção.

Para pontos extras, certifique-se de que se você chamar `withBoxUnlocked` quando a caixa já estiver destrancada, a caixa permaneça destrancada.

"Algumas pessoas, quando confrontadas com um problema, pensam 'Eu sei, vou usar expressões regulares.' Agora elas têm dois problemas."

—Jamie Zawinski

Chapter 9

Expressões Regulares

Ferramentas e técnicas de programação sobrevivem e se espalham de maneira caótica e evolutiva. Nem sempre são as melhores ou mais brilhantes que vencem, mas sim aquelas que funcionam bem o suficiente dentro do nicho certo ou que acontecem de estar integradas com outra tecnologia bem-sucedida.

Neste capítulo, discutirei uma dessas ferramentas, *expressões regulares*. Expressões regulares são uma maneira de descrever padrãoes em dados de *string*. Elas formam uma linguagem pequena e separada que faz parte do JavaScript e de muitas outras linguagens e sistemas.

Expressões regulares são ao mesmo tempo terrivelmente estranhas e extremamente úteis. Sua sintaxe é críptica e a interface de programação que o JavaScript fornece para elas é desajeitada. Mas são uma ferramenta poderosa para inspecionar e processar *strings*. Entender adequadamente expressões regulares fará de você um programador mais eficiente.

Criando uma expressão regular

Uma expressão regular é um tipo de objeto. Ela pode ser construída com o construtor `RegExp` ou escrita como um valor literal envolvendo um padrão em caracteres de barra (`/`).

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Ambos os objetos de expressão regular representam o mesmo padrão: um caractere *a* seguido de um *b* seguido de um *c*.

Ao usar o construtor `RegExp`, o padrão é escrito como uma *string* normal, então as regras usuais se aplicam para barras invertidas.

A segunda notação, onde o padrão aparece entre caracteres de barra, trata barras invertidas de forma um pouco diferente. Primeiro, como uma barra encerra o padrão, precisamos colocar uma barra invertida antes de qualquer barra que queiramos que seja *parte* do padrão. Além disso, barras invertidas

que não fazem parte de códigos de caracteres especiais (como `\n`) serão *preservadas*, em vez de ignoradas como são em *strings*, e mudam o significado do padrão. Alguns caracteres, como interrogações e sinais de mais, têm significados especiais em expressões regulares e devem ser precedidos por uma barra invertida se pretendem representar o próprio caractere.

```
let aPlus = /A\+/;
```

Testando correspondências

Objetos de expressão regular possuem vários métodos. O mais simples é `test`. Se você passar uma *string*, ele retornará um booleano dizendo se a *string* contém uma correspondência do padrão na expressão.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

Uma expressão regular consistindo apenas de caracteres não-especiais simplesmente representa aquela sequência de caracteres. Se *abc* ocorre em qualquer lugar na *string* contra a qual estamos testando (não apenas no início), `test` retornará `true`.

Conjuntos de caracteres

Descobrir se uma *string* contém *abc* poderia ser feito igualmente com uma chamada a `indexOf`. Expressões regulares são úteis porque nos permitem descrever padrãoes mais complicados.

Digamos que queremos encontrar qualquer número. Em uma expressão regular, colocar um conjunto de caracteres entre colchetes faz com que aquela parte da expressão corresponda a qualquer um dos caracteres entre os colchetes.

Ambas as expressões a seguir correspondem a todas as *strings* que contêm um dígito:

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

Dentro de colchetes, um hífen (`-`) entre dois caracteres pode ser usado para indicar um intervalo de caracteres, onde a ordenação é determinada pelo número

Unicode do caractere. Os caracteres 0 a 9 ficam um ao lado do outro nessa ordenação (códigos 48 a 57), então `[0-9]` cobre todos eles e corresponde a qualquer dígito.

Vários grupos comuns de caracteres têm seus próprios atalhos embutidos. Dígitos são um deles: `\d` significa a mesma coisa que `[0-9]`.

`\d`	Qualquer caractere de dígito
`\w`	Um caractere alfanumérico ("caractere de palavra")
`\s`	Qualquer caractere de espaço em branco (espaço, tab, nova linha e similares)
`\D`	Um caractere que *não* é um dígito
`\W`	Um caractere não-alfanumérico
`\S`	Um caractere que não é espaço em branco
`.`	Qualquer caractere exceto nova linha

Você poderia corresponder a um formato de data e hora como 01-30-2003 15:20 com a seguinte expressão:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

Essa expressão regular parece completamente horrível, não é? Metade dela são barras invertidas, produzindo um ruído de fundo que dificulta identificar o padrão real expresso. Veremos uma versão um pouco melhorada dessa expressão mais adiante.

Esses códigos de barra invertida também podem ser usados dentro de colchetes. Por exemplo, `[\d.]` significa qualquer dígito ou um caractere de ponto. O ponto em si, entre colchetes, perde seu significado especial. O mesmo vale para outros caracteres especiais, como o sinal de mais (`+`).

Para *inverter* um conjunto de caracteres — isto é, expressar que você quer corresponder a qualquer caractere *exceto* os do conjunto — você pode escrever um caractere circunflexo (`^`) após o colchete de abertura.

```
let nonBinary = /[^01]/;
console.log(nonBinary.test("1100100010100110"));
// → false
console.log(nonBinary.test("0111010112101001"));
// → true
```

Caracteres internacionais

Devido à implementação inicial simplista do JavaScript e ao fato de que essa abordagem simplista foi mais tarde consolidada como comportamento padrão, as expressões regulares do JavaScript são bastante limitadas em relação a caracteres que não aparecem na língua inglesa. Por exemplo, no que diz respeito às expressões regulares do JavaScript, um "caractere de palavra" é apenas um dos 26 caracteres do alfabeto latino (maiúsculo ou minúsculo), dígitos decimais e, por algum motivo, o caractere sublinhado. Coisas como *é* ou *ß*, que são definitivamente caracteres de palavra, não correspondem a `\w` (e *correspondem* a `\W` maiúsculo, a categoria de não-palavra).

Por um estranho acidente histórico, `\s` (espaço em branco) não tem esse problema e corresponde a todos os caracteres que o padrão Unicode considera espaço em branco, incluindo coisas como o espaço não-quebrável e o separador de vogais mongol.

É possível usar `\p` em uma expressão regular para corresponder a todos os caracteres aos quais o padrão Unicode atribui uma dada propriedade. Isso nos permite corresponder a coisas como letras de uma maneira mais cosmopolita. Porém, novamente devido à compatibilidade com os padrões originais da linguagem, estes são reconhecidos apenas quando se coloca um caractere `u` (de Unicode) após a expressão regular.

`\p{L}`	Qualquer letra
`\p{N}`	Qualquer caractere numérico
`\p{P}`	Qualquer caractere de pontuação
`\P{L}`	Qualquer não-letra (P maiúsculo inverte)
`\p{Script=Hangul}`	Qualquer caractere do script dado (veja Capítulo 5)

Usar `\w` para processamento de texto que pode precisar lidar com texto não-inglês (ou até texto inglês com palavras emprestadas como "cliché") é um risco, pois não tratará caracteres como "é" como letras. Embora tendam a ser um pouco mais verbosos, os grupos de propriedades `\p` são mais robustos.

```
console.log(/\p{L}/u.test("α"));
// → true
console.log(/\p{L}/u.test("!"));
// → false
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
```

Por outro lado, se você está correspondendo números para fazer algo com eles, frequentemente quer `\d` para dígitos, pois converter caracteres numéricos ar-

bitrários em um número JavaScript não é algo que uma função como `Number` pode fazer por você.

Repetindo partes de um padrão

Agora sabemos como corresponder a um único dígito. E se quisermos corresponder a um número inteiro — uma sequência de um ou mais dígitos?

Quando você coloca um sinal de mais (`+`) após algo em uma expressão regular, isso indica que o elemento pode se repetir mais de uma vez. Assim, `/\d+/` corresponde a um ou mais caracteres de dígito.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

O asterisco (`*`) tem um significado semelhante, mas também permite que o padrão corresponda zero vezes. Algo com um asterisco depois nunca impede que um padrão corresponda — ele simplesmente corresponderá zero instâncias se não encontrar nenhum texto adequado.

Uma interrogação (`?`) torna uma parte de um padrão *opcional*, significando que pode ocorrer zero vezes ou uma vez. No exemplo a seguir, o caractere *u* é permitido mas o padrão também corresponde quando ele está ausente:

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

Para indicar que um padrão deve ocorrer um número preciso de vezes, use chaves. Colocar `{4}` após um elemento, por exemplo, exige que ocorra exatamente quatro vezes. Também é possível especificar um intervalo dessa forma: `{2,4}` significa que o elemento deve ocorrer pelo menos duas vezes e no máximo quatro vezes.

Aqui está outra versão do padrão de data e hora que permite tanto dígitos simples quanto duplos para dias, meses e horas. Também é um pouco mais fácil de decifrar.

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("1-30-2003 8:45"));
// → true
```

Você também pode especificar intervalos abertos ao usar chaves, omitindo o número após a vírgula. Por exemplo, `{5,}` significa cinco ou mais vezes.

Agrupando subexpressões

Para usar um operador como `*` ou `+` em mais de um elemento por vez, você deve usar parênteses. Uma parte de uma expressão regular entre parênteses conta como um único elemento para os operadores que a seguem.

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

O primeiro e o segundo `+` se aplicam apenas ao segundo `o` em `boo` e `hoo`, respectivamente. O terceiro `+` se aplica a todo o grupo `(hoo+)`, correspondendo a uma ou mais sequências assim.

O `i` no final da expressão no exemplo torna essa expressão regular insensível a maiúsculas e minúsculas, permitindo que corresponda ao B maiúsculo na *string* de entrada, mesmo que o padrão em si seja todo minúsculo.

Correspondências e grupos

O método `test` é a maneira absolutamente mais simples de corresponder uma expressão regular. Ele diz apenas se correspondeu e nada mais. Expressões regulares também possuem um método `exec` (executar) que retornará `null` se nenhuma correspondência foi encontrada e retornará um objeto com informações sobre a correspondência caso contrário.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

Um objeto retornado de `exec` tem uma propriedade `index` que nos diz *onde* na *string* a correspondência bem-sucedida começa. Fora isso, o objeto parece (e de fato é) um *array* de *strings*, cujo primeiro elemento é a *string* que foi correspondida. No exemplo anterior, esta é a sequência de dígitos que procurávamos.

Valores de *string* têm um método `match` que se comporta de forma semelhante.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

Quando a expressão regular contém subexpressões agrupadas com parênteses, o texto que correspondeu a esses grupos também aparecerá no *array*. A correspondência inteira é sempre o primeiro elemento. O próximo elemento é a parte correspondida pelo primeiro grupo (aquele cujo parêntese de abertura vem primeiro na expressão), depois o segundo grupo, e assim por diante.

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

Quando um grupo não é correspondido de forma alguma (por exemplo, quando seguido por uma interrogação), sua posição no *array* de saída conterá `undefined` . Quando um grupo é correspondido múltiplas vezes (por exemplo, quando seguido por um `+`), apenas a última correspondência acaba no *array*.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

Se você quiser usar parênteses puramente para agrupamento, sem que apareçam no *array* de correspondências, pode colocar `?:` após o parêntese de abertura.

```
console.log(/(?:na)+/.exec("banana"));
// → ["nana"]
```

Grupos podem ser úteis para extrair partes de uma *string*. Se não queremos apenas verificar se uma *string* contém uma data, mas também extraí-la e construir um objeto que a represente, podemos envolver parênteses ao redor dos padrões de dígitos e extrair diretamente a data do resultado de `exec`.

Mas primeiro faremos um breve desvio para discutir a maneira embutida de representar valores de data e hora em JavaScript.

A classe Date

O JavaScript tem uma classe `Date` padrão para representar datas, ou melhor, pontos no tempo. Se você simplesmente criar um objeto date usando `new`, obtém a data e hora atuais.

```
console.log(new Date());
// → Fri Feb 02 2024 18:03:06 GMT+0100 (CET)
```

Você também pode criar um objeto para um momento específico.

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

O JavaScript usa uma convenção onde os números dos meses começam em zero (então dezembro é 11), mas os números dos dias começam em um. Isso é confuso e ridículo. Tome cuidado.

Os últimos quatro argumentos (horas, minutos, segundos e milissegundos) são opcionais e considerados zero quando não fornecidos.

Timestamps são armazenados como o número de milissegundos desde o início de 1970, no fuso horário UTC. Isso segue uma convenção estabelecida pelo "tempo Unix", que foi inventado por volta daquela época. Você pode usar números negativos para tempos antes de 1970. O método `getTime` em um objeto date retorna esse número. É grande, como você pode imaginar.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

Se você der ao construtor `Date` um único argumento, esse argumento é tratado como tal contagem de milissegundos. Você pode obter a contagem de milissegundos atual criando um novo objeto `Date` e chamando `getTime` nele ou chamando a função `Date.now`.

Objetos Date fornecem métodos como `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes` e `getSeconds` para extrair seus componentes. Além de `getFullYear` há também `getYear`, que dá o ano menos 1900 (como `98` ou `125`) e é em grande parte inútil.

Colocando parênteses ao redor das partes da expressão que nos interessam, podemos agora criar um objeto date a partir de uma *string*.

```
function getDate(string) {
  let [_, month, day, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

A *binding* sublinhado (`_`) é ignorada e usada apenas para pular o elemento de correspondência completa no *array* retornado por `exec`.

Limites e look-ahead

Infelizmente, `getDate` também extrairá uma data da *string* `"100-1-30000"`. Uma correspondência pode acontecer em qualquer lugar na *string*, então neste caso, ela simplesmente começará no segundo caractere e terminará no penúltimo.

Se quisermos forçar que a correspondência abranja toda a *string*, podemos adicionar os marcadores `^` e `$`. O circunflexo corresponde ao início da *string* de entrada, enquanto o cifrão corresponde ao final. Assim, `/^\d+$/` corresponde a uma *string* consistindo inteiramente de um ou mais dígitos, `/^!/` corresponde a qualquer *string* que comece com ponto de exclamação, e `/x^/` não corresponde a nenhuma *string* (não pode haver um `x` antes do início da *string*).

Há também um marcador `\b` que corresponde a *limites de palavra*, posições que têm um caractere de palavra de um lado e um caractere não-palavra do outro. Infelizmente, estes usam o mesmo conceito simplista de caracteres de palavra que `\w` e, portanto, não são muito confiáveis.

Note que esses marcadores de limite não correspondem a nenhum caractere real. Eles apenas impõem que uma dada condição se mantenha no lugar onde aparecem no padrão.

Testes de *look-ahead* fazem algo semelhante. Eles fornecem um padrão e farão a correspondência falhar se a entrada não corresponder àquele padrão, mas não avançam a posição de correspondência. São escritos entre `(?=` e `)`.

```
console.log(/a(?=e)/.exec("braeburn"));
// → ["a"]
console.log(/a(?! )/.exec("a b"));
// → null
```

O `e` no primeiro exemplo é necessário para corresponder, mas não faz parte da *string* correspondida. A notação `(?! )` expressa um look-ahead *negativo*. Isso corresponde apenas se o padrão entre parênteses *não* corresponder, fazendo com que o segundo exemplo corresponda apenas a caracteres `a` que não tenham um espaço depois deles.

Padrões de escolha

Digamos que queremos saber se um trecho de texto contém não apenas um número, mas um número seguido de uma das palavras *pig*, *cow* ou *chicken*, ou qualquer uma de suas formas plurais.

Poderíamos escrever três expressões regulares e testá-las em sequência, mas há uma maneira mais elegante. O caractere pipe (`|`) denota uma escolha entre o padrão à sua esquerda e o padrão à sua direita. Podemos usá-lo em expressões como esta:

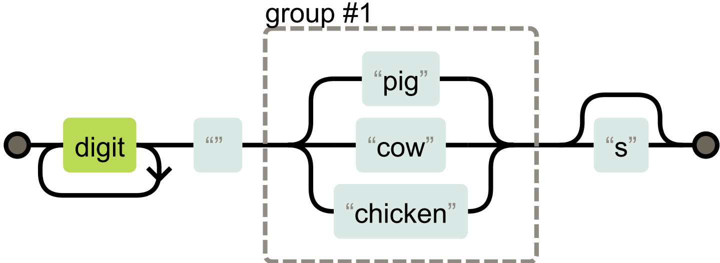
```
let animalCount = /\d+ (pig|cow|chicken)s?/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pugs"));
// → false
```

Parênteses podem ser usados para limitar a parte do padrão à qual o operador pipe se aplica, e você pode colocar múltiplos desses operadores lado a lado para expressar uma escolha entre mais de duas alternativas.

A mecânica da correspondência

Conceitualmente, quando você usa `exec` ou `test`, o motor de expressão regular procura uma correspondência na sua *string* tentando corresponder a expressão primeiro a partir do início da *string*, depois a partir do segundo caractere, e assim por diante até encontrar uma correspondência ou alcançar o final da *string*. Ele retornará a primeira correspondência encontrada ou falhará em encontrar qualquer correspondência.

Para fazer a correspondência real, o motor trata uma expressão regular como um diagrama de fluxo. Este é o diagrama para a expressão de animais do exemplo anterior:

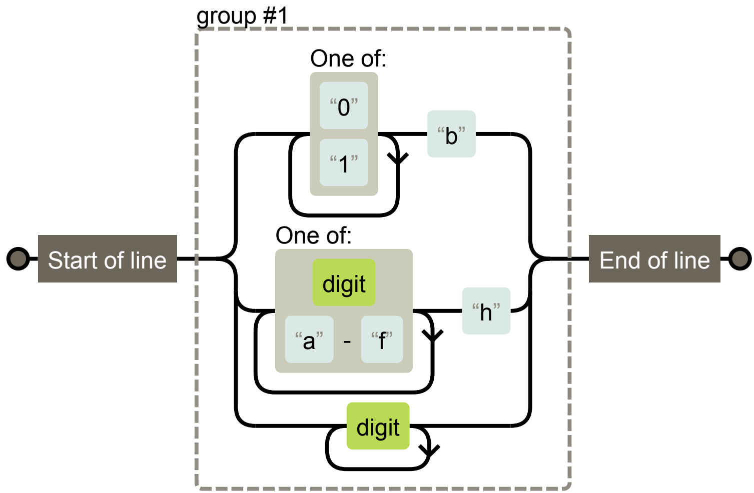


Se conseguirmos encontrar um caminho do lado esquerdo do diagrama ao

lado direito, nossa expressão corresponde. Mantemos uma posição atual na *string* e, toda vez que passamos por uma caixa, verificamos que a parte da *string* após nossa posição atual corresponde àquela caixa.

Retrocesso

A expressão regular `/^([01]+b|[\da-f]+h|\d+)$/` corresponde a um número binário seguido de um `b`, um número hexadecimal (isto é, base 16, com as letras `a` a `f` representando os dígitos 10 a 15) seguido de um `h`, ou um número decimal regular sem caractere de sufixo. Este é o diagrama correspondente:



Ao corresponder essa expressão, o ramo superior (binário) frequentemente será tentado mesmo que a entrada não contenha um número binário. Ao corresponder a *string* `"103"`, por exemplo, fica claro apenas no `3` que estamos no ramo errado. A *string corresponde* à expressão, apenas não ao ramo em que estamos atualmente.

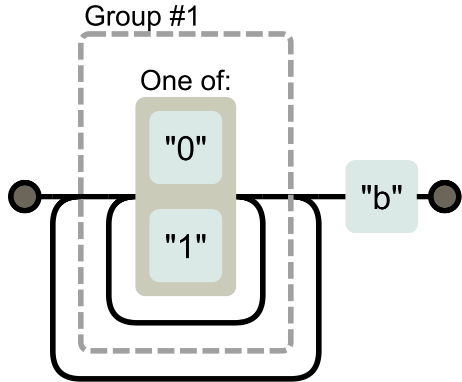
Então o motor *retrocede*. Ao entrar em um ramo, ele lembra sua posição atual (neste caso, no início da *string*, logo após a primeira caixa de limite no diagrama) para que possa voltar e tentar outro ramo se o atual não funcionar. Para a *string* `"103"`, após encontrar o caractere `3`, o motor começa a tentar o ramo para números hexadecimais, que falha novamente porque não há um `h` após o número. Então tenta o ramo de números decimais. Esse se encaixa, e uma correspondência é reportada.

O motor para assim que encontra uma correspondência completa. Isso significa que, se múltiplos ramos poderiam potencialmente corresponder a uma *string*, apenas o primeiro (ordenado por onde os ramos aparecem na expressão

regular) é usado.

O retrocesso também acontece para operadores de repetição como + e `*`. Se você corresponder `/^.*x/` contra `"abcxe"`, a parte `.*` primeiro tentará consumir toda a *string*. O motor então perceberá que precisa de um `x` para corresponder ao padrão. Como não há `x` após o final da *string*, o operador asterisco tenta corresponder um caractere a menos. Mas o motor não encontra um `x` após `abcx` também, então retrocede novamente, correspondendo o operador asterisco apenas a `abc`. *Agora* ele encontra um `x` onde precisa e reporta uma correspondência bem-sucedida das posições 0 a 4.

É possível escrever expressões regulares que farão *muito* retrocesso. Esse problema ocorre quando um padrão pode corresponder a um trecho de entrada de muitas maneiras diferentes. Por exemplo, se ficarmos confusos ao escrever uma expressão regular de número binário, podemos acidentalmente escrever algo como `/([01]+)+b/`.



Se isso tentar corresponder a uma longa série de zeros e uns sem um caractere *b* no final, o motor primeiro percorre o *loop* interno até ficar sem dígitos. Então percebe que não há `b`, então retrocede uma posição, percorre o *loop* externo uma vez e desiste novamente, tentando retroceder do *loop* interno mais uma vez. Ele continuará tentando toda rota possível através desses dois *loops*. Isso significa que a quantidade de trabalho *dobra* com cada caractere adicional. Para apenas algumas dezenas de caracteres, a correspondência resultante levará praticamente uma eternidade.

O método replace

Valores de *string* possuem um método `replace` que pode ser usado para substituir parte da *string* por outra *string*.

```
console.log("papa".replace("p", "m"));
// → mapa
```

O primeiro argumento também pode ser uma expressão regular, caso em que a primeira correspondência da expressão regular é substituída. Quando uma opção `g` (de *global*) é adicionada após a expressão regular, *todas* as correspondências na *string* serão substituídas, não apenas a primeira.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

O verdadeiro poder de usar expressões regulares com `replace` vem do fato de que podemos nos referir a grupos correspondidos na *string* de substituição. Por exemplo, digamos que temos uma *string* grande contendo nomes de pessoas, um nome por linha, no formato `Sobrenome, Nome`. Se quisermos trocar esses nomes e remover a vírgula para obter o formato `Nome Sobrenome`, podemos usar o seguinte código:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nMilner, Robin"
    .replace(/(\p{L}+), (\p{L}+)/gu, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Robin Milner
```

O `$1` e `$2` na *string* de substituição se referem aos grupos entre parênteses no padrão. `$1` é substituído pelo texto que correspondeu ao primeiro grupo, `$2` pelo segundo, e assim por diante, até `$9`. A correspondência inteira pode ser referenciada com `$&`.

É possível passar uma função — em vez de uma *string* — como segundo argumento de `replace`. Para cada substituição, a função será chamada com os grupos correspondidos (assim como a correspondência inteira) como argumentos, e seu valor de retorno será inserido na nova *string*.

Aqui está um exemplo:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // só restou um, remover o 's'
    unit = unit.slice(0, unit.length - 1);
  } else if (amount == 0) {
    amount = "no";
  }
  return amount + " " + unit;
}
```

```
console.log(stock.replace(/(\d+) (\p{L}+)/gu, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

Este código pega uma *string*, encontra todas as ocorrências de um número seguido de uma palavra alfanumérica e retorna uma *string* que tem uma unidade a menos de cada quantidade.

O grupo `(\d+)` acaba como o argumento `amount` da função, e o grupo `(\p{L}+)` é vinculado a `unit`. A função converte `amount` para um número — o que sempre funciona, pois correspondeu a `\d+` anteriormente — e faz alguns ajustes caso reste apenas um ou zero.

Ganância

Podemos usar `replace` para escrever uma função que remove todos os comentários de um trecho de código JavaScript. Aqui está uma primeira tentativa:

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1  1
```

A parte antes do operador `|` corresponde a duas barras seguidas de qualquer número de caracteres que não sejam nova linha. A parte para comentários de múltiplas linhas é mais complexa. Usamos `[^]` (qualquer caractere que não esteja no conjunto vazio de caracteres) como uma maneira de corresponder a qualquer caractere. Não podemos simplesmente usar um ponto aqui porque comentários de bloco podem continuar em uma nova linha, e o caractere ponto não corresponde a caracteres de nova linha.

Mas a saída da última linha parece ter dado errado. Por quê?

A parte `[^]*` da expressão, como descrevi na seção sobre retrocesso, primeiro tentará corresponder o máximo possível. Se isso fizer a próxima parte do padrão falhar, o motor volta um caractere e tenta novamente de lá. No exemplo, o motor primeiro tenta corresponder todo o restante da *string* e então volta de lá. Ele encontrará uma ocorrência de `*/` após voltar quatro caracteres e corresponder a isso. Não é isso que queríamos — a intenção era corresponder a um único comentário, não ir até o final do código e encontrar o final do último comentário de bloco.

Por causa desse comportamento, dizemos que os operadores de repetição (`+`, `*`, `?` e `{}`) são *gananciosos*, significando que correspondem o máximo possível e retrocedem a partir daí. Se você colocar uma interrogação após eles (`+?`, `*?`, `??`, `{}?`), eles se tornam não-gananciosos e começam correspondendo o mínimo possível, correspondendo mais apenas quando o padrão restante não se encaixa na correspondência menor.

E é exatamente isso que queremos neste caso. Fazendo o asterisco corresponder o menor trecho de caracteres que nos leve a um `*/`, consumimos um comentário de bloco e nada mais.

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Muitos bugs em programas de expressão regular podem ser rastreados ao uso não intencional de um operador ganancioso onde um não-ganancioso funcionaria melhor. Ao usar um operador de repetição, prefira a variante não-gananciosa.

Criando objetos RegExp dinamicamente

Em alguns casos, você pode não saber o padrão exato que precisa corresponder quando está escrevendo seu código. Digamos que quer testar o nome do usuário em um trecho de texto. Você pode montar uma *string* e usar o construtor `RegExp` nela.

```
let name = "harry";
let regexp = new RegExp("(^|\\s)" + name + "($|\\s)", "gi");
console.log(regexp.test("Harry is a dodgy character."));
// → true
```

Ao criar a parte `\s` da *string*, precisamos usar duas barras invertidas porque estamos escrevendo em uma *string* normal, não em uma expressão regular delimitada por barras. O segundo argumento do construtor `RegExp` contém as opções para a expressão regular — neste caso, `"gi"` para global e insensível a maiúsculas.

Mas e se o nome for `"dea+hl[]rd"` porque nosso usuário é um adolescente nerd? Isso resultaria em uma expressão regular sem sentido que não corresponderá ao nome do usuário.

Para contornar isso, podemos adicionar barras invertidas antes de qualquer caractere que tenha significado especial.

```
let name = "dea+hl[]rd";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("(^|\\s)" + escaped + "($|\\s)",
                        "gi");
let text = "This dea+hl[]rd guy is super annoying.";
console.log(regexp.test(text));
// → true
```

O método search

Embora o método `indexOf` em *strings* não possa ser chamado com uma expressão regular, existe outro método, `search`, que espera uma expressão regular. Como `indexOf`, ele retorna o primeiro índice no qual a expressão foi encontrada, ou -1 quando não foi encontrada.

```
console.log("  word".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

Infelizmente, não há como indicar que a correspondência deve começar em um dado deslocamento (como podemos com o segundo argumento de `indexOf`), o que seria frequentemente útil.

A propriedade lastIndex

O método `exec` similarmente não fornece uma maneira conveniente de começar a busca a partir de uma dada posição na *string*. Mas fornece uma maneira *in*conveniente.

Objetos de expressão regular possuem propriedades. Uma dessas propriedades é `source`, que contém a *string* a partir da qual a expressão foi criada. Outra propriedade é `lastIndex`, que controla, em algumas circunstâncias limitadas, onde a próxima correspondência começará.

Essas circunstâncias são que a expressão regular deve ter a opção global (`g`) ou sticky (`y`) habilitada, e a correspondência deve acontecer através do método `exec`. Novamente, uma solução menos confusa teria sido simplesmente permitir que um argumento extra fosse passado a `exec`, mas confusão é uma característica essencial da interface de expressões regulares do JavaScript.

```
let pattern = /y/g;
pattern.lastIndex = 3;
```

```
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

Se a correspondência foi bem-sucedida, a chamada a `exec` atualiza automaticamente a propriedade `lastIndex` para apontar após a correspondência. Se nenhuma correspondência foi encontrada, `lastIndex` é redefinido para 0, que também é o valor que tem em um objeto de expressão regular recém-construído.

A diferença entre as opções global e sticky é que, quando sticky está habilitada, a correspondência só terá sucesso se começar diretamente em `lastIndex`, enquanto com global, ela buscará adiante por uma posição onde uma correspondência possa começar.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

Ao usar um valor de expressão regular compartilhado para múltiplas chamadas `exec`, essas atualizações automáticas na propriedade `lastIndex` podem causar problemas. Sua expressão regular pode estar acidentalmente começando em um índice deixado por uma chamada anterior.

```
let digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

Outro efeito interessante da opção global é que ela muda o funcionamento do método `match` em *strings*. Quando chamado com uma expressão global, em vez de retornar um *array* semelhante ao retornado por `exec`, `match` encontrará *todas* as correspondências do padrão na *string* e retornará um *array* contendo as *strings* correspondidas.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Portanto, tenha cuidado com expressões regulares globais. Os casos em que são necessárias — chamadas a `replace` e lugares onde você quer usar `lastIndex` explicitamente — são tipicamente as situações onde você quer usá-las.

Uma coisa comum a fazer é encontrar todas as correspondências de uma expressão regular em uma *string*. Podemos fazer isso usando o método `matchAll`
.

```
let input = "A string with 3 numbers in it... 42 and 88.";
let matches = input.matchAll(/\d+/g);
for (let match of matches) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
//   Found 42 at 33
//   Found 88 at 40
```

Este método retorna um *array* de *arrays* de correspondência. A expressão regular passada a `matchAll` *deve* ter `g` habilitado.

Analisando um arquivo INI

Para concluir o capítulo, veremos um problema que exige expressões regulares. Imagine que estamos escrevendo um programa para coletar automaticamente informações sobre nossos inimigos da internet. (Na verdade, não escreveremos esse programa aqui, apenas a parte que lê o arquivo de configuração. Desculpe.) O arquivo de configuração se parece com isto:

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7

; comentários são precedidos por ponto e vírgula...
; cada seção diz respeito a um inimigo individual
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Davaeorn
type=evil wizard
outputdir=/home/marijn/enemies/davaeorn
```

As regras exatas desse formato — que é um formato de arquivo amplamente usado, geralmente chamado de arquivo *INI* — são as seguintes:

- Linhas em branco e linhas começando com ponto e vírgula são ignoradas.
- Linhas entre `[` e `]` iniciam uma nova seção.

- Linhas contendo um identificador alfanumérico seguido de um caractere = adicionam uma configuração à seção atual.
- Qualquer outra coisa é inválida.

Nossa tarefa é converter uma *string* como esta em um objeto cujas propriedades armazenem *strings* para configurações escritas antes do primeiro cabeçalho de seção e subobjetos para seções, com esses subobjetos armazenando as configurações da seção.

Como o formato precisa ser processado linha por linha, dividir o arquivo em linhas separadas é um bom começo. Vimos o método `split` no Capítulo 4. Alguns sistemas operacionais, porém, usam não apenas um caractere de nova linha para separar linhas, mas um caractere de retorno de carro seguido de uma nova linha (`"\r\n"`). Dado que o método `split` também aceita uma expressão regular como argumento, podemos usar uma expressão regular como `/\r?\n/` para dividir de uma maneira que permita tanto `"\n"` quanto `"\r\n"` entre linhas.

```
function parseINI(string) {
  // Começar com um objeto para armazenar os campos de nível
     superior
  let result = {};
  let section = result;
  for (let line of string.split(/\r?\n/)) {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^\[(.*)\]$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;|$)/.test(line)) {
      throw new Error("Line '" + line + "' is not valid.");
    }
  };
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

O código percorre as linhas do arquivo e constrói um objeto. Propriedades no topo são armazenadas diretamente nesse objeto, enquanto propriedades encon-

tradas em seções são armazenadas em um objeto de seção separado. A *binding* `section` aponta para o objeto da seção atual.

Há dois tipos de linhas significativas — cabeçalhos de seção ou linhas de propriedade. Quando uma linha é uma propriedade regular, ela é armazenada na seção atual. Quando é um cabeçalho de seção, um novo objeto de seção é criado e `section` é definido para apontar para ele.

Note o uso recorrente de `^` e `$` para garantir que a expressão corresponda à linha inteira, não apenas parte dela. Omitir esses resulta em código que funciona na maior parte, mas se comporta de forma estranha para certas entradas, o que pode ser um *bug* difícil de rastrear.

O padrão `if (match = string.match(...))` faz uso do fato de que o valor de uma expressão de atribuição (`=`) é o valor atribuído. Você frequentemente não tem certeza de que sua chamada a `match` terá sucesso, então só pode acessar o objeto resultante dentro de uma declaração `if` que testa isso. Para não quebrar a cadeia agradável de formas `else if`, atribuímos o resultado da correspondência a uma *binding* e imediatamente usamos essa atribuição como o teste da declaração `if`.

Se uma linha não é um cabeçalho de seção nem uma propriedade, a função verifica se é um comentário ou uma linha vazia usando a expressão `/^\s*(;|$)/` para corresponder a linhas que contêm apenas espaço em branco, ou espaço em branco seguido de ponto e vírgula (tornando o restante da linha um comentário). Quando uma linha não corresponde a nenhuma das formas esperadas, a função lança uma exceção.

Unidades de código e caracteres

Outro erro de design que foi padronizado nas expressões regulares do JavaScript é que, por padrão, operadores como `.` ou `?` trabalham com unidades de código (como discutido no Capítulo 5), não caracteres reais. Isso significa que caracteres compostos por duas unidades de código se comportam de maneira estranha.

```
console.log(/{3}/.test(""));
// → false
console.log(/<.>/.test("<>"));
// → false
console.log(/<.>/u.test("<>"));
// → true
```

O problema é que o na primeira linha é tratado como duas unidades de código, e `{3}` é aplicado apenas à segunda unidade. Da mesma forma, o ponto corre-

sponde a uma única unidade de código, não às duas que compõem o emoji da rosa.

Você deve adicionar a opção `u` (Unicode) à sua expressão regular para fazê-la tratar tais caracteres adequadamente.

```
console.log(/{3}/u.test(""));
// → true
```

Resumo

Expressões regulares são objetos que representam padrões em *strings*. Elas usam sua própria linguagem para expressar esses padrões.

`/abc/`	Uma sequência de caracteres
`/[abc]/`	Qualquer caractere de um conjunto
`/[^abc]/`	Qualquer caractere *não* em um conjunto
`/[0-9]/`	Qualquer caractere em um intervalo
`/x+/`	Uma ou mais ocorrências do padrão `x`
`/x+?/`	Uma ou mais ocorrências, não-ganancioso
`/x*/`	Zero ou mais ocorrências
`/x?/`	Zero ou uma ocorrência
`/x{2,4}/`	Duas a quatro ocorrências
`/(abc)/`	Um grupo
`/a|b|c/`	Qualquer um de vários padrões
`/\d/`	Qualquer caractere de dígito
`/\w/`	Um caractere alfanumérico ("caractere de palavra")
`/\s/`	Qualquer caractere de espaço em branco
`/./`	Qualquer caractere exceto nova linha
`/\p{L}/u`	Qualquer caractere de letra
`/^/`	Início da entrada
`/$/`	Fim da entrada
`/(?=a)/`	Um teste de look-ahead

Uma expressão regular tem um método `test` para testar se uma dada *string* a corresponde. Ela também tem um método `exec` que, quando uma correspondência é encontrada, retorna um *array* contendo todos os grupos correspondidos. Tal *array* tem uma propriedade `index` que indica onde a correspondência começou.

Strings têm um método `match` para corresponder contra uma expressão regular e um método `search` para buscar por uma, retornando apenas a posição inicial da correspondência. Seu método `replace` pode substituir correspondên-

cias de um padrão por uma *string* de substituição ou função.

Expressões regulares podem ter opções, que são escritas após a barra de fechamento. A opção `i` torna a correspondência insensível a maiúsculas e minúsculas. A opção `g` torna a expressão *global*, o que, entre outras coisas, faz com que o método `replace` substitua todas as instâncias em vez de apenas a primeira. A opção `y` torna uma expressão sticky, o que significa que ela não buscará adiante e pulará parte da *string* ao procurar uma correspondência. A opção `u` ativa o modo Unicode, que habilita a sintaxe `\p` e corrige diversos problemas em torno do tratamento de caracteres que ocupam duas unidades de código.

Expressões regulares são uma ferramenta afiada com um cabo estranho. Elas simplificam algumas tarefas tremendamente, mas podem rapidamente se tornar incontroláveis quando aplicadas a problemas complexos. Parte de saber como usá-las é resistir à vontade de tentar encaixar nelas coisas que elas não conseguem expressar de forma limpa.

Exercícios

É quase inevitável que, no decorrer do trabalho com esses exercícios, você ficará confuso e frustrado com algum comportamento inexplicável de expressão regular. Às vezes ajuda inserir sua expressão em uma ferramenta online como *debuggex.com* para ver se sua visualização corresponde ao que você pretendia e experimentar como ela responde a várias *strings* de entrada.

Golfe de regexp

Golfe de código é um termo usado para o jogo de tentar expressar um programa particular no menor número de caracteres possível. Da mesma forma, *golfe de regexp* é a prática de escrever a expressão regular mais curta possível para corresponder a um dado padrão e *apenas* aquele padrão.

Para cada um dos itens a seguir, escreva uma expressão regular para testar se o padrão dado ocorre em uma *string*. A expressão regular deve corresponder apenas a *strings* contendo o padrão. Quando sua expressão funcionar, veja se consegue torná-la menor.

1. *car* e *cat*
2. *pop* e *prop*
3. *ferret*, *ferry* e *ferrari*

4. Qualquer palavra terminando em *ious*
5. Um caractere de espaço em branco seguido de um ponto, vírgula, dois-pontos ou ponto e vírgula
6. Uma palavra com mais de seis letras
7. Uma palavra sem a letra *e* (ou *E*)

Consulte a tabela no resumo do capítulo para ajuda. Teste cada solução com algumas *strings* de teste.

Estilo de citação

Imagine que você escreveu uma história e usou aspas simples ao longo do texto para marcar trechos de diálogo. Agora quer substituir todas as aspas de diálogo por aspas duplas, mantendo as aspas simples usadas em contrações como *aren't*.

Pense em um padrão que distinga esses dois tipos de uso de aspas e construa uma chamada ao método `replace` que faça a substituição adequada.

Números novamente

Escreva uma expressão que corresponda apenas a números no estilo JavaScript. Ela deve suportar um sinal de menos *ou* mais opcional na frente do número, o ponto decimal e notação de expoente — `5e-3` ou `1E10` — novamente com um sinal opcional na frente do expoente. Também note que não é necessário haver dígitos antes ou depois do ponto, mas o número não pode ser apenas um ponto. Ou seja, `.5` e `5.` são números JavaScript válidos, mas um ponto sozinho não é.

"Escreva código que seja fácil de deletar, não fácil de estender."
—Tef, programming is terrible

Chapter 10

Módulos

Idealmente, um programa tem uma estrutura clara e direta. A maneira como funciona é fácil de explicar, e cada parte desempenha um papel bem definido.

Na prática, programas crescem organicamente. Funcionalidades são adicionadas conforme o programador identifica novas necessidades. Manter tal programa bem estruturado requer atenção e trabalho constantes. Este é um trabalho que só compensará no futuro, na *próxima* vez que alguém trabalhar no programa, então é tentador negligenciá-lo e permitir que as várias partes do programa se tornem profundamente emaranhadas.

Isso causa dois problemas práticos. Primeiro, entender um sistema emaranhado é difícil. Se tudo pode tocar em tudo o mais, é difícil olhar para qualquer parte isoladamente. Você é forçado a construir um entendimento holístico da coisa toda. Segundo, se quiser usar qualquer funcionalidade de tal programa em outra situação, reescrevê-la pode ser mais fácil do que tentar desemaranhá-la de seu contexto.

A frase "grande bola de lama" é frequentemente usada para tais programas grandes e sem estrutura. Tudo gruda junto, e quando você tenta pegar um pedaço, a coisa toda desmorona e você só consegue fazer uma bagunça.

Programas modulares

Módulos são uma tentativa de evitar esses problemas. Um módulo é um pedaço de programa que especifica de quais outros pedaços ele depende e que funcionalidade fornece para outros módulos usarem (sua *interface*).

Interfaces de módulos têm muito em comum com interfaces de objetos, como vimos no Capítulo 6. Elas tornam parte do módulo disponível para o mundo exterior e mantêm o restante privado.

Mas a interface que um módulo fornece para outros usarem é apenas metade da história. Um bom sistema de módulos também requer que módulos especifiquem qual código *eles* usam de outros módulos. Essas relações são chamadas de *dependências*. Se o módulo A usa funcionalidade do módulo B, diz-se que

depende desse módulo. Quando estas são claramente especificadas no próprio módulo, podem ser usadas para descobrir quais outros módulos precisam estar presentes para poder usar um dado módulo e para carregar automaticamente as dependências.

Quando as maneiras como módulos interagem entre si são explícitas, um sistema se torna mais como LEGO, onde peças interagem através de conectores bem definidos, e menos como lama, onde tudo se mistura com tudo.

Módulos ES

A linguagem JavaScript original não tinha nenhum conceito de módulo. Todos os scripts rodavam no mesmo escopo, e acessar uma função definida em outro script era feito referenciando as *bindings* globais criadas por aquele script. Isso encorajava ativamente o emaranhamento acidental e difícil de ver do código e convidava problemas como scripts não relacionados tentando usar o mesmo nome de *binding*.

Desde o ECMAScript 2015, o JavaScript suporta dois tipos diferentes de programas. *Scripts* se comportam da maneira antiga: suas *bindings* são definidas no escopo global e não têm como referenciar diretamente outros scripts. *Módulos* obtêm seu próprio escopo separado e suportam as palavras-chave `import` e `export`, que não estão disponíveis em scripts, para declarar suas dependências e interface. Esse sistema de módulos é geralmente chamado de *módulos ES* (onde *ES* significa ECMAScript).

Um programa modular é composto por vários desses módulos, conectados via seus *imports* e *exports*.

O módulo de exemplo a seguir converte entre nomes de dias e números (como retornado pelo método `getDay` de `Date`). Ele define uma constante que não faz parte de sua interface e duas funções que fazem. Não tem dependências.

```
const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"];

export function dayName(number) {
  return names[number];
}
export function dayNumber(name) {
  return names.indexOf(name);
}
```

A palavra-chave `export` pode ser colocada na frente de uma definição de função, classe ou *binding* para indicar que aquela *binding* faz parte da interface do mó-

dulo. Isso torna possível que outros módulos usem aquela *binding* importando-a.

```
import {dayName} from "./dayname.js";
let now = new Date();
console.log(`Today is ${dayName(now.getDay())}`);
// → Today is Monday
```

A palavra-chave `import`, seguida de uma lista de nomes de *bindings* entre chaves, torna *bindings* de outro módulo disponíveis no módulo atual. Módulos são identificados por *strings* entre aspas.

Como tal nome de módulo é resolvido para um programa real difere por plataforma. O *browser* os trata como endereços web, enquanto o Node.js os resolve como arquivos. Quando você executa um módulo, todos os outros módulos dos quais ele depende — e os módulos dos quais *aqueles* dependem — são carregados, e as *bindings* exportadas são disponibilizadas para os módulos que as importam.

Declarações de *import* e *export* não podem aparecer dentro de funções, *loops* ou outros blocos. Elas são resolvidas imediatamente quando o módulo é carregado, independentemente de como o código no módulo é executado. Para refletir isso, elas devem aparecer apenas no corpo externo do módulo.

Assim, a interface de um módulo consiste em uma coleção de *bindings* nomeadas, que outros módulos que dependem do módulo podem acessar. *Bindings* importadas podem ser renomeadas para receber um novo nome local usando `as` após seu nome.

```
import {dayName as nomDeJour} from "./dayname.js";
console.log(nomDeJour(3));
// → Wednesday
```

Um módulo também pode ter um *export* especial chamado `default`, que é frequentemente usado para módulos que exportam apenas uma única *binding*. Para definir um *export* padrão, escreva `export default` antes de uma expressão, declaração de função ou declaração de classe.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

Tal *binding* é importada omitindo as chaves ao redor do nome do *import*.

```
import seasonNames from "./seasonname.js";
```

Para importar todas as *bindings* de um módulo de uma vez, você pode usar `import *`. Você fornece um nome, e esse nome será vinculado a um objeto contendo todas as exportações do módulo. Isso pode ser útil quando se usa

muitas exportações diferentes.

```
import * as dayName from "./dayname.js";
console.log(dayName.dayName(3));
// → Wednesday
```

Pacotes

Uma das vantagens de construir um programa a partir de pedaços separados e poder executar alguns desses pedaços por conta própria é que você pode usar o mesmo pedaço em programas diferentes.

Mas como configurar isso? Digamos que quero usar a função `parseINI` do Capítulo 9 em outro programa. Se está claro do que a função depende (neste caso, nada), posso simplesmente copiar aquele módulo para meu novo projeto e usá-lo. Mas então, se encontrar um erro no código, provavelmente o corrigirei no programa em que estiver trabalhando no momento e esquecerei de corrigi-lo no outro programa.

Uma vez que você começa a duplicar código, rapidamente se verá desperdiçando tempo e energia movendo cópias e mantendo-as atualizadas. É aí que os *pacotes* entram. Um pacote é um pedaço de código que pode ser distribuído (copiado e instalado). Ele pode conter um ou mais módulos e tem informações sobre de quais outros pacotes depende. Um pacote também geralmente vem com documentação explicando o que faz, para que pessoas que não o escreveram ainda possam usá-lo.

Quando um problema é encontrado em um pacote ou uma nova funcionalidade é adicionada, o pacote é atualizado. Agora os programas que dependem dele (que também podem ser pacotes) podem copiar a nova versão para obter as melhorias feitas no código.

Trabalhar dessa forma requer infraestrutura. Precisamos de um lugar para armazenar e encontrar pacotes e uma maneira conveniente de instalá-los e atualizá-los. No mundo JavaScript, essa infraestrutura é fornecida pelo NPM (*https://npmjs.com*).

O NPM é duas coisas: um serviço online onde você pode baixar (e enviar) pacotes, e um programa (incluído com o Node.js) que ajuda a instalá-los e gerenciá-los.

No momento da escrita, há mais de três milhões de pacotes diferentes disponíveis no NPM. Uma grande parte deles é lixo, para ser justo. Mas quase todo pacote JavaScript útil e publicamente disponível pode ser encontrado no NPM. Por exemplo, um analisador de arquivo INI, semelhante ao que construímos no

Capítulo 9, está disponível sob o nome de pacote `ini`.

O Capítulo 20 mostrará como instalar tais pacotes localmente usando o programa de linha de comando `npm`.

Ter pacotes de qualidade disponíveis para download é extremamente valioso. Significa que frequentemente podemos evitar reinventar um programa que 100 pessoas já escreveram antes e obter uma implementação sólida e bem testada pressionando algumas teclas.

Software é barato de copiar, então uma vez que alguém o escreveu, distribuí-lo a outras pessoas é um processo eficiente. Escrevê-lo em primeiro lugar *é* trabalho, porém, e responder a pessoas que encontraram problemas no código ou que querem propor novas funcionalidades é ainda mais trabalho.

Por padrão, você possui o copyright do código que escreve, e outras pessoas podem usá-lo apenas com sua permissão. Mas como algumas pessoas são simplesmente legais e porque publicar bom software pode ajudá-lo a se tornar um pouco famoso entre programadores, muitos pacotes são publicados sob uma licença que explicitamente permite que outras pessoas os usem.

A maioria do código no NPM é licenciado dessa forma. Algumas licenças exigem que você também publique código que constrói sobre o pacote sob a mesma licença. Outras são menos exigentes, requerendo apenas que você mantenha a licença com o código ao distribuí-lo. A comunidade JavaScript usa em grande parte o último tipo de licença. Ao usar pacotes de outras pessoas, certifique-se de estar ciente de suas licenças.

Agora, em vez de escrever nosso próprio analisador de arquivo INI, podemos usar um do NPM.

```
import {parse} from "ini";

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

Módulos CommonJS

Antes de 2015, quando a linguagem JavaScript não tinha um sistema de módulos embutido, as pessoas já estavam construindo sistemas grandes em JavaScript. Para tornar isso viável, elas *precisavam* de módulos.

A comunidade projetou seus próprios sistemas de módulos improvisados sobre a linguagem. Esses usam funções para criar um escopo local para os módulos e objetos regulares para representar interfaces de módulos.

Inicialmente, as pessoas simplesmente envolviam manualmente todo o seu módulo em uma "expressão de função imediatamente invocada" para criar o

escopo do módulo e atribuíam seus objetos de interface a uma única variável global.

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

Esse estilo de módulos fornece isolamento, até certo ponto, mas não declara dependências. Em vez disso, apenas coloca sua interface no escopo global e espera que suas dependências, se houver, façam o mesmo. Isso não é ideal.

Se implementarmos nosso próprio carregador de módulos, podemos fazer melhor. A abordagem mais amplamente usada para módulos JavaScript acoplados é chamada de *módulos CommonJS*. O Node.js usou esse sistema de módulos desde o início (embora agora também saiba como carregar módulos ES), e é o sistema de módulos usado por muitos pacotes no NPM.

Um módulo CommonJS parece um script regular, mas tem acesso a duas *bindings* que usa para interagir com outros módulos. A primeira é uma função chamada `require`. Quando você a chama com o nome do módulo de sua dependência, ela garante que o módulo seja carregado e retorna sua interface. A segunda é um objeto chamado `exports`, que é o objeto de interface do módulo. Ele começa vazio e você adiciona propriedades a ele para definir valores exportados.

Este módulo CommonJS de exemplo fornece uma função de formatação de data. Ele usa dois pacotes do NPM — `ordinal` para converter números em *strings* como `"1st"` e `"2nd"`, e `date-names` para obter os nomes em inglês para dias da semana e meses. Ele exporta uma única função, `formatDate`, que recebe um objeto `Date` e uma *string* de template.

A *string* de template pode conter códigos que direcionam o formato, como `YYYY` para o ano completo e `Do` para o dia ordinal do mês. Você poderia dar a ela uma *string* como `"MMMM Do YYYY"` para obter uma saída como `November 22 nd 2017`.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");
```

```
exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

A interface de `ordinal` é uma única função, enquanto `date-names` exporta um objeto contendo múltiplas coisas — `days` e `months` são *arrays* de nomes. A desestruturação é muito conveniente ao criar *bindings* para interfaces importadas.

O módulo adiciona sua função de interface a `exports` para que módulos que dependem dele tenham acesso a ela. Poderíamos usar o módulo assim:

```
const {formatDate} = require("./format-date.js");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

O CommonJS é implementado com um carregador de módulos que, ao carregar um módulo, envolve seu código em uma função (dando-lhe seu próprio escopo local) e passa as *bindings* `require` e `exports` para essa função como argumentos.

Se assumirmos que temos acesso a uma função `readFile` que lê um arquivo pelo nome e nos dá seu conteúdo, podemos definir uma forma simplificada de `require` assim:

```
function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let exports = require.cache[name] = {};
    let wrapper = Function("require, exports", code);
    wrapper(require, exports);
  }
  return require.cache[name];
}
require.cache = Object.create(null);
```

`Function` é uma função embutida do JavaScript que recebe uma lista de argumentos (como uma *string* separada por vírgulas) e uma *string* contendo o corpo da função e retorna um valor de função com esses argumentos e esse corpo. Este é um conceito interessante — permite que um programa crie novos

trechos de programa a partir de dados de *string* — mas também perigoso, pois se alguém puder enganar seu programa para colocar uma *string* que forneça em `Function`, poderá fazer o programa fazer qualquer coisa que quiser.

O JavaScript padrão não fornece tal função como `readFile`, mas diferentes ambientes JavaScript, como o *browser* e o Node.js, fornecem suas próprias maneiras de acessar arquivos. O exemplo apenas finge que `readFile` existe.

Para evitar carregar o mesmo módulo múltiplas vezes, `require` mantém um armazenamento (cache) de módulos já carregados. Quando chamada, primeiro verifica se o módulo solicitado já foi carregado e, se não, o carrega. Isso envolve ler o código do módulo, envolvê-lo em uma função e chamá-la.

Ao definir `require` e `exports` como parâmetros para a função wrapper gerada (e passar os valores apropriados ao chamá-la), o carregador garante que essas *bindings* estejam disponíveis no escopo do módulo.

Uma diferença importante entre este sistema e os módulos ES é que os *imports* de módulos ES acontecem antes de o script de um módulo começar a executar, enquanto `require` é uma função normal, invocada quando o módulo já está em execução. Diferentemente das declarações `import`, chamadas a `require` *podem* aparecer dentro de funções, e o nome da dependência pode ser qualquer expressão que avalie para uma *string*, enquanto `import` permite apenas *strings* simples entre aspas.

A transição da comunidade JavaScript do estilo CommonJS para módulos ES tem sido lenta e um tanto áspera. Felizmente, agora estamos em um ponto onde a maioria dos pacotes populares no NPM fornece seu código como módulos ES, e o Node.js permite que módulos ES importem de módulos CommonJS. Embora código CommonJS ainda seja algo que você encontrará, não há mais razão real para escrever novos programas nesse estilo.

Construção e empacotamento

Muitos pacotes JavaScript não são tecnicamente escritos em JavaScript. Extensões de linguagem como TypeScript, o dialeto de verificação de tipos mencionado no Capítulo 8, são amplamente usadas. As pessoas também frequentemente começam a usar funcionalidades planejadas da linguagem muito antes de serem adicionadas às plataformas que realmente executam JavaScript. Para tornar isso possível, elas *compilam* seu código, traduzindo-o de seu dialeto JavaScript escolhido para JavaScript puro — ou até para uma versão anterior de JavaScript — para que browsers possam executá-lo.

Incluir um programa modular que consiste em 200 arquivos diferentes em uma página web produz seus próprios problemas. Se buscar um único arquivo

pela rede leva 50 milissegundos, carregar o programa inteiro leva 10 segundos, ou talvez metade disso se puder carregar vários arquivos simultaneamente. Isso é muito tempo desperdiçado. Como buscar um único arquivo grande tende a ser mais rápido do que buscar muitos pequenos, programadores web começaram a usar ferramentas que combinam seus programas (que eles meticulosamente dividiram em módulos) em um único arquivo grande antes de publicá-lo na web. Tais ferramentas são chamadas de *bundlers*.

E podemos ir mais longe. Além do número de arquivos, o *tamanho* dos arquivos também determina a rapidez com que podem ser transferidos pela rede. Assim, a comunidade JavaScript inventou *minificadores*. Estas são ferramentas que pegam um programa JavaScript e o tornam menor, removendo automaticamente comentários e espaços em branco, renomeando *bindings* e substituindo trechos de código por código equivalente que ocupa menos espaço.

Não é incomum que o código que você encontra em um pacote NPM ou que roda em uma página web tenha passado por *múltiplos* estágios de transformação — convertendo de JavaScript moderno para JavaScript histórico, combinando os módulos em um único arquivo e minificando o código. Não entraremos nos detalhes dessas ferramentas neste livro, pois há muitas delas, e qual é popular muda regularmente. Apenas esteja ciente de que tais coisas existem e procure-as quando precisar.

Design de módulos

Estruturar programas é um dos aspectos mais sutis da programação. Qualquer funcionalidade não-trivial pode ser organizada de várias maneiras.

O bom design de programa é subjetivo — há compensações envolvidas e questões de gosto. A melhor maneira de aprender o valor de um design bem estruturado é ler ou trabalhar em muitos programas e notar o que funciona e o que não funciona. Não assuma que uma bagunça dolorosa é "simplesmente assim". Você pode melhorar a estrutura de quase tudo colocando mais pensamento nisso.

Um aspecto do design de módulos é a facilidade de uso. Se você está projetando algo que pretende ser usado por múltiplas pessoas — ou até por você mesmo, em três meses quando não se lembrar mais dos detalhes do que fez — é útil que sua interface seja simples e previsível.

Isso pode significar seguir convenções existentes. Um bom exemplo é o pacote `ini`. Esse módulo imita o objeto padrão `JSON` fornecendo funções `parse` e `stringify` (para escrever um arquivo INI) e, como `JSON`, converte entre *strings* e objetos simples. A interface é pequena e familiar, e depois de trabalhar com

ela uma vez, é provável que se lembre de como usá-la.

Mesmo que não haja uma função padrão ou pacote amplamente usado para imitar, você pode manter seus módulos previsíveis usando estruturas de dados simples e fazendo uma única coisa focada. Muitos dos módulos de análise de arquivo INI no NPM fornecem uma função que lê diretamente tal arquivo do disco rígido e o analisa, por exemplo. Isso torna impossível usar tais módulos no *browser*, onde não temos acesso direto ao sistema de arquivos, e adiciona complexidade que seria melhor resolvida *compondo* o módulo com alguma função de leitura de arquivo.

Isso aponta para outro aspecto útil do design de módulos — a facilidade com que algo pode ser composto com outro código. Módulos focados que computam valores são aplicáveis em uma gama mais ampla de programas do que módulos maiores que realizam ações complicadas com efeitos colaterais. Um leitor de arquivo INI que insiste em ler o arquivo do disco é inútil em um cenário onde o conteúdo do arquivo vem de outra fonte.

Relacionadamente, objetos com estado são às vezes úteis ou até necessários, mas se algo pode ser feito com uma função, use uma função. Vários dos leitores de arquivo INI no NPM fornecem um estilo de interface que requer que você primeiro crie um objeto, depois carregue o arquivo em seu objeto e finalmente use métodos especializados para obter os resultados. Esse tipo de coisa é comum na tradição orientada a objetos, e é terrível. Em vez de fazer uma única chamada de função e seguir em frente, você tem que realizar o ritual de mover seu objeto através de seus vários estados. E como os dados agora estão envolvidos em um tipo de objeto especializado, todo código que interage com ele precisa conhecer esse tipo, criando interdependências desnecessárias.

Frequentemente, definir novas estruturas de dados não pode ser evitado — apenas algumas básicas são fornecidas pelo padrão da linguagem, e muitos tipos de dados precisam ser mais complexos do que um *array* ou um map. Mas quando um *array* basta, use um *array*.

Um exemplo de uma estrutura de dados um pouco mais complexa é o grafo do Capítulo 7. Não há uma única maneira óbvia de representar um grafo em JavaScript. Naquele capítulo, usamos um objeto cujas propriedades armazenam *arrays* de *strings* — os outros nós alcançáveis a partir daquele nó.

Há vários pacotes de busca de caminho diferentes no NPM, mas nenhum deles usa esse formato de grafo. Eles geralmente permitem que as arestas do grafo tenham um peso, que é o custo ou distância associado a ela. Isso não é possível em nossa representação.

Por exemplo, existe o pacote `dijkstrajs`. Uma abordagem bem conhecida para busca de caminho, bastante similar à nossa função `findRoute`, é chamada de *algoritmo de Dijkstra*, em homenagem a Edsger Dijkstra, que primeiro o

escreveu. O sufixo `js` é frequentemente adicionado a nomes de pacotes para indicar que são escritos em JavaScript. Esse pacote `dijkstrajs` usa um formato de grafo semelhante ao nosso, mas em vez de *arrays*, usa objetos cujos valores de propriedade são números — os pesos das arestas.

Se quiséssemos usar esse pacote, teríamos que garantir que nosso grafo estivesse armazenado no formato que ele espera. Todas as arestas recebem o mesmo peso, já que nosso modelo simplificado trata cada estrada como tendo o mesmo custo (um turno).

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

Isso pode ser uma barreira à composição — quando vários pacotes usam estruturas de dados diferentes para descrever coisas semelhantes, combiná-los é difícil. Portanto, se quiser projetar para composabilidade, descubra quais estruturas de dados outras pessoas estão usando e, quando possível, siga o exemplo delas.

Projetar uma estrutura de módulos adequada para um programa pode ser difícil. Na fase em que você ainda está explorando o problema, tentando coisas diferentes para ver o que funciona, pode querer não se preocupar muito com isso, pois manter tudo organizado pode ser uma grande distração. Uma vez que você tenha algo que pareça sólido, esse é um bom momento para dar um passo atrás e organizá-lo.

Resumo

Módulos fornecem estrutura a programas maiores separando o código em pedaços com interfaces e dependências claras. A interface é a parte do módulo que é visível para outros módulos, e as dependências são os outros módulos que ele utiliza.

Como o JavaScript historicamente não fornecia um sistema de módulos, o sistema CommonJS foi construído sobre ele. Então, em algum momento, ele

ganhou um sistema embutido, que agora coexiste de forma desconfortável com o sistema CommonJS.

Um pacote é um pedaço de código que pode ser distribuído por conta própria. O NPM é um repositório de pacotes JavaScript. Você pode baixar todos os tipos de pacotes úteis (e inúteis) dele.

Exercícios

Um robô modular

Estas são as *bindings* que o projeto do Capítulo 7 cria:

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

Se você fosse escrever esse projeto como um programa modular, quais módulos criaria? Qual módulo dependeria de qual outro, e como seriam suas interfaces?

Quais peças provavelmente estariam disponíveis já prontas no NPM? Você preferiria usar um pacote NPM ou escrevê-las você mesmo?

Módulo de estradas

Escreva um módulo ES baseado no exemplo do Capítulo 7 que contenha o *array* de estradas e exporte a estrutura de dados de grafo que as representa como `roadGraph`. Ele depende de um módulo `./graph.js` que exporta uma função `buildGraph`, usada para construir o grafo. Essa função espera um *array* de *arrays* de dois elementos (os pontos de início e fim das estradas).

Dependências circulares

Uma dependência circular é uma situação onde o módulo A depende de B, e B também, direta ou indiretamente, depende de A. Muitos sistemas de módulos simplesmente proíbem isso porque qualquer que seja a ordem que você escolha

para carregar tais módulos, não pode garantir que as dependências de cada módulo tenham sido carregadas antes de ele executar.

Módulos CommonJS permitem uma forma limitada de dependências cíclicas. Desde que os módulos não acessem a interface um do outro até que terminem de carregar, dependências cíclicas estão OK.

A função `require` fornecida anteriormente neste capítulo suporta esse tipo de ciclo de dependência. Você consegue ver como ela lida com ciclos?

*"Quem consegue esperar quieto enquanto a lama assenta?
Quem consegue permanecer imóvel até o momento da ação?"*

—Laozi, Tao Te Ching

Chapter 11

Programação Assíncrona

A parte central de um computador, a parte que executa os passos individuais que compõem nossos programas, é chamada de *processador*. Os programas que vimos até agora manterão o processador ocupado até terminarem seu trabalho. A velocidade com que algo como um *loop* que manipula números pode ser executado depende quase inteiramente da velocidade do processador e da memória do computador.

Mas muitos programas interagem com coisas fora do processador. Por exemplo, podem se comunicar por uma rede de computadores ou solicitar dados do disco rígido — que é muito mais lento do que obtê-los da memória.

Quando tal coisa está acontecendo, seria uma pena deixar o processador ocioso — pode haver outro trabalho que ele poderia fazer enquanto isso. Em parte, isso é tratado pelo seu sistema operacional, que alternará o processador entre múltiplos programas em execução. Mas isso não ajuda quando queremos que um *único* programa consiga progredir enquanto espera por uma requisição de rede.

Assincronicidade

Em um modelo de programação *síncrona*, as coisas acontecem uma de cada vez. Quando você chama uma função que realiza uma ação de longa duração, ela retorna somente quando a ação terminou e pode retornar o resultado. Isso paralisa seu programa pelo tempo que a ação leva.

Um modelo *assíncrono* permite que múltiplas coisas aconteçam ao mesmo tempo. Quando você inicia uma ação, seu programa continua executando. Quando a ação termina, o programa é informado e obtém acesso ao resultado (por exemplo, os dados lidos do disco).

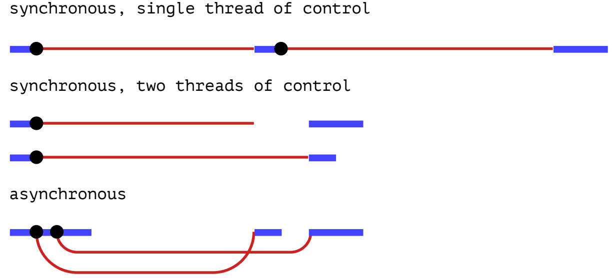
Podemos comparar a programação síncrona e assíncrona usando um pequeno exemplo: um programa que faz duas requisições pela rede e depois combina os resultados.

Em um ambiente síncrono, onde a função de requisição retorna somente

depois de ter feito seu trabalho, a maneira mais fácil de realizar essa tarefa é fazer as requisições uma após a outra. Isso tem a desvantagem de que a segunda requisição só será iniciada quando a primeira terminar. O tempo total levado será no mínimo a soma dos dois tempos de resposta.

A solução para esse problema, em um sistema síncrono, é iniciar threads adicionais de controle. Uma *thread* é outro programa em execução cuja execução pode ser intercalada com outros programas pelo sistema operacional — já que a maioria dos computadores modernos contém múltiplos processadores, múltiplas threads podem até executar ao mesmo tempo, em processadores diferentes. Uma segunda thread poderia iniciar a segunda requisição, e então ambas as threads esperam seus resultados voltarem, após o que se ressincronizam para combinar seus resultados.

No diagrama a seguir, as linhas grossas representam o tempo que o programa gasta executando normalmente, e as linhas finas representam o tempo gasto esperando pela rede. No modelo síncrono, o tempo gasto pela rede é *parte* da linha do tempo para uma dada thread de controle. No modelo assíncrono, iniciar uma ação de rede permite que o programa continue executando enquanto a comunicação de rede acontece ao lado dele, notificando o programa quando terminar.



Outra maneira de descrever a diferença é que esperar por ações terminarem é *implícito* no modelo síncrono, enquanto é *explícito* — sob nosso controle — no assíncrono.

A assincronicidade corta para os dois lados. Ela facilita expressar programas que não se encaixam no modelo linear de controle, mas também pode tornar mais estranho expressar programas que seguem uma linha reta. Veremos algumas maneiras de reduzir essa estranheza mais adiante no capítulo.

Ambas as plataformas proeminentes de programação JavaScript — browsers e Node.js — tornam assíncronas operações que podem levar algum tempo, em vez de depender de threads. Como programar com threads é notoriamente difícil (entender o que um programa faz é muito mais difícil quando ele faz múltiplas coisas ao mesmo tempo), isso é geralmente considerado algo bom.

Callbacks

Uma abordagem à programação assíncrona é fazer com que funções que precisam esperar por algo recebam um argumento extra, uma *função de callback*. A função assíncrona inicia um processo, configura as coisas para que a função de *callback* seja chamada quando o processo terminar e então retorna.

Como exemplo, a função `setTimeout`, disponível tanto no Node.js quanto em *browsers*, espera um dado número de milissegundos e então chama uma função.

```
setTimeout(() => console.log("Tick"), 500);
```

Esperar geralmente não é um trabalho importante, mas pode ser muito útil quando você precisa organizar algo para acontecer em um certo momento ou verificar se alguma ação está levando mais tempo do que o esperado.

Outro exemplo de uma operação assíncrona comum é ler um arquivo do armazenamento de um dispositivo. Imagine que você tem uma função `readTextFile` que lê o conteúdo de um arquivo como *string* e o passa para uma função de *callback*.

```
readTextFile("shopping_list.txt", content => {
  console.log(`Shopping List:\n${content}`);
});
// → Shopping List:
// → Peanut butter
// → Bananas
```

A função `readTextFile` não faz parte do JavaScript padrão. Veremos como ler arquivos no *browser* e no Node.js em capítulos posteriores.

Realizar múltiplas ações assíncronas em sequência usando *callbacks* significa que você precisa continuar passando novas funções para lidar com a continuação da computação após as ações. Uma função assíncrona que compara dois arquivos e produz um booleano indicando se o conteúdo deles é o mesmo poderia parecer assim:

```
function compareFiles(fileA, fileB, callback) {
  readTextFile(fileA, contentA => {
    readTextFile(fileB, contentB => {
      callback(contentA == contentB);
    });
  });
}
```

Esse estilo de programação é viável, mas o nível de indentação aumenta a cada ação assíncrona porque você acaba dentro de outra função. Fazer coisas mais

complicadas, como envolver ações assíncronas em um *loop*, pode ficar estranho.

De certa forma, a assincronicidade é *contagiosa*. Qualquer função que chama uma função que funciona assincronamente deve ela própria ser assíncrona, usando um *callback* ou mecanismo semelhante para entregar seu resultado. Chamar um *callback* é um pouco mais complexo e propenso a erros do que simplesmente retornar um valor, então precisar estruturar grandes partes do seu programa dessa forma não é ótimo.

Promises

Uma maneira ligeiramente diferente de construir um programa assíncrono é ter funções assíncronas que retornam um objeto que representa seu resultado (futuro) em vez de passar funções de *callback*. Dessa forma, tais funções realmente retornam algo significativo, e o formato do programa se assemelha mais ao de programas síncronos.

É para isso que serve a classe padrão `Promise`. Uma *promise* é um recibo representando um valor que pode não estar disponível ainda. Ela fornece um método `then` que permite registrar uma função que deve ser chamada quando a ação que ela aguarda terminar. Quando a *promise* é *resolvida*, significando que seu valor se torna disponível, tais funções (pode haver múltiplas) são chamadas com o valor do resultado. É possível chamar `then` em uma *promise* que já foi resolvida — sua função ainda será chamada.

A maneira mais fácil de criar uma *promise* é chamando `Promise.resolve`. Essa função garante que o valor que você dá a ela seja envolvido em uma *promise*. Se já for uma *promise*, é simplesmente retornada. Caso contrário, você obtém uma nova *promise* que resolve imediatamente com seu valor como resultado.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

Para criar uma *promise* que não resolve imediatamente, você pode usar `Promise` como construtor. Ele tem uma interface um tanto estranha: o construtor espera uma função como argumento, que ele chama imediatamente, passando a ela uma função que ela pode usar para resolver a *promise*.

Por exemplo, é assim que você poderia criar uma interface baseada em *promises* para a função `readTextFile`:

```
function textFile(filename) {
  return new Promise(resolve => {
```

```
    readTextFile(filename, text => resolve(text));
  });
}

textFile("plans.txt").then(console.log);
```

Note como, em contraste com funções no estilo *callback*, esta função assíncrona retorna um valor significativo — uma *promise* de dar a você o conteúdo do arquivo em algum momento no futuro.

Uma coisa útil sobre o método `then` é que ele próprio retorna outra *promise*. Esta resolve para o valor retornado pela função de *callback* ou, se o valor retornado for uma *promise*, para o valor para o qual essa *promise* resolve. Assim, você pode "encadear" múltiplas chamadas a `then` para configurar uma sequência de ações assíncronas.

Esta função, que lê um arquivo cheio de nomes de arquivos e retorna o conteúdo de um arquivo aleatório naquela lista, mostra esse tipo de *pipeline* assíncrono de *promises*:

```
function randomFile(listFile) {
  return textFile(listFile)
    .then(content => content.trim().split("\n"))
    .then(ls => ls[Math.floor(Math.random() * ls.length)])
    .then(filename => textFile(filename));
}
```

A função retorna o resultado dessa cadeia de chamadas `then`. A *promise* inicial busca a lista de arquivos como *string*. A primeira chamada `then` transforma essa *string* em um *array* de linhas, produzindo uma nova *promise*. A segunda chamada `then` escolhe uma linha aleatória, produzindo uma terceira *promise* que contém um único nome de arquivo. A chamada `then` final lê esse arquivo, então o resultado da função como um todo é uma *promise* que retorna o conteúdo de um arquivo aleatório.

Nesse código, as funções usadas nas duas primeiras chamadas `then` retornam um valor regular que será imediatamente passado para a *promise* retornada por `then` quando a função retornar. A última chamada `then` retorna uma *promise* (`textFile(filename)`), tornando-a um passo assíncrono real.

Também teria sido possível realizar todos esses passos dentro de um único *callback* `then`, pois apenas o último passo é realmente assíncrono. Mas o tipo de wrappers `then` que apenas fazem alguma transformação síncrona de dados é frequentemente útil, como quando se quer retornar uma *promise* que produz uma versão processada de algum resultado assíncrono.

```
function jsonFile(filename) {
```

```
  return textFile(filename).then(JSON.parse);
}

jsonFile("package.json").then(console.log);
```

Geralmente, é útil pensar em uma *promise* como um dispositivo que permite ao código ignorar a questão de quando um valor vai chegar. Um valor normal precisa realmente existir antes de podermos referenciá-lo. Um valor prometido é um valor que *pode* já estar lá ou pode aparecer em algum ponto no futuro. Computações definidas em termos de *promises*, conectando-as com chamadas `then`, são executadas assincronamente conforme suas entradas se tornam disponíveis.

Falha

Computações JavaScript regulares podem falhar lançando uma exceção. Computações assíncronas frequentemente precisam de algo assim. Uma requisição de rede pode falhar, um arquivo pode não existir, ou algum código que faz parte da computação assíncrona pode lançar uma exceção.

Um dos problemas mais urgentes com o estilo *callback* de programação assíncrona é que torna extremamente difícil garantir que falhas sejam adequadamente reportadas aos *callbacks*.

Uma convenção comum é usar o primeiro argumento do *callback* para indicar que a ação falhou, e o segundo para passar o valor produzido pela ação quando foi bem-sucedida.

```
someAsyncFunction((error, value) => {
  if (error) handleError(error);
  else processValue(value);
});
```

Tais funções de *callback* devem sempre verificar se receberam uma exceção e garantir que quaisquer problemas que causem, incluindo exceções lançadas por funções que chamam, sejam capturados e dados à função correta.

Promises tornam isso mais fácil. Elas podem ser resolvidas (a ação terminou com sucesso) ou rejeitadas (ela falhou). Handlers de resolução (registrados com `then`) são chamados apenas quando a ação é bem-sucedida, e rejeições são propagadas para a nova *promise* retornada por `then`. Quando um handler lança uma exceção, isso automaticamente faz com que a *promise* produzida por sua chamada `then` seja rejeitada. Se qualquer elemento em uma cadeia de ações assíncronas falhar, o resultado da cadeia inteira é marcado como rejeitado, e nenhum handler de sucesso é chamado além do ponto onde falhou.

Assim como resolver uma *promise* fornece um valor, rejeitar uma também fornece um valor, geralmente chamado de *razão* da rejeição. Quando uma exceção em uma função handler causa a rejeição, o valor da exceção é usado como razão. Da mesma forma, quando um handler retorna uma *promise* que é rejeitada, essa rejeição flui para a próxima *promise*. Existe uma função `Promise.reject` que cria uma nova *promise* imediatamente rejeitada.

Para lidar explicitamente com tais rejeições, *promises* possuem um método `catch` que registra um handler para ser chamado quando a *promise* é rejeitada, semelhante a como handlers `then` lidam com resolução normal. Também é muito parecido com `then` no sentido de que retorna uma nova *promise*, que resolve para o valor da *promise* original quando resolve normalmente e para o resultado do handler `catch` caso contrário. Se um handler `catch` lança um erro, a nova *promise* também é rejeitada.

Como atalho, `then` também aceita um handler de rejeição como segundo argumento, para que você possa instalar ambos os tipos de handlers em uma única chamada de método: `.then(acceptHandler, rejectHandler)`.

Uma função passada ao construtor `Promise` recebe um segundo argumento, além da função de resolução, que pode usar para rejeitar a nova *promise*.

Quando nossa função `readTextFile` encontra um problema, ela passa o erro para sua função de *callback* como segundo argumento. Nosso wrapper `textFile` deveria na verdade verificar esse argumento para que uma falha faça a *promise* que retorna ser rejeitada.

```
function textFile(filename) {
  return new Promise((resolve, reject) => {
    readTextFile(filename, (text, error) => {
      if (error) reject(error);
      else resolve(text);
    });
  });
}
```

As cadeias de valores de *promise* criadas por chamadas a `then` e `catch` formam assim um *pipeline* através do qual valores assíncronos ou falhas se movem. Como tais cadeias são criadas registrando handlers, cada elo tem um handler de sucesso ou um handler de rejeição (ou ambos) associado a ele. Handlers que não correspondem ao tipo de resultado (sucesso ou falha) são ignorados. Handlers que correspondem são chamados, e seu resultado determina que tipo de valor vem em seguida — sucesso quando retornam um valor que não é *promise*, rejeição quando lançam uma exceção, e o resultado da *promise* quando retornam uma *promise*.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Handler 1:", value))
  .catch(reason => {
    console.log("Caught failure " + reason);
    return "nothing";
  })
  .then(value => console.log("Handler 2:", value));
// → Caught failure Error: Fail
// → Handler 2: nothing
```

A primeira função handler `then` não é chamada porque nesse ponto do *pipeline* a *promise* contém uma rejeição. O handler `catch` lida com essa rejeição e retorna um valor, que é dado à segunda função handler `then`.

Assim como uma exceção não capturada é tratada pelo ambiente, ambientes JavaScript podem detectar quando uma rejeição de *promise* não é tratada e reportarão isso como um erro.

Carla

É um dia ensolarado em Berlim. A pista do antigo aeroporto desativado está repleta de ciclistas e patinadores. Na grama perto de um contêiner de lixo, um bando de corvos ruidosamente se movimenta, tentando convencer um grupo de turistas a abrir mão de seus sanduíches.

Uma das corvos se destaca — uma fêmea grande e desgrenhada com algumas penas brancas na asa direita. Ela engana as pessoas com uma habilidade e confiança que sugerem que faz isso há muito tempo. Quando um senhor idoso é distraído pelas artimanhas de outra corvo, ela calmamente mergulha, agarra o pãozinho meio-comido da mão dele e voa embora.

Ao contrário do resto do grupo, que parece feliz em passar o dia fazendo palhaçadas ali, a corvo grande parece determinada. Carregando seu espólio, ela voa direto para o telhado do prédio do hangar, desaparecendo em uma abertura de ventilação.

Dentro do prédio, ouve-se um som estranho de batidas — suave, mas persistente. Ele vem de um espaço estreito sob o telhado de uma escadaria inacabada. A corvo está sentada ali, cercada por seus lanches roubados, meia dúzia de smartphones (vários dos quais estão ligados) e uma bagunça de cabos. Ela bate rapidamente na tela de um dos telefones com o bico. Palavras estão aparecendo nele. Se você não soubesse melhor, pensaria que ela está digitando.

Essa corvo é conhecida por seus pares como “cāāw-krö". Mas como esses sons são pouco adequados para cordas vocais humanas, vamos nos referir a ela como Carla.

Carla é uma corvo um tanto peculiar. Na juventude, era fascinada pela linguagem humana, bisbilhotando as pessoas até ter um bom domínio do que diziam. Mais tarde na vida, seu interesse mudou para a tecnologia humana, e ela começou a roubar telefones para estudá-los. Seu projeto atual é aprender a programar. O texto que está digitando em seu laboratório escondido é, na verdade, um trecho de código JavaScript assíncrono.

Invadindo

Carla adora a internet. Infelizmente, o telefone em que está trabalhando está prestes a ficar sem dados pré-pagos. O prédio tem uma rede sem fio, mas requer um código para acessar.

Felizmente, os roteadores sem fio do prédio têm 20 anos e são mal protegidos. Pesquisando um pouco, Carla descobre que o mecanismo de autenticação da rede tem uma falha que ela pode usar. Ao entrar na rede, um dispositivo deve enviar a senha correta de seis dígitos. O ponto de acesso responderá com uma mensagem de sucesso ou falha dependendo de se o código correto foi fornecido. Porém, ao enviar um código parcial (digamos, apenas três dígitos), a resposta é diferente dependendo de se aqueles dígitos são o início correto do código ou não. Enviar números incorretos retorna imediatamente uma mensagem de falha. Ao enviar os corretos, o ponto de acesso espera por mais dígitos.

Isso torna possível acelerar enormemente a adivinhação do número. Carla pode encontrar o primeiro dígito tentando cada número por vez, até encontrar um que não retorne falha imediatamente. Tendo um dígito, ela pode encontrar o segundo da mesma forma, e assim por diante, até saber toda a senha.

Assuma que Carla tem uma função `joinWifi`. Dado o nome da rede e a senha (como *string*), a função tenta entrar na rede, retornando uma *promise* que resolve se bem-sucedida e rejeita se a autenticação falhou. A primeira coisa que ela precisa é uma maneira de envolver uma *promise* de modo que ela automaticamente rejeite se levar muito tempo, para permitir que o programa avance rapidamente se o ponto de acesso não responder.

```
function withTimeout(promise, time) {
  return new Promise((resolve, reject) => {
    promise.then(resolve, reject);
    setTimeout(() => reject("Timed out"), time);
  });
}
```

Isso usa o fato de que uma *promise* pode ser resolvida ou rejeitada apenas uma vez. Se a *promise* dada como argumento resolver ou rejeitar primeiro,

esse resultado será o resultado da *promise* retornada por `withTimeout`. Se, por outro lado, o `setTimeout` disparar primeiro, rejeitando a *promise*, quaisquer chamadas posteriores de resolução ou rejeição são ignoradas.

Para encontrar a senha inteira, o programa precisa repetidamente procurar o próximo dígito tentando cada dígito. Se a autenticação tem sucesso, sabemos que encontramos o que procurávamos. Se falha imediatamente, sabemos que aquele dígito estava errado e devemos tentar o próximo. Se a requisição expira, encontramos outro dígito correto e devemos continuar adicionando outro dígito.

Como você não pode esperar por uma *promise* dentro de um *loop* `for`, Carla usa uma função recursiva para conduzir esse processo. A cada chamada, essa função recebe o código como o conhecemos até agora, bem como o próximo dígito a tentar. Dependendo do que acontece, ela pode retornar um código terminado ou chamar a si mesma, para começar a descobrir a próxima posição no código ou para tentar novamente com outro dígito.

```
function crackPasscode(networkID) {
  function nextDigit(code, digit) {
    let newCode = code + digit;
    return withTimeout(joinWifi(networkID, newCode), 50)
      .then(() => newCode)
      .catch(failure => {
        if (failure == "Timed out") {
          return nextDigit(newCode, 0);
        } else if (digit < 9) {
          return nextDigit(code, digit + 1);
        } else {
          throw failure;
        }
      });
  }
  return nextDigit("", 0);
}
```

O ponto de acesso tende a responder a requisições de autenticação inválidas em cerca de 20 milissegundos, então para segurança, esta função espera 50 milissegundos antes de expirar uma requisição.

```
crackPasscode("HANGAR 2").then(console.log);
// → 555555
```

Carla inclina a cabeça e suspira. Isso teria sido mais satisfatório se o código fosse um pouco mais difícil de adivinhar.

Funções async

Mesmo com *promises*, esse tipo de código assíncrono é irritante de escrever. *Promises* frequentemente precisam ser conectadas de maneiras verbosas e aparentemente arbitrárias. Para criar um *loop* assíncrono, Carla foi forçada a introduzir uma função recursiva.

O que a função de quebra de senha realmente faz é completamente linear — ela sempre espera que a ação anterior complete antes de iniciar a próxima. Em um modelo de programação síncrona, seria mais direto de expressar.

A boa notícia é que o JavaScript permite escrever código pseudossíncrono para descrever computação assíncrona. Uma função `async` retorna implicitamente uma *promise* e pode, em seu corpo, usar `await` em outras *promises* de uma maneira que *parece* síncrona.

Podemos reescrever `crackPasscode` assim:

```
async function crackPasscode(networkID) {
  for (let code = "";;) {
    for (let digit = 0;; digit++) {
      let newCode = code + digit;
      try {
        await withTimeout(joinWifi(networkID, newCode), 50);
        return newCode;
      } catch (failure) {
        if (failure == "Timed out") {
          code = newCode;
          break;
        } else if (digit == 9) {
          throw failure;
        }
      }
    }
  }
}
```

Essa versão mostra mais claramente a estrutura de *loop* duplo da função (o *loop* interno tenta os dígitos de 0 a 9 e o *loop* externo adiciona dígitos à senha).

Uma função `async` é marcada pela palavra `async` antes da palavra-chave `function`. Métodos também podem ser tornados `async` escrevendo `async` antes de seu nome. Quando tal função ou método é chamado, retorna uma *promise*. Assim que a função retorna algo, essa *promise* é resolvida. Se o corpo lança uma exceção, a *promise* é rejeitada.

Dentro de uma função `async`, a palavra `await` pode ser colocada na frente de uma expressão para esperar que uma *promise* resolva e só então continuar a

execução da função. Se a *promise* rejeitar, uma exceção é levantada no ponto do `await`.

Tal função não executa mais do início ao fim de uma só vez como uma função JavaScript regular. Em vez disso, pode ser *congelada* em qualquer ponto que tenha um `await` e retomada em um momento posterior.

Para a maioria do código assíncrono, essa notação é mais conveniente do que usar *promises* diretamente. Você ainda precisa de um entendimento de *promises*, já que em muitos casos ainda interagirá com elas diretamente. Mas ao conectá-las, funções `async` são geralmente mais agradáveis de escrever do que cadeias de chamadas `then`.

Geradores

Essa capacidade de funções serem pausadas e depois retomadas não é exclusiva de funções `async`. O JavaScript também tem um recurso chamado funções *geradoras*. Essas são semelhantes, mas sem as *promises*.

Quando você define uma função com `function*` (colocando um asterisco após a palavra `function`), ela se torna um gerador. Quando você chama um gerador, ele retorna um iterador, que já vimos no Capítulo 6.

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}

for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

Inicialmente, quando você chama `powers`, a função é congelada em seu início. Toda vez que você chama `next` no iterador, a função executa até atingir uma expressão `yield`, que a pausa e faz com que o valor *yielded* se torne o próximo valor produzido pelo iterador. Quando a função retorna (a do exemplo nunca retorna), o iterador está concluído.

Escrever iteradores é frequentemente muito mais fácil quando se usam funções geradoras. O iterador para a classe `Group` (do exercício no Capítulo 6) pode ser escrito com este gerador:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

Não há mais necessidade de criar um objeto para armazenar o estado da iteração — geradores automaticamente salvam seu estado local toda vez que fazem *yield*.

Tais expressões `yield` podem ocorrer apenas diretamente na própria função geradora e não em uma função interna que você define dentro dela. O estado que um gerador salva ao fazer *yield* é apenas seu ambiente *local* e a posição onde fez *yield*.

Uma função `async` é um tipo especial de gerador. Ela produz uma *promise* quando chamada, que é resolvida quando retorna (termina) e rejeitada quando lança uma exceção. Sempre que faz *yield* (espera) uma *promise*, o resultado dessa *promise* (valor ou exceção lançada) é o resultado da expressão `await`.

Um Projeto de Arte Corvídea

Uma manhã, Carla acorda com um barulho desconhecido do asfalto do lado de fora de seu hangar. Pulando para a borda do telhado, ela vê que os humanos estão montando algo. Há muita fiação elétrica, um palco e algum tipo de grande parede preta sendo construída.

Sendo uma corvo curiosa, Carla examina a parede mais de perto. Ela parece consistir em vários dispositivos grandes com vidro frontal conectados a cabos. Na parte de trás, os dispositivos dizem “LedTec SIG-5030”.

Uma rápida pesquisa na internet revela um manual do usuário para esses dispositivos. Eles parecem ser placas de trânsito, com uma matriz programável de LEDs âmbar. A intenção dos humanos é provavelmente exibir algum tipo de informação neles durante seu evento. Curiosamente, as telas podem ser programadas por uma rede sem fio. Será que estão conectadas à rede local do prédio?

Cada dispositivo em uma rede recebe um *endereço IP*, que outros dispositivos podem usar para enviar mensagens a ele. Falamos mais sobre isso no Capítulo 13. Carla percebe que seus próprios telefones recebem endereços como `10.0.0.20` ou `10.0.0.33`. Pode valer a pena tentar enviar mensagens para todos esses endereços e ver se algum responde à interface descrita no manual das placas.

O Capítulo 18 mostra como fazer requisições reais em redes reais. Neste capítulo, usaremos uma função simplificada fictícia chamada `request` para co-

municação de rede. Esta função recebe dois argumentos — um endereço de rede e uma mensagem, que pode ser qualquer coisa que possa ser enviada como JSON — e retorna uma *promise* que resolve para uma resposta da máquina no endereço dado, ou rejeita se houve um problema.

De acordo com o manual, você pode mudar o que é exibido em uma placa SIG-5030 enviando uma mensagem com conteúdo como `{"command": "display", "data": [0, 0, 3, …]}`, onde `data` contém um número por ponto LED, fornecendo seu brilho — 0 significa desligado, 3 significa brilho máximo. Cada placa tem 50 luzes de largura e 30 de altura, então um comando de atualização deve enviar 1.500 números.

Este código envia uma mensagem de atualização de exibição para todos os endereços na rede local, para ver o que pega. Cada um dos números em um endereço IP pode ir de 0 a 255. Nos dados enviados, ele ativa um número de luzes correspondente ao último número do endereço de rede.

```
for (let addr = 1; addr < 256; addr++) {
  let data = [];
  for (let n = 0; n < 1500; n++) {
    data.push(n < addr ? 3 : 0);
  }
  let ip = `10.0.0.${addr}`;
  request(ip, {command: "display", data})
    .then(() => console.log(`Request to ${ip} accepted`))
    .catch(() => {});
}
```

Como a maioria desses endereços não existirá ou não aceitará tais mensagens, a chamada `catch` garante que erros de rede não travem o programa. As requisições são todas enviadas imediatamente, sem esperar que outras terminem, para não desperdiçar tempo quando algumas máquinas não respondem.

Tendo disparado sua varredura de rede, Carla volta para fora para ver o resultado. Para sua alegria, todas as telas agora mostram uma faixa de luz em seus cantos superiores esquerdos. Elas *estão* na rede local e *aceitam* comandos. Ela rapidamente anota os números mostrados em cada tela. Há nove telas, dispostas em três de altura por três de largura. Elas têm os seguintes endereços de rede:

```
const screenAddresses = [
  "10.0.0.44", "10.0.0.45", "10.0.0.41",
  "10.0.0.31", "10.0.0.40", "10.0.0.42",
  "10.0.0.48", "10.0.0.47", "10.0.0.46"
];
```

Agora isso abre possibilidades para todo tipo de travessura. Ela poderia mostrar "corvos mandam, humanos babam" na parede em letras gigantes. Mas isso parece um pouco grosseiro. Em vez disso, ela planeja mostrar um vídeo de uma corvo voando cobrindo todas as telas à noite.

Carla encontra um clipe de vídeo adequado, no qual um segundo e meio de filmagem pode ser repetido para criar um vídeo em *loop* mostrando a batida de asa de uma corvo. Para caber nas nove telas (cada uma podendo mostrar 50x30 pixels), Carla corta e redimensiona os vídeos para obter uma série de imagens de 150x90, 10 por segundo. Cada uma é então cortada em nove retângulos e processada de modo que os pontos escuros no vídeo (onde a corvo está) mostrem uma luz brilhante, e os pontos claros (sem corvo) fiquem escuros, o que deve criar o efeito de uma corvo âmbar voando contra um fundo preto.

Ela configurou a variável `clipImages` para conter um *array* de quadros, onde cada quadro é representado com um *array* de nove conjuntos de pixels — um para cada tela — no formato que as placas esperam.

Para exibir um único quadro do vídeo, Carla precisa enviar uma requisição para todas as telas ao mesmo tempo. Mas ela também precisa esperar pelo resultado dessas requisições, tanto para não começar a enviar o próximo quadro antes que o atual tenha sido adequadamente enviado quanto para perceber quando requisições estão falhando.

`Promise` tem um método estático `all` que pode ser usado para converter um *array* de *promises* em uma única *promise* que resolve para um *array* de resultados. Isso fornece uma maneira conveniente de ter algumas ações assíncronas acontecendo lado a lado, esperar que todas terminem e então fazer algo com seus resultados (ou pelo menos esperar por elas para garantir que não falhem).

```
function displayFrame(frame) {
  return Promise.all(frame.map((data, i) => {
    return request(screenAddresses[i], {
      command: "display",
      data
    });
  }));
}
```

Isso mapeia sobre as imagens em `frame` (que é um *array* de *arrays* de dados de exibição) para criar um *array* de *promises* de requisição. Então retorna uma *promise* que combina todas elas.

Para poder parar um vídeo em reprodução, o processo é envolvido em uma classe. Essa classe tem um método assíncrono `play` que retorna uma *promise* que resolve apenas quando a reprodução é parada novamente via o método

`stop`.

```
function wait(time) {
  return new Promise(accept => setTimeout(accept, time));
}

class VideoPlayer {
  constructor(frames, frameTime) {
    this.frames = frames;
    this.frameTime = frameTime;
    this.stopped = true;
  }

  async play() {
    this.stopped = false;
    for (let i = 0; !this.stopped; i++) {
      let nextFrame = wait(this.frameTime);
      await displayFrame(this.frames[i % this.frames.length]);
      await nextFrame;
    }
  }

  stop() {
    this.stopped = true;
  }
}
```

A função `wait` envolve `setTimeout` em uma *promise* que resolve após o número dado de milissegundos. Isso é útil para controlar a velocidade de reprodução.

```
let video = new VideoPlayer(clipImages, 100);
video.play().catch(e => {
  console.log("Playback failed: " + e);
});
setTimeout(() => video.stop(), 15000);
```

Durante toda a semana que a parede de telas fica de pé, toda noite, quando escurece, um enorme pássaro laranja brilhante misteriosamente aparece nela.

O loop de eventos

Um programa assíncrono começa executando seu script principal, que frequentemente configura *callbacks* para serem chamados depois. Esse script principal, assim como os *callbacks*, executam até o fim de uma só vez, sem interrupção. Mas entre eles, o programa pode ficar ocioso, esperando algo

acontecer.

Então *callbacks* não são chamados diretamente pelo código que os agendou. Se eu chamar `setTimeout` de dentro de uma função, aquela função já terá retornado no momento em que a função de *callback* é chamada. E quando o *callback* retorna, o controle não volta para a função que o agendou.

O comportamento assíncrono acontece em sua própria pilha de chamadas vazia. Esta é uma das razões pelas quais, sem *promises*, gerenciar exceções através de código assíncrono é tão difícil. Como cada *callback* começa com uma pilha praticamente vazia, seus handlers `catch` não estarão na pilha quando lançarem uma exceção.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (e) {
  // Isso não executará
  console.log("Caught", e);
}
```

Não importa quão próximos no tempo eventos — como timeouts ou requisições recebidas — aconteçam, um ambiente JavaScript executará apenas um programa por vez. Você pode pensar nisso como ele executando um grande *loop ao redor* do seu programa, chamado de *loop de eventos*. Quando não há nada a ser feito, esse *loop* é pausado. Mas conforme eventos chegam, eles são adicionados a uma fila, e seu código é executado um após o outro. Como nada executa ao mesmo tempo, código de execução lenta pode atrasar o tratamento de outros eventos.

Este exemplo define um timeout mas então demora além do momento pretendido do timeout, fazendo o timeout atrasar.

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

Promises sempre resolvem ou rejeitam como um novo evento. Mesmo se uma *promise* já estiver resolvida, esperar por ela fará com que seu *callback* execute após o script atual terminar, em vez de imediatamente.

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
// → Me first!
// → Done
```

Em capítulos posteriores veremos vários outros tipos de eventos que executam no *loop* de eventos.

Bugs assíncronos

Quando seu programa executa sincronamente, de uma só vez, não há mudanças de estado acontecendo exceto aquelas que o próprio programa faz. Para programas assíncronos isso é diferente — eles podem ter *lacunas* em sua execução durante as quais outro código pode executar.

Vejamos um exemplo. Esta é uma função que tenta reportar o tamanho de cada arquivo em um *array* de arquivos, garantindo que todos sejam lidos ao mesmo tempo em vez de em sequência.

```
async function fileSizes(files) {
  let list = "";
  await Promise.all(files.map(async fileName => {
    list += fileName + ": " +
      (await textFile(fileName)).length + "\n";
  }));
  return list;
}
```

A parte `async fileName =>` mostra como arrow functions também podem ser tornadas `async` colocando a palavra `async` na frente delas.

O código não parece imediatamente suspeito... ele mapeia a *arrow function* `async` sobre o *array* de nomes, criando um *array* de *promises*, e então usa `Promise.all` para esperar por todas antes de retornar a lista que constroem.

Mas esse programa está completamente quebrado. Ele sempre retornará apenas uma única linha de saída, listando o arquivo que levou mais tempo para ser lido.

Você consegue descobrir por quê?

O problema está no operador `+=`, que pega o valor *atual* de `list` no momento em que a declaração começa a executar e então, quando o `await` termina, define a *binding* `list` como aquele valor mais a *string* adicionada.

Mas entre o momento em que a declaração começa a executar e o momento em que termina, há uma lacuna assíncrona. A expressão `map` executa antes de qualquer coisa ter sido adicionada à lista, então cada um dos operadores

`+=` começa de uma *string* vazia e acaba, quando seu armazenamento termina, definindo `list` como o resultado de adicionar sua linha à *string* vazia.

Isso poderia ter sido facilmente evitado retornando as linhas das *promises* mapeadas e chamando `join` no resultado de `Promise.all`, em vez de construir a lista alterando uma *binding*. Como de costume, computar novos valores é menos propenso a erros do que alterar valores existentes.

```
async function fileSizes(files) {
  let lines = files.map(async fileName => {
    return fileName + ": " +
      (await textFile(fileName)).length;
  });
  return (await Promise.all(lines)).join("\n");
}
```

Erros como esse são fáceis de cometer, especialmente quando se usa `await`, e você deve estar ciente de onde estão as lacunas em seu código. Uma vantagem da assincronicidade *explícita* do JavaScript (seja através de *callbacks*, *promises* ou `await`) é que identificar essas lacunas é relativamente fácil.

Resumo

A programação assíncrona torna possível expressar a espera por ações de longa duração sem congelar o programa todo. Ambientes JavaScript tipicamente implementam esse estilo de programação usando *callbacks*, funções que são chamadas quando as ações completam. Um *loop* de eventos agenda tais *callbacks* para serem chamados quando apropriado, um após o outro, para que suas execuções não se sobreponham.

Programar assincronamente é facilitado por *promises*, objetos que representam ações que podem completar no futuro, e funções `async`, que permitem escrever um programa assíncrono como se fosse síncrono.

Exercícios

Tempos Tranquilos

Há uma câmera de segurança perto do laboratório de Carla que é ativada por um sensor de movimento. Ela está conectada à rede e começa a enviar um fluxo de vídeo quando está ativa. Como prefere não ser descoberta, Carla montou um sistema que percebe esse tipo de tráfego de rede sem fio e acende uma luz

em seu esconderijo sempre que há atividade do lado de fora, para que ela saiba quando ficar quieta.

Ela também tem registrado os horários em que a câmera é acionada por um tempo e quer usar essas informações para visualizar quais horários, em uma semana média, tendem a ser tranquilos e quais tendem a ser movimentados. O registro é armazenado em arquivos contendo um *timestamp* (como retornado por `Date.now()`) por linha.

```
1695709940692
1695701068331
1695701189163
```

O arquivo `"camera_logs.txt"` contém uma lista de arquivos de log. Escreva uma função assíncrona `activityTable(day)` que para um dado dia da semana retorna um *array* de 24 números, um para cada hora do dia, que contém o número de observações de tráfego de rede da câmera vistas naquela hora do dia. Dias são identificados por número usando o sistema de `Date.getDay`, onde domingo é 0 e sábado é 6.

A função `activityGraph`, fornecida pela *sandbox*, resume tal tabela em uma *string*.

Para ler os arquivos, use a função `textFile` definida anteriormente — dado um nome de arquivo, ela retorna uma *promise* que resolve para o conteúdo do arquivo. Lembre-se de que `new Date(timestamp)` cria um objeto `Date` para aquele momento, que tem métodos `getDay` e `getHours` retornando o dia da semana e a hora do dia.

Ambos os tipos de arquivos — a lista de arquivos de log e os arquivos de log em si — têm cada dado em sua própria linha, separados por caracteres de nova linha (`"\n"`).

Promises Reais

Reescreva a função do exercício anterior sem `async`/`await`, usando métodos simples de `Promise`.

Neste estilo, usar `Promise.all` será mais conveniente do que tentar modelar um *loop* sobre os arquivos de log. Na função `async`, simplesmente usar `await` em um *loop* é mais simples. Se ler um arquivo leva algum tempo, qual dessas duas abordagens levará menos tempo para executar?

Se um dos arquivos listados na lista de arquivos tiver um erro de digitação e a leitura falhar, como essa falha acaba no objeto `Promise` que sua função retorna?

Construindo Promise.all

Como vimos, dado um *array* de promises, `Promise.all` retorna uma *promise* que espera todas as *promises* no *array* terminarem. Ela então tem sucesso, produzindo um *array* de valores resultado. Se uma *promise* no *array* falhar, a *promise* retornada por `all` também falha, passando adiante a razão da falha da *promise* que falhou.

Implemente algo assim você mesmo como uma função regular chamada `Promise_all`
.

Lembre-se de que depois que uma *promise* tem sucesso ou falha, ela não pode ter sucesso ou falhar novamente, e chamadas adicionais às funções que a resolvem são ignoradas. Isso pode simplificar a maneira como você lida com a falha da sua *promise*.

"The evaluator, which determines the meaning of expressions in a programming language, is just another program."

—Hal Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs

Chapter 12

Projeto: Uma Linguagem de Programação

Construir sua própria linguagem de programação é surpreendentemente fácil (desde que você não mire muito alto) e muito esclarecedor.

A principal coisa que quero mostrar neste capítulo é que não há nenhuma mágica envolvida na construção de uma linguagem de programação. Muitas vezes senti que algumas invenções humanas eram tão imensamente inteligentes e complicadas que eu nunca seria capaz de entendê-las. Mas com um pouco de leitura e experimentação, elas frequentemente se revelam bastante comuns.

Vamos construir uma linguagem de programação chamada Egg. Será uma linguagem minúscula e simples — mas poderosa o suficiente para expressar qualquer computação que você possa imaginar. Ela permitirá abstração simples baseada em funçãoões.

Análise Sintática

A parte mais imediatamente visível de uma linguagem de programação é sua *sintaxe*, ou notação. Um *parser* é um programa que lê um trecho de texto e produz uma estrutura de dados que reflete a estrutura do programa contido naquele texto. Se o texto não formar um programa válido, o parser deve apontar o erro.

Nossa linguagem terá uma sintaxe simples e uniforme. Tudo em Egg é uma expressão. Uma expressão pode ser o nome de uma vinculação, um número, uma string, ou uma *aplicação*. Aplicações são usadas para chamadas de função, mas também para construções como `if` ou `while`.

Para manter o parser simples, strings em Egg não suportam nada como sequências de escape com barra invertida. Uma string é simplesmente uma sequência de caracteres que não são aspas duplas, envolvida por aspas duplas. Um número é uma sequência de dígitos. Nomes de vinculações podem consistir de qualquer caractere que não seja espaço em branco e que não tenha um significado especial na sintaxe.

Aplicações são escritas da mesma forma que em JavaScript, colocando parên-

teses após uma expressão e tendo qualquer número de argumentos entre esses parênteses, separados por vírgulas.

```
do(define(x, 10),
   if(>(x, 5),
      print("large"),
      print("small")))
```

A uniformidade da linguagem Egg significa que coisas que são operadores em JavaScript (como `>`) são vinculações normais nesta linguagem, aplicadas assim como outras funçãoões. Como a sintaxe não tem o conceito de bloco, precisamos de uma construção `do` para representar a execução de múltiplas coisas em sequência.

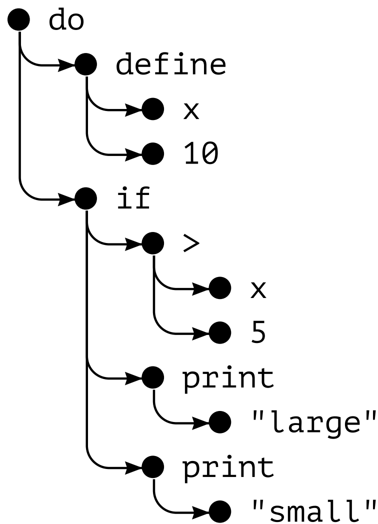
A estrutura de dados que o parser usará para descrever um programa consiste em objetos de expressão, cada um com uma propriedade `type` indicando o tipo de expressão, e outras propriedades para descrever seu conteúdo.

Expressões do tipo `"value"` representam strings ou números literais. Sua propriedade `value` contém o valor da string ou número que representam. Expressões do tipo `"word"` são usadas para identificadores (nomes). Tais objetos têm uma propriedade `name` que contém o nome do identificador como uma string. Finalmente, expressões `"apply"` representam aplicações. Elas têm uma propriedade `operator` que se refere à expressão que está sendo aplicada, assim como uma propriedade `args` que contém um array de expressões de argumento.

A parte `>(x, 5)` do programa anterior seria representada assim:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

Tal estrutura de dados é chamada de *árvore sintática*. Se você imaginar os objetos como pontos e as ligações entre eles como linhas entre esses pontos, conforme mostrado no diagrama a seguir, a estrutura tem uma forma de árvore. O fato de que expressões contêm outras expressões, que por sua vez podem conter mais expressões, é similar à maneira como galhos de árvores se dividem e se dividem novamente.



Compare isso com o parser que escrevemos para o formato de arquivo de configuração no Capítulo 9, que tinha uma estrutura simples: ele dividia a entrada em linhas e tratava essas linhas uma de cada vez. Havia apenas algumas formas simples que uma linha podia ter.

Aqui precisamos encontrar uma abordagem diferente. Expressões não são separadas em linhas e têm uma estrutura recursiva. Expressões de aplicação *contêm* outras expressões.

Felizmente, esse problema pode ser resolvido muito bem escrevendo uma função de parser que é recursiva de uma forma que reflete a natureza recursiva da linguagem.

Definimos uma função `parseExpression` que recebe uma string como entrada. Ela retorna um objeto contendo a estrutura de dados para a expressão no início da string, junto com a parte da string restante após a análise dessa expressão. Ao analisar subexpressões (o argumento de uma aplicação, por exemplo), essa função pode ser chamada novamente, produzindo a expressão do argumento assim como o texto que resta. Esse texto pode, por sua vez, conter mais argumentos ou ser o parêntese de fechamento que termina a lista de argumentos.

Esta é a primeira parte do parser:

```
function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^"([^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^\d+\b/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
  } else if (match = /^[^\s(),#"]+/.exec(program)) {
```

```
    expr = {type: "word", name: match[0]};
  } else {
    throw new SyntaxError("Unexpected syntax: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Como Egg, assim como JavaScript, permite qualquer quantidade de espaço em branco entre seus elementos, temos que cortar repetidamente o espaço em branco do início da string do programa. A função `skipSpace` ajuda com isso.

Depois de pular qualquer espaço inicial, `parseExpression` usa três expressões regulares para identificar os três elementos atômicos que Egg suporta: strings, números e palavras. O parser constrói um tipo diferente de estrutura de dados dependendo de qual expressão corresponde. Se a entrada não corresponder a nenhuma dessas três formas, não é uma expressão válida, e o parser lança um erro. Usamos o construtor `SyntaxError` aqui. Esta é uma classe de exceção definida pelo padrão, como `Error`, mas mais específica.

Em seguida, cortamos a parte que correspondeu da string do programa e passamos isso, junto com o objeto da expressão, para `parseApply`, que verifica se a expressão é uma aplicação. Se for, ela analisa uma lista de argumentos entre parênteses.

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    let arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",") {
      program = skipSpace(program.slice(1));
    } else if (program[0] != ")") {
```

```
      throw new SyntaxError("Expected ',' or ')'");
    }
  }
  return parseApply(expr, program.slice(1));
}
```

Se o próximo caractere no programa não for um parêntese de abertura, isso não é uma aplicação, e `parseApply` retorna a expressão que recebeu. Caso contrário, ela pula o parêntese de abertura e cria o objeto de árvore sintática para essa expressão de aplicação. Em seguida, chama recursivamente `parseExpression` para analisar cada argumento até que um parêntese de fechamento seja encontrado. A recursão é indireta, através de `parseApply` e `parseExpression` chamando uma à outra.

Como uma expressão de aplicação pode ela mesma ser aplicada (como em `multiplier(2)(1)`), `parseApply` deve, após ter analisado uma aplicação, chamar a si mesma novamente para verificar se outro par de parênteses segue.

Isso é tudo que precisamos para analisar Egg. Encapsulamos isso em uma função conveniente `parse` que verifica que alcançamos o final da string de entrada após analisar a expressão (um programa Egg é uma única expressão), e que nos dá a estrutura de dados do programa.

```
function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Unexpected text after program");
  }
  return expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

Funciona! Não nos dá informações muito úteis quando falha e não armazena a linha e coluna onde cada expressão começa, o que poderia ser útil ao reportar erros depois, mas é bom o suficiente para nossos propósitos.

O avaliador

O que podemos fazer com a árvore sintática de um programa? Executá-la, é claro! E é isso que o avaliador faz. Você dá a ele uma árvore sintática e um

objeto de escopo que associa nomes a valores, e ele avaliará a expressão que a árvore representa e retornará o valor que isso produz.

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

O avaliador tem código para cada um dos tipos de expressão. Uma expressão de valor literal produz seu valor. (Por exemplo, a expressão `100` avalia para o número 100.) Para uma vinculação, devemos verificar se ela está realmente definida no escopo e, se estiver, buscar o valor da vinculação.

Aplicações são mais envolvidas. Se são uma forma especial, como `if`, não avaliamos nada — apenas passamos as expressões dos argumentos, junto com o escopo, para a função que trata essa forma. Se é uma chamada normal, avaliamos o operador, verificamos que é uma função e a chamamos com os argumentos avaliados.

Usamos valores de função JavaScript simples para representar os valores de função de Egg. Voltaremos a isso mais adiante, quando a forma especial `fun` for definida.

A estrutura recursiva de `evaluate` se assemelha à estrutura do parser, e

ambas espelham a estrutura da própria linguagem. Também seria possível combinar o parser e o avaliador em uma única função e avaliar durante a análise, mas separá-los dessa forma torna o programa mais claro e flexível.

Isso é realmente tudo o que é necessário para interpretar Egg. É simples assim. Mas sem definir algumas formas especiais e adicionar alguns valores úteis ao ambiente, você não pode fazer muita coisa com essa linguagem ainda.

Formas especiais

O objeto `specialForms` é usado para definir sintaxe especial em Egg. Ele associa palavras a funções que avaliam tais formas. Atualmente está vazio. Vamos adicionar `if`.

```
specialForms.if = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

A construção `if` de Egg espera exatamente três argumentos. Ela avaliará o primeiro e, se o resultado não for o valor `false`, avaliará o segundo. Caso contrário, o terceiro é avaliado. Essa forma `if` é mais similar ao operador ternário `?:` de JavaScript do que ao `if` de JavaScript. É uma expressão, não uma instrução, e produz um valor — a saber, o resultado do segundo ou terceiro argumento.

Egg também difere de JavaScript na forma como trata o valor da condição para `if`. Ele tratará apenas o valor `false` como falso, não coisas como zero ou a string vazia.

A razão pela qual precisamos representar `if` como uma forma especial em vez de uma função regular é que todos os argumentos de funções são avaliados antes que a função seja chamada, enquanto `if` deve avaliar apenas *ou* seu segundo *ou* seu terceiro argumento, dependendo do valor do primeiro.

A forma `while` é similar.

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
```

```
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Como undefined não existe em Egg, retornamos false,
  // por falta de um resultado significativo
  return false;
};
```

Outro bloco de construção básico é `do`, que executa todos os seus argumentos de cima para baixo. Seu valor é o valor produzido pelo último argumento.

```
specialForms.do = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};
```

Para poder criar vinculações e dar-lhes novos valores, também criamos uma forma chamada `define`. Ela espera uma palavra como seu primeiro argumento e uma expressão que produz o valor a ser atribuído àquela palavra como seu segundo argumento. Como `define`, como tudo mais, é uma expressão, ela deve retornar um valor. Faremos com que retorne o valor que foi atribuído (assim como o operador = de JavaScript).

```
specialForms.define = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Incorrect use of define");
  }
  let value = evaluate(args[1], scope);
  scope[args[0].name] = value;
  return value;
};
```

O ambiente

O escopo aceito por `evaluate` é um objeto com propriedades cujos nomes correspondem a nomes de vinculações e cujos valores correspondem aos valores a que essas vinculações estão ligadas. Vamos definir um objeto para representar o escopo global.

Para poder usar a construção `if` que acabamos de definir, devemos ter acesso

a valores Booleanos. Como existem apenas dois valores booleanos, não precisamos de sintaxe especial para eles. Simplesmente vinculamos dois nomes aos valores `true` e `false` e os usamos.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

Agora podemos avaliar uma expressão simples que nega um valor booleano.

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

Para fornecer operadores básicos de aritmética e comparação, também adicionaremos alguns valores de função ao escopo. No interesse de manter o código curto, usaremos `Function` para sintetizar um conjunto de funções de operador em um loop, em vez de defini-las individualmente.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

Também é útil ter uma forma de imprimir valores, então encapsularemos `console.log` em uma função e a chamaremos de `print`.

```
topScope.print = value => {
  console.log(value);
  return value;
};
```

Isso nos dá ferramentas elementares suficientes para escrever programas simples. A função a seguir fornece uma forma conveniente de analisar um programa e executá-lo em um escopo novo:

```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

Usaremos cadeias de protótipos de objetos para representar escopos aninhados, de forma que o programa possa adicionar vinculações ao seu escopo local sem alterar o escopo de nível superior.

```
run(`
do(define(total, 0),
   define(count, 1),
```

```
  while(<(count, 11),
        do(define(total, +(total, count)),
           define(count, +(count, 1)))),
  print(total))
`);
// → 55
```

Este é o programa que já vimos várias vezes antes, que calcula a soma dos números de 1 a 10, expresso em Egg. É claramente mais feio do que o programa JavaScript equivalente — mas nada mal para uma linguagem implementada em menos de 150 linhas de código.

Funções

Uma linguagem de programação sem funções é uma linguagem de programação pobre, de fato. Felizmente, não é difícil adicionar uma construção `fun`, que trata seu último argumento como o corpo da função e usa todos os argumentos antes dele como os nomes dos parâmetros da função.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
    return expr.name;
  });

  return function(...args) {
    if (args.length != params.length) {
      throw new TypeError("Wrong number of arguments");
    }
    let localScope = Object.create(scope);
    for (let i = 0; i < args.length; i++) {
      localScope[params[i]] = args[i];
    }
    return evaluate(body, localScope);
  };
};
```

Funções em Egg obtêm seu próprio escopo local. A função produzida pela

forma `fun` cria esse escopo local e adiciona as vinculações dos argumentos a ele. Em seguida, avalia o corpo da função nesse escopo e retorna o resultado.

```
run(`
do(define(plusOne, fun(a, +(a, 1))),
   print(plusOne(10)))
`);
// → 11

run(`
do(define(pow, fun(base, exp,
     if(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1)))))),
   print(pow(2, 10)))
`);
// → 1024
```

Compilação

O que construímos é um interpretador. Durante a avaliação, ele atua diretamente sobre a representação do programa produzida pelo parser.

Compilação é o processo de adicionar outro passo entre a análise e a execução de um programa, que transforma o programa em algo que pode ser avaliado mais eficientemente fazendo o máximo de trabalho possível antecipadamente. Por exemplo, em linguagens bem projetadas, é óbvio, para cada uso de uma vinculação, a qual vinculação se está referindo, sem realmente executar o programa. Isso pode ser usado para evitar procurar a vinculação pelo nome toda vez que ela é acessada, buscando-a diretamente de alguma localização de memória predeterminada.

Tradicionalmente, a compilação envolve converter o programa em código de máquina, o formato bruto que o processador de um computador pode executar. Mas qualquer processo que converta um programa em uma representação diferente pode ser considerado compilação.

Seria possível escrever uma estratégia de avaliação alternativa para Egg, uma que primeiro converta o programa em um programa JavaScript, use `Function` para invocar o compilador JavaScript nele, e execute o resultado. Quando feito corretamente, isso faria Egg rodar muito rápido, enquanto ainda seria bastante simples de implementar.

Se você está interessado nesse tópico e disposto a dedicar algum tempo a isso, encorajo você a tentar implementar tal compilador como um exercício.

Trapaceando

Quando definimos `if` e `while`, você provavelmente notou que eram wrappers mais ou menos triviais em torno do próprio `if` e `while` de JavaScript. Da mesma forma, os valores em Egg são apenas valores JavaScript comuns. Preencher a lacuna até um sistema mais primitivo, como o código de máquina que o processador entende, requer mais esforço — mas a forma como funciona se assemelha ao que estamos fazendo aqui.

Embora a linguagem de brinquedo neste capítulo não faça nada que não poderia ser feito melhor em JavaScript, *existem* situações em que escrever pequenas linguagens ajuda a realizar trabalho real.

Tal linguagem não precisa se parecer com uma linguagem de programação típica. Se JavaScript não viesse equipado com expressões regulares, por exemplo, você poderia escrever seu próprio parser e avaliador para expressões regulares.

Ou imagine que você está construindo um programa que torna possível criar parsers rapidamente fornecendo uma descrição lógica da linguagem que eles precisam analisar. Você poderia definir uma notação específica para isso, e um compilador que a compila em um programa de parser.

```
expr = number | string | name | application

number = digit+

name = letter+

string = '"' (! '"')* '"'

application = expr '(' (expr (',' expr)*)? ')'
```

Isso é o que normalmente se chama de *linguagem de domínio específico*, uma linguagem feita sob medida para expressar um domínio estreito de conhecimento. Tal linguagem pode ser mais expressiva do que uma linguagem de propósito geral porque é projetada para descrever exatamente as coisas que precisam ser descritas em seu domínio e nada mais.

Exercícios

Arrays

Adicione suporte para arrays em Egg adicionando as três funções a seguir ao escopo global: `array(...values)` para construir um array contendo os val-

ores dos argumentos, `length(array)` para obter o comprimento de um array, e `element(array, n)` para buscar o *n*-ésimo elemento de um array.

Closure

A forma como definimos `fun` permite que funções em Egg referenciem o escopo ao redor, permitindo que o corpo da função use valores locais que eram visíveis no momento em que a função foi definida, assim como funções JavaScript fazem.

O programa a seguir ilustra isso: a função `f` retorna uma função que adiciona seu argumento ao argumento de `f`, o que significa que ela precisa de acesso ao escopo local dentro de `f` para poder usar a vinculação `a`.

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
   print(f(4)(5)))
`);
// → 9
```

Volte à definição da forma `fun` e explique qual mecanismo faz isso funcionar.

Comentários

Seria bom se pudéssemos escrever comentários em Egg. Por exemplo, sempre que encontrarmos um sinal de cerquilha (`#`), poderíamos tratar o resto da linha como um comentário e ignorá-lo, similar ao `//` em JavaScript.

Não precisamos fazer grandes mudanças no parser para suportar isso. Podemos simplesmente mudar `skipSpace` para pular comentários como se fossem espaço em branco, de forma que todos os pontos onde `skipSpace` é chamado agora também pularão comentários. Faça essa mudança.

Corrigindo o escopo

Atualmente, a única forma de atribuir um valor a uma vinculação é `define`. Essa construção age como uma forma tanto de definir novas vinculações quanto de dar a vinculações existentes um novo valor.

Essa ambiguidade causa um problema. Quando você tenta dar a uma vinculação não-local um novo valor, você acaba definindo uma local com o mesmo nome em vez disso. Algumas linguagens funcionam assim por design, mas sempre achei essa uma forma estranha de lidar com escopo.

Adicione uma forma especial `set`, similar a `define`, que dá a uma vinculação um novo valor, atualizando a vinculação em um escopo externo se ela não existir

no escopo interno. Se a vinculação não estiver definida de forma alguma, lance um `ReferenceError` (outro tipo de erro padrão).

A técnica de representar escopos como objetos simples, que tem tornado as coisas convenientes até agora, vai atrapalhar um pouco neste ponto. Você pode querer usar a função `Object.getPrototypeOf`, que retorna o protótipo de um objeto. Lembre-se também que você pode usar `Object.hasOwn` para descobrir se um dado objeto possui uma propriedade.

"The dream behind the web is of a common information space in which we communicate by sharing information. Its universality is essential: the fact that a hypertext link can point to anything, be it personal, local or global, be it draft or highly polished."

—Tim Berners-Lee, The World Wide Web: A Very Short Personal History

Chapter 13

JavaScript e o Navegador

Os próximos capítulos deste livro discutirão navegadores web. Sem navegadores, não haveria JavaScript — ou se houvesse, ninguém jamais teria prestado atenção nele.

A tecnologia web tem sido descentralizada desde o início, não apenas tecnicamente, mas também na forma como evoluiu. Vários fabricantes de navegadores adicionaram novas funcionalidades de forma improvisada e às vezes mal pensada, as quais foram então — às vezes — adotadas por outros, e finalmente estabelecidas em padrões.

Isso é tanto uma bênção quanto uma maldição. Por um lado, é empoderador não ter uma entidade central controlando um sistema, mas tê-lo sendo melhorado por diversas partes trabalhando em colaboração frouxa (ou ocasionalmente, hostilidade aberta). Por outro lado, a forma desordenada como a web foi desenvolvida significa que o sistema resultante não é exatamente um exemplo brilhante de consistência interna. Algumas partes são francamente confusas e mal projetadas.

Redes e a Internet

Redes de computadores existem desde os anos 1950. Se você colocar cabos entre dois ou mais computadores e permitir que eles enviem dados de um lado para outro através desses cabos, você pode fazer todo tipo de coisas maravilhosas.

Se conectar duas máquinas no mesmo prédio nos permite fazer coisas maravilhosas, conectar máquinas por todo o planeta deve ser ainda melhor. A tecnologia para começar a implementar essa visão foi desenvolvida nos anos 1980, e a rede resultante é chamada de *internet*. Ela correspondeu à sua promessa.

Um computador pode usar essa rede para enviar bits a outro computador. Para que qualquer comunicação efetiva surja desse envio de bits, os computadores em ambos os lados devem saber o que os bits devem representar. O significado de qualquer sequência de bits depende inteiramente do tipo de coisa que está tentando expressar e do mecanismo de codificação usado.

Um *protocolo de rede* descreve um estilo de comunicação sobre uma rede. Existem protocolos para enviar e-mail, para buscar e-mail, para compartilhar arquivos, e até para controlar computadores que foram infectados por software malicioso.

O *Protocolo de Transferência de Hipertexto* (HTTP) é um protocolo para recuperar recursos nomeados (pedaços de informação, como páginas web ou imagens). Ele especifica que o lado que faz a requisição deve começar com uma linha como esta, nomeando o recurso e a versão do protocolo que está tentando usar:

```
GET /index.html HTTP/1.1
```

Existem muito mais regras sobre como o requisitante pode incluir mais informações na requisição e como o outro lado, que retorna o recurso, empacota seu conteúdo. Veremos o HTTP com um pouco mais de detalhe no Capítulo 18.

A maioria dos protocolos é construída sobre outros protocolos. O HTTP trata a rede como um dispositivo semelhante a um fluxo no qual você pode colocar bits e fazer com que cheguem ao destino correto na ordem correta. Fornecer essas garantias sobre o envio primitivo de dados que a rede oferece já é um problema bastante complicado.

O *Protocolo de Controle de Transmissão* (TCP) é um protocolo que resolve esse problema. Todos os dispositivos conectados à internet o "falam", e a maior parte da comunicação na internet é construída sobre ele.

Uma conexão TCP funciona da seguinte forma: um computador deve estar esperando, ou *escutando*, que outros computadores comecem a falar com ele. Para poder escutar diferentes tipos de comunicação ao mesmo tempo em uma única máquina, cada ouvinte tem um número (chamado de *porta*) associado a ele. A maioria dos protocolos especifica qual porta deve ser usada por padrão. Por exemplo, quando queremos enviar um e-mail usando o protocolo SMTP, a máquina pela qual o enviamos deve estar escutando na porta 25.

Outro computador pode então estabelecer uma conexão conectando-se à máquina alvo usando o número de porta correto. Se a máquina alvo puder ser alcançada e estiver escutando naquela porta, a conexão é criada com sucesso. O computador que escuta é chamado de *servidor*, e o computador que se conecta é chamado de *cliente*.

Tal conexão funciona como um tubo de mão dupla por onde bits podem fluir — as máquinas em ambos os lados podem colocar dados nele. Uma vez que os bits são transmitidos com sucesso, eles podem ser lidos novamente pela máquina do outro lado. Este é um modelo conveniente. Pode-se dizer que o TCP fornece uma abstração da rede.

A Web

A *World Wide Web* (que não deve ser confundida com a internet como um todo) é um conjunto de protocolos e formatos que nos permite visitar páginas web em um navegador. A palavra *Web* se refere ao fato de que tais páginas podem facilmente se vincular umas às outras, conectando-se assim em uma enorme malha pela qual os usuários podem navegar.

Para se tornar parte da web, tudo que você precisa fazer é conectar uma máquina à internet e fazê-la escutar na porta 80 com o protocolo HTTP para que outros computadores possam pedir documentos a ela.

Cada documento na web é nomeado por um *localizador uniforme de recursos* (URL), que se parece com algo assim:

```
  http://eloquentjavascript.net/13_browser.html
 |      |                      |               |
 protocolo     servidor               caminho
```

A primeira parte nos diz que essa URL usa o protocolo HTTP (ao contrário, por exemplo, de HTTP criptografado, que seria *https://*). Em seguida vem a parte que identifica de qual servidor estamos solicitando o documento. Por último vem uma string de caminho que identifica o documento (ou *recurso*) em que estamos interessados.

Máquinas conectadas à internet recebem um *endereço IP*, um número que pode ser usado para enviar mensagens àquela máquina, e se parece com algo como `149.210.142.219` ou `2001:4860:4860::8888`. Como listas de números mais ou menos aleatórios são difíceis de lembrar e estranhas de digitar, você pode registrar um nome de *domínio* para um endereço ou conjunto de endereços. Registrei *eloquentjavascript.net* para apontar para o endereço IP de uma máquina que controlo e posso assim usar esse nome de domínio para servir páginas web.

Se você digitar essa URL na barra de endereço do seu navegador, o navegador tentará recuperar e exibir o documento naquela URL. Primeiro, seu navegador precisa descobrir a que endereço *eloquentjavascript.net* se refere. Então, usando o protocolo HTTP, ele fará uma conexão com o servidor naquele endereço e pedirá o recurso */13_browser.html*. Se tudo correr bem, o servidor enviará de volta um documento, que seu navegador então exibirá na sua tela.

HTML

HTML, que significa Linguagem de Marcação de Hipertexto, é o formato de documento usado para páginas web. Um documento HTML contém texto,

assim como *tags* que dão estrutura ao texto, descrevendo coisas como links, parágrafos e cabeçalhos.

Um documento HTML curto pode se parecer com isto:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

É assim que tal documento ficaria no navegador:

My home page

Hello, I am Marijn and this is my home page.

I also wrote a book! Read it here.

As tags, envolvidas em colchetes angulares (< e >, os símbolos de *menor que* e *maior que*), fornecem informações sobre a estrutura do documento. O outro texto é apenas texto simples.

O documento começa com <!doctype html>, que diz ao navegador para interpretar a página como HTML *moderno*, em oposição a estilos obsoletos usados no passado.

Documentos HTML têm um cabeçalho (head) e um corpo (body). O cabeçalho contém informações *sobre* o documento, e o corpo contém o documento em si. Neste caso, o cabeçalho declara que o título deste documento é "My home page" e que usa a codificação UTF-8, que é uma forma de codificar texto Unicode como dados binários. O corpo do documento contém um cabeçalho (<h1>, significando "cabeçalho 1" — <h2> a <h6> produzem subcabeçalhos) e dois parágrafos (<p>).

Tags vêm em diversas formas. Um elemento, como o corpo, um parágrafo ou um link, é iniciado por uma *tag de abertura* como <p> e terminado por uma *tag de fechamento* como </p>. Algumas tags de abertura, como a de link (<a>), contêm informações extras na forma de pares nome="valor". Estes são

chamados de *atributos*. Neste caso, o destino do link é indicado com href="http://eloquentjavascript.net", onde href significa “referência de hipertexto”.

Alguns tipos de tags não envolvem nada e portanto não precisam ser fechadas. A tag de metadados <meta charset="utf-8"> é um exemplo disso.

Para poder incluir colchetes angulares no texto de um documento, mesmo que eles tenham um significado especial em HTML, mais uma forma de notação especial precisa ser introduzida. Um colchete angular de abertura simples é escrito como < (“menor que”), e um de fechamento como > (“maior que”). Em HTML, um caractere de e-comercial (&) seguido por um nome ou código de caractere e um ponto e vírgula (;) é chamado de *entidade* e será substituído pelo caractere que codifica.

Isso é análogo à forma como barras invertidas são usadas em strings JavaScript. Como esse mecanismo dá aos caracteres de e-comercial um significado especial também, eles precisam ser escapados como &. Dentro de valores de atributos, que são envolvidos em aspas duplas, " pode ser usado para inserir um caractere de aspas literal.

HTML é analisado de forma notavelmente tolerante a erros. Quando tags que deveriam estar lá estão faltando, o navegador as adiciona automaticamente. A forma como isso é feito foi padronizada, e você pode confiar que todos os navegadores modernos farão da mesma forma.

O documento a seguir será tratado da mesma forma que o mostrado anteriormente:

```
<!doctype html>

<meta charset=utf-8>
<title>My home page</title>

<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentjavascript.net>here</a>.
```

As tags <html>, <head> e <body> sumiram completamente. O navegador sabe que <meta> e <title> pertencem ao cabeçalho e que <h1> significa que o corpo começou. Além disso, não estou mais fechando explicitamente os parágrafos, já que abrir um novo parágrafo ou terminar o documento os fechará implicitamente. As aspas em torno dos valores dos atributos também foram removidas.

Este livro geralmente omitirá as tags <html>, <head> e <body> dos exemplos para mantê-los curtos e livres de desordem. Mas eu *vou* fechar tags e incluir aspas em torno de atributos.

Também geralmente omitirei a declaração de doctype e de charset. Não

tome isso como incentivo para removê-las de documentos HTML. Navegadores frequentemente farão coisas ridículas quando você as esquece. Considere o doctype e os metadados de charset como implicitamente presentes nos exemplos, mesmo quando não são mostrados no texto.

HTML e JavaScript

No contexto deste livro, a tag HTML mais importante é <script>, que nos permite incluir um trecho de JavaScript em um documento.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

Tal script será executado assim que sua tag <script> for encontrada enquanto o navegador lê o HTML. Esta página exibirá um diálogo quando aberta — a função alert se assemelha a prompt, pois abre uma pequena janela, mas apenas mostra uma mensagem sem pedir entrada.

Incluir programas grandes diretamente em documentos HTML é frequentemente impraticável. A tag <script> pode receber um atributo src para buscar um arquivo de script (um arquivo de texto contendo um programa JavaScript) de uma URL.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

O arquivo *code/hello.js* incluído aqui contém o mesmo programa — alert("hello!"). Quando uma página HTML referencia outras URLs como parte de si mesma, como um arquivo de imagem ou um script, navegadores web as recuperarão imediatamente e as incluirão na página.

Uma tag de script deve sempre ser fechada com </script>, mesmo que faça referência a um arquivo de script e não contenha nenhum código. Se você esquecer disso, o restante da página será interpretado como parte do script.

Você pode carregar módulos ES (veja Capítulo 10) no navegador dando à sua tag de script um atributo type="module". Tais módulos podem depender de outros módulos usando URLs relativas a si mesmos como nomes de módulo em declarações import.

Alguns atributos também podem conter um programa JavaScript. A tag <button> (que aparece como um botão) suporta um atributo onclick. O valor do atributo será executado sempre que o botão for clicado.

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

Note que tive que usar aspas simples para a string no atributo `onclick` porque aspas duplas já estão sendo usadas para delimitar o atributo inteiro. Eu também poderia ter usado `"` para escapar as aspas internas.

Na sandbox

Executar programas baixados da internet é potencialmente perigoso. Você não sabe muito sobre as pessoas por trás da maioria dos sites que visita, e elas não necessariamente têm boas intenções. Executar programas de agentes maliciosos é como você tem seu computador infectado por vírus, seus dados roubados e suas contas hackeadas.

No entanto, a atração da web é que você pode navegá-la sem necessariamente confiar em todas as páginas que visita. É por isso que navegadores limitam severamente as coisas que um programa JavaScript pode fazer: ele não pode olhar os arquivos no seu computador ou modificar qualquer coisa não relacionada à página web em que está incorporado.

Isolar um ambiente de programação dessa forma é chamado de *sandbox*, a ideia sendo que o programa está brincando inofensivamente em uma caixa de areia. Mas você deve imaginar esse tipo particular de caixa de areia como tendo uma gaiola de barras de aço grossas sobre ela, para que os programas brincando nela não possam realmente escapar.

A parte difícil do sandboxing é permitir que programas tenham espaço suficiente para serem úteis enquanto os restringe de fazer qualquer coisa perigosa. Muitas funcionalidades úteis, como se comunicar com outros servidores ou ler o conteúdo da área de transferência, também podem ser usadas para fins problemáticos que invadem a privacidade.

De vez em quando, alguém aparece com uma nova forma de contornar as limitações de um navegador e fazer algo prejudicial, variando de vazar informações privadas menores a tomar controle da máquina inteira em que o navegador está rodando. Os desenvolvedores do navegador respondem corrigindo a falha, e tudo fica bem novamente — até que o próximo problema seja descoberto, e esperançosamente divulgado publicamente em vez de secretamente explorado por alguma agência governamental ou organização criminosa.

Compatibilidade e as guerras dos navegadores

Nos estágios iniciais da web, um navegador chamado Mosaic dominava o mercado. Depois de alguns anos, o equilíbrio mudou para o Netscape, que foi, por sua vez, amplamente suplantado pelo Internet Explorer da Microsoft. Em

qualquer momento em que um único navegador era dominante, o fabricante desse navegador se sentia no direito de inventar unilateralmente novas funcionalidades para a web. Como a maioria dos usuários usava o navegador mais popular, sites simplesmente começavam a usar essas funcionalidades — sem se importar com os outros navegadores.

Essa foi a era sombria da compatibilidade, frequentemente chamada de *guerras dos navegadores*. Desenvolvedores web ficaram com não uma web unificada, mas duas ou três plataformas incompatíveis. Para piorar as coisas, os navegadores em uso por volta de 2003 eram todos cheios de bugs, e é claro que os bugs eram diferentes para cada navegador. A vida era difícil para pessoas que escreviam páginas web.

O Mozilla Firefox, uma ramificação sem fins lucrativos do Netscape, desafiou a posição do Internet Explorer no final dos anos 2000. Como a Microsoft não estava particularmente interessada em se manter competitiva na época, o Firefox tomou grande parte de sua fatia de mercado. Mais ou menos na mesma época, o Google lançou seu navegador Chrome e o Safari da Apple ganhou popularidade, levando a uma situação em que havia quatro grandes participantes, em vez de um.

Os novos participantes tinham uma atitude mais séria em relação a padrões e melhores práticas de engenharia, nos dando menos incompatibilidade e menos bugs. A Microsoft, vendo sua participação de mercado desmoronar, adotou essas atitudes em seu navegador Edge, que substituiu o Internet Explorer. Se você está começando a aprender desenvolvimento web hoje, considere-se sortudo. As versões mais recentes dos principais navegadores se comportam de forma bastante uniforme e têm relativamente poucos bugs.

Infelizmente, com a participação de mercado do Firefox ficando cada vez menor, e o Edge se tornando apenas um invólucro em torno do núcleo do Chrome em 2018, essa uniformidade pode novamente tomar a forma de um único fornecedor — Google, dessa vez — tendo controle suficiente sobre o mercado de navegadores para impor sua ideia do que a web deveria ser ao resto do mundo.

Pelo que vale, essa longa cadeia de eventos históricos e acidentes produziu a plataforma web que temos hoje. Nos próximos capítulos, vamos escrever programas para ela.

> *"Too bad! Same old story! Once you've finished building your house you notice you've accidentally learned something that you really should have known—before you started."*
>
> —Friedrich Nietzsche, Beyond Good and Evil

Chapter 14

O Modelo de Objeto do Documento

Quando você abre uma página web, seu navegador recupera o texto HTML da página e o analisa, da mesma forma que nosso parser do Capítulo 12 analisava programas. O navegador constrói um modelo da estrutura do documento e usa esse modelo para desenhar a página na tela.

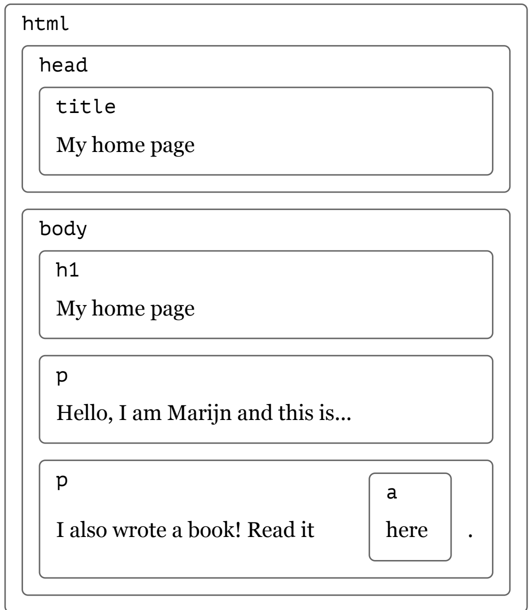
Essa representação do documento é um dos brinquedos que um programa JavaScript tem disponível em sua sandbox. É uma estrutura de dados que você pode ler ou modificar. Ela funciona como uma estrutura de dados *ao vivo*: quando é modificada, a página na tela é atualizada para refletir as mudanças.

Estrutura do documento

Você pode imaginar um documento HTML como um conjunto aninhado de caixas. Tags como `<body>` e `</body>` envolvem outras tags, que por sua vez contêm outras tags ou texto. Aqui está o documento de exemplo do capítulo anterior:

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

Esta página tem a seguinte estrutura:



A estrutura de dados que o navegador usa para representar o documento segue essa forma. Para cada caixa, existe um objeto, com o qual podemos interagir para descobrir coisas como qual tag HTML ele representa e quais caixas e texto ele contém. Essa representação é chamada de *Modelo de Objeto do Documento*, ou *DOM* para abreviar.

A vinculação global `document` nos dá acesso a esses objetos. Sua propriedade `documentElement` se refere ao objeto representando a tag `<html>`. Como todo documento HTML tem um cabeçalho e um corpo, ele também tem propriedades `head` e `body` apontando para esses elementos.

Árvores

Pense nas árvores sintáticas do Capítulo 12 por um momento. Suas estruturas são notavelmente similares à estrutura do documento de um navegador. Cada *nó* pode se referir a outros nós, *filhos*, que por sua vez podem ter seus próprios filhos. Essa forma é típica de estruturas aninhadas, onde elementos podem conter subelementos que são similares a eles mesmos.

Chamamos uma estrutura de dados de *árvore* quando ela tem uma estrutura ramificada, sem ciclos (um nó não pode conter a si mesmo, direta ou indiretamente), e uma única *raiz* bem definida. No caso do DOM, `document.documentElement` serve como a raiz.

Árvores aparecem muito em ciência da computação. Além de representar estruturas recursivas como documentos HTML ou programas, elas são frequentemente usadas para manter conjuntos ordenados de dados, pois elementos geral-

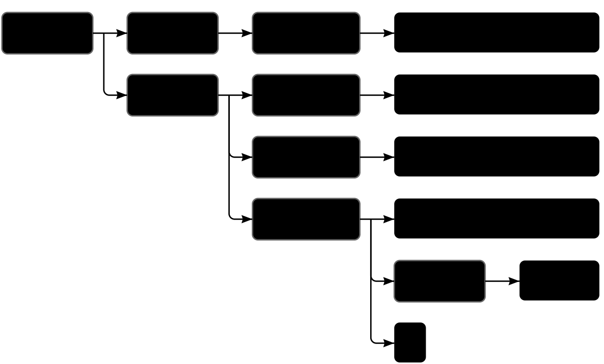
mente podem ser encontrados ou inseridos mais eficientemente em uma árvore do que em um array plano.

Uma árvore típica tem diferentes tipos de nós. A árvore sintática da linguagem Egg tinha identificadores, valores e nós de aplicação. Nós de aplicação podem ter filhos, enquanto identificadores e valores são *folhas*, ou nós sem filhos.

O mesmo vale para o DOM. Nós de *elementos*, que representam tags HTML, determinam a estrutura do documento. Eles podem ter nó filhos. Um exemplo de tal nó é `document.body`. Alguns desses filhos podem ser nós folha, como pedaços de texto ou nós de comentário.

Cada objeto de nó DOM tem uma propriedade `nodeType`, que contém um código (número) que identifica o tipo do nó. Elementos têm o código 1, que também é definido como a propriedade constante `Node.ELEMENT_NODE`. Nós de texto, representando uma seção de texto no documento, recebem o código 3 (`Node.TEXT_NODE`). Comentários têm o código 8 (`Node.COMMENT_NODE`).

Outra forma de visualizar nossa árvore do documento é a seguinte:



As folhas são nós de texto, e as setas indicam relacionamentos pai-filho entre nós.

O padrão

Usar códigos numéricos crípticos para representar tipos de nó não é algo muito ao estilo do JavaScript. Mais adiante neste capítulo, veremos que outras partes da interface do DOM também parecem desajeitadas e estranhas. Isso é porque a interface do DOM não foi projetada apenas para JavaScript. Em vez disso, ela tenta ser uma interface neutra em relação à linguagem que pode ser usada em outros sistemas também — não apenas para HTML, mas também para XML, que é um formato de dados genérico com uma sintaxe similar ao HTML.

Isso é lamentável. Padrões são frequentemente úteis. Mas neste caso, a

vantagem (consistência entre linguagens) não é tão convincente. Ter uma interface que é devidamente integrada com a linguagem que você está usando economizará mais tempo do que ter uma interface familiar entre linguagens.

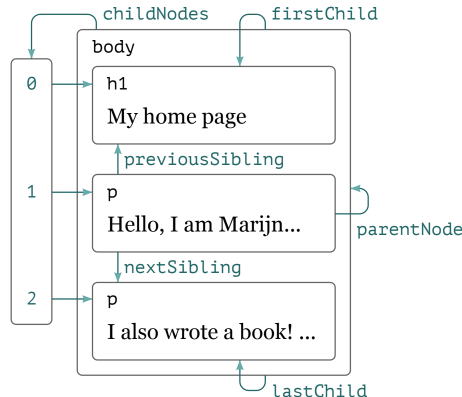
Como exemplo dessa integração precária, considere a propriedade `childNodes` que nós de elemento no DOM possuem. Essa propriedade contém um objeto semelhante a um array, com uma propriedade `length` e propriedades rotuladas por números para acessar os nós filhos. Mas é uma instância do tipo `NodeList`, não um array real, então não possui métodos como `slice` e `map`.

Depois há problemas simplesmente causados por design ruim. Por exemplo, não há como criar um novo nó e imediatamente adicionar filhos ou atributos a ele. Em vez disso, você precisa primeiro criá-lo e depois adicionar filhos e atributos um por um, usando efeitos colaterais. Código que interage pesadamente com o DOM tende a ficar longo, repetitivo e feio.

Mas essas falhas não são fatais. Como JavaScript nos permite criar nossas próprias abstraçãoões, é possível projetar formas melhoradas de expressar as operações que estamos realizando. Muitas bibliotecas destinadas à programação em navegadores vêm com tais ferramentas.

Navegando pela árvore

Nós DOM contêm uma riqueza de links para outros nós próximos. O diagrama a seguir ilustra isso:



Embora o diagrama mostre apenas um link de cada tipo, cada nó tem uma propriedade `parentNode` que aponta para o nó do qual faz parte, se houver. Da mesma forma, cada nó de elemento (tipo 1) tem uma propriedade `childNodes` que aponta para um objeto semelhante a array contendo seus filhos.

Em teoria, você poderia se mover para qualquer lugar na árvore usando apenas esses links de pai e filho. Mas JavaScript também dá acesso a vários links

de conveniência adicionais. As propriedades `firstChild` e `lastChild` apontam para os primeiros e últimos elementos filhos, ou têm o valor `null` para nós sem filhos. Da mesma forma, `previousSibling` e `nextSibling` apontam para nós adjacentes, que são nós com o mesmo pai que aparecem imediatamente antes ou depois do próprio nó. Para um primeiro filho, `previousSibling` será null, e para um último filho, `nextSibling` será null.

Existe também a propriedade `children`, que é como `childNodes` mas contém apenas filhos de elemento (tipo 1), não outros tipos de nós filhos. Isso pode ser útil quando você não está interessado em nós de texto.

Ao lidar com uma estrutura de dados aninhada como esta, funções recursivas são frequentemente úteis. A função a seguir varre um documento em busca de nós de texto contendo uma string dada e retorna `true` quando encontra um:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let child of node.childNodes) {
      if (talksAbout(child, string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

A propriedade `nodeValue` de um nó de texto contém a string de texto que ele representa.

Encontrando elementos

Navegar por esses links entre pais, filhos e irmãos é frequentemente útil. Mas se quisermos encontrar um nó específico no documento, alcançá-lo começando em `document.body` e seguindo um caminho fixo de propriedades é uma má ideia. Fazer isso incorpora suposições em nosso programa sobre a estrutura precisa do documento — uma estrutura que você pode querer mudar depois. Outro fator complicante é que nós de texto são criados mesmo para os espaços em branco entre nós. A tag `<body>` do documento de exemplo não tem apenas três filhos (`<h1>` e dois elementos `<p>`), mas sete: esses três, mais os espaços antes,

depois e entre eles.

Se quisermos obter o atributo `href` do link naquele documento, não queremos dizer algo como “Pegue o segundo filho do sexto filho do corpo do documento”. Seria melhor se pudéssemos dizer “Pegue o primeiro link no documento”. E podemos.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Todos os nós de elemento têm um método `getElementsByTagName`, que coleta todos os elementos com o nome de tag dado que são descendentes (filhos diretos ou indiretos) daquele nó e os retorna como um objeto semelhante a array.

Para encontrar um *único* nó específico, você pode dar a ele um atributo `id` e usar `document.getElementById` em vez disso.

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

Um terceiro método similar é `getElementsByClassName`, que, assim como `getElementsByTagName`, busca através dos conteúdos de um nó de elemento e recupera todos os elementos que têm a string dada em seu atributo `class`.

Alterando o documento

Quase tudo sobre a estrutura de dados do DOM pode ser alterado. A forma da árvore do documento pode ser modificada alterando relacionamentos pai-filho. Nós têm um método `remove` para removê-los de seu nó pai atual. Para adicionar um nó filho a um nó de elemento, podemos usar `appendChild`, que o coloca no final da lista de filhos, ou `insertBefore`, que insere o nó dado como primeiro argumento antes do nó dado como segundo argumento.

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

Um nó pode existir no documento em apenas um lugar. Assim, inserir o parágrafo *Three* na frente do parágrafo *One* irá primeiro removê-lo do final do documento e depois inseri-lo na frente, resultando em *Three*/*One*/*Two*. Todas as operações que inserem um nó em algum lugar irão, como efeito colateral, fazer com que ele seja removido de sua posição atual (se tiver uma).

O método `replaceChild` é usado para substituir um nó filho por outro. Ele recebe como argumentos dois nós: um novo nó e o nó a ser substituído. O nó substituído deve ser um filho do elemento no qual o método é chamado. Note que tanto `replaceChild` quanto `insertBefore` esperam o nó *novo* como seu primeiro argumento.

Criando nós

Digamos que queremos escrever um script que substitua todas as imagens (`tags <img>`) no documento pelo texto contido em seus atributos `alt`, que especifica uma representação textual alternativa da imagem. Isso envolve não apenas remover as imagens, mas também adicionar um novo nó de texto para substituí-las.

```
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Dado uma string, `createTextNode` nos dá um nó de texto que podemos inserir no documento para fazê-lo aparecer na tela.

O loop que percorre as imagens começa do final da lista. Isso é necessário porque a lista de nós retornada por um método como `getElementsByTagName` (ou uma propriedade como `childNodes`) é *ao vivo*. Ou seja, ela é atualizada

conforme o documento muda. Se começássemos do início, remover a primeira imagem faria a lista perder seu primeiro elemento, de forma que na segunda vez que o loop se repetisse, quando `i` fosse 1, ele pararia porque o comprimento da coleção agora também é 1.

Se você quer uma coleção *sólida* de nós, em oposição a uma ao vivo, pode converter a coleção em um array real chamando `Array.from`.

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

Para criar nós de elemento, você pode usar o método `document.createElement`. Esse método recebe um nome de tag e retorna um novo nó vazio do tipo dado.

O exemplo a seguir define uma utilidade `elt`, que cria um nó de elemento e trata o restante de seus argumentos como filhos daquele nó. Essa função é então usada para adicionar uma atribuição a uma citação.

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", preface to the second edition of ",
        elt("em", "The Open Society and Its Enemies"),
        ", 1950"));
</script>
```

É assim que o documento resultante fica:

No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it.
—**Karl Popper**, preface to the second editon of *The Open Society and Its Enemies*, 1950

Atributos

Alguns atributos de elemento, como `href` para links, podem ser acessados através de uma propriedade de mesmo nome no objeto DOM do elemento. Esse é o caso para a maioria dos atributos padrão comumente usados.

HTML permite que você defina qualquer atributo que quiser em nós. Isso pode ser útil porque permite armazenar informações extras em um documento. Para ler ou alterar atributos personalizados, que não estão disponíveis como propriedades regulares de objeto, você deve usar os métodos `getAttribute` e `setAttribute`.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

É recomendado prefixar os nomes de tais atributos inventados com `data-` para garantir que não conflitem com nenhum outro atributo.

Existe um atributo comumente usado, `class`, que é uma palavra-chave na linguagem JavaScript. Por razões históricas — algumas implementações antigas de JavaScript não conseguiam lidar com nomes de propriedade que coincidissem com palavras-chave — a propriedade usada para acessar esse atributo é chamada `className`. Você também pode acessá-lo por seu nome real, `"class"`, com os métodos `getAttribute` e `setAttribute`.

Layout

Você pode ter notado que diferentes tipos de elementos são dispostos de maneira diferente. Alguns, como parágrafos (`<p>`) ou cabeçalhos (`<h1>`), ocupam toda

a largura do documento e são renderizados em linhas separadas. Estes são chamados de elementos de *bloco*. Outros, como links (<a>) ou o elemento <strong>, são renderizados na mesma linha que o texto ao redor. Tais elementos são chamados de elementos *inline*.

Para qualquer documento, navegadores são capazes de calcular um layout, que dá a cada elemento um tamanho e posição baseados em seu tipo e conteúdo. Esse layout é então usado para realmente desenhar o documento.

O tamanho e posição de um elemento podem ser acessados a partir do JavaScript. As propriedades `offsetWidth` e `offsetHeight` fornecem o espaço que o elemento ocupa em *pixels*. Um pixel é a unidade básica de medida no navegador. Tradicionalmente corresponde ao menor ponto que a tela pode desenhar, mas em displays modernos, que podem desenhar pontos *muito* pequenos, isso pode não ser mais o caso, e um pixel do navegador pode abranger múltiplos pontos do display.

Da mesma forma, `clientWidth` e `clientHeight` fornecem o tamanho do espaço *dentro* do elemento, ignorando a largura da borda.

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  // → 19
  console.log("offsetHeight:", para.offsetHeight);
  // → 25
</script>
```

Dar uma borda a um parágrafo faz com que um retângulo seja desenhado ao redor dele.

I'm boxed in

A forma mais eficaz de encontrar a posição precisa de um elemento na tela é o método `getBoundingClientRect`. Ele retorna um objeto com propriedades `top`, `bottom`, `left` e `right`, indicando as posições em pixels dos lados do elemento em relação ao canto superior esquerdo da tela. Se você quiser posições em pixels relativas ao documento inteiro, deve adicionar a posição de rolagem atual, que pode ser encontrada nas vinculações `pageXOffset` e `pageYOffset`.

Calcular o layout de um documento pode ser bastante trabalhoso. No interesse da velocidade, motores de navegador não recalculam imediatamente o layout de um documento toda vez que você o altera, mas esperam o máximo

que podem antes de fazê-lo. Quando um programa JavaScript que alterou o documento termina de executar, o navegador terá que calcular um novo layout para desenhar o documento alterado na tela. Quando um programa *pede* a posição ou tamanho de algo lendo propriedades como `offsetHeight` ou chamando `getBoundingClientRect`, fornecer essa informação também requer calcular um layout.

Um programa que alterna repetidamente entre ler informações de layout do DOM e alterar o DOM força muitos cálculos de layout e consequentemente rodará muito devagar. O código a seguir é um exemplo disso. Ele contém dois programas diferentes que constroem uma linha de caracteres *X* com 2.000 pixels de largura e medem o tempo que cada um leva.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Tempo atual em milissegundos
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

Estilos

Vimos que diferentes elementos HTML são desenhados de formas diferentes. Alguns são exibidos como blocos, outros inline. Alguns adicionam estilo —

`<strong>` torna seu conteúdo negrito, e `<a>` o torna azul e sublinhado.

A forma como uma tag `<img>` mostra uma imagem ou uma tag `<a>` faz um link ser seguido quando clicado está fortemente ligada ao tipo do elemento. Mas podemos alterar o estilo associado a um elemento, como a cor do texto ou o sublinhado. Aqui está um exemplo que usa a propriedade `style`:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

O segundo link ficará verde em vez da cor de link padrão:

Normal link

Green link

Um atributo de estilo pode conter uma ou mais *declaraçãoões*, que são uma propriedade (como `color`) seguida por dois-pontos e um valor (como `green`). Quando há mais de uma declaração, elas devem ser separadas por ponto e vírgulas, como em `"color: red; border: none"`.

Muitos aspectos do documento podem ser influenciados pela estilização. Por exemplo, a propriedade `display` controla se um elemento é exibido como bloco ou como elemento inline.

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

A tag `block` ficará em sua própria linha, já que elementos de bloco não são exibidos inline com o texto ao redor. A última tag não é exibida de forma alguma — `display: none` impede que um elemento apareça na tela. Esta é uma forma de esconder elementos. Frequentemente é preferível a removê-los do documento inteiramente, porque torna fácil revelá-los novamente mais tarde.

This text is displayed **inline**,
as a block
, and .

Código JavaScript pode manipular diretamente o estilo de um elemento através da propriedade `style` do elemento. Essa propriedade contém um objeto que tem propriedades para todas as propriedades de estilo possíveis. Os valores dessas propriedades são strings, nas quais podemos escrever para alterar um aspecto particular do estilo do elemento.

```
<p id="para" style="color: purple">
  Nice text
```

```
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Alguns nomes de propriedades de estilo contêm hífens, como `font-family`. Como tais nomes de propriedade são estranhos de trabalhar em JavaScript (você teria que escrever `style["font-family"]`), os nomes das propriedades no objeto `style` para tais propriedades têm seus hífens removidos e as letras após eles capitalizadas (`style.fontFamily`).

Estilos em cascata

O sistema de estilos para HTML é chamado de *CSS*, de *Cascading Style Sheets* (Folhas de Estilo em Cascata). Uma *folha de estilo* é um conjunto de regras sobre como estilizar elementos em um documento. Ela pode ser escrita dentro de uma tag `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

O *cascata* no nome se refere ao fato de que múltiplas regras são combinadas para produzir o estilo final de um elemento. No exemplo, o estilo padrão para tags `<strong>`, que lhes dá `font-weight: bold`, é sobreposto pela regra na tag `<style>`, que adiciona `font-style` e `color`.

Quando múltiplas regras definem um valor para a mesma propriedade, a regra lida mais recentemente obtém uma precedência maior e vence. Por exemplo, se a regra na tag `<style>` incluísse `font-weight: normal`, contradizendo a regra padrão de `font-weight`, o texto seria normal, *não* negrito. Estilos em um atributo `style` aplicado diretamente ao nó têm a precedência mais alta e sempre vencem.

É possível direcionar coisas além de nomes de tag em regras CSS. Uma regra para `.abc` se aplica a todos os elementos com `"abc"` em seu atributo `class`. Uma regra para `#xyz` se aplica ao elemento com um atributo `id` de `"xyz"` (que

deve ser único dentro do documento).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* elementos p com id main e com classes a e b */
p#main.a.b {
  margin-bottom: 20px;
}
```

A regra de precedência que favorece a regra definida mais recentemente se aplica apenas quando as regras têm a mesma *especificidade*. A especificidade de uma regra é uma medida de quão precisamente ela descreve os elementos correspondentes, determinada pelo número e tipo (tag, classe ou ID) de aspectos do elemento que ela requer. Por exemplo, uma regra que direciona `p.a` é mais específica do que regras que direcionam `p` ou apenas `.a` e, portanto, teria precedência sobre elas.

A notação `p > a …{}` aplica os estilos dados a todas as tags `<a>` que são filhos diretos de tags `<p>`. Da mesma forma, `p a …{}` se aplica a todas as tags `<a>` dentro de tags `<p>`, sejam eles filhos diretos ou indiretos.

Seletores de consulta

Não usaremos folhas de estilo muito neste livro. Entendê-las é útil ao programar no navegador, mas são complexas o suficiente para justificar um livro separado. A principal razão pela qual introduzi a sintaxe de *seletor* — a notação usada em folhas de estilo para determinar a quais elementos um conjunto de estilos se aplica — é que podemos usar essa mesma minilinguagem como uma forma eficaz de encontrar elementos DOM.

O método `querySelectorAll`, que é definido tanto no objeto `document` quanto em nós de elemento, recebe uma string de seletor e retorna um `NodeList` contendo todos os elementos que ele corresponde.

```
<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
```

```
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // Todos os elementos <p>
  // → 4
  console.log(count(".animal"));     // Classe animal
  // → 2
  console.log(count("p .animal"));   // Animal dentro de <p>
  // → 2
  console.log(count("p > .animal")); // Filho direto de <p>
  // → 1
</script>
```

Ao contrário de métodos como `getElementsByTagName`, o objeto retornado por `querySelectorAll` *não* é ao vivo. Ele não mudará quando você alterar o documento. Ainda não é um array real, porém, então você precisa chamar `Array.from` se quiser tratá-lo como um.

O método `querySelector` (sem a parte `All`) funciona de forma similar. Este é útil se você quer um único elemento específico. Ele retornará apenas o primeiro elemento correspondente, ou `null` quando nenhum elemento corresponder.

Posicionamento e animação

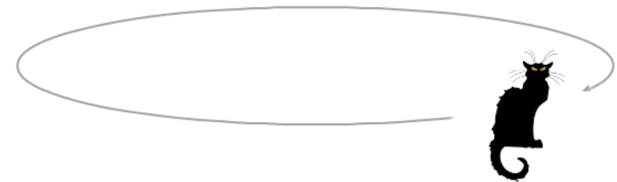
A propriedade de estilo `position` influencia o layout de forma poderosa. Ela tem um valor padrão de `static`, significando que o elemento fica em seu lugar normal no documento. Quando é definida como `relative`, o elemento ainda ocupa espaço no documento, mas agora as propriedades de estilo `top` e `left` podem ser usadas para movê-lo em relação àquele lugar normal. Quando `position` é definida como `absolute`, o elemento é removido do fluxo normal do documento — ou seja, ele não ocupa mais espaço e pode se sobrepor a outros elementos. Suas propriedades `top` e `left` podem ser usadas para posicioná-lo absolutamente em relação ao canto superior esquerdo do elemento envolvente mais próximo cuja propriedade `position` não seja `static`, ou em relação ao documento se nenhum elemento envolvente assim existir.

Podemos usar isso para criar uma animação. O documento a seguir exibe uma imagem de um gato que se move em uma elipse:

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
```

```
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

A seta cinza mostra o caminho ao longo do qual a imagem se move.



Nossa imagem está centralizada na página e tem `position` definida como `relative`. Atualizaremos repetidamente os estilos `top` e `left` dessa imagem para movê-la.

O script usa `requestAnimationFrame` para agendar a função `animate` para rodar sempre que o navegador estiver pronto para repintar a tela. A própria função `animate` chama `requestAnimationFrame` novamente para agendar a próxima atualização. Quando a janela (ou aba) do navegador está ativa, isso causará atualizações a uma taxa de cerca de 60 por segundo, o que tende a produzir uma animação de boa aparência.

Se apenas atualizássemos o DOM em um loop, a página congelaria e nada apareceria na tela. Navegadores não atualizam seu display enquanto um programa JavaScript está rodando, nem permitem qualquer interação com a página. É por isso que precisamos de `requestAnimationFrame` — ele informa ao navegador que terminamos por agora, e ele pode prosseguir e fazer as coisas que navegadores fazem, como atualizar a tela e responder a ações do usuário.

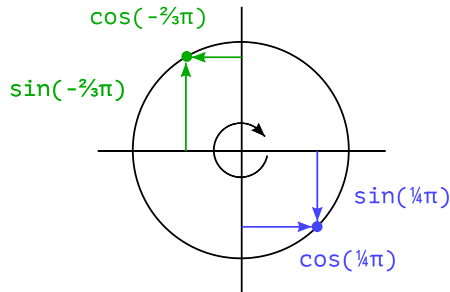
A função de animação recebe o tempo atual como argumento. Para garantir que o movimento do gato por milissegundo seja estável, ela baseia a velocidade na qual o ângulo muda na diferença entre o tempo atual e a última vez que a função foi executada. Se simplesmente movesse o ângulo por uma quantidade fixa por passo, o movimento engasgaria quando, por exemplo, outra tarefa

pesada rodando no mesmo computador impedisse a função de rodar por uma fração de segundo.

Mover-se em círculos é feito usando as funções trigonométricas `Math.cos` e `Math.sin`. Para aqueles que não estão familiarizados com estas, vou apresentá-las brevemente, já que as usaremos ocasionalmente neste livro.

`Math.cos` e `Math.sin` são úteis para encontrar pontos que ficam em um círculo ao redor do ponto (0, 0) com raio 1. Ambas as funções interpretam seu argumento como a posição neste círculo, com 0 denotando o ponto no extremo direito do círculo, indo no sentido horário até que 2π (cerca de 6,28) tenha nos levado ao redor de todo o círculo. `Math.cos` diz a coordenada x do ponto que corresponde à posição dada, e `Math.sin` produz a coordenada y. Posições (ou ângulos) maiores que 2π ou menores que 0 são válidas — a rotação se repete, de modo que $a+2\pi$ se refere ao mesmo ângulo que a.

Essa unidade para medir ângulos é chamada de radianos — um círculo completo é 2π radianos, similar a como são 360 graus quando medimos em graus. A constante π está disponível como `Math.PI` em JavaScript.



O código de animação do gato mantém um contador, `angle`, para o ângulo atual da animação e o incrementa toda vez que a função `animate` é chamada. Ele pode então usar esse ângulo para calcular a posição atual do elemento de imagem. O estilo `top` é calculado com `Math.sin` e multiplicado por 20, que é o raio vertical da nossa elipse. O estilo `left` é baseado em `Math.cos` e multiplicado por 200, de modo que a elipse é muito mais larga do que alta.

Note que estilos geralmente precisam de *unidades*. Neste caso, temos que acrescentar `"px"` ao número para dizer ao navegador que estamos contando em pixels (em oposição a centímetros, "ems" ou outras unidades). Isso é fácil de esquecer. Usar números sem unidades fará com que seu estilo seja ignorado — a menos que o número seja 0, que sempre significa a mesma coisa, independentemente da unidade.

Resumo

Programas JavaScript podem inspecionar e interferir no documento que o navegador está exibindo através de uma estrutura de dados chamada DOM. Essa estrutura de dados representa o modelo do documento pelo navegador, e um programa JavaScript pode modificá-la para alterar o documento visível.

O DOM é organizado como uma árvore, onde elementos são arranjados hierarquicamente de acordo com a estrutura do documento. Os objetos representando elementos têm propriedades como `parentNode` e `childNodes`, que podem ser usadas para navegar por essa árvore.

A forma como um documento é exibido pode ser influenciada por *estilos*, tanto anexando estilos diretamente a nós quanto definindo regras que correspondam a certos nós. Há muitas propriedades de estilo diferentes, como `color` ou `display`. Código JavaScript pode manipular o estilo de um elemento diretamente através de sua propriedade `style`.

Exercícios

Construa uma tabela

Uma tabela HTML é construída com a seguinte estrutura de tags:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

Para cada *linha*, a tag `<table>` contém uma tag `<tr>`. Dentro dessas tags `<tr>`, podemos colocar elementos de célula: tanto células de cabeçalho (`<th>`) quanto células regulares (`<td>`).

Dado um conjunto de dados de montanhas, um array de objetos com propriedades `name`, `height` e `place`, gere a estrutura DOM para uma tabela que enumere os objetos. Ela deve ter uma coluna por chave e uma linha por objeto, mais uma linha de cabeçalho com elementos `<th>` no topo, listando os nomes

das colunas.

Escreva isso de forma que as colunas sejam automaticamente derivadas dos objetos, pegando os nomes das propriedades do primeiro objeto nos dados.

Mostre a tabela resultante no documento adicionando-a ao elemento que tem um atributo `id` de `"mountains"`.

Quando tiver isso funcionando, alinhe à direita as células que contêm valores numéricos definindo sua propriedade `style.textAlign` como `"right"`.

Elementos por nome de tag

O método `document.getElementsByTagName` retorna todos os elementos filhos com um dado nome de tag. Implemente sua própria versão disso como uma função que recebe um nó e uma string (o nome da tag) como argumentos e retorna um array contendo todos os nós de elemento descendentes com o dado nome de tag. Sua função deve percorrer o documento por si mesma. Ela não pode usar um método como `querySelectorAll` para fazer o trabalho.

Para encontrar o nome da tag de um elemento, use sua propriedade `nodeName`. Mas note que isso retornará o nome da tag todo em maiúsculas. Use os métodos de string `toLowerCase` ou `toUpperCase` para compensar isso.

O chapéu do gato

Estenda a animação do gato definida anteriormente para que tanto o gato quanto seu chapéu (`<img src="img/hat.png">`) orbitem em lados opostos da elipse.

Ou faça o chapéu circular ao redor do gato. Ou altere a animação de alguma outra forma interessante.

Para facilitar o posicionamento de múltiplos objetos, você provavelmente vai querer mudar para posicionamento absoluto. Isso significa que `top` e `left` são contados em relação ao canto superior esquerdo do documento. Para evitar usar coordenadas negativas, que fariam a imagem se mover para fora da página visível, você pode adicionar um número fixo de pixels aos valores de posição.

> *"You have power over your mind—not outside events. Realize this, and you will find strength."*
>
> —Marcus Aurelius, Meditations

Chapter 15

Manipulando Eventos

Alguns programas trabalham com entrada direta do usuário, como ações de mouse e teclado. Esse tipo de entrada não está disponível antecipadamente, como uma estrutura de dados bem organizada — ela chega pedaço por pedaço, em tempo real, e o programa deve responder a ela conforme acontece.

Manipuladores de eventos

Imagine uma interface onde a única forma de descobrir se uma tecla no teclado está sendo pressionada é ler o estado atual daquela tecla. Para poder reagir a pressionamentos de tecla, você teria que constantemente ler o estado da tecla para pegá-la antes que fosse solta. Seria perigoso realizar outras computações que consumam tempo, já que você poderia perder um pressionamento de tecla.

Algumas máquinas primitivas lidam com entrada assim. Um passo acima disso é o hardware ou sistema operacional notar o pressionamento de tecla e colocá-lo em uma fila. Um programa pode então verificar periodicamente a fila em busca de novos eventos e reagir ao que encontrar lá.

Claro, o programa precisa lembrar de verificar a fila, e fazer isso com frequência, porque qualquer tempo entre a tecla ser pressionada e o programa notar o evento fará com que o software pareça não responsivo. Essa abordagem é chamada de *polling*. A maioria dos programadores prefere evitá-la.

Um mecanismo melhor é o sistema notificar ativamente o código quando um evento ocorre. Navegadores fazem isso nos permitindo registrar funções como *manipuladores* para eventos específicos.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

A vinculação `window` se refere a um objeto embutido fornecido pelo navegador. Ele representa a janela do navegador que contém o documento. Chamar seu método `addEventListener` registra o segundo argumento para ser chamado sempre que o evento descrito pelo primeiro argumento ocorrer.

Eventos e nós DOM

Cada manipulador de evento do navegador é registrado em um contexto. No exemplo anterior, chamamos `addEventListener` no objeto `window` para registrar um manipulador para toda a janela. Tal método também pode ser encontrado em elementos DOM e alguns outros tipos de objetos. Ouvintes de eventos são chamados apenas quando o evento acontece no contexto do objeto no qual foram registrados.

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

Esse exemplo anexa um manipulador ao nó do botão. Cliques no botão fazem esse manipulador rodar, mas cliques no resto do documento não.

Dar a um nó um atributo `onclick` tem um efeito similar. Isso funciona para a maioria dos tipos de eventos — você pode anexar um manipulador através do atributo cujo nome é o nome do evento com `on` na frente.

Mas um nó pode ter apenas um atributo `onclick`, então você pode registrar apenas um manipulador por nó dessa forma. O método `addEventListener` permite que você adicione qualquer número de manipuladores, o que significa que é seguro adicionar manipuladores mesmo se já houver outro manipulador no elemento.

O método `removeEventListener`, chamado com argumentos similares a `addEventListener`, remove um manipulador.

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
```

```
  }
  button.addEventListener("click", once);
</script>
```

A função passada para `removeEventListener` deve ser o mesmo valor de função dado a `addEventListener`. Quando você precisa cancelar o registro de um manipulador, vai querer dar à função manipuladora um nome (`once`, no exemplo) para poder passar o mesmo valor de função para ambos os métodos.

Objetos de evento

Embora tenhamos ignorado isso até agora, funções manipuladoras de evento recebem um argumento: o *objeto de evento*. Esse objeto contém informações adicionais sobre o evento. Por exemplo, se quisermos saber *qual* botão do mouse foi pressionado, podemos olhar a propriedade `button` do objeto de evento.

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button");
    } else if (event.button == 1) {
      console.log("Middle button");
    } else if (event.button == 2) {
      console.log("Right button");
    }
  });
</script>
```

A informação armazenada em um objeto de evento difere por tipo de evento. (Discutiremos diferentes tipos mais adiante no capítulo.) A propriedade `type` do objeto sempre contém uma string identificando o evento (como `"click"` ou `"mousedown"`).

Propagação

Para a maioria dos tipos de eventos, manipuladores registrados em nós com filhos também receberão eventos que acontecem nos filhos. Se um botão dentro de um parágrafo é clicado, manipuladores de evento no parágrafo também verão o evento de clique.

Mas se tanto o parágrafo quanto o botão tiverem um manipulador, o manip-

ulador mais específico — o do botão — tem vez primeiro. Diz-se que o evento *propaga* para fora a partir do nó onde aconteceu para o nó pai daquele nó e daí para a raiz do documento. Finalmente, depois que todos os manipuladores registrados em um nó específico tiveram sua vez, manipuladores registrados na janela inteira recebem a chance de responder ao evento.

A qualquer momento, um manipulador de evento pode chamar o método `stopPropagation` no objeto de evento para impedir que manipuladores mais acima recebam o evento. Isso pode ser útil quando, por exemplo, você tem um botão dentro de outro elemento clicável e não quer que cliques no botão ativem o comportamento de clique do elemento externo.

O exemplo a seguir registra manipuladores `"mousedown"` tanto em um botão quanto no parágrafo ao redor dele. Quando clicado com o botão direito do mouse, o manipulador do botão chama `stopPropagation`, que impedirá o manipulador do parágrafo de rodar. Quando o botão é clicado com outro botão do mouse, ambos os manipuladores rodarão.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

A maioria dos objetos de evento tem uma propriedade `target` que se refere ao nó onde se originaram. Você pode usar essa propriedade para garantir que não está acidentalmente manipulando algo que propagou de um nó que você não quer manipular.

Também é possível usar a propriedade `target` para lançar uma rede ampla para um tipo específico de evento. Por exemplo, se você tem um nó contendo uma longa lista de botões, pode ser mais conveniente registrar um único manipulador de clique no nó externo e usar a propriedade `target` para descobrir se um botão foi clicado, em vez de registrar manipuladores individuais em todos os botões.

```
<button>A</button>
<button>B</button>
<button>C</button>
```

```
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent);
    }
  });
</script>
```

Ações padrão

Muitos eventos têm uma ação padrão. Se você clicar em um link, será levado ao destino do link. Se pressionar a seta para baixo, o navegador rolará a página para baixo. Se clicar com o botão direito, verá um menu de contexto. E assim por diante.

Para a maioria dos tipos de eventos, os manipuladores de evento JavaScript são chamados *antes* que o comportamento padrão ocorra. Se o manipulador não quiser que esse comportamento normal aconteça, tipicamente porque já cuidou de manipular o evento, ele pode chamar o método `preventDefault` no objeto de evento.

Isso pode ser usado para implementar seus próprios atalhos de teclado ou menu de contexto. Também pode ser usado para interferir de forma desagradável no comportamento que os usuários esperam. Por exemplo, aqui está um link que não pode ser seguido:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Tente não fazer coisas assim sem uma razão realmente boa. Será desagradável para as pessoas que usam sua página quando o comportamento esperado é quebrado.

Dependendo do navegador, alguns eventos não podem ser interceptados de forma alguma. No Chrome, por exemplo, o atalho de teclado para fechar a aba atual (CTRL-W ou COMMAND-W) não pode ser manipulado por JavaScript.

Eventos de tecla

Quando uma tecla no teclado é pressionada, seu navegador dispara um evento "keydown". Quando ela é solta, você recebe um evento "keyup".

```
<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

Apesar do nome, "keydown" dispara não apenas quando a tecla é fisicamente pressionada. Quando uma tecla é pressionada e mantida, o evento dispara novamente toda vez que a tecla *repete*. Às vezes você precisa ter cuidado com isso. Por exemplo, se você adicionar um botão ao DOM quando uma tecla é pressionada e removê-lo quando a tecla é solta, pode acidentalmente adicionar centenas de botões quando a tecla é mantida pressionada por mais tempo.

O exemplo anterior observa a propriedade key do objeto de evento para ver sobre qual tecla o evento é. Essa propriedade contém uma string que, para a maioria das teclas, corresponde ao que pressionar aquela tecla digitaria. Para teclas especiais como ENTER, ela contém uma string que nomeia a tecla (" Enter", neste caso). Se você segura SHIFT enquanto pressiona uma tecla, isso também pode influenciar o nome da tecla — "v" se torna "V", e "1" pode se tornar "!", se é isso que pressionar SHIFT-1 produz no seu teclado.

Teclas modificadoras como SHIFT, CTRL, ALT e META (COMMAND no Mac) geram eventos de tecla assim como teclas normais. Ao procurar combinações de teclas, você também pode descobrir se essas teclas estão pressionadas olhando as propriedades shiftKey, ctrlKey, altKey e metaKey dos eventos de teclado e mouse.

```
<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
```

```
    }
  });
</script>
```

O nó DOM onde um evento de tecla se origina depende do elemento que tem foco quando a tecla é pressionada. A maioria dos nós não pode ter foco a menos que você lhes dê um atributo `tabindex`, mas coisas como links, botões e campos de formulário podem. Voltaremos a campos de formulário no Capítulo 18. Quando nada em particular tem foco, `document.body` funciona como o nó alvo dos eventos de tecla.

Quando o usuário está digitando texto, usar eventos de tecla para descobrir o que está sendo digitado é problemático. Algumas plataformas, notavelmente o teclado virtual em telefones Android, não disparam eventos de tecla. Mas mesmo quando você tem um teclado tradicional, alguns tipos de entrada de texto não correspondem a pressionamentos de tecla de forma direta, como software de *editor de método de entrada* (*IME*) usado por pessoas cujos sistemas de escrita não cabem em um teclado, onde múltiplos pressionamentos de tecla são combinados para criar caracteres.

Para notar quando algo foi digitado, elementos nos quais você pode digitar, como as tags `<input>` e `<textarea>`, disparam eventos `"input"` sempre que o usuário muda seu conteúdo. Para obter o conteúdo real que foi digitado, é melhor lê-lo diretamente do campo focado, o que discutimos no Capítulo 18.

Eventos de ponteiro

Existem atualmente duas formas amplamente usadas de apontar para coisas em uma tela: mouses (incluindo dispositivos que agem como mouses, como touchpads e trackballs) e telas de toque. Estes produzem diferentes tipos de eventos.

Cliques do mouse

Pressionar um botão do mouse faz uma série de eventos dispararem. Os eventos `"mousedown"` e `"mouseup"` são similares a `"keydown"` e `"keyup"` e disparam quando o botão é pressionado e solto. Eles acontecem nos nós DOM que estão imediatamente abaixo do ponteiro do mouse quando o evento ocorre.

Após o evento `"mouseup"`, um evento `"click"` dispara no nó mais específico que continha tanto o pressionamento quanto a liberação do botão. Por exemplo, se eu pressionar o botão do mouse em um parágrafo e depois mover o ponteiro para outro parágrafo e soltar o botão, o evento `"click"` acontecerá no elemento

que contém ambos os parágrafos.

Se dois cliques acontecem próximos um do outro, um evento `"dblclick"` (duplo clique) também dispara, após o segundo evento de clique.

Para obter informações precisas sobre o local onde um evento de mouse aconteceu, você pode olhar suas propriedades `clientX` e `clientY`, que contêm as coordenadas do evento (em pixels) relativas ao canto superior esquerdo da janela, ou `pageX` e `pageY`, que são relativas ao canto superior esquerdo do documento inteiro (que pode ser diferente quando a janela foi rolada).

O programa a seguir implementa um aplicativo de desenho primitivo. Toda vez que você clicar no documento, ele adiciona um ponto sob o ponteiro do mouse.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* arredonda cantos */
    background: teal;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

Criaremos um aplicativo de desenho menos primitivo no Capítulo 19.

Movimento do mouse

Toda vez que o ponteiro do mouse se move, um evento `"mousemove"` dispara. Esse evento pode ser usado para rastrear a posição do mouse. Uma situação comum em que isso é útil é ao implementar alguma forma de funcionalidade de arrastar com o mouse.

Como exemplo, o programa a seguir exibe uma barra e configura manipuladores de evento para que arrastar para a esquerda ou direita nessa barra a

torne mais estreita ou mais larga:

```
<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // Rastreia a última posição X observada do mouse
  let bar = document.querySelector("div");
  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
      window.addEventListener("mousemove", moved);
      event.preventDefault(); // Prevenir seleção
    }
  });

  function moved(event) {
    if (event.buttons == 0) {
      window.removeEventListener("mousemove", moved);
    } else {
      let dist = event.clientX - lastX;
      let newWidth = Math.max(10, bar.offsetWidth + dist);
      bar.style.width = newWidth + "px";
      lastX = event.clientX;
    }
  }
</script>
```

A página resultante se parece com isto:

Drag the bar to change its width:

Note que o manipulador `"mousemove"` é registrado na janela inteira. Mesmo se o mouse sair da barra durante o redimensionamento, enquanto o botão estiver pressionado, ainda queremos atualizar seu tamanho.

Devemos parar de redimensionar a barra quando o botão do mouse é solto. Para isso, podemos usar a propriedade `buttons` (note o plural), que nos diz sobre os botões que estão atualmente pressionados. Quando é 0, nenhum botão está pressionado. Quando botões estão pressionados, o valor da propriedade `buttons` é a soma dos códigos desses botões — o botão esquerdo tem código 1, o direito 2 e o do meio 4. Com os botões esquerdo e direito pressionados, por exemplo, o valor de `buttons` será 3.

Note que a ordem desses códigos é diferente da usada por `button`, onde o

botão do meio vinha antes do direito. Como mencionado, consistência não é um ponto forte da interface de programação do navegador.

Eventos de toque

O estilo de navegador gráfico que usamos foi projetado com interfaces de mouse em mente, numa época em que telas de toque eram raras. Para fazer a web "funcionar" nos primeiros telefones com tela de toque, navegadores para esses dispositivos fingiam, até certo ponto, que eventos de toque eram eventos de mouse. Se você tocar sua tela, receberá eventos `"mousedown"`, `"mouseup"` e `"click"`.

Mas essa ilusão não é muito robusta. Uma tela de toque não funciona como um mouse: ela não tem múltiplos botões, você não pode rastrear o dedo quando ele não está na tela (para simular `"mousemove"`), e permite que múltiplos dedos estejam na tela ao mesmo tempo.

Eventos de mouse cobrem interação por toque apenas em casos simples — se você adicionar um manipulador `"click"` a um botão, usuários de toque ainda poderão usá-lo. Mas algo como a barra redimensionável no exemplo anterior não funciona em uma tela de toque.

Existem tipos específicos de eventos disparados por interação de toque. Quando um dedo começa a tocar a tela, você recebe um evento `"touchstart"`. Quando ele é movido enquanto toca, eventos `"touchmove"` disparam. Finalmente, quando ele para de tocar a tela, você verá um evento `"touchend"`.

Como muitas telas de toque podem detectar múltiplos dedos ao mesmo tempo, esses eventos não têm um único conjunto de coordenadas associado a eles. Em vez disso, seus objetos de evento têm uma propriedade `touches`, que contém um objeto semelhante a array de pontos, cada um com suas próprias propriedades `clientX`, `clientY`, `pageX` e `pageY`.

Você poderia fazer algo assim para mostrar círculos vermelhos ao redor de cada dedo tocando:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Touch this page</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
```

```
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

Frequentemente você vai querer chamar `preventDefault` em manipuladores de eventos de toque para sobrescrever o comportamento padrão do navegador (que pode incluir rolar a página ao deslizar) e para impedir que os eventos de mouse sejam disparados, para os quais você pode *também* ter um manipulador.

Eventos de rolagem

Sempre que um elemento é rolado, um evento `"scroll"` é disparado nele. Isso tem vários usos, como saber o que o usuário está vendo atualmente (para desabilitar animaçãoões fora da tela ou enviar relatórios de espionagem para seu quartel-general maligno) ou mostrar alguma indicação de progresso (destacando parte de um índice ou mostrando um número de página).

O exemplo a seguir desenha uma barra de progresso acima do documento e a atualiza para se encher conforme você rola para baixo:

```
<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Criar algum conteúdo
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
```

```
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Dar a um elemento uma `position` de `fixed` age de forma semelhante a uma posição `absolute`, mas também impede que ele role junto com o resto do documento. O efeito é fazer nossa barra de progresso ficar no topo. Sua largura é alterada para indicar o progresso atual. Usamos `%`, em vez de `px`, como unidade ao definir a largura para que o elemento seja dimensionado em relação à largura da página.

A vinculação global `innerHeight` nos dá a altura da janela, que devemos subtrair da altura total rolável — você não pode continuar rolando quando atingir o final do documento. Há também um `innerWidth` para a largura da janela. Dividindo `pageYOffset`, a posição de rolagem atual, pela posição máxima de rolagem e multiplicando por 100, obtemos a porcentagem para a barra de progresso.

Chamar `preventDefault` em um evento de rolagem não impede que a rolagem aconteça. Na verdade, o manipulador de evento é chamado apenas *após* a rolagem ocorrer.

Eventos de foco

Quando um elemento recebe foco, o navegador dispara um evento `"focus"` nele. Quando ele perde o foco, o elemento recebe um evento `"blur"`.

Ao contrário dos eventos discutidos anteriormente, esses dois eventos não propagam. Um manipulador em um elemento pai não é notificado quando um elemento filho ganha ou perde foco.

O exemplo a seguir exibe texto de ajuda para o campo de texto que atualmente tem foco:

```
<p>Name: <input type="text" data-help="Your full name"></p>
<p>Age: <input type="text" data-help="Your age in years"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
```

```
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>
```

Esta captura de tela mostra o texto de ajuda para o campo de idade:

Name: Hieronimus

Age:

Age in years

O objeto window receberá eventos `"focus"` e `"blur"` quando o usuário mover de ou para a aba ou janela do navegador em que o documento é mostrado.

Evento de carregamento

Quando uma página termina de carregar, o evento `"load"` dispara no objeto window e no corpo do documento. Isso é frequentemente usado para agendar ações de inicialização que requerem que o documento inteiro tenha sido construído. Lembre-se que o conteúdo de tags `<script>` é executado imediatamente quando a tag é encontrada. Isso pode ser cedo demais, por exemplo quando o script precisa fazer algo com partes do documento que aparecem após a tag `<script>`.

Elementos como imagens e tags de script que carregam um arquivo externo também têm um evento `"load"` que indica que os arquivos que referenciam foram carregados. Como os eventos relacionados a foco, eventos de carregamento não propagam.

Quando você fecha uma página ou navega para fora dela (por exemplo, seguindo um link), um evento `"beforeunload"` dispara. O uso principal desse evento é impedir que o usuário perca trabalho acidentalmente ao fechar um documento. Se você prevenir o comportamento padrão nesse evento *e* definir a propriedade `returnValue` no objeto de evento como uma string, o navegador mostrará ao usuário um diálogo perguntando se ele realmente quer sair da página. Esse diálogo pode incluir sua string, mas como alguns sites maliciosos tentam usar esses diálogos para confundir pessoas a ficar em suas páginas para ver anúncios duvidosos de perda de peso, a maioria dos navegadores não os exibe mais.

Eventos e o loop de eventos

No contexto do loop de eventos, como discutido no Capítulo 11, manipuladores de eventos do navegador se comportam como outras notificações assíncronas. Eles são agendados quando o evento ocorre, mas devem esperar que outros scripts que estão rodando terminem antes de terem a chance de rodar.

O fato de que eventos só podem ser processados quando nada mais está rodando significa que, se o loop de eventos está preso com outro trabalho, qualquer interação com a página (que acontece através de eventos) será atrasada até que haja tempo para processá-la. Então, se você agendar trabalho demais, seja com manipuladores de evento de longa duração ou com muitos de curta duração, a página se tornará lenta e desagradável de usar.

Para casos em que você *realmente* quer fazer algo demorado em segundo plano sem congelar a página, navegadores fornecem algo chamado *web workers*. Um worker é um processo JavaScript que roda junto ao script principal, em sua própria linha do tempo.

Imagine que elevar um número ao quadrado é uma computação pesada e de longa duração que queremos realizar em uma thread separada. Poderíamos escrever um arquivo chamado `code/squareworker.js` que responde a mensagens calculando um quadrado e enviando uma mensagem de volta.

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

Para evitar os problemas de ter múltiplas threads tocando os mesmos dados, workers não compartilham seu escopo global ou quaisquer outros dados com o ambiente do script principal. Em vez disso, você precisa se comunicar com eles enviando mensagens de um lado para outro.

Este código cria um worker rodando aquele script, envia algumas mensagens a ele e mostra as respostas.

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

A função `postMessage` envia uma mensagem, que causará um evento `"message"` no receptor. O script que criou o worker envia e recebe mensagens através do objeto `Worker`, enquanto o worker conversa com o script que o criou enviando e ouvindo diretamente em seu escopo global. Apenas valores que podem ser

representados como JSON podem ser enviados como mensagens — o outro lado receberá uma *cópia* deles, em vez do valor em si.

Temporizadores

A função `setTimeout` que vimos no Capítulo 11 agenda outra função para ser chamada depois, após um dado número de milissegundos. Às vezes você precisa cancelar uma função que agendou. Pode fazer isso armazenando o valor retornado por `setTimeout` e chamando `clearTimeout` nele.

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% de chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

A função `cancelAnimationFrame` funciona da mesma forma que `clearTimeout`. Chamá-la com um valor retornado por `requestAnimationFrame` cancelará aquele frame (assumindo que ele ainda não tenha sido chamado).

Um conjunto similar de funções, `setInterval` e `clearInterval`, é usado para definir temporizadores que devem se repetir a cada X milissegundos.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Debouncing

Alguns tipos de eventos têm o potencial de disparar rapidamente muitas vezes seguidas, como os eventos `"mousemove"` e `"scroll"`. Ao manipular tais eventos, você deve ter cuidado para não fazer nada que consuma muito tempo, ou seu manipulador ocupará tanto tempo que a interação com o documento começará a parecer lenta.

Se você precisa fazer algo não trivial em tal manipulador, pode usar `setTimeout` para garantir que não está fazendo isso com muita frequência. Isso é geralmente chamado de *debouncing* do evento. Existem várias abordagens ligeiramente diferentes para isso.

Por exemplo, suponha que queremos reagir quando o usuário digitou algo, mas não queremos fazer isso imediatamente para cada evento de entrada. Quando estão digitando rápido, queremos apenas esperar até que ocorra uma pausa. Em vez de executar imediatamente uma ação no manipulador de evento, definimos um temporizador. Também limpamos o temporizador anterior (se houver) para que, quando eventos ocorrem próximos uns dos outros (mais perto do que nosso atraso do temporizador), o temporizador do evento anterior seja cancelado.

```
<textarea>Type something here...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);
  });
</script>
```

Passar um valor indefinido para `clearTimeout` ou chamá-lo em um temporizador que já disparou não tem efeito. Assim, não precisamos ter cuidado sobre quando chamá-lo, e simplesmente o fazemos para cada evento.

Podemos usar um padrão ligeiramente diferente se quisermos espaçar as respostas para que sejam separadas por pelo menos um certo período de tempo, mas quisermos dispará-las *durante* uma série de eventos, não apenas depois. Por exemplo, podemos querer responder a eventos `"mousemove"` mostrando as coordenadas atuais do mouse, mas apenas a cada 250 milissegundos.

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
```

```
</script>
```

Resumo

Manipuladores de eventos tornam possível detectar e reagir a eventos acontecendo em nossa página web. O método `addEventListener` é usado para registrar tal manipulador.

Cada evento tem um tipo (`"keydown"`, `"focus"`, e assim por diante) que o identifica. A maioria dos eventos é chamada em um elemento DOM específico e depois propaga para os ancestrais daquele elemento, permitindo que manipuladores associados a esses elementos os tratem.

Quando um manipulador de evento é chamado, recebe um objeto de evento com informações adicionais sobre o evento. Esse objeto também tem métodos que nos permitem parar a propagação adicional (`stopPropagation`) e impedir o tratamento padrão do navegador para o evento (`preventDefault`).

Pressionar uma tecla dispara eventos `"keydown"` e `"keyup"`. Pressionar um botão do mouse dispara eventos `"mousedown"`, `"mouseup"` e `"click"`. Mover o mouse dispara eventos `"mousemove"`. Interação com tela de toque resultará em eventos `"touchstart"`, `"touchmove"` e `"touchend"`.

Rolagem pode ser detectada com o evento `"scroll"`, e mudanças de foco podem ser detectadas com os eventos `"focus"` e `"blur"`. Quando o documento termina de carregar, um evento `"load"` dispara no objeto window.

Exercícios

Balão

Escreva uma página que exiba um balão (usando o emoji de balão,

`font-sizestyle.fontSize10px`

`"ArrowUp""ArrowDown"`

Rastro do mouse

`<div>`

`"mousemove"`

Abas

`asTabs<button>data-tabnamedisplaynone`

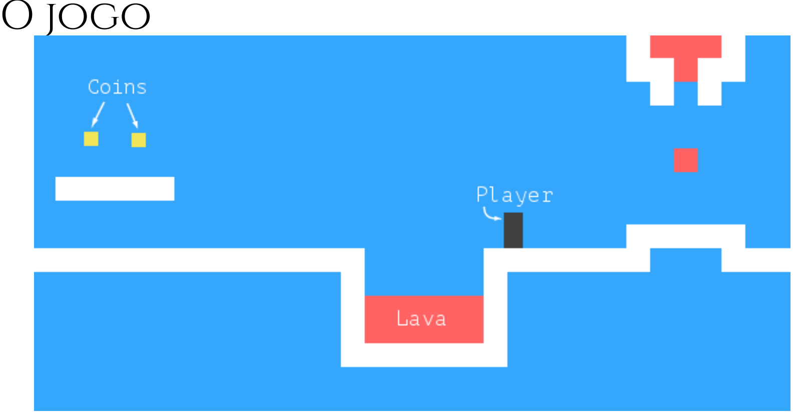
"All reality is a game."

—Iain Banks, The Player of Games

Chapter 16

Projeto: Um Jogo de Plataforma

O jogo



A tecnologia

<canvas>

Níveis

```
let simpleLevelPlan = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
```

```
......##############..
......................`;
```

#@=

|v

Lendo um nível

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
        let type = levelChars[ch];
        if (typeof type != "string") {
          let pos = new Vec(x, y);
          this.startActors.push(type.create(pos, ch));
          type = "empty";
        }
        return type;
      });
    });
  }
}
```

trim

rows"empty""wall""lava"

map

LevellevelCharstypecreatestartActors"empty"

Vecxy

State

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }

  static start(level) {
    return new State(level, level.startActors, "playing");
  }
```

```
  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

status"lost""won"

ATORESsizeposupdate

update

type"player""coin""lava"

createLevelLava

Vec

```
class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vec(this.x * factor, this.y * factor);
  }
}
```

times

update

speed

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vec(0, -0.5)),
                      new Vec(0, 0));
  }
}

Player.prototype.size = new Vec(0.8, 1.5);
```

@

sizePlayertypeVec

Lavareset

createLevel

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vec(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vec(1, 1);
```

Coinwobblepos

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vec(0.6, 0.6);
```

Math.sin

Math.sinπMath.random

levelChars

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

Level

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

Desenhando

DOMDisplay

style

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

actorLayer

scale

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
```

```
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type})))
  ));
}
```

<table>rows<tr><td>elt

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

table-layoutborder-spacingpadding

backgroundwhitergb(R, G, B)rgb(52, 166, 251).lava

scale

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

actorlava

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

syncState

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

```
.lost .player {
  background: rgb(160, 64, 64);
```

```
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

scrollPlayerIntoView

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

scrollPlayerIntoViewscrollLeftscrollTop

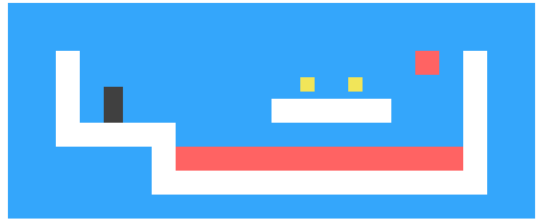
```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // O viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
};
```

Vec

scrollLeft-100

```
<link rel="stylesheet" href="css/game.css">
```

```
<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```

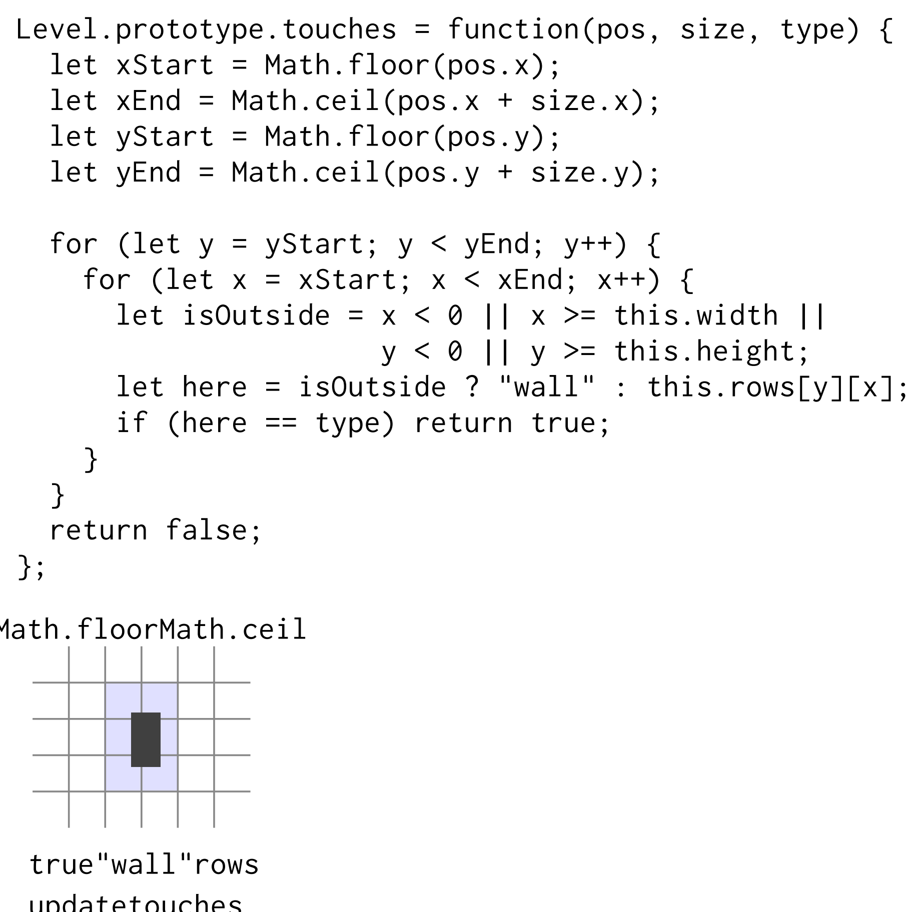


<link>rel="stylesheet"game.css

Movimento e colisão

```
Level.prototype.touches = function(pos, size, type) {
  let xStart = Math.floor(pos.x);
  let xEnd = Math.ceil(pos.x + size.x);
  let yStart = Math.floor(pos.y);
  let yEnd = Math.ceil(pos.y + size.y);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
                      y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

Math.floorMath.ceil



true"wall"rows

updatetouches

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status != "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
  return newState;
};
```

update

overlaptrue

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

collide"lost""won"

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

Atualizações dos atores

updatekeysLavakeys

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
```

```
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

updatereset-1

update

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
                  this.basePos, wobble);
};
```

wobbleMath.sin

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vec(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
  }
  return new Player(pos, new Vec(xSpeed, ySpeed));
};
```

ySpeed

Rastreando teclas

preventDefault

"keydown""keyup"

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

type"keydown""keyup"

Executando o jogo

requestAnimationFramerequestAnimationFrame

runAnimationfalse

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

requestAnimationFramelastTimetime

runLevelLeveldocument.bodyrunLevel

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
```

```
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

runLevelrunGameasync

GAME_LEVELSrunGame

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Exercícios

Game over

runGameconsole.log

Pausando o jogorunLevel

runAnimationrunLevel

arrowKeystrackKeysrunLevel

“Desenhar é enganar.”

—M.C. Escher, cited by Bruno Ernst in The Magic Mirror of M.C. Escher

Chapter 17

Desenhando no Canvas

transform

SVG

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

xmlns<circle><rect>

Normal HTML here.

<circle>

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

O elemento canvas<canvas>widthheight

<canvas>"2d""webgl""webgpu"

getContext<canvas>

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
```

```
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

Before canvas.

After canvas.

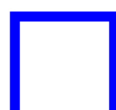
Linhas e superfícies

`fillRect` `strokeRect`

`fillStyle`

`strokeStyle` `lineWidth`

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```



`width` `height`

Caminhos

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

strokelineTomoveTomoveTo

fillmoveTo

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

closePath

Curvas

quadraticCurveTo

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // controle=(60, 10) destino=(90, 90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

bezierCurveTo

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // controle1=(10, 10) controle2=(90, 10) destino=(50, 90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

arc

π2 * Math.PIπ

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // centro=(50, 50) raio=40 ângulo=0 a 7
  cx.arc(50, 50, 40, 0, 7);
  // centro=(150, 50) raio=40 ângulo=0 a π½
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

arc

arcmoveTo

Desenhando um gráfico de pizza

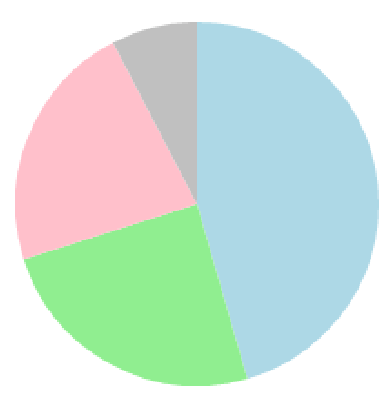
results

```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
```

```
    {name: "Unsatisfied", count: 510, color: "pink"},
    {name: "No comment", count: 175, color: "silver"}
  ];
```

π

```
  <canvas width="200" height="200"></canvas>
  <script>
    let cx = document.querySelector("canvas").getContext("2d");
    let total = results
      .reduce((sum, {count}) => sum + count, 0);
    // Começar no topo
    let currentAngle = -0.5 * Math.PI;
    for (let result of results) {
      let sliceAngle = (result.count / total) * 2 * Math.PI;
      cx.beginPath();
      // centro=100,100, raio=100
      // do ângulo atual, sentido horário pelo ângulo da fatia
      cx.arc(100, 100, 100,
             currentAngle, currentAngle + sliceAngle);
      currentAngle += sliceAngle;
      cx.lineTo(100, 100);
      cx.fillStyle = result.color;
      cx.fill();
    }
  </script>
```



TEXTO`fillText``strokeText``fillText``fillStyle`

```
  <canvas></canvas>
  <script>
    let cx = document.querySelector("canvas").getContext("2d");
    cx.font = "28px Georgia";
    cx.fillStyle = "fuchsia";
    cx.fillText("I can draw text, too!", 10, 50);
```

```
</script>
```

fontitalicbold

fillTextstrokeTexttextAlign"end""center"textBaseline"top""middle""bottom
"

Imagens

drawImage<img><img>"load"

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

drawImage

drawImage

clearRectfillRect

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // retângulo de origem
                   cycle * spriteW, 0, spriteW, spriteH,
                   // retângulo de destino
                   0,               0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
```

```
  });
</script>
```

cycle

Transformação

scale

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

scale

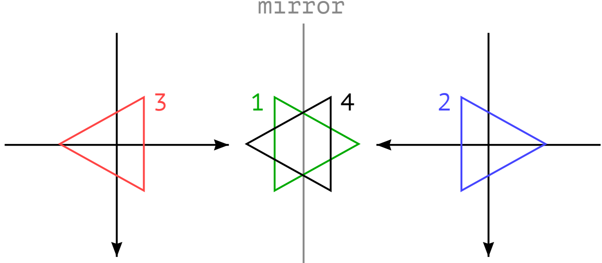
cx.scale(-1, 1)drawImagedrawImage

scalerotatetranslate

π

translate(50, 50)
rotate(0.1*Math.PI)
rotate(0.1*Math.PI)
translate(50, 50)

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```



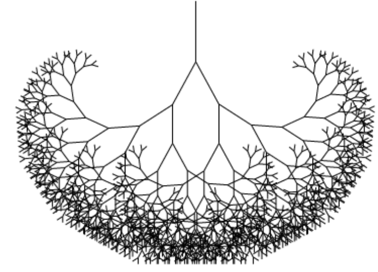
flipHorizontallytranslate

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

Armazenando e limpando transformações

saverestoresaverestoreresetTransform
branch

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```



saverestorebranch

De volta ao jogo

drawImage

CanvasDisplayDOMDisplaysyncStateclear

DOMDisplayflipPlayer

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

syncState

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

DOMDisplay

updateViewportscrollPlayerIntoViewDOMDisplay

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                         state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
};
```

Math.maxMath.minMath.max(x, 0)Math.min

```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height);
};
```

touches

```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
```

```
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

drawImageotherSprites

DOMDisplayscale

drawImage

playerXOverlap

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
    this.flipPlayer = player.speed.x < 0;
  }

  let tile = 8;
  if (player.speed.y != 0) {
    tile = 9;
  } else if (player.speed.x != 0) {
    tile = Math.floor(Date.now() / 60) % 8;
  }

  this.cx.save();
  if (this.flipPlayer) {
    flipHorizontally(this.cx, x + width / 2);
  }
  let tileX = tile * width;
  this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                   x,     y, width, height);
```

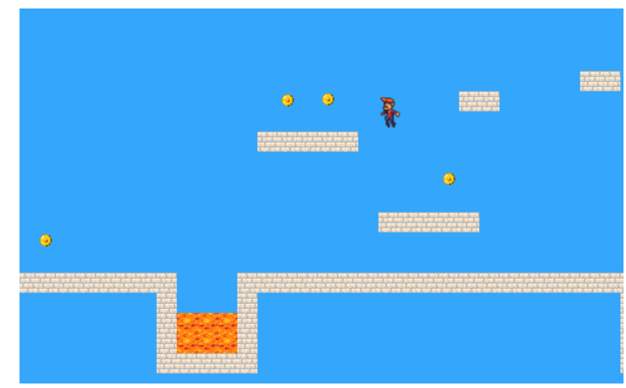
```
    this.cx.restore();
  };
```

`drawPlayerdrawActors`

```
CanvasDisplay.prototype.drawActors = function(actors) {
  for (let actor of actors) {
    let width = actor.size.x * scale;
    let height = actor.size.y * scale;
    let x = (actor.pos.x - this.viewport.left) * scale;
    let y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(actor, x, y, width, height);
    } else {
      let tileX = (actor.type == "coin" ? 2 : 1) * scale;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }
};
```

`scale`

`translate`



Escolhendo uma interface gráfica

Resumo<canvas>

`getContext`

`fillStylestrokeStylelineWidth`

`fillRect` `strokeRect` `fillText` `strokeText`

`beginPath` `lineTo` `fill` `stroke`

`drawImage`

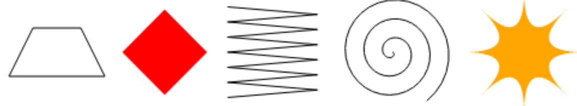
`translate` `scale` `rotate` `save` `restore`

`clearRect`

Exercícios

Formas

π



`Math.cos` `Math.sin`

O gráfico de pizza

`Math.sin` `Math.cos`

Uma bola quicando `requestAnimationFrame`

Espelhamento pré-computado

`drawImage`

"O que frequentemente era difícil para as pessoas entenderem sobre o design era que não havia nada além de URLs, HTTP e HTML. Não havia um computador central 'controlando' a web, nenhuma rede única na qual esses protocolos funcionavam, nem mesmo uma organização em algum lugar que 'administrava' a Web. A Web não era uma 'coisa' física que existia em um certo 'lugar'. Era um 'espaço' no qual informações podiam existir."

—Tim Berners-Lee

Chapter 18

HTTP e Formulários

O protocolo

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

```
HTTP/1.1 200 OK
Content-Length: 87320
Content-Type: text/html
Last-Modified: Fri, 13 Oct 2023 10:05:41 GMT

<!doctype html>
... the rest of the document
```

<body>

```
GET /18_http.html HTTP/1.1
```

GETDELETEPUTPOSTDELETE

HTTP/1.1

```
HTTP/1.1 200 OK
```

nome: valor

```
Content-Length: 87320
Content-Type: text/html
Last-Modified: Fri, 13 Oct 2023 10:05:41 GMT
```

Content-Type
GETDELETEPUTPOST

Navegadores e HTTP

GET

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

method<form>GETaction

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

name&

%3FencodeURIComponentdecodeURIComponent

```
console.log(encodeURIComponent("Yes?"));
// → Yes%3F
console.log(decodeURIComponent("Yes%3F"));
// → Yes?
```

methodPOSTPOST

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

GETPOSTPOSTGET

FETCHfetch

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

fetchResponseMapheaders.get("Content-Type")headers.get("content-TYPE")

fetch

fetch

text

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
```

```
// → This is the content of data.txt
```

json

fetchGETexample/data.txt

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

PUTPOSTbodyheadersRange

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → the content
```

Sandboxing HTTP

```
Access-Control-Allow-Origin: *
```

Apreciando o HTTP

addUserPUT/users/larryPUTGET/users/larry

Segurança e HTTPS

Campos de formulário

<form>

<input>type<input>

text

password text

checkbox

color

date

radio

file

<form>

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="color" value="orange"> (color)</p>
<p><input type="date" value="2023-10-13"> (date)</p>
<p><input type="radio" value="A" name="choice">
   <input type="radio" value="B" name="choice" checked>
   <input type="radio" value="C" name="choice"> (radio)</p>
<p><input type="file"> (file)</p>
```

abc (text)

••• (password)

(checkbox)

(radio)

Choose File snippets.txt (file)

<textarea><textarea></textarea>value

```
<textarea>
one
two
three
</textarea>
```

<select>

```
<select>
  <option>Pancakes</option>
  <option>Pudding</option>
  <option>Ice cream</option>
</select>
```

Pancakes

Pancakes
Pancakes
Pudding
Ice cream

"change"

FOCO

<select>

focusblurdocument.activeElement

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
```

```
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

autofocus

tabindex

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

tabindextabindex

Campos desabilitadosdisabled

```
<button>I'm all right</button>
<button disabled>I'm out</button>
```

I'm all right I'm out

O formulário como um todo<form>form<form>elements

nameelements

```
<form action="example/submit.html">
  Name: <input type="text" name="name"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Log in</button>
</form>
<script>
  let form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

typesubmit

actionGETPOST"submit"preventDefault

```
<form>
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.value.value);
```

```
    event.preventDefault();
  });
</script>
```

"submit"fetch

Campos de texto

<textarea><input>textpasswordvalue

selectionStartselectionEndvalue

<textarea>

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    if (event.key == "F2") {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // Colocar o cursor após a palavra
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
  }
</script>
```

replaceSelection

"change""input"

```
<input type="text"> length: <span id="length">0</span>
<script>
  let text = document.querySelector("input");
  let output = document.querySelector("#length");
  text.addEventListener("input", () => {
    output.textContent = text.value.length;
  });
</script>
```

Checkboxes e botões de rádio

checked

```
<label>
  <input type="checkbox" id="purple"> Make this page purple
</label>
<script>
  let checkbox = document.querySelector("#purple");
```

```
  checkbox.addEventListener("change", () => {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

<label>

name

```
Color:
<label>
  <input type="radio" name="color" value="orange"> Orange
</label>
<label>
  <input type="radio" name="color" value="lightgreen"> Green
</label>
<label>
  <input type="radio" name="color" value="lightblue"> Blue
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

querySelectorAllname"color"

Campos de seleção<select>

multiple<select>multiple

<option>valuevalue<select>multiple

<option><select>optionsselected

multiple

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
```

```
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

Campos de arquivo

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("You chose", file.name);
      if (file.type) console.log("It has type", file.type);
    }
  });
</script>
```

`filesfilemultiple`

`filesnamesizetypetext/plainimage/jpeg`

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

`FileReader"load"readAsTextresult`

`FileReader"error"error`

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
```

```
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

Armazenando dados no lado do cliente

localStorage

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

localStorageremoveItem

localStorage

localStorage

```
Notes: <select></select> <button>Add</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(newState) {
    list.textContent = "";
    for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
    }
    note.value = newState.notes[newState.selected];

    localStorage.setItem("Notes", JSON.stringify(newState));
    state = newState;
  }
  setState(JSON.parse(localStorage.getItem("Notes")) ?? {
    notes: {"shopping list": "Carrots\nRaisins"},
    selected: "shopping list"
  });
```

```
    list.addEventListener("change", () => {
      setState({notes: state.notes, selected: list.value});
    });
    note.addEventListener("change", () => {
      let {selected} = state;
      setState({
        notes: {...state.notes, [selected]: note.value},
        selected
      });
    });
    document.querySelector("button")
      .addEventListener("click", () => {
        let name = prompt("Note name");
        if (name) setState({
          notes: {...state.notes, [name]: ""},
          selected: name
        });
      });
  </script>
```

"Notes"localStoragelocalStoragenullnullJSON.parse"null"null??

setStatelocalStorage

...state.notes

localStoragesessionStoragesessionStorage

ResumoGET

fetch

```
fetch("/18_http.html").then(r => r.text()).then(text => {
  console.log(`The page starts with ${text.slice(0, 15)}`);
});
```

GET

"change""input"valuechecked

"submit"preventDefault

FileReader

localStoragesessionStorage

Exercícios

Negociação de conteúdoAccept

text/plaintext/htmlapplication/json

headersfetchAccept

application/rainbows+unicorns

Uma bancada de JavaScript

<textarea>Function

Jogo da Vida de Conway

-
-
-

`Math.random`

> *"Olho para as muitas cores diante de mim. Olho para minha tela em branco. Então, tento aplicar cores como palavras que formam poemas, como notas que formam música."*
>
> —Joan Miró

Chapter 19

Projeto: Um Editor de Pixel Art

Componentes<canvas><select>

dom

syncState

O EstadoPicturetoolcolor

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
```

```
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
  draw(pixels) {
    let copy = this.pixels.slice();
    for (let {x, y, color} of pixels) {
      copy[x + y * this.width] = color;
    }
    return new Picture(this.width, this.height, copy);
  }
}
```

drawxycolorslice

emptyArrayfill

#fillStyle

"#000000""#ff00ff"ff

{color: field.value}

```
function updateState(state, action) {
  return {...state, ...action};
}
```

Construção de DOMelt

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child != "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

onclick

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("click")
    }, "The button"));
  </script>
</body>
```

O canvas

```
const scale = 10;

class PictureCanvas {
  constructor(picture, pointerDown) {
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    this.syncState(picture);
  }
  syncState(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  }
}
```

scalesyncState

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

pointerDown

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button != 0) return;
  let pos = pointerPosition(downEvent, this.dom);
  let onMove = onDown(pos);
  if (!onMove) return;
  let move = moveEvent => {
    if (moveEvent.buttons == 0) {
      this.dom.removeEventListener("mousemove", move);
    } else {
      let newPos = pointerPosition(moveEvent, this.dom);
      if (newPos.x == pos.x && newPos.y == pos.y) return;
      pos = newPos;
```

```
      onMove(newPos);
    }
  };
  this.dom.addEventListener("mousemove", move);
};

function pointerPosition(pos, domNode) {
  let rect = domNode.getBoundingClientRect();
  return {x: Math.floor((pos.clientX - rect.left) / scale),
          y: Math.floor((pos.clientY - rect.top) / scale)};
}
```

getBoundingClientRectclientXclientY

preventDefault"touchstart"

```
PictureCanvas.prototype.touch = function(startEvent,
                                         onDown) {
  let pos = pointerPosition(startEvent.touches[0], this.dom);
  let onMove = onDown(pos);
  startEvent.preventDefault();
  if (!onMove) return;
  let move = moveEvent => {
    let newPos = pointerPosition(moveEvent.touches[0],
                                 this.dom);
    if (newPos.x == pos.x && newPos.y == pos.y) return;
    pos = newPos;
    onMove(newPos);
  };
  let end = () => {
    this.dom.removeEventListener("touchmove", move);
    this.dom.removeEventListener("touchend", end);
  };
  this.dom.addEventListener("touchmove", move);
  this.dom.addEventListener("touchend", end);
};
```

clientXclientYtouches

A APLICAÇÃO

<select>dispatch

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
```

```
      let tool = tools[this.state.tool];
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                   ...this.controls.reduce(
                     (a, c) => a.concat(" ", c.dom), []));
  }
  syncState(state) {
    this.state = state;
    this.canvas.syncState(state.picture);
    for (let ctrl of this.controls) ctrl.syncState(state);
  }
}
```

PictureCanvas

this.controlsreduce

<select>"change"

```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, " Tool: ", this.select);
  }
  syncState(state) { this.select.value = state.tool; }
}
```

<label>

<input>typecolor"#RRGGBB"

color

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
    });
    this.dom = elt("label", null, " Color: ", this.input);
  }
  syncState(state) { this.input.value = state.color; }
}
```

Ferramentas de desenho

```
function draw(pos, state, dispatch) {
  function drawPixel({x, y}, state) {
    let drawn = {x, y, color: state.color};
    dispatch({picture: state.picture.draw([drawn])});
  }
  drawPixel(pos, state);
  return drawPixel;
}
```

drawPixel

rectangle

```
function rectangle(start, state, dispatch) {
  function drawRectangle(pos) {
    let xStart = Math.min(start.x, pos.x);
    let yStart = Math.min(start.y, pos.y);
    let xEnd = Math.max(start.x, pos.x);
    let yEnd = Math.max(start.y, pos.y);
```

```
    let drawn = [];
    for (let y = yStart; y <= yEnd; y++) {
      for (let x = xStart; x <= xEnd; x++) {
        drawn.push({x, y, color: state.color});
      }
    }
    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}
```

```
const around = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function fill({x, y}, state, dispatch) {
  let targetColor = state.picture.pixel(x, y);
  let drawn = [{x, y, color: state.color}];
  let visited = new Set();
  for (let done = 0; done < drawn.length; done++) {
    for (let {dx, dy} of around) {
      let x = drawn[done].x + dx, y = drawn[done].y + dy;
      if (x >= 0 && x < state.picture.width &&
          y >= 0 && y < state.picture.height &&
          !visited.has(x + "," + y) &&
          state.picture.pixel(x, y) == targetColor) {
        drawn.push({x, y, color: state.color});
        visited.add(x + "," + y);
      }
    }
  }
  dispatch({picture: state.picture.draw(drawn)});
}
```

drawn

```
function pick(pos, state, dispatch) {
```

```
    dispatch({color: state.picture.pixel(pos.x, pos.y)});
  }
```

Salvando e carregando

```
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, " Save");
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  syncState(state) { this.picture = state.picture; }
}
```

<canvas>

toDataURLdata:http:https:

download

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, " Load");
  }
  syncState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
```

```
}
```

FileReader<img>Picture

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
    });
  });
  reader.readAsDataURL(file);
}
```

<canvas>getImageDataPicture

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
  }
  return new Picture(width, height, pixels);
}
```

datagetImageData
toStringn.toString(16)hexpadStart

Histórico de desfazer

done

doneAt

```
function historyUpdateState(state, action) {
  if (action.undo == true) {
    if (state.done.length == 0) return state;
    return {
      ...state,
      picture: state.done[0],
      done: state.done.slice(1),
      doneAt: 0
    };
  } else if (action.picture &&
             state.doneAt < Date.now() - 1000) {
    return {
      ...state,
      ...action,
      done: [state.picture, ...state.done],
      doneAt: Date.now()
    };
  } else {
    return {...state, ...action};
  }
}
```

doneAt

donedoneAt

```
class UndoButton {
  constructor(state, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, " Undo");
  }
  syncState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}
```

VAMOS DESENHAR PixelEditor

```
const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};
```

```
const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                           tools = baseTools,
                           controls = baseControls}) {
  let app = new PixelEditor(state, {
    tools,
    controls,
    dispatch(action) {
      state = historyUpdateState(state, action);
      app.syncState(state);
    }
  });
  return app.dom;
}
```

=undefinedstartPixelEditortoolstoolsbaseTools

```
<div></div>
<script>
  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

Por que isso é tão difícil?

Exercícios

Atalhos de teclado

PixelEditortabIndex<div>tabindextabIndexelt

ctrlKeymetaKey

Desenho eficiente

drawPicture

syncStatePictureCanvas

drawPicture

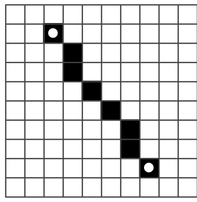
<canvas>widthheight

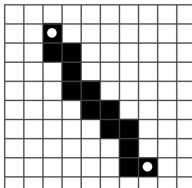
Círculos

circle

Linhas corretas

```
draw"mousemove""touchmove"
draw
```





```
line
```

> *"Um estudante perguntou: 'Os programadores de antigamente usavam apenas máquinas simples e nenhuma linguagem de programação, mas mesmo assim faziam programas bonitos. Por que usamos máquinas complicadas e linguagens de programação?' Fu-Tzu respondeu: 'Os construtores de antigamente usavam apenas gravetos e argila, mas mesmo assim faziam cabanas bonitas."'*
>
> —Master Yuan-Ma, The Book of Programming

Chapter 20

Node.js

Contexto

O comando node`nodehello.js`

```
let message = "Hello world";
console.log(message);
```

`node`

```
$ node hello.js
Hello world
```

`console.lognode`

`node`

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

`processconsoleexitnode`

`process.argvnodeshowargv.jsconsole.log(process.argv)`

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

`ArrayMathJSONdocumentprompt`

Módulos`consoleprocess`

`require.js`

`.mjsimportexportrequire`

requireimport/./../...//"./graph.mjs"/tmp/robot/robot.mjs/tmp/robot/graph.mjs

node_modules"node:fs""robot"node_modules/robot/

main.mjs

```
import {reverse} from "./reverse.mjs";

// O índice 2 contém o primeiro argumento real da linha de comando
let argument = process.argv[2];

console.log(reverse(argument));
```

reverse.mjs

```
export function reverse(string) {
  return Array.from(string).reverse().join("");
}
```

exportmain.mjs

```
$ node main.mjs JavaScript
tpircSavaJ
```

Instalando com NPMnpm

ini

```
$ npm install ini
added 1 package in 723ms

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

npm installnode_modulesini"ini"parse

Arquivos de pacotenpm installnode_modulespackage.jsonnpm init

package.json

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulation of a package-delivery robot",
  "version": "1.0.0",
  "main": "run.mjs",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
```

```
  "license": "ISC"
}
```

npm installpackage.jsonpackage.json

VERSÕESpackage.json

2.3.0

^package.json"^2.3.0"

npmnpm publishpackage.json

O MÓDULO DE SISTEMA DE ARQUIVOSnode:fs

readFile

```
import {readFile} from "node:fs";
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("The file contains:", text);
});
```

readFile"utf8"Buffer

```
import {readFile} from "node:fs";
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

writeFile

```
import {writeFile} from "node:fs";
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

writeFileBuffer

node:fsreaddirstatrenameunlink

node:fs/promisesnode:fs

```
import {readFile} from "node:fs/promises";
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text));
```

node:fsSyncreadFilereadFileSync

```
import {readFileSync} from "node:fs";
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

O MÓDULO HTTPnode:http

```
import {createServer} from "node:http";
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

createServerrequestresponseurl

responsewriteHeadContent-Type

response.writeresponse.end

server.listen

node

method

node:httprequestfetchfetchfetch

STREAMSwriteBufferend

createWriteStreamnode:fswritewriteFile

request

onaddEventListener

"data""end"createReadStreamnode:fs

```
import {createServer} from "node:http";
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

chunkBuffertoString

```
fetch("http://localhost:8000/", {
  method: "POST",
  body: "Hello server"
}).then(resp => resp.text()).then(console.log);
```

```
// → HELLO SERVER
```

Um servidor de arquivos

GETPUTDELETE

/tmp/public/C:\tmp\public\/file.txt/tmp/public/file.txtC:\tmp\public
\file.txt

methodsasync

```
import {createServer} from "node:http";

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request).catch(error => {
    if (error.status != null) return error;
    return {body: String(error), status: 500};
  }).then(({body, status = 200, type = "text/plain"}) => {
    response.writeHead(status, {"Content-Type": type});
    if (body?.pipe) body.pipe(response);
    else response.end(body);
  });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

catch

statustype

bodypipenullend

urlPathURLrequest.url"/file.txt"%20

```
import {resolve, sep} from "node:path";

const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = new URL(url, "http://d");
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
```

```
  }
  return path;
}
```

..//../secret_file

urlPathresolvenode:pathprocess.cwdcwdsepnode:path

GET

Content-Typemime-typestext/plain

npmmime

```
$ npm install mime-types@2.1.0
```

stat

```
import {createReadStream} from "node:fs";
import {stat, readdir} from "node:fs/promises";
import {lookup} from "mime-types";

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: lookup(path)};
  }
};
```

statnode:fs/promisesnode:fs

statcode"ENOENT"

statsstatsizemtimeisDirectory

readdircreateReadStreammime

DELETE

```
import {rmdir, unlink} from "node:fs/promises";

methods.DELETE = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
```

```
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

PUT

```
import {createWriteStream} from "node:fs";

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

pipepipepipeStreampipe

createWriteStream"error""error"pipe"finish"

curl-X-d

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d CONTENT http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
CONTENT
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

file.txtPUTDELETE

Resumo

`npm` `node:fs` `node:http`

`readFileSync` `node:fs/promises`

Exercícios

Ferramenta de busca `grep`

`grep`

Criação de diretório `DELETE` `rmdir`

`MKCOL` `mkdir` `node:fs` `MKCOL`

Um espaço público na web `Content-Type`

> *“Se você tem conhecimento, deixe que outros acendam suas velas nele.”*
>
> —Margaret Fuller

Chapter 21

Projeto: Website de Compartilhamento de Habilidades

Design

Long polling

Interface HTTP

/talks/talks

GET/talks

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]
```

PUT/talks/UnituningPUTpresentersummary

encodeURIComponent

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

GETDELETE

POST/talks/Unituning/commentsauthormessage

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
 "message": "Will you talk about raising a cycle?"}
```

GET/talksETagIf-None-Match

ETagIf-None-Match

Prefer: wait=90

ETag

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(tempo passa)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295
```

```
[...]
```

O servidor

ROTEAMENTO`createServer``if`

`PUT``/^\/talks\/([^\/]+)$/``/talks/`

```
router.mjs
export class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  async resolve(request, context) {
    let {pathname} = new URL(request.url, "http://d");
    for (let {method, url, handler} of this.routes) {
      let match = url.exec(pathname);
      if (!match || request.method != method) continue;
      let parts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...parts, request);
    }
  }
}
```

`Router``add``resolve`

`context``resolve``%20`

SERVINDO ARQUIVOS`public``PUT``DELETE`

`serve-static``serve-static``request``response``"node:http"`

```
import {createServer} from "node:http";
import serveStatic from "serve-static";

function notFound(request, response) {
  response.writeHead(404, "Not found");
  response.end("<h1>Not found</h1>");
}

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];
```

```
    let fileServer = serveStatic("./public");
    this.server = createServer((request, response) => {
      serveFromRouter(this, request, response, () => {
        fileServer(request, response,
                   () => notFound(request, response));
      });
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

serveFromRouterfileServer(request, response, next)notFound

serveFromRouter

```
import {Router} from "./router.mjs";

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

async function serveFromRouter(server, request,
                               response, next) {
  let resolved = await router.resolve(request, server)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(err), status: 500};
    });
  if (!resolved) return next();
  let {body, status = 200, headers = defaultHeaders} =
    await resolved;
  response.writeHead(status, headers);
  response.end(body);
}
```

Palestras como recursos

talks/talks/<título>

GET

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (Object.hasOwn(server.talks, title)) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
```

```
      return {status: 404, body: `No talk '${title}' found`};
    }
  });
```

talks

```
  router.add("DELETE", talkPath, async (server, title) => {
    if (Object.hasOwn(server.talks, title)) {
      delete server.talks[title];
      server.updated();
    }
    return {status: 204};
  });
```

updated

PUTpresentersummary

talksupdated

json"node:stream/consumers"textbufferjsonreadJSON

```
  import {json as readJSON} from "node:stream/consumers";

  router.add("PUT", talkPath,
             async (server, title, request) => {
    let talk = await readJSON(request);
    if (!talk ||
        typeof talk.presenter != "string" ||
        typeof talk.summary != "string") {
      return {status: 400, body: "Bad talk data"};
    }
    server.talks[title] = {
      title,
      presenter: talk.presenter,
      summary: talk.summary,
      comments: []
    };
    server.updated();
    return {status: 204};
  });
```

readJSON

```
  router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
             async (server, title, request) => {
    let comment = await readJSON(request);
    if (!comment ||
        typeof comment.author != "string" ||
        typeof comment.message != "string") {
      return {status: 400, body: "Bad comment data"};
    } else if (Object.hasOwn(server.talks, title)) {
```

```
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Suporte a long pollingGET/talks

ETag

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = Object.keys(this.talks)
    .map(title => this.talks[title]);
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`,
              "Cache-Control": "no-store"}
  };
};
```

If-None-MatchPrefer

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

Prefer

waitingwaitForChanges

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
```

```
};
```

updatedversion

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

SkillShareServerpublic/talks

```
new SkillShareServer({}).start(8000);
```

O CLIENTE

HTMLindex.htmlserve-static/./public/index.html./public

public/index.html

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

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

AÇÕES{talks, user}

handleAction

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return {...state, user: action.user};
  } else if (action.type == "setTalks") {
    return {...state, talks: action.talks};
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
```

```
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

localStorage

fetchfetchOK

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

reportErrorcatch

```
function reportError(error) {
  alert(String(error));
}
```

Renderizando componentes

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

elt

```
function renderTalk(talk, dispatch) {
```

```
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

"submit"form.reset"newComment"

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
```

```
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

POLLINGETag/talks

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

async

setTimeoutasync

ETag

A APLICAÇÃO

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
```

```
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

Exercícios`npm install`

Persistência em disco

Reinicialização de campo de comentário

Exercise Hints

Estrutura do Programa

Fazendo um triângulo com loop`for`

`+= 1"#""##"+= "#"`

FizzBuzz`%`

`ifelse ifelse`

`||`

Tabuleiro de xadrez`"""\n"`

`% 2`

Funções

Mínimo

`return`

Recursão`findfindSolutionifelse ifelseelsereturn`

Contando feijões`< string.length`

`letconst`

Estruturas de Dados: Objetos e Arrays

A soma de um intervalo`[]push`

`<=<`

`=`

`range>=<=`

`range(5, 2)`

Invertendo um array`reverseArrayunshiftpushfor(let i = array.length - 1; i >= 0; i--)`

`reverseArrayarray.slice()`

`Math.flooriarray.length - 1 - i`

Uma lista`arrayToListlist = {value: X, rest: list}`

`listToArraynthfor`

```
for (let node = list; node; node = node.rest) {}
```

`nodevaluenodenull`

`nthvaluevaluerest`

Comparação profunda`typeof x == "object" && x != null===`

`Object.keys`

`falsetrue`

Funções de Ordem Superior

Everything`&&everybreakreturnfalsetrue`

`everysomea && b!(!a || !b)`

Direção de escrita dominante`textScriptscharacterScript`

`reducereduce`

A Vida Secreta dos Objetos

Um tipo vetor`Rabbitclass`

`get`$\sqrt{x^2 + y^2}$`Math.sqrtx ** 2`

Grupos`includesindexOf`

`addpush`

`deletefilter`

`fromforofadd`

Grupos iteráveis`GroupIteratornext`

`GroupSymbol.iterator`

Projeto: Um Robô

Medindo um robô`runRobot`

`console.log`

Eficiência do robô`goalOrientedRobot`

Grupo persistente

`concat`

`empty`

`empty`

Bugs e Erros

Repetir`primitiveMultiplytrycatchMultiplicatorUnitFailure`

`look`

A caixa trancada`finallytryfinally`

Expressões Regulares

Estilo de citação`/\P{L}'|'\P{L}/u`

`\P{L}$1$2`

Números novamente

`[+\-]?(\+|-|)`

`"5."".5""."|`

`i[eE]`

Módulos

Um robô modular

`graph.js` `dijkstrajs` `dijkstrajs` `buildGraph` `buildGraph`

`roads.js` `roads` `roadGraph` `./graph.js`

`VillageState` `state.js` `./roads.js` `randomPick` `state.js` `randomRobot` `random-item` `runRobot` `VillageState` `runRobot`

`mailRoute` `example-robots.js` `./roads.js` `goalOrientedRobot` `dijkstrajs`

`VillageState`

`random-item`

Módulo de estradas

`import` `buildGraph` `const`

`roadGraph` `export` `buildGraph` `roads`

Dependências circulares

`require` `require`

Programação Assíncrona

Tempos Tranquilos

`split` `textFile` `new Date`

`await` `Promise`

Promises Reais

`then` `textFile` `Promise.all` `Promise.all` `then`

`then`

```
function activityTable(day) {
  return textFile("camera_logs.txt").then(files => {
    return Promise.all(files.split("\n").map(textFile));
  }).then(logs => {
    // analisar...
  });
}
```

`Promise.all`

```
function activityTable(day) {
  let table = []; // inicializar...
  return textFile("camera_logs.txt").then(files => {
    return Promise.all(files.split("\n").map(name => {
      return textFile(name).then(log => {
        // analisar...
      });
    }));
  }).then(() => table);
```

}

awaitPromise.all

textFilePromise.allthenthenthen

Construindo Promise.allPromisethen

rejectcatchthen

Projeto: Uma Linguagem de Programação

Arrays

array

ClosureFunscope

Comentários

execmatch

Corrigindo o escopoObject.getPrototypeOfObject.hasOwnnamesetset

Object.getPrototypeOfnull

O Modelo de Objeto do Documento

Construa uma tabeladocument.createElementdocument.createTextNodeappendChild

Object.keys

document.getElementByIddocument.querySelector"#mountains"

Elementos por nome de tagtalksAbout

byTagname

Node.ELEMENT_NODE

O chapéu do gatoMath.cosMath.sinπMath.PI

Manipulando Eventos

Balão"keydown"event.key

replaceChildtextContent

Rastro do mouse

"mousemove"% elements.length

"mousemove"requestAnimationFrame

AbaschildNodeschildNodeschildrenchildNodes

Projeto: Um Jogo de Plataforma

Pausando o jogofalserunAnimationrunAnimation

runAnimation

trackKeysaddEventListenerremoveEventListenerhandlertrackKeys

`trackKeys`

Um monstro

`update`

`state.actors` `collide` `Coin` `Lava` `"lost"`

Desenhando no Canvas

Formas

`rotate` `flipHorizontally` `translate`

`lineTo` `% 2`

`quadraticCurveTo` `quadraticCurveTo`

O gráfico de pizza

`fillText` `textAlign` `textBaseline`

`currentAngle + 0.5 * sliceAngle`

```
let middleAngle = currentAngle + 0.5 * sliceAngle;
let textX = Math.cos(middleAngle) * 120 + centerX;
let textY = Math.sin(middleAngle) * 120 + centerY;
```

`textBaseline` `"middle"` `textAlign` `"right"` `"left"`

`Math.cos`

Uma bola quicando

`strokeRect` `arc(x, y, raio, 0, 7)`

`Vec`

`clearRect`

Espelhamento pré-computado

`drawImage` `<canvas>`

`"load"`

HTTP e Formulários

Negociação de conteúdo

`fetch`

`Accept`

Uma bancada de JavaScript

`document.querySelector` `document.getElementById`

`"click"` `"mousedown"` `value` `Function`

`Function` `try`

`textContent` `document.createTextNode`

Jogo da Vida de Conway

`<table>` `<br>`

Projeto: Um Editor de Pixel Art

Atalhos de tecladokey

"keydown"tools

preventDefault

Desenho eficiente

updatePicturedrawPicture

widthheightnull

Círculosrectangle

Math.sqrtx ** 2

Linhas corretasπ

Math.abs

```
[start, end] = [end, start];
```

draw

Node.js

Ferramenta de buscaprocess.argv[2]RegExp

readFileSyncnode:fs/promisesasync

statstatSyncisDirectory

readdirreaddirSyncreaddirSync

readdirsepnode:pathjoin

Criação de diretórioDELETEMKCOLmkdir

Um espaço público na web<textarea>GETfetch

<form>"submit"PUT<textarea>

<select><option>GET/"change"

Projeto: Website de Compartilhamento de Habilidades

Persistência em discotalkswriteFileupdated

./talks.jsonreadFile

Reinicialização de campo de comentáriosyncStatesyncState

Index