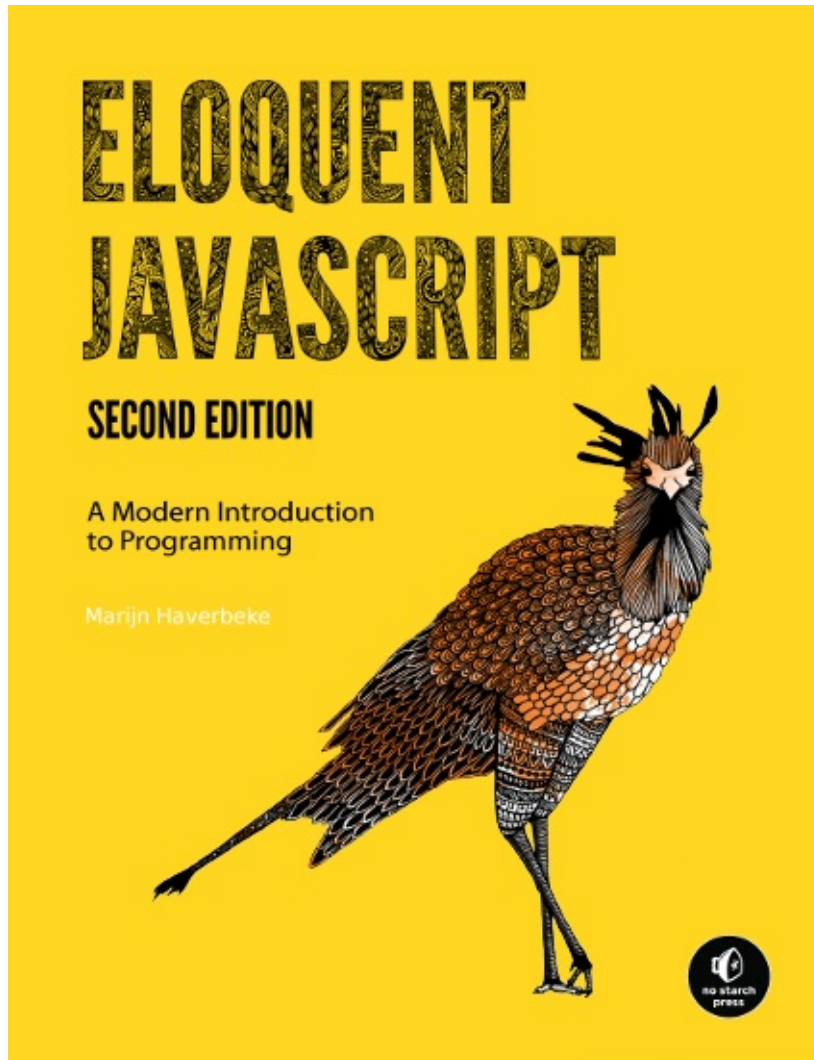

Table of Contents

Introduction	1.1
Valores, Tipos e Operadores	1.2
Estrutura do Programa	1.3
Funções	1.4
Estrutura de Dados: Objeto e Array	1.5
Funções de Ordem Superior	1.6
A Vida Secreta dos Objetos	1.7
Prática: Vida Eletrônica	1.8
Erros e Manipulação de Erros	1.9
Expressões Regulares	1.10
Módulos	1.11
Prática: A Linguagem de Programação	1.12
JavaScript e o Navegador	1.13
O Document Object Model	1.14
Manipulando Eventos	1.15
Projeto: Plataforma de Jogo	1.16
Desenhando no Canvas	1.17
HTTP	1.18
Formulários e Campos de Formulários	1.19
Projeto: Um Programa de Pintura	1.20
Node.js	1.21
Projeto: Website de Compartilhamento de Habilidades	1.22

JavaScript Eloquent - 2ª edição

Uma moderna introdução ao JavaScript, programação e maravilhas digitais.



Conteúdo do Livro

Introdução

1. [Valores, Tipos e Operadores - \(Parte 1: Linguagem\)](#)
2. [Estrutura do Programa](#)
3. [Funções](#)
4. [Estrutura de Dados: Objeto e Array](#)
5. [Funções de Ordem Superior](#)
6. [A Vida Secreta dos Objetos](#)
7. [Prática: Vida Eletrônica](#)
8. [Erros e Manipulação de Erros](#)
9. [Expressões Regulares](#)
10. [Módulos](#)
11. [Prática: A Linguagem de Programação](#)

12. [JavaScript e o Navegador - \(Parte 2: Navegador\)](#)
13. [O Document Object Model](#)
14. [Manipulando Eventos](#)
15. [Projeto: Plataforma de Jogo](#)
16. [Desenhando no Canvas](#)
17. [HTTP](#)
18. [Formulários e Campos de Formulários](#)
19. [Projeto: Um Programa de Pintura](#)
20. [Node.js - \(Parte 3: Node.js\)](#)
21. [Projeto: Website de Compartilhamento de Habilidades](#)

Status Geral do Projeto

As informações sobre o status e log de cada capítulo estão organizadas [nessa issue](#).

Atualmente, estamos melhorando o que já está traduzido, focando na qualidade e precisão da tradução e entendimento do texto como um todo, além de tentar aplicar a gramática mais correta possível. Vários [contribuidores](#) ajudaram em diferentes partes do livro e, por isso, existem diversas oportunidades de melhorias.

Como Contribuir?

Se você tiver interesse em ajudar, criamos um [guia](#) para ajudá-lo e, se tiver qualquer dúvida, basta abrir uma issue.

Informações Importantes

- Autor: **Marijn Haverbeke**
- [Versão original](#) deste livro.

Licenciado sob a licença [Creative Commons attribution-noncommercial](#).

Todo código neste livro também pode ser considerado sob a [licença MIT](#).

Valores, Tipos e Operadores

Abaixo da parte superficial da máquina, o programa se movimenta. Sem esforço, ele se expande e se contrai. Com grande harmonia, os elétrons se espalham e se reagrupam. As formas no monitor são como ondulações na água. A essência permanece invisível por baixo.

— Master Yuan-Ma, The Book of Programming

Dentro do mundo do computador, há somente dados. Você pode ler, modificar e criar novos dados, entretanto, qualquer coisa que não seja um dado simplesmente não existe. Todos os dados são armazenados em longas sequências de bits e são, fundamentalmente, parecidos.

Bits podem ser qualquer tipo de coisa representada por dois valores, normalmente descritos como zeros e uns. Dentro do computador, eles representam formas tais como uma carga elétrica alta ou baixa, um sinal forte ou fraco ou até um ponto na superfície de um CD que possui brilho ou não. Qualquer pedaço de informação pode ser reduzido a uma sequência de zeros e uns e, então, representados por bits.

Como um exemplo, pense sobre a maneira que o número 13 pode ser armazenado em bits. A forma usual de se fazer esta analogia é a forma de escrevermos números decimais, mas ao invés de 10 dígitos, temos apenas 2. E, ao invés de o valor de um dígito aumentar dez vezes sobre o dígito após ele, o valor aumenta por um fator de 2. Estes são os bits que compõem o número treze, com o valor dos dígitos mostrados abaixo deles:

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

Assim, este é o número binário 00001101, ou $8 + 4 + 1$, que equivale a 13.

Valores

Imagine um mar de bits, um oceano deles. Um computador moderno possui mais de trinta bilhões de bits em seu armazenamento volátil de dados. Já em armazenamento de dados não voláteis, sendo eles o disco rígido ou algo equivalente, tende a ter uma ordem de magnitude ainda maior.



Para que seja possível trabalhar com tais quantidades de bits sem ficar perdido, você pode separá-los em partes que representam pedaços de informação. No ambiente JavaScript, essas partes são chamadas de *valores*. Embora todos os valores sejam compostos por bits, cada valor exerce um papel diferente e todo valor possui um tipo que determina o seu papel. Existem seis tipos básicos de valores no JavaScript: números, *Strings*, *Booleanos*, objetos, funções e valores indefinidos.

Para criar um valor, você deve simplesmente invocar o seu nome. Isso é bastante conveniente, pois você não precisa de nenhum material extra para construí-los e muito menos ter que pagar algo por eles. Você só chama por ele e pronto, você o tem. É claro que eles não são criados do nada. Todo valor precisa estar armazenado em algum lugar e, se você quiser usar uma quantidade enorme deles ao mesmo tempo, você pode acabar ficando sem bits. Felizmente, esse é um problema apenas se você precisa deles simultaneamente. A medida que você não utiliza um valor, ele será dissipado, fazendo com que seus bits sejam reciclados, disponibilizando-os para serem usados na construção de outros novos valores.

Esse capítulo introduz os elementos que representam os átomos dos programas JavaScript, que são os simples tipos de valores e os operadores que podem atuar sobre eles.

Números

Valores do tipo *número* são, sem muitas surpresas, valores numéricos. Em um programa JavaScript, eles são escritos assim:

```
13
```

Coloque isso em um programa e isso fará com que padrões de bits referentes ao número 13 sejam criados e passe a existir dentro da memória do computador.

O JavaScript utiliza um número fixo de bits, mais precisamente 64 deles, para armazenar um único valor numérico. Existem apenas algumas maneiras possíveis que você pode combinar esses 64 bits, ou seja, a quantidade de números diferentes que podem ser representados é limitada. Para um valor N de dígitos decimais, a quantidade de números que pode ser representada é 10^N . De forma similar, dado 64 dígitos binários, você pode representar 2^{64} número diferentes, que é aproximadamente 18 quintilhões (o número 18 com 18 zeros após ele). Isso é muito.

A memória do computador costumava ser bem menor e, por isso, as pessoas usavam grupos de 8 ou 16 bits para representar os números. Por isso, era muito fácil extrapolar essa capacidade de armazenamento tão pequena usando números que não cabiam nesse espaço. Hoje em dia, até os computadores pessoais possuem memória suficiente, possibilitando usar grupos de 64 bits, sendo apenas necessário se preocupar em exceder o espaço quando estiver lidando com números extremamente grandes.

Entretanto, nem todos os números inteiros menores do que 18 quintilhões cabem em um número no JavaScript. Os bits também armazenam números negativos, sendo que um desses bits indica o sinal do número. Um grande problema é que números fracionários também precisam ser representados. Para fazer isso, alguns bits são usados para armazenar a posição do ponto decimal. Na realidade, o maior número inteiro que pode ser armazenado está na região de 9 quatrilhões (15 zeros), que ainda assim é extremamente grande.

Números fracionários são escritos usando um ponto.

```
9.81
```

Para números muito grandes ou pequenos, você pode usar a notação científica adicionando um “e” (de “expoente”) seguido do valor do expoente:

```
2.998e8
```

Isso é $2.998 \times 10^8 = 299800000$.

Cálculos usando números inteiros menores que os 9 quatrilhões mencionados, serão com certeza precisos. Infelizmente, cálculos com número fracionários normalmente não são precisos. Da mesma forma que π (pi) não pode ser expresso de forma precisa por uma quantidade finita de dígitos decimais, muitos números perdem sua precisão

quando existem apenas 64 bits disponíveis para armazená-los. Isso é vergonhoso, porém causa problemas apenas em algumas situações específicas. O mais importante é estar ciente dessa limitação e tratar números fracionários como aproximações e não como valores precisos.

Aritmética

A principal coisa para se fazer com números são cálculos aritméticos. As operações como adição ou multiplicação recebem dois valores numéricos e produzem um novo número a partir deles. Elas são representadas dessa forma no JavaScript:

```
100 + 4 * 11
```

Os símbolos `+` e `*` são chamados de *operadores*. O primeiro é referente à adição e o segundo à multiplicação. Colocar um operador entre dois valores irá aplicá-lo a esses valores e produzirá um novo valor.

O significado do exemplo anterior é “adicione 4 e 100 e, em seguida, multiplique esse resultado por 11” ou a multiplicação é realizada antes da adição? Como você deve ter pensado, a multiplicação acontece primeiro. Entretanto, como na matemática, você pode mudar esse comportamento envolvendo a adição com parênteses.

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

Para subtração existe o operador `-` e para a divisão usamos o operador `/`.

Quando os operadores aparecem juntos sem parênteses, a ordem que eles serão aplicados é determinada pela *precedência* deles. O exemplo mostra que a multiplicação ocorre antes da adição. O operador `/` possui a mesma precedência que `*` e, de forma similar, os operadores `+` e `-` possuem a mesma precedência entre si. Quando vários operadores de mesma precedência aparecem próximos uns aos outros, como por exemplo `1 - 2 + 1`, eles são aplicados da esquerda para a direita: `(1 - 2) + 1`.

Essas regras de precedência não são coisas com as quais você deve se preocupar. Quando estiver em dúvida, apenas adicione os parênteses.

Existe mais um operador aritmético que você talvez não reconheça 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` produz `0`. A precedência do operador resto é a mesma da multiplicação e divisão. Você ouvirá com frequência esse operador ser chamado de *modulo* mas, tecnicamente falando, *resto* é o termo mais preciso.

Números Especiais

Existem três valores especiais no JavaScript que são considerados números, mas não se comportam como números normais.

Os dois primeiros são `Infinity` e `-Infinity`, que são usados para representar os infinitos positivo e negativo. O cálculo `Infinity - 1` continua sendo `Infinity`, assim como qualquer outra variação dessa conta. Entretanto, não confie muito em cálculos baseados no valor infinito, pois esse valor não é matematicamente sólido e rapidamente nos levará ao próximo número especial: `NaN`.

`NaN` é a abreviação de “*not a number*” (não é um número), mesmo sabendo que ele é um valor do tipo número. Você receberá esse valor como resultado quando, por exemplo, tentar calcular `0 / 0` (zero dividido por zero), `Infinity - Infinity` ou, então, realizar quaisquer outras operações numéricas que não resultem em um número preciso e significativo.

Strings

O próximo tipo básico de dado é a *String*. *Strings* são usadas para representar texto, e são escritas delimitando o seu conteúdo entre aspas.

```
"Patch my boat with chewing gum"  
'Monkeys wave goodbye'
```

Ambas as aspas simples e duplas podem ser usadas para representar *Strings*, contanto que as aspas abertas sejam iguais no início e no fim.

Quase tudo pode ser colocado entre aspas e o JavaScript criará um valor do tipo *String* com o que quer que seja. Entretanto, alguns caracteres são mais difíceis. Você pode imaginar como deve ser complicado colocar aspas dentro de aspas. Além disso, os caracteres *newlines* (quebra de linhas, usados quando você aperta *Enter*), também não podem ser colocados entre aspas. As *Strings* devem permanecer em uma única linha.

Para que seja possível incluir tais caracteres em uma *String*, a seguinte notação é utilizada: toda vez que um caractere de barra invertida (`\`) for encontrado dentro de um texto entre aspas, ele indicará que o caractere seguinte possui um significado especial. Isso é chamado de *escapar* o caractere. Uma aspa que se encontra logo após uma barra invertida não representará o fim da *String* e, ao invés disso, será considerada como parte do texto dela. Quando um caractere `n` aparecer após uma barra invertida, ele será interpretado como uma quebra de linha e, de forma similar, um `t` significará um caractere de tabulação. Considere a seguinte *String*:

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

O texto na verdade será:

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

Existe, obviamente, situações nas quais você vai querer que a barra invertida em uma *String* seja apenas uma barra invertida e não um código especial. Nesse caso, se duas barras invertidas estiverem seguidas uma da outra, elas se anulam e apenas uma será deixada no valor da *String* resultante. Essa é a forma na qual a *String* " A newline character is written like `"\n"`. " pode ser representada:

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

Strings não podem ser divididas, multiplicadas nem subtraídas, entretanto, o operador `+` pode ser usado nelas. Ele não efetua a adição, mas *concatena*, ou seja, junta duas *Strings* em uma única *String*. O próximo exemplo produzirá a *String* "concatenate" :

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

Existem outras maneiras de manipular as *Strings*, as quais serão discutidas quando chegarmos aos métodos no [Capítulo 4](#).

Operadores Unários

Nem todos os operadores são símbolos, sendo que alguns são escritos como palavras. Um exemplo é o operador `typeof` , que produz um valor do tipo *String* contendo o nome do tipo do valor que você está verificando.

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

Nós vamos usar `console.log` nos códigos de exemplo para indicar que desejamos ver o resultado da avaliação de algo. Quando você executar tais códigos, o valor produzido será mostrado na tela, entretanto, a forma como ele será apresentado vai depender do ambiente JavaScript que você usar para rodar os códigos.

Todos os operadores que vimos operavam em dois valores, mas `typeof` espera um único valor. Operadores que usam dois valores são chamados de operadores *binários*, enquanto que aqueles que recebem apenas um, são chamados de operadores *unários*. O operador `-` pode ser usado tanto como binário quanto como unário.

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

Valores *Booleanos*

Você frequentemente precisará de um valor para distinguir entre duas possibilidades, como por exemplo “sim” e “não”, ou “ligado” e “desligado”. Para isso, o JavaScript possui o tipo *Booleano*, que tem apenas dois valores: verdadeiro e falso (que são escritos como `true` e `false` respectivamente).

Comparações

Essa é uma maneira de produzir valores *Booleanos*:

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

Os sinais `>` e `<` são tradicionalmente símbolos para representar “é maior que” e “é menor que” respectivamente. Eles são operadores binários, e o resultado da aplicação deles é um valor *Booleano* que indica se a operação é verdadeira nesse caso.

Strings podem ser comparadas da mesma forma.

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

A forma na qual as *Strings* são ordenadas é mais ou menos alfabética. Letras maiúsculas serão sempre “menores” que as minúsculas, portanto, `"z" < "a"` é verdadeiro. Além disso, caracteres não alfabéticos (!, -, e assim por diante) também são incluídos nessa ordenação. A comparação de fato, é baseada no padrão *Unicode*, que atribui um número para todos os caracteres que você possa precisar, incluindo caracteres do Grego, Árabe, Japonês, Tâmil e por aí vai. Possuir tais números é útil para armazenar as *Strings* dentro do computador, pois faz com que seja possível representá-las como uma sequência de números. Quando comparamos *Strings*, o JavaScript inicia da esquerda para a direita, comparando os códigos numéricos dos caracteres um por um.

Outros operadores parecidos são `>=` (maior que ou igual a), `<=` (menor que ou igual a), `==` (igual a) e `!=` (não igual a).

```
console.log("Itchy" != "Scratchy")
// → true
```


Existe apenas um valor no JavaScript que não é igual a ele mesmo, que é o valor `NaN`. Ele significa “*not a number*”, que em português seria traduzido como “não é um número”.

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

`NaN` é supostamente usado para indicar o resultado de alguma operação que não tenha sentido e, por isso, ele não será igual ao resultado de quaisquer *outras* operações sem sentido.

Operadores Lógicos

Existem também operadores que podem ser aplicados aos valores *Booleanos*. O JavaScript dá suporte a três operadores lógicos: *and*, *or* e *not*, que podem ser traduzidos para o português como *e*, *ou* e *não*. Eles podem ser usados para “pensar” de forma lógica sobre *Booleanos*.

O operador `&&` representa o valor lógico *and* ou, em português, *e*. Ele é um operador binário, e seu resultado é apenas verdadeiro se ambos os valores dados à ele forem verdadeiros.

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

O operador `||` indica o valor lógico *or* ou, em português, *ou*. Ele produz um valor verdadeiro se qualquer um dos valores dados à ele for verdadeiro.

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

Not, em português *não*, é escrito usando um ponto de exclamação (`!`). Ele é um operador unário que inverte o valor que é dado à ele. Por exemplo, `!true` produz `false` e `!false` produz `true`.

Quando misturamos esses operadores *Booleanos* com operadores aritméticos e outros tipos de operadores, nem sempre é óbvio quando devemos usar ou não os parênteses. Na prática, você normalmente não terá problemas sabendo que, dos operadores que vimos até agora, `||` possui a menor precedência, depois vem o operador `&&`, em seguida vêm os operadores de comparação (`>`, `==`, etc) e, por último, quaisquer outros operadores. Essa ordem foi escolhida de tal forma que, em expressões típicas como o exemplo a seguir, poucos parênteses são realmente necessários:

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

O último operador lógico que iremos discutir não é unário nem binário, mas *ternário*, operando em três valores. Ele é escrito usando um ponto de interrogação e dois pontos, como mostrado abaixo:

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

Esse operador é chamado de operador *condicional* (algumas vezes é chamado apenas de operador *ternário*, já que é o único operador desse tipo na linguagem). O valor presente à esquerda do ponto de interrogação “seleciona” qual dos outros dois valores será retornado. Quando ele for verdadeiro, o valor do meio é escolhido e, quando for falso, o valor à direita é retornado.

Valores Indefinidos

Existem dois valores especiais, `null` e `undefined`, que são usados para indicar a ausência de um valor com significado. Eles são valores por si sós, mas não carregam nenhum tipo de informação.

Muitas operações na linguagem que não produzem um valor com significado (você verá alguns mais para frente) retornarão `undefined` simplesmente porque eles precisam retornar *algum* valor.

A diferença de significado entre `undefined` e `null` é um acidente que foi criado no design do JavaScript, e não faz muita diferença na maioria das vezes. Nos casos em que você deve realmente se preocupar com esses valores, recomendo tratá-los como valores idênticos (vamos falar mais sobre isso em breve).

Conversão Automática de Tipo

Na introdução, mencionei que o JavaScript tentar fazer o seu melhor para aceitar quase todos os programas que você fornecer, inclusive aqueles que fazem coisas bem estranhas. Isso pode ser demonstrado com as 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 a um tipo de valor “errado”, o JavaScript converterá, de forma silenciosa, esse valor para o tipo que ele desejar, usando uma série de regras que muitas vezes não é o que você deseja ou espera. Esse comportamento é chamado de *coerção de tipo* (ou *conversão de tipo*). Portanto, na primeira expressão, `null` se torna `0` e, na segunda, a `String` `"5"` se torna o número `5`. Já na terceira expressão, o operador `+` tenta efetuar uma concatenação de `String` antes de tentar executar a adição numérica e, por isso, o número `1` é convertido para a `String` `"1"`.

Quando algo que não pode ser mapeado como um número de forma óbvia (tais como `"five"` ou `undefined`) é convertido para um número, o valor `NaN` é produzido. Quaisquer outras operações aritméticas realizadas com `NaN` continuam produzindo `NaN`, portanto, quando você perceber que está recebendo esse valor em algum lugar inesperado, procure por conversões de tipo acidentais.

Quando comparamos valores do mesmo tipo usando o operador `==`, o resultado é fácil de se prever: você receberá verdadeiro quando ambos os valores forem o mesmo, exceto no caso de `NaN`. Por outro lado, quando os tipos forem diferentes, o JavaScript usa um conjunto de regras complicadas e confusas para determinar o que fazer, sendo que, na maioria dos casos, ele tenta apenas converter um dos valores para o mesmo tipo do outro valor. Entretanto, quando `null` ou `undefined` aparece em algum dos lados do operador, será produzido verdadeiro apenas se ambos os lados forem `null` ou `undefined`.

```
console.log(null == undefined);
```

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

O último exemplo é um comportamento que normalmente é bastante útil. Quando quiser testar se um valor possui um valor real ao invés de `null` ou `undefined`, você pode simplesmente compará-lo a `null` com o operador `==` (ou `!=`).

Mas e se você quiser testar se algo se refere ao valor preciso `false`? As regras de conversão de *Strings* e números para valores *Booleanos* afirmam que `0`, `NaN` e *Strings* vazias contam como `false`, enquanto todos os outros valores contam como `true`. Por causa disso, expressões como `0 == false` e `"" == false` retornam `true`. Para casos assim, onde você **não** quer qualquer conversão automática de tipos acontecendo, existem dois tipos extras de operadores: `===` e `!==`. O primeiro testa se o valor é precisamente igual ao outro, e o segundo testa se ele não é precisamente igual. Então `"" === false` é falso como esperado.

Usar os operadores de comparação de três caracteres defensivamente, para prevenir inesperadas conversões de tipo que o farão tropeçar, é algo que eu recomendo. Mas quando você tem certeza de que os tipos de ambos os lados serão iguais, ou que eles vão ser ambos `null / undefined`, não há problemas em usar os operadores curtos.

O Curto-Circuito de && e ||

Os operadores lógicos `&&` e `||` tem uma maneira peculiar de lidar com valores de tipos diferentes. Eles vão converter o valor à sua esquerda para o tipo *Booleano* a fim de decidir o que fazer, mas então, dependendo do operador e do resultado da conversão, eles ou retornam o valor à esquerda *original*, ou o valor à direita.

O operador `||` vai retornar o valor à sua esquerda quando ele puder ser convertido em `true`, caso contrário, retorna o valor à sua direita. Ele faz a coisa certa para valores *Booleanos*, e vai fazer algo análogo para valores de outros tipos. Isso é muito útil, pois permite que o operador seja usado para retornar um determinado valor predefinido.

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

O operador `&&` trabalha similarmente, mas ao contrário. Quando o valor à sua esquerda é algo que se torne `false`, ele retorna o valor e caso contrário, ele retorna o valor à sua direita.

Outra importante propriedade destes 2 operadores é que a expressão à sua direita é avaliada somente quando necessário. No caso de `true || x`, não importa o que `x` é - pode ser uma expressão que faça algo *terrível* - o resultado vai ser verdadeiro, e `x` nunca é avaliado. O mesmo acontece para `false && x`, que é falso, e vai ignorar `x`.

Resumo

Nós vimos 4 tipos de valores do JavaScript neste capítulo. Números, *Strings*, *Booleanos* e valores indefinidos.

Alguns valores são criados digitando seu nome (`true`, `null`) ou valores (`13`, `"abc"`). Você pode combinar e transformar valores com operadores. Nós vimos operadores binários para operações aritméticas (`+`, `-`, `*`, `/`, e `%`), um para concatenação de *String* (`+`), comparação (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) e lógica (`&&`,

`||`), assim como vários operadores unários (`-` para negatizar um número, `!` para negar uma lógica, `typeof` para encontrar o tipo do valor) e o operador ternário (`?:`) para retornar um de dois valores, baseando-se em um terceiro valor.

Isto lhe dá informação suficiente para usar o JavaScript como uma calculadora de bolso, mas não muito mais. O próximo capítulo vai começar a amarrar essas operações básicas conjuntamente dentro de programas básicos.

Estrutura do Programa

O meu coração vermelho brilha nitidamente sob minha pele e ele têm que administrar 10cc de JavaScript para fazer com que eu volte (Eu respondi bem a toxinas no sangue). Cara, esse negócio vai chutar os pêssegos de direita para fora!

- `_why`, *Why's (Poignant) Guide to Ruby*

Este é o ponto onde nós começamos a fazer coisas que podem realmente ser chamadas de programação. Nós vamos expandir nosso domínio da linguagem JavaScript para além dos substantivos e fragmentos de sentenças que nós vimos anteriormente, para o ponto onde poderemos realmente expressar algo mais significativo.

Expressões e Afirmações

No [Capítulo 1](#) nós criamos alguns valores e então aplicamos operadores para obter novos valores. Criar valores desta forma é uma parte essencial de todo programa JavaScript, mas isso é somente uma parte. Um fragmento de código que produz um valor é chamado de *expressão*. Todo valor que é escrito literalmente (como `22` ou `"psychoanalysis"`) é uma expressão. Uma expressão entre parênteses é também uma expressão, e também um operador binário aplicado a duas expressões, ou um unário aplicado a uma.

Isso mostra parte da beleza da interface baseada na linguagem. Expressões podem ser encadeadas de forma semelhante às subfrases usadas na linguagem humana - uma subfrase pode conter sua própria subfrase, e assim por diante. Isto nos permite combinar expressões para expressar computações complexas arbitrariamente.

Se uma expressão corresponde a um fragmento de sentença, uma *afirmação*, no JavaScript, corresponde a uma frase completa em linguagem humana. Um programa é simplesmente uma lista de afirmações.

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

```
1;  
!false;
```

É um programa inútil, entretanto. Uma expressão pode ser apenas para produzir um valor, que pode então ser usado para fechar a expressão. Uma declaração vale por si só, e só equivale a alguma coisa se ela afeta em algo. Ela pode mostrar algo na tela - que conta como mudar algo - ou pode mudar internamente o estado da máquina de uma forma que vai afetar outras declarações que irão vir. Estas mudanças são chamadas *efeitos colaterais*. As afirmações nos exemplos anteriores somente produzem o valor `1` e `true` e então imediatamente os jogam fora novamente. Não deixam nenhuma impressão no mundo. Quando executamos o programa, nada acontece.

Ponto e vírgula

Em alguns casos, o JavaScript permite que você omita o ponto e vírgula no fim de uma declaração. Em outros casos ele deve estar lá ou coisas estranhas irão acontecer. As regras para quando ele pode ser seguramente omitido são um pouco complexas e propensas a erro. Neste livro todas as declarações que precisam de ponto e vírgula vão sempre terminar com um. Eu recomendo a você fazer o mesmo em seus programas, ao menos até você aprender mais sobre as sutilezas envolvidas em retirar o ponto e vírgula.

Variáveis

Como um programa mantém um estado interno? Como ele se lembra das coisas? Nós vimos como produzir novos valores com valores antigos, mas isso não altera os valores antigos, e o valor novo deve ser imediatamente usado ou vai ser dissipado. Para pegar e guardar valores, o JavaScript fornece uma coisa chamada *variável*.

```
var caught = 5 * 5;
```

E isso nos dá um segundo tipo de declaração. A palavra especial (palavra-chave) `var` indica que esta sentença vai definir uma variável. Ela é seguida pelo nome da variável e, se nós quisermos dá-la imediatamente um valor, por um operador `=` e uma expressão.

A declaração anterior criou uma variável chamada `caught` e a usou para armazenar o valor que foi produzido pela multiplicação 5 por 5.

Depois de uma variável ter sido definida, seu nome pode ser usado como uma expressão. O valor da expressão é o valor atual mantido pela variável. Aqui temos um exemplo:

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

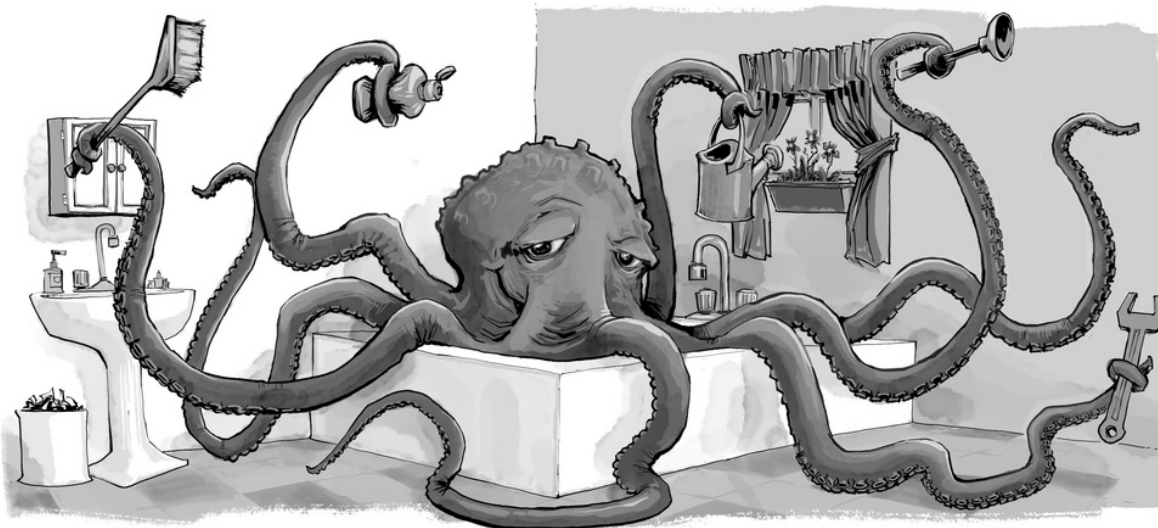
Nomes de variáveis podem ser quase qualquer palavra, menos as reservadas para palavras-chave (como `var`). Não pode haver espaços incluídos. Dígitos podem também ser parte dos nomes de variáveis - `catch22` é um nome válido, por exemplo - mas um nome não pode iniciar com um dígito. O nome de uma variável não pode incluir pontuação, exceto pelos caracteres `$` e `_`.

Quando uma variável aponta para um valor, isso não significa que estará ligada ao valor para sempre. O operador `=` pode ser usado a qualquer hora em variáveis existentes para desconectá-las de seu valor atual e então apontá-las para um novo:

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

Você deve imaginar variáveis como tentáculos, ao invés de caixas. Elas não *contêm* valores; elas os *agarram* - duas variáveis podem referenciar o mesmo valor. Somente os valores que o programa mantém tem o poder de ser acessado por ele. Quando você precisa se lembrar de algo, você aumenta o tentáculo para segurar ou recoloca um de seus tentáculos existentes para fazer isso.

Quando você define uma variável sem fornecer um valor a ela, o tentáculo fica conceitualmente no ar - ele não tem nada para segurar. Quando você pergunta por um valor em um lugar vazio, você recebe o valor `undefined`.



Um exemplo. Para lembrar da quantidade de dólares que Luigi ainda lhe deve, você cria uma variável. E então quando ele lhe paga 35 dólares, você dá a essa variável um novo valor.

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

Palavras-chave e Palavras Reservadas

Palavras que tem um significado especial, como `var`, não podem ser usadas como nomes de variáveis. Estas são chamadas *keywords* (palavras-chave). Existe também algumas palavras que são reservadas para uso em futuras versões do JavaScript. Estas também não são oficialmente autorizadas a serem utilizadas como nomes de variáveis, embora alguns ambientes JavaScript as permitam. A lista completa de palavras-chave e palavras reservadas é um pouco longa:

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

Não se preocupe em memorizá-las, mas lembre-se que este pode ser o problema quando algo não funcionar como o esperado.

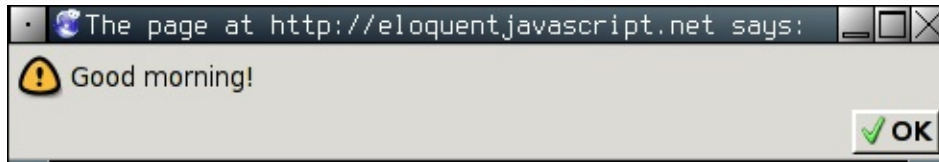
O Ambiente

A coleção de variáveis e seus valores que existem em um determinado tempo é chamado de `environment` (ambiente). Quando um programa inicia, o ambiente não está vazio. Ele irá conter no mínimo o número de variáveis que fazem parte do padrão da linguagem. E na maioria das vezes haverá um conjunto adicional de variáveis que fornecem maneiras de interagir com o sistema envolvido. Por exemplo, em um navegador, existem variáveis que apontam para funcionalidades que permitem a você inspecionar e influenciar no atual carregamento do website, e ler a entrada do mouse e teclado da pessoa que está usando o navegador.

Funções

Muitos dos valores fornecidos no ambiente padrão são do tipo `function` (função). Uma função é um pedaço de programa envolvido por um valor. Este valor pode ser aplicado a fim de executar o programa envolvido. Por exemplo, no ambiente do navegador, a variável `alert` detém uma função que mostra uma pequena caixa de diálogo com uma mensagem. É usada da seguinte forma:

```
alert("Good morning!");
```



Executar uma função é denominado *invocar*, *chamar* ou *aplicar* uma função. Você pode chamar uma função colocando os parênteses depois da expressão que produz um valor de função. Normalmente você irá usar o nome da variável que contém uma função diretamente. Os valores entre os parênteses são passados ao programa dentro da função. No exemplo, a função `alert` usou a `string` que foi passada como o texto a ser mostrado na caixa de diálogo. Os valores passados para funções são chamados de `arguments` (argumentos). A função `alert` precisa somente de um deles, mas outras funções podem precisar de diferentes quantidades ou tipos de argumentos.

A Função `console.log`

A função `alert` pode ser útil como saída do dispositivo quando experimentada, mas clicar sempre em todas estas pequenas janelas vai lhe irritar. Nos exemplos passados, nós usamos `console.log` para saída de valores. A maioria dos sistemas JavaScript (incluindo todos os navegadores modernos e o Node.js), fornecem uma função `console.log` que escreve seus argumentos como texto na saída do dispositivo. Nos navegadores, a saída fica no console JavaScript. Esta parte da interface do `browser` fica oculta por padrão, mas muitos browsers abrem quando você pressiona `F12`, ou no Mac, quando você pressiona `Command + option + I`. Se isso não funcionar, busque no menu algum item pelo nome de *web console* ou *developer tools*.

Quando rodarmos os exemplos ou seu próprio código nas páginas deste livro, o `console.log` vai mostrar embaixo o exemplo, ao invés de ser no console JavaScript.

```
var x = 30;
console.log("o valor de x é ", x);
// o valor de x é 30
```

Embora eu tenha afirmado que nomes de variáveis não podem conter pontos, `console.log` claramente contém um ponto. Eu não tinha mentido para você. Esta não é uma simples variável, mas na verdade uma expressão que retorna o campo `log` do valor contido na variável `console`. Nós vamos entender o que isso significa no capítulo 4.

Retornando Valores

Mostrar uma caixa de diálogo ou escrever texto na tela é um efeito colateral. Muitas funções são úteis por causa dos efeitos que elas produzem. É também possível para uma função produzir um valor, no caso de não ser necessário um efeito colateral. Por exemplo, temos a função `Math.max`, que pega dois números e retorna o maior entre eles:

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


Quando uma função produz um valor, é dito que ela *retorna* (`return`) ele. Em JavaScript, tudo que produz um valor é uma expressão, o que significa que chamadas de função podem ser usadas dentro de expressões maiores. No exemplo abaixo, uma chamada para a função `Math.min`, que é o oposto de `Math.max`, é usada como uma das entradas para o operador de soma:

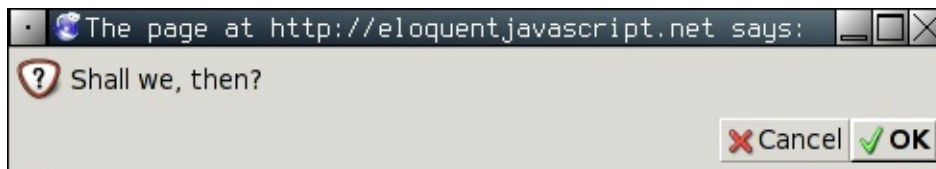
```
console.log(Math.min(2, 4) + 100);
```

O próximo capítulo explica como nós podemos escrever nossas próprias funções.

prompt e confirm

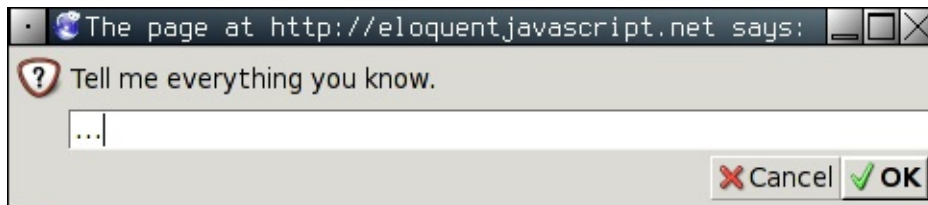
O ambiente fornecido pelos navegadores contém algumas outras funções para mostrar janelas. Você pode perguntar a um usuário uma questão Ok/Cancel usando `confirm`. Isto retorna um valor booleano: `true` se o usuário clica em OK e `false` se o usuário clica em *Cancel*.

```
confirm("Shall we, then?");
```



`prompt` pode ser usado para criar uma questão "aberta". O primeiro argumento é a questão; o segundo é o texto que o usuário inicia. Uma linha do texto pode ser escrita dentro da janela de diálogo, e a função vai retornar isso como uma string.

```
prompt("Tell me everything you know.", "...");
```



Estas duas funções não são muito usadas na programação moderna para web, principalmente porque você não tem controle sobre o modo que a janela vai aparecer, mas elas são úteis para experimentos.

Fluxo de Controle

Quando seu programa contém mais que uma declaração, as declarações são executadas, previsivelmente, de cima para baixo. Como um exemplo básico, este programa tem duas declarações. A primeira pergunta ao usuário por um número, e a segunda, que é executada posteriormente, mostra o quadrado deste número:

```
var theNumber = Number(prompt("Pick a number", ""));  
alert("Your number is the square root of " + theNumber * theNumber);
```

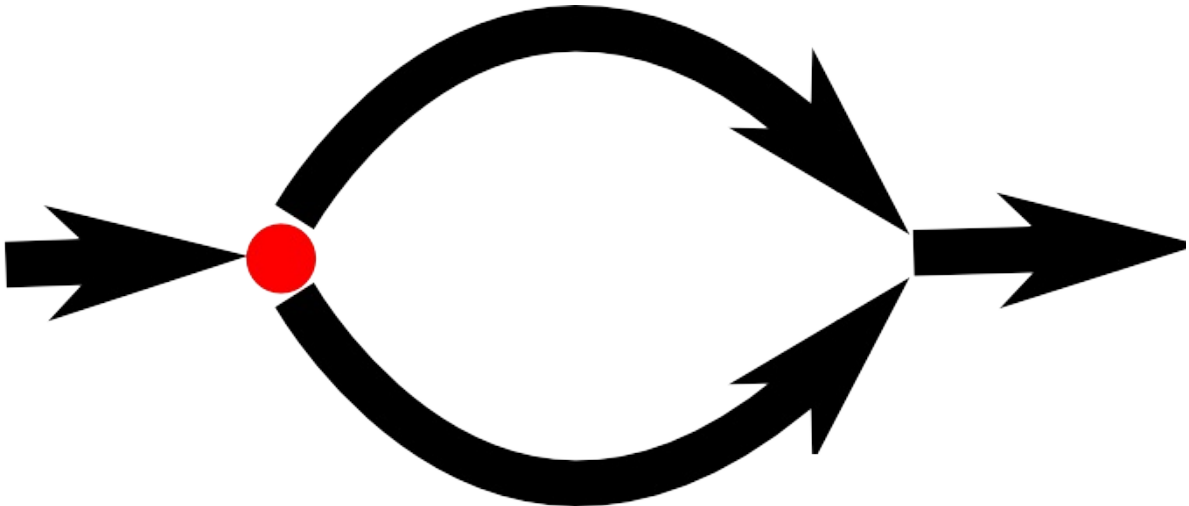
A função `Number` converte o valor para um número. Nós precisamos dessa conversão pois o resultado de `prompt` é um valor do tipo `string`, e nós queremos um número. Existem funções similares chamadas `String` e `Boolean` que convertem valores para estes tipos.

Aqui podemos ver uma representação bem trivial do fluxo de controle em linha reta:



Execução Condicional

Executar declarações em ordem linear não é a única opção que temos. Uma alternativa é a *execução condicional*, onde escolhemos entre duas rotas diferentes baseado em um valor booleano, como ilustra a seguir:



A execução condicional é escrita, em JavaScript, com a palavra-chave `if`. No caso mais simples, nós queremos que algum código seja executado se, e somente se, uma certa condição existir. No programa anterior, por exemplo, podemos mostrar o quadrado do dado fornecido como entrada apenas se ele for realmente um número.

```
var theNumber = Number(prompt("Pick a number", ""));  
if (!isNaN(theNumber))  
    alert("Your number is the square root of " +  
          theNumber * theNumber);
```

Com essa modificação, se você fornecer "queijo" como argumento de entrada, nenhuma saída será retornada.

A palavra-chave `if` executa ou não uma declaração baseada no resultado de uma expressão Booleana. Tal expressão é escrita entre parênteses logo após a palavra-chave e seguida por uma declaração a ser executada.

A função `isNaN` é uma função padrão do JavaScript que retorna `true` apenas se o argumento fornecido for `NaN`. A função `Number` retorna `NaN` quando você fornece a ela uma string que não representa um número válido. Por isso, a condição se traduz a "a não ser que `theNumber` não seja um número, faça isso".

Você frequentemente não terá código que executa apenas quando uma condição for verdadeira, mas também código que lida com o outro caso. Esse caminho alternativo é representado pela segunda seta no diagrama. A palavra-chave `else` pode ser usada, juntamente com `if`, para criar dois caminhos distintos de execução.

```
var theNumber = Number(prompt("Pick a number", ""));  
if (!isNaN(theNumber))  
    alert("Your number is the square root of " +  
          theNumber * theNumber);  
else  
    alert("Hey. why didn't you give me a number?");
```

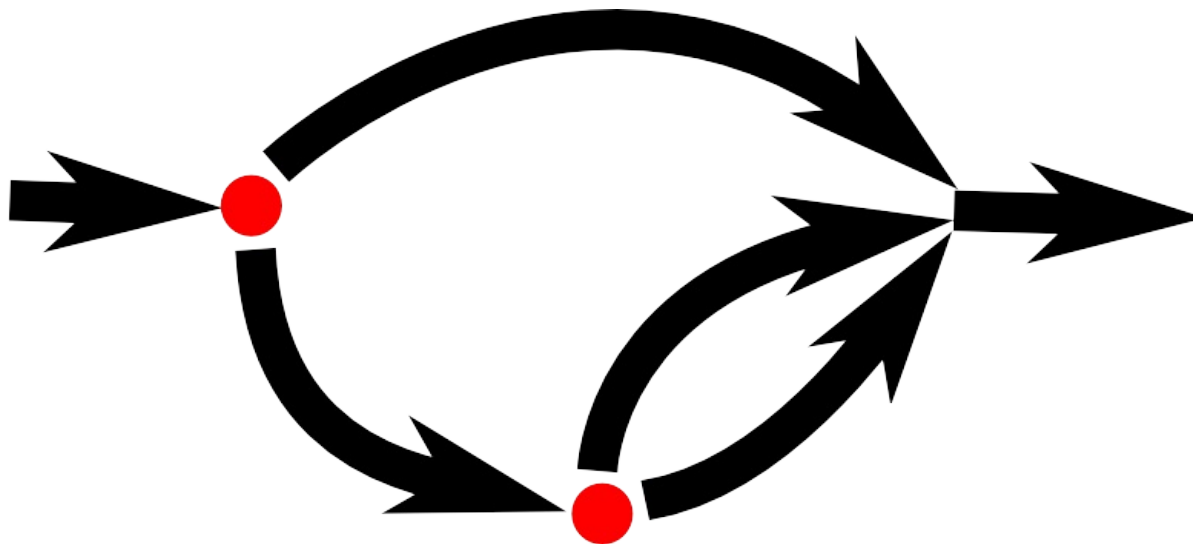
Se tivermos mais que dois caminhos a escolher, múltiplos pares de `if / else` podem ser "encadeados". Aqui temos um exemplo:

```
var num = Number(prompt("Pick a number", "0"));

if (num < 10)
  alert("Small");
else if (num < 100)
  alert("Medium");
else
  alert("Large");
```

O programa irá primeiramente verificar se `num` é menor que 10. Se for, ele escolhe esse caminho, mostra "Small" e termina sua execução. Se não for, ele escolhe o caminho `else`, que contém o segundo `if`. Se a segunda condição (`< 100`) for verdadeira, o número está entre 10 e 100, e "Medium" será mostrado. Caso contrário, o segundo e último `else` será escolhido.

O esquema de setas para este programa parece com algo assim:

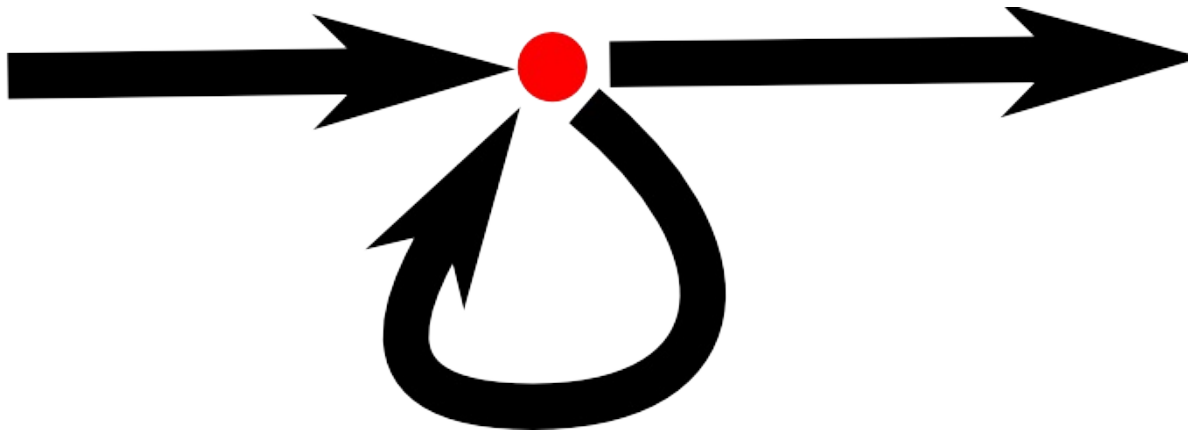


Loops While e Do

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

```
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 com que algo seja *menos* trabalhoso, e não o contrário. Se precisarmos de todos os números pares menores do que 1.000, essa abordagem seria inviável. O que precisamos é de uma maneira de repetir código. Essa forma de fluxo de controle é chamada de *laço de repetição* (loop).



O fluxo de controle do loop nos permite voltar a um mesmo ponto no programa onde estávamos anteriormente e repeti-lo no estado atual do programa. Se combinarmos isso a uma variável contadora, conseguimos fazer algo assim:

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

Uma declaração que inicia com a palavra-chave `while` cria um loop. A palavra `while` é acompanhada por uma expressão entre parênteses e seguida por uma declaração, similar ao `if`. O loop continua executando a declaração enquanto a expressão produzir um valor que, após convertido para o tipo Booleano, seja `true`.

Nesse loop, queremos imprimir o número atual e somar dois em nossa variável. Sempre que precisarmos executar múltiplas declarações dentro de um loop, nós as envolvemos com chaves (`{` e `}`). As chaves, para declarações, são similares aos parênteses para as expressões, agrupando e fazendo com que sejam tratadas como uma única declaração. Uma sequência de declarações envolvidas por chaves é chamada de *bloco*.

Muitos programadores JavaScript envolvem cada `if` e loop com chaves. Eles fazem isso tanto para manter a consistência quanto para evitar que seja necessário adicionar ou remover chaves quando houver alterações posteriores no número de declarações. Nesse livro, para sermos mais breves, iremos escrever sem chaves a maioria das declarações compostas por uma única linha. Fique a vontade para escolher o estilo que preferir.

A variável `number` demonstra uma maneira na qual variáveis podem verificar o progresso de um programa. Toda vez que o loop se repete, `number` é incrementado por `2`. No início de cada repetição, ele é comparado com o número `12` para decidir se o programa terminou de executar todo o trabalho esperado.

Como um exemplo de algo que seja útil, podemos escrever um programa que calcula e mostra o valor de 2^{10} (2 elevado à décima potência). Nós usamos duas variáveis: uma para armazenar o resultado e outra para contar quantas vezes multiplicamos esse resultado por 2. O loop testa se a segunda variável já atingiu o valor 10 e então atualiza ambas as variáveis.

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

O contador pode também iniciar com `1` e checar o valor com `<= 10`, mas por razões que iremos ver no [Capítulo 4](#), é uma boa ideia se acostumar a usar a contagem iniciando com zero.

O loop `do` é uma estrutura de controle similar ao `while`. A única diferença entre eles é que o `do` sempre executa suas declarações ao menos uma vez e inicia o teste para verificar se deve parar ou não apenas após a primeira execução. Para demonstrar isso, o teste aparece após o corpo do loop:

```
do {  
  var name = prompt("Who are you?");  
} while (!name);  
console.log(name);
```

Esse programa irá forçar você a informar um nome. Ele continuará pedindo até que seja fornecido um valor que não seja uma string vazia. Aplicar o operador `!` faz com que o valor seja convertido para o tipo Booleano antes de negá-lo, e todas as strings exceto `""` convertem para `true`.

Indentando Código

Você deve ter reparado nos espaços que coloco em algumas declarações. No JavaScript, eles não são necessários e o computador irá aceitar o programa sem eles. De fato, até as quebras de linhas são opcionais. Se você quiser, pode escrever um programa inteiro em uma única linha. O papel da indentação dentro dos blocos é fazer com que a estrutura do código se destaque. Em códigos complexos, onde temos blocos dentro de blocos, pode se tornar extremamente difícil distinguir onde um bloco começa e o outro termina. Com a indentação adequada, o formato visual do programa corresponde ao formato dos blocos contidos nele. Gosto de usar dois espaços para cada bloco, mas essa preferência pode variar — algumas pessoas usam quatro espaços e outras usam caracteres "tab".

Loops For

Vários loops seguem o padrão visto nos exemplos anteriores do `while`. Primeiramente uma variável "contadora" é criada para monitorar o progresso do loop. Em seguida, temos o loop `while` que contém uma expressão de teste que normalmente checa se o contador alcançou algum limite. O contador é atualizado no final do corpo do loop, permitindo acompanhar o progresso.

Por esse padrão ser muito comum, o JavaScript e linguagens similares fornecem uma forma um pouco mais curta e compreensiva chamada de loop `for`.

```
for (var number = 0; number <= 12; number = number + 2)  
  console.log(number);  
// → 0  
// → 2  
// ... etcetera
```

Esse programa é equivalente ao exemplo anterior que imprime números pares. A única diferença é que todas as declarações relacionadas ao "estado" do loop estão agora agrupadas.

Os parênteses após a palavra-chave `for` devem conter dois pontos e vírgulas. A parte anterior ao primeiro ponto e vírgula *inicializa* o loop, normalmente definindo uma variável. A segunda parte é a expressão que *verifica* se o loop deve continuar ou não. A parte final *atualiza* o estado do loop após cada iteração. Na maioria dos casos, essa construção é menor e mais clara que a do `while`.

Aqui está o código que calcula 2^{10} usando `for` ao invés de `while`:

```
var result = 1;
```

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

Repare que mesmo não abrindo o bloco com `{`, a declaração no loop continua indentada com dois espaços para deixar claro que ela "pertence" à linha anterior a ela.

Quebrando a execução de um Loop

Ter uma condição que produza um resultado `false` não é a única maneira que um loop pode parar. Existe uma declaração especial chamada `break` que tem o efeito de parar a execução e sair do loop em questão.

Esse programa ilustra o uso da declaração `break`. Ele encontra o primeiro número que é, ao mesmo tempo, maior ou igual a 20 e divisível por 7.

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

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

A construção do `for` nesse exemplo não contém a parte que checa pelo fim do loop. Isso significa que o loop não vai parar de executar até que a declaração `break` contida nele seja executada.

Se você não incluir a declaração `break` ou acidentalmente escrever uma condição que sempre produza um resultado `true`, seu programa ficará preso em um *loop infinito*. Um programa preso em um loop infinito nunca vai terminar sua execução, o que normalmente é uma coisa ruim.

Se você criar um loop infinito em algum dos exemplos destas páginas, você normalmente será perguntado se deseja interromper a execução do script após alguns segundos. Se isso não funcionar, você deverá fechar a aba que está trabalhando, ou em alguns casos, fechar o navegador para recuperá-lo.

A palavra-chave `continue` é similar ao `break`, de modo que também influencia o progresso de um loop. Quando `continue` é encontrado no corpo de um loop, o controle de execução pula para fora do corpo e continua executando a próxima iteração do loop.

Atualizando variáveis sucintamente

Um programa, especialmente quando em loop, muitas vezes precisa de atualizar uma variável para armazenar um valor baseado no valor anterior dessa variável.

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

O JavaScript fornece um atalho para isso:

```
counter += 1;
```

Atalhos similares funcionam para outros operadores, como `result *= 2` para dobrar o `result` ou `counter -= 1` para diminuir um.

Isto nos permite encurtar nosso exemplo de contagem um pouco mais:

```
for (var number = 0; number <= 12; number += 2)
    console.log(number);
```

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

Resolvendo um valor com `switch`

É comum que o código fique assim:

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

Há um construtor chamado `switch` que se destina a resolver o envio de valores de uma forma mais direta. Infelizmente, a sintaxe JavaScript usada para isso (que foi herdada na mesma linha de linguagens de programação, C e Java) é um pouco estranha - frequentemente uma cadeia de declarações `if` continua parecendo 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;
}
```

Dentro do bloco aberto pelo `switch`, você pode colocar qualquer número de rótulo no `case`. O programa vai pular para o rótulo correspondente ao valor que `switch` fornece, ou para `default` se nenhum valor for encontrado. Então ele começa a executar as declarações, e continua a passar pelos rótulos, até encontrar uma declaração `break`. Em alguns casos, como no exemplo `case "sunny"`, pode ser usado para compartilhar algum código entre os `cases` (ele recomenda "ir lá fora" para ambos os tempos `sunny` e `cloudy`). Mas tenha cuidado: é fácil esquecer de um `break`, o que fará com que o programa execute código que você não gostaria de executar.

Capitalização

Nomes de variáveis não podem conter espaços, no entanto é muito útil usar múltiplas palavras para descrever claramente o quê a variável representa. Estas são praticamente suas escolhas para escrever nomes de variáveis com várias palavras:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

O primeiro estilo é difícil de ler. Pessoalmente, eu gosto de usar sublinhados, embora esse estilo seja um pouco doloroso de escrever. O padrão das funções em JavaScript, e o da maioria dos programadores JavaScript, é seguir o último estilo - eles capitalizam toda palavra exceto a primeira. Não é difícil se acostumar com coisas pequenas assim, e o código com estilos de nomenclaturas mistas pode se tornar desagradável para leitura, então vamos seguir esta convenção.

Em alguns casos, como a função `Number`, a primeira letra da variável é capitalizada também. Isso é feito para marcar a função como um construtor. O que é um construtor será esclarecido no [capítulo 6](#). Por enquanto, o importante é não ser incomodado por esta aparente falta de consistência.

Comentários

Frequentemente, o código puro não transmite todas as informações necessárias que você gostaria que tivessem para leitores humanos, ou ele se transmite de uma forma tão enigmática que as pessoas realmente não conseguem entendê-lo. Em outras ocasiões, você está apenas se sentindo poético ou quer anotar alguns pensamentos como parte de seu programa. Os comentários são para isto.

O comentário é um pedaço de texto que é parte de um programa mas é completamente ignorado pelo computador. No JavaScript temos duas maneiras de escrever os comentários. Para escrever em uma única linha de comentário, você pode usar dois caracteres barra (`//`) e então o comentário após.

```
var accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
var 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 até o final da linha. Uma seção de texto entre `/*` e `*/` será ignorado, independentemente se ele contém quebras de linha. Isto geralmente é útil para adicionar blocos de informação sobre um arquivo ou um pedaço do programa.

```
/*
I first found this number scrawled on the back of one of
my notebooks a few years ago. Since then, it has
occasionally dropped by, showing up in phone numbers and
the serial numbers of products that I bought. It
obviously likes me, so I've decided to keep it.
*/

var theNumber = 11213;
```

Resumo

Você agora sabe que um programa é construído de declarações, que as vezes contém mais declarações. Declarações tendem a conter expressões, que podem ser feitas de pequenas expressões.

Colocar declarações uma após a outra nos dá um programa que é executado de cima para baixo. Você pode causar transtornos no fluxo de controle usando declarações condicionais (`if` , `else` e `switch`) e loops (`while` , `do` e `for`).

As variáveis podem ser usadas para arquivar pedaços de dados sob um nome, e são úteis para rastrear o estado de um programa. O ambiente é um conjunto de variáveis que são definidas. O sistema JavaScript sempre coloca um número padrão de variáveis úteis dentro do seu ambiente.

Funções são valores especiais que encapsulam um pedaço do programa. Você pode invocá-las escrevendo `function Name (argument1, argument2) {}`. Essa chamada de função é uma expressão, que pode produzir um valor.

Exercícios

Se você está inseguro sobre como testar suas soluções para os exercícios, consulte a [introdução](#).

Cada exercício começa com a descrição de um problema. Leia e tente resolvê-lo. Se você tiver dificuldades, considere a leitura das dicas abaixo do exercício. As soluções completas para os exercícios não estão inclusas neste livro, mas você pode procurar elas online em eloquentjavascript.net/code. Se você quer aprender algo, eu recomendo que veja as soluções somente após ter resolvido o exercício, ou pelo menos, depois que tentou por um período longo e duro o suficiente para dar uma pequena dor de cabeça.

Triângulo com Loop

Escreva um programa que faça sete chamadas a `console.log()` para retornar o seguinte triângulo:

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

Uma maneira interessante para saber o comprimento de uma `string` é escrevendo `.length` após ela.

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

A maioria dos exercícios contém um pedaço de código que pode ser utilizada para alterar e resolver o exercício. Lembre-se que você pode clicar em um bloco de código para editá-lo.

```
// Your code here.
```

Dicas:

Você pode começar com um programa que simplesmente imprime os números de 1 a 7, na qual você pode derivar algumas modificações no exemplo de impressão de números dado no início do capítulo aqui, onde o loop foi introduzido.

Agora, considere a equivalência entre números e cadeias em um `hash` de caracteres. Você pode ir de 1 para 2 adicionando 1 (`+ = 1`). Você pode ir de `"#"` para `"##"`, adicionando um caractere (`+ = "#"`). Assim, a solução pode acompanhar de perto o número, de impressão do programa.

FizzBuzz

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

Quando o programa estiver funcionando, modifique-o para imprimir `FizzBuzz` para números que são divisíveis ambos por 3 e 5 (e continue imprimindo `Fizz` e `Buzz` para números divisíveis por apenas um deles).

(Isto é na verdade uma pergunta de entrevista usada para eliminar uma porcentagem significativa de candidatos programadores. Então se você resolvê-la, você está autorizado de se sentir bem consigo mesmo).

Dica:

Interar sobre os números é trabalho claro de um loop, e selecionar o que imprimir é uma questão de execução condicional. Lembre-se do truque de usar o operador restante (`%`) para verificar se um número é divisível por outro número (terá zero de resto).

Na primeira versão, existem três resultados possíveis para cada número, então você irá criar uma cadeia de `if/else if/else`.

Na segunda versão o programa tem uma solução simples e uma inteligente. A maneira mais simples é adicionar um outro "ramo" para um teste preciso da condição dada. Para o método inteligente é construir uma sequência de caracteres contendo palavra ou palavras para a saída, que imprima a palavra ou o número, caso não haja palavra, fazendo o uso do operador elegante `||`.

Tabuleiro de Xadrez

Escreva um programa que cria uma `string` que representa uma grade 8x8, usando novas linhas para separar os caracteres. A cada posição da grade existe um espaço ou um caractere "#". Esses caracteres formam um tabuleiro de xadrez.

Passando esta `string` para o `console.log` deve mostrar algo como isto:

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

Quando você tiver o programa que gere este padrão, defina a variável `size = 8` e altere programa para que ele funcione para qualquer `size`, a saída da grade de largura e altura.

```
// Your code here.
```

Dica:

A sequência pode ser construída iniciando vazia ("") e repetidamente adicionando caracteres. O caracter para uma nova linha é escrito assim `\n`.

Utilize `console.log` para visualizar a saída do seu programa.

Para trabalhar com duas dimensões, você irá precisar de um loop dentro de outro loop. Coloque entre chaves os "corpos" dos loops para se tornar mais fácil de visualizar quando inicia e quando termina. Tente recuar adequadamente esses "corpos". A ordem dos loops deve seguir a ordem que usamos para construir a string (linha

por linha, esquerda para direita, cima para baixo). Então o loop mais externo manipula as linhas e o loop interno manipula os caracteres por linha.

Você vai precisar de duas variáveis para acompanhar seu progresso. Para saber se coloca um espaço ou um "#" em uma determinada posição, você pode testar se a soma dos dois contadores ainda é divisível por (`% 2`).

Encerrando uma linha com um caracter de nova linha acontece após a linha de cima ser construída, faça isso após o loop interno, mas dentro do loop externo.

Funções

“As pessoas pensam que Ciência da Computação é a arte de gênios. Na realidade é o oposto, são várias pessoas fazendo coisas que dependem uma das outras, como um muro de pequenas pedras.” — Donald Knuth

Você já viu valores de funções como `alert`, e como invocá-las. Funções são essenciais na programação JavaScript. O conceito de encapsular uma parte do programa em um valor tem vários usos. É uma ferramenta usada para estruturar aplicações de larga escala, reduzir repetição de código, associar nomes a subprogramas e isolar esses subprogramas uns dos outros.

A aplicação mais óbvia das funções é quando queremos definir novos vocabulários. Criar novas palavras no nosso dia a dia geralmente não é uma boa ideia, porém em programação é indispensável.

Um adulto típico tem por volta de 20.000 palavras em seu vocabulário. Apenas algumas linguagens de programação possuem 20.000 conceitos embutidos, sendo que o vocabulário que se tem disponível tende a ser bem definido e, por isso, menos flexível do que a linguagem usada por humanos. Por isso, normalmente temos que adicionar conceitos do nosso próprio vocabulário para evitar repetição.

Definindo Uma Função

Uma definição de função nada mais é do que uma definição normal de uma variável, na qual o valor recebido pela variável é uma função. Por exemplo, o código a seguir define uma variável `square` que se refere a uma função que retorna o quadrado do número dado:

```
var square = function(x) {  
  return x * x;  
};  
  
console.log(square(12));  
// → 144
```

Uma função é criada por meio de uma expressão que se inicia com a palavra-chave `function`. Funções podem receber uma série de parâmetros (nesse caso, somente `x`) e um "corpo", contendo as declarações que serão executadas quando a função for invocada. O "corpo" da função deve estar sempre envolvido por chaves, mesmo quando for formado por apenas uma simples declaração (como no exemplo anterior).

Uma função pode receber múltiplos parâmetros ou nenhum parâmetro. No exemplo a seguir, `makeNoise` não recebe nenhum parâmetro, enquanto `power` recebe dois:

```
var makeNoise = function() {  
  console.log("Pling!");  
};  
  
makeNoise();  
// → Pling!  
  
var power = function(base, exponent) {  
  var result = 1;  
  for (var count = 0; count < exponent; count++)  
    result *= base;  
  return result;  
};  
  
console.log(power(2, 10));
```

```
// → 1024
```

Algumas funções produzem um valor, como as funções `power` e `square` acima, e outras não, como no exemplo de `makeNoise`, que produz apenas um “efeito colateral”. A declaração `return` é usada para determinar o valor de retorno da função. Quando o controle de execução interpreta essa declaração, ele sai imediatamente do contexto da função atual e disponibiliza o valor retornado para o código que invocou a função. A palavra-chave `return` sem uma expressão após, irá fazer com que o retorno da função seja `undefined`.

Parâmetros e Escopos

Os parâmetros de uma função comportam-se como variáveis regulares. Seu valor inicial é informado por quem invocou a função e não pelo código da função em si.

Uma propriedade importante das funções é que variáveis definidas dentro do “corpo” delas, incluindo seus parâmetros, são *locais* à própria função. Isso significa, por exemplo, que a variável `result` no exemplo `power` será criada novamente toda vez que a função for invocada, sendo que as diferentes execuções não interferem umas nas outras.

Essa característica de localidade das variáveis se aplica somente aos parâmetros e às variáveis que forem declaradas usando a palavra-chave `var` dentro do “corpo” de uma função. Variáveis declaradas fora do contexto de alguma função são chamadas de *globais* (não locais), pois elas são visíveis em qualquer parte da aplicação. É possível acessar variáveis *globais* dentro de qualquer função, contanto que você não tenha declarado uma variável local com o mesmo nome.

O código a seguir demonstra esse conceito. Ele define e executa duas funções em que ambas atribuem um valor à variável `x`. A primeira função `f1` declara a variável como local e então muda apenas seu valor. Já a segunda função `f2` não declara `x` localmente, portanto sua referência a `x` está associada à variável global `x` definida no topo do exemplo:

```
var x = "outside";

var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x);
// → outside

var f2 = function() {
  x = "inside f2";
};
f2();
console.log(x);
// → inside f2
```

Esse comportamento ajuda a prevenir interferências acidentais entre funções. Se todas as variáveis fossem compartilhadas por toda a aplicação, seria muito trabalhoso garantir que o mesmo nome não fosse utilizado em duas situações com propósitos diferentes. Além disso, se fosse o caso de reutilizar uma variável com o mesmo nome, talvez você pudesse se deparar com efeitos estranhos de códigos que alteram o valor da sua variável. Assumindo que variáveis locais existem apenas dentro do contexto da função, a linguagem torna possível ler e entender funções como “pequenos universos”, sem termos que nos preocupar com o código da aplicação inteira de uma só vez.

Escopo Aninhado

O JavaScript não se distingue apenas pela diferenciação entre variáveis *locais* e *globais*. Funções também podem ser criadas dentro de outras funções, criando vários níveis de “localidades”.

Por exemplo, a função `landscape` possui duas funções, `flat` e `mountain`, declaradas dentro do seu corpo:

```
var landscape = function() {
  var result = "";
  var flat = function(size) {
    for (var count = 0; count < size; count++)
      result += "_";
  };
  var mountain = function(size) {
    result += "/";
    for (var count = 0; count < size; count++)
      result += "'";
    result += "\\\\";
  };

  flat(3);
  mountain(4);
  flat(6);
  mountain(1);
  flat(1);
  return result;
};

console.log(landscape());
// → _ _ _ / ' ' ' ' \ \ \ \ / ' \ _
```

As funções `flat` e `mountain` podem “ver” a variável `result` porque elas estão dentro do mesmo escopo da função que as definiu. Entretanto, elas não conseguem ver a variável `count` uma da outra (somente a sua própria), pois elas estão definidas em escopos diferentes. O ambiente externo à função `landscape` não consegue ver as variáveis definidas dentro de `landscape`.

Em resumo, cada escopo local pode também ver todos os escopos locais que o contêm. O conjunto de variáveis visíveis dentro de uma função é determinado pelo local onde aquela função está escrita na aplicação. Todas as variáveis que estejam em blocos ao redor de definições de funções, são visíveis aos corpos dessas funções e também àqueles que estão no mesmo nível. Essa abordagem em relação à visibilidade de variáveis é chamada de *escopo léxico*.

Pessoas com experiência em outras linguagens de programação podem talvez esperar que qualquer bloco de código entre chaves produza um novo “ambiente local”. Entretanto, no JavaScript, as funções são as únicas coisas que podem criar novos escopos. Também é permitido a utilização de “blocos livres”:

```
var something = 1;
{
  var something = 2;
  // Do stuff with variable something...
}
// Outside of the block again...
```

Entretanto, a variável `something` dentro do bloco faz referência à mesma variável fora do bloco. Na realidade, embora blocos como esse sejam permitidos, eles são úteis somente para agrupar o corpo de uma declaração condicional `if` ou um laço de repetição.

Se você acha isso estranho, não se preocupe, pois não está sozinho. A próxima versão do JavaScript vai introduzir a palavra-chave `let`, que funcionará como `var`, mas criará uma variável que é local ao *bloco* que a contém e não à *função* que a contém.

Funções Como Valores

As variáveis de função, normalmente, atuam apenas como nomes para um pedaço específico de um programa. Tais variáveis são definidas uma vez e nunca se alteram. Isso faz com que seja fácil confundir a função com seu próprio nome.

Entretanto, são duas coisas distintas. Um valor de função pode fazer todas as coisas que outros valores podem fazer - você pode usá-lo em expressões arbitrárias e não apenas invocá-la. É possível armazenar um valor de função em um novo local, passá-lo como argumento para outra função e assim por diante. Não muito diferente, uma variável que faz referência a uma função continua sendo apenas uma variável regular e pode ser atribuída a um novo valor, como mostra o exemplo abaixo:

```
var launchMissiles = function(value) {  
    missileSystem.launch("now");  
};  
  
if (safeMode)  
    launchMissiles = function(value) { /* do nothing */};
```

No capítulo 5, nós vamos discutir as coisas maravilhosas que podem ser feitas quando passamos valores de função para outras funções.

Notação Por Declaração

Existe uma maneira mais simples de expressar “`var square = function...`”. A palavra-chave `function` também pode ser usada no início da declaração, como demonstrado abaixo:

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

Isso é uma *declaração de função*. Ela define a variável `square` e faz com que ela referencie a função em questão. Até agora tudo bem, porém existe uma pequena diferença nessa maneira de definir uma função.

```
console.log("The future says:", future());  
  
function future() {  
    return "We STILL have no flying cars.";  
}
```

O exemplo acima funciona, mesmo sabendo que a função foi definida *após* o código que a executa. Isso ocorre porque as declarações de funções não fazem parte do fluxo normal de controle, que é executado de cima para baixo. Elas são conceitualmente movidas para o topo do escopo que as contém e podem ser usadas por qualquer código no mesmo escopo. Isso pode ser útil em algumas situações, porque nos permite ter a liberdade de ordenar o código de uma maneira que seja mais expressiva, sem nos preocuparmos muito com o fato de ter que definir todas as funções antes de usá-las.

O que acontece quando definimos uma declaração de função dentro de um bloco condicional (`if`) ou um laço de repetição? Bom, não faça isso. Diferentes plataformas JavaScript usadas em diferentes navegadores têm tradicionalmente feito coisas diferentes nessas situações, e a última versão basicamente proíbe essa prática. Se você deseja que seu programa se comporte de forma consistente, use somente essa forma de definição de função no bloco externo de uma outra função ou programa.

```
function example() {
```

```
function a() {} // Okay
if (something) {
  function b() {} // Danger!
}
}
```

A Pilha de Chamadas

Será muito útil observarmos como o fluxo de controle flui por meio das execuções das funções. Aqui, temos um simples programa fazendo algumas chamadas de funções:

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

A execução desse programa funciona da seguinte forma: a chamada à função `greet` faz com que o controle pule para o início dessa função (linha 2). Em seguida, é invocado `console.log` (uma função embutida no navegador), que assume o controle, faz seu trabalho e então retorna o controle para a linha 2 novamente. O controle chega ao fim da função `greet` e retorna para o local onde a função foi invocada originalmente (linha 4). Por fim, o controle executa uma nova chamada a `console.log`.

Podemos representar o fluxo de controle, esquematicamente, assim:

```
top
  greet
    console.log
  greet
top
  console.log
top
```

Devido ao fato de que a função deve retornar ao local onde foi chamada após finalizar a sua execução, o computador precisa se lembrar do contexto no qual a função foi invocada originalmente. Em um dos casos, `console.log` retorna o controle para a função `greet`. No outro caso, ela retorna para o final do programa.

O local onde o computador armazena esse contexto é chamado de *call stack* (pilha de chamadas). Toda vez que uma função é invocada, o contexto atual é colocado no topo dessa "pilha" de contextos. Quando a função finaliza sua execução, o contexto no topo da pilha é removido e utilizado para continuar o fluxo de execução.

O armazenamento dessa pilha de contextos necessita de espaço na memória do computador. Quando a pilha começar a ficar muito grande, o computador reclamará com uma mensagem do tipo *out of stack space* (sem espaço na pilha) ou *too much recursion* (muitas recursões). O código a seguir demonstra esse problema fazendo uma pergunta muito difícil para o computador, que resultará em um ciclo infinito de chamadas entre duas funções. Se o computador tivesse uma pilha de tamanho infinito, isso poderia ser possível, no entanto, eventualmente chegaremos ao limite de espaço e explodiremos a "pilha".

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


Argumentos Opcionais

O código abaixo é permitido e executa sem problemas:

```
alert("Hello", "Good Evening", "How do you do?");
```

A função `alert`, oficialmente, aceita somente um argumento. No entanto, quando você a chama assim, ela não reclama. Ela simplesmente ignora os outros argumentos e lhe mostra o seu "Hello".

O JavaScript é extremamente tolerante com a quantidade de argumentos que você passa para uma função. Se você passar mais argumentos que o necessário, os extras serão ignorados. Se você passar menos argumentos, os parâmetros faltantes simplesmente receberão o valor `undefined`.

A desvantagem disso é que, possivelmente - e provavelmente - você passará um número errado de argumentos, de forma acidental, para as funções e nada irá alertá-lo sobre isso.

A vantagem é que esse comportamento pode ser usado em funções que aceitam argumentos opcionais. Por exemplo, a versão seguinte de `power` pode ser chamada com um ou dois argumentos. No caso de ser invocada com apenas um argumento, ela assumirá o valor 2 para o expoente e a função se comportará com um expoente ao quadrado.

```
function power(base, exponent) {  
  if (exponent == undefined)  
    exponent = 2;  
  var result = 1;  
  for (var count = 0; count < exponent; count++)  
    result *= base;  
  return result;  
}  
  
console.log(power(4));  
// → 16  
console.log(power(4, 3));  
// → 64
```

No [próximo capítulo](#), veremos uma maneira de acessar a lista que contém todos os argumentos que foram passados para uma função. Isso é útil, pois torna possível uma função aceitar qualquer número de argumentos. Por exemplo, `console.log` tira proveito disso, imprimindo todos os valores que foram passados.

```
console.log("R", 2, "D", 2);  
// → R 2 D 2
```

Closure

A habilidade de tratar funções como valores, combinada com o fato de que variáveis locais são recriadas toda vez que uma função é invocada; isso traz à tona uma questão interessante.

O que acontece com as variáveis locais quando a função que as criou não está mais ativa?

O código a seguir mostra um exemplo disso. Ele define uma função `wrapValue` que cria uma variável local e retorna uma função que acessa e retorna essa variável.

```
function wrapValue(n) {  
  var localVariable = n;  
  return function() { return localVariable; };  
}
```

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

Isso é permitido e funciona como você espera: a variável ainda pode ser acessada. Várias instâncias da variável podem coexistir, o que é uma boa demonstração do conceito de que variáveis locais são realmente recriadas para cada nova chamada, sendo que as chamadas não interferem nas variáveis locais umas das outras.

A funcionalidade capaz de referenciar uma instância específica de uma variável local após a execução de uma função é chamada de *closure*. Uma função que *closes over* (fecha sobre) variáveis locais é chamada de *closure*.

Esse comportamento faz com que você não tenha que se preocupar com o tempo de vida das variáveis, como também permite usos criativos de valores de função.

Com uma pequena mudança, podemos transformar o exemplo anterior, possibilitando a criação de funções que se multiplicam por uma quantidade arbitrária.

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

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

A variável explícita `localVariable` do exemplo na função `wrapValue` não é necessária, pois o parâmetro em si já é uma variável local.

Pensar em programas que funcionam dessa forma requer um pouco de prática. Um bom modelo mental é pensar que a palavra-chave `function` "congela" o código que está em seu corpo e o envolve em um pacote (o valor da função). Quando você ler `return function(...) {...}`, pense como se estivesse retornando um manipulador que possibilita executar instruções computacionais que foram "congeladas" para um uso posterior.

No exemplo, `multiplier` retorna um pedaço de código "congelado" que fica armazenado na variável `twice`. A última linha do exemplo chama o valor armazenado nessa variável, fazendo com que o código "congelado" (`return number * factor;`) seja executado. Ele continua tendo acesso à variável `factor` que foi criada na chamada de `multiplier` e, além disso, tem acesso ao argumento que foi passado a ele (o valor 5) por meio do parâmetro `number`.

Recursão

É perfeitamente aceitável uma função invocar a si mesma, contanto que se tenha cuidado para não sobrecarregar a pilha de chamadas. Uma função que invoca a si mesma é denominada *recursiva*. A recursividade permite que as funções sejam escritas em um estilo diferente. Veja neste exemplo uma implementação alternativa de `power`:

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

```
console.log(power(2, 3));  
// → 8
```

Essa é a maneira mais próxima da forma como os matemáticos definem a exponenciação, descrevendo o conceito de uma forma mais elegante do que a variação que usa um laço de repetição. A função chama a si mesma várias vezes com diferentes argumentos para alcançar a multiplicação repetida.

Entretanto, há um grave problema: em implementações típicas no JavaScript, a versão recursiva é aproximadamente dez vezes mais lenta do que a variação que utiliza um laço de repetição. Percorrer um laço de repetição simples é mais rápido do que invocar uma função múltiplas vezes.

O dilema velocidade versus elegância é bastante interessante. Você pode interpretá-lo como uma forma de transição gradual entre acessibilidade para humanos e máquina. Praticamente todos os programas podem se tornar mais rápidos quando se tornam maiores e mais complexos, cabendo ao desenvolvedor decidir qual o balanço ideal entre ambos.

No caso da [versão anterior](#) da implementação de `power`, a versão menos elegante (usando laço de repetição) é bem simples e fácil de ser lida, não fazendo sentido substituí-la pela versão recursiva. Porém, frequentemente lidamos com aplicações mais complexas e sacrificar um pouco a eficiência para tornar o código mais legível e simples acaba se tornando uma escolha atrativa.

A regra básica que tem sido repetida por muitos programadores e com a qual eu concordo plenamente, é não se preocupar com eficiência até que você saiba, com certeza, que o programa está muito lento. Quando isso acontecer, encontre quais partes estão consumindo maior tempo de execução e comece a trocar elegância por eficiência nessas partes.

É evidente que essa regra não significa que se deva ignorar a performance completamente. Em muitos casos, como na função `power`, não há muitos benefícios em usar a abordagem mais elegante. Em outros casos, um programador experiente pode identificar facilmente, que uma abordagem mais simples nunca será rápida o suficiente.

A razão pela qual estou enfatizando isso é que, surpreendentemente, muitos programadores iniciantes focam excessivamente em eficiência até nos menores detalhes. Isso acaba gerando programas maiores, mais complexos e muitas vezes menos corretos, que demoram mais tempo para serem escritos e, normalmente, executam apenas um pouco mais rapidamente do que as variações mais simples e diretas.

Porém, muitas vezes a recursão não é uma alternativa menos eficiente do que um laço de repetição. É muito mais simples resolver alguns problemas com recursão do que com laços de repetição. A maioria desses problemas envolve exploração ou processamento de várias ramificações, as quais podem se dividir em novas ramificações e assim por diante.

Considere este quebra-cabeça: iniciando com o número 1 e repetidamente adicionando 5 ou multiplicando por 3, uma infinita quantidade de novos números pode ser produzida. Como você implementaria uma função que, dado um número, tenta achar a sequência de adições e multiplicações que produzem esse número? Por exemplo, o número 13 pode ser produzido multiplicando-se por 3 e adicionando-se 5 duas vezes. Já o número 15 não pode ser produzido de nenhuma forma.

Aqui está uma solução recursiva:

```
function findSolution(target) {  
  function find(start, history) {  
    if (start == target)  
      return history;  
    else if (start > target)  
      return null;  
    else  
      return find(start + 5, "(" + history + " + 5)") ||  
             find(start * 3, "(" + history + " * 3)");  
  }  
}
```

```
    return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

Note que esse programa não necessariamente encontra a menor sequência de operações. Ele termina sua execução quando encontra a primeira solução possível.

Eu não espero que você entenda como isso funciona imediatamente, mas vamos analisar o exemplo, pois é um ótimo exercício para entender o pensamento recursivo.

A função interna `find` é responsável pela recursão. Ela recebe dois argumentos (o número atual e uma string que registra como chegamos a esse número) e retorna uma string que mostra como chegar no número esperado ou `null`.

Para fazer isso, a função executa uma entre três ações possíveis. Se o número atual é o número esperado, o histórico atual reflete uma possível sequência para alcançar o número esperado, então ele é simplesmente retornado. Se o número atual é maior que o número esperado, não faz sentido continuar explorando o histórico, já que adicionar ou multiplicar o número atual gerará um número ainda maior. Por fim, se nós tivermos um número menor do que o número esperado, a função tentará percorrer todos os caminhos possíveis que iniciam do número atual, chamando ela mesma duas vezes, uma para cada próximo passo que seja permitido. Se a primeira chamada retornar algo que não seja `null`, ela é retornada. Caso contrário, a segunda chamada é retornada, independentemente se ela produzir string ou `null`.

Para entender melhor como essa função produz o resultado que estamos esperando, vamos analisar todas as chamadas a `find` que são feitas quando procuramos a solução para o número 13.

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

A indentação reflete a profundidade da pilha de chamadas. A primeira chamada do `find` invoca a si mesma duas vezes, explorando as soluções que começam com `(1 + 5)` e `(1 * 3)`. A primeira chamada tenta achar a solução que começa com `(1 + 5)` e, usando recursão, percorre todas as possíveis soluções que produzam um número menor ou igual ao número esperado. Como ele não encontra uma solução para o número esperado, o valor `null` é retornado até retornar para a chamada inicial. Nesse momento, o operador `||` faz com que a pilha de chamadas inicie o processo de exploração pelo outro caminho `(1 * 3)`. Essa busca tem resultados satisfatórios, porque após duas chamadas recursivas acaba encontrando o número 13. Essa chamada recursiva mais interna retorna uma string e cada operador `||` nas chamadas intermediárias passa essa string adiante, retornando no final a solução esperada.

Funções Crescentes

Existem duas razões naturais para as funções serem introduzidas nos programas.

A primeira delas é quando você percebe que está escrevendo o mesmo código várias vezes. Nós queremos evitar isso, pois quanto mais código, maiores são as chances de erros e mais linhas de código há para as pessoas lerem e entenderem o programa. Por isso, nós extraímos a funcionalidade repetida, encontramos um bom nome para ela e colocamos dentro de uma função.

A segunda razão é quando você precisa de uma funcionalidade que ainda não foi escrita e que merece ser encapsulada em uma função própria. Você começa dando um nome à função e, em seguida, escreve o seu corpo. Às vezes, você pode até começar escrevendo o código que usa a função antes mesmo de defini-la.

A dificuldade de encontrar um bom nome para uma função é um bom indicativo de quão claro é o conceito que você está tentando encapsular. Vamos analisar um exemplo.

Nós queremos escrever um programa que imprima dois números, sendo eles o número de vacas e galinhas em uma fazenda com as palavras *Cows* (vacas) e *Chickens* (galinhas) depois deles e algarismos zeros antes de ambos os números para que sejam sempre números de três dígitos.

```
007 Cows
011 Chickens
```

Bom, claramente, isso é uma função que exige dois argumentos. Vamos codar.

```
function printFarmInventory(cows, chickens) {
  var cowString = String(cows);
  while (cowString.length < 3)
    cowString = "0" + cowString;
  console.log(cowString + " Cows");
  var chickenString = String(chickens);
  while (chickenString.length < 3)
    chickenString = "0" + chickenString;
  console.log(chickenString + " Chickens");
}
printFarmInventory(7, 11);
```

Adicionar `.length` após o valor de uma `string` nos fornecerá o tamanho (quantidade de caracteres) daquela `string`. Por isso, o laço de repetição `while` continua adicionando zeros no início da `string` que representa o número até que a mesma tenha três caracteres.

Missão cumprida! Porém, no momento em que iríamos enviar o código ao fazendeiro (juntamente com uma grande cobrança, é claro), ele nos ligou dizendo que começou a criar porcos, e perguntou, se poderíamos estender a funcionalidade do software para também imprimir os porcos?

É claro que podemos. Antes de entrar no processo de copiar e colar essas mesmas quatro linhas outra vez, vamos parar e reconsiderar. Deve existir uma forma melhor. Aqui está a primeira tentativa:

```
function printZeroPaddedWithLabel(number, label) {
  var 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);
```

Funcionou! Mas o nome `printZeroPaddedWithLabel` é um pouco estranho. Ele é uma combinação de três coisas - imprimir, adicionar zeros e adicionar a label correta - em uma única função.

Ao invés de tentarmos abstrair a parte repetida do nosso programa como um todo, vamos tentar selecionar apenas um conceito.

```
function zeroPad(number, width) {
  var 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);
```

Ter uma função com um bom nome descritivo como `zeroPad` torna fácil para qualquer um ler e entender o código. Além disso, ele pode ser útil em outras situações, além desse programa específico. Você pode usá-lo, por exemplo, para imprimir números corretamente alinhados em uma tabela.

O quão inteligente e versátil as nossas funções deveriam ser? Nós poderíamos escrever funções extremamente simples, que apenas adicionam algarismos para que o número tenha três caracteres, até funções complicadas, para formatação de números fracionários, números negativos, alinhamento de casas decimais, formatação com diferentes caracteres e por aí vai.

Um princípio útil é não adicionar funcionalidades, a menos que você tenha certeza absoluta de que irá precisar delas. Pode ser tentador escrever soluções genéricas para cada funcionalidade com que você se deparar. Resista a essa vontade. Você não vai ganhar nenhum valor real com isso e vai acabar escrevendo muitas linhas de código que nunca serão usadas.

Funções e Efeitos Colaterais

Funções podem ser divididas naquelas que são invocadas para produzir um efeito colateral e naquelas que são invocadas para gerar um valor de retorno (embora também seja possível termos funções que produzam efeitos colaterais e que retornem um valor).

A primeira função auxiliar no exemplo da fazenda, `printZeroPaddedWithLabel`, é invocada para produzir um efeito colateral: imprimir uma linha. A segunda versão, `zeroPad`, é chamada para produzir um valor de retorno. Não é coincidência que a segunda versão é útil em mais situações do que a primeira. Funções que criam valores são mais fáceis de serem combinadas de diferentes maneiras do que funções que produzem efeitos colaterais diretamente.

Uma função "pura" é um tipo específico de função que produz valores e que não gera efeitos colaterais, como também não depende de efeitos colaterais de outros códigos — por exemplo, ela não utiliza variáveis globais que podem ser alteradas por outros códigos. Uma função pura tem a característica de, ser sempre chamada com os mesmos argumentos, produzir o mesmo valor (e não fará nada além disso). Isso acaba fazendo com que seja fácil de entendermos como ela funciona. Uma chamada para tal função pode ser mentalmente substituída pelo seu resultado, sem alterar o significado do código. Quando você não tem certeza se uma função pura está funcionando corretamente, você pode testá-la simplesmente invocando-a. Sabendo que ela funciona nesse contexto, funcionará em qualquer outro contexto. Funções que não são "puras" podem retornar valores diferentes baseados em vários tipos de fatores e produzem efeitos colaterais que podem fazer com que seja difícil de testar e pensar sobre elas.

Mesmo assim, não há necessidade de se sentir mal ao escrever funções que não são "puras" ou começar uma "guerra santa" para eliminar códigos impuros. Efeitos colaterais são úteis em algumas situações. Não existe uma versão "pura" de `console.log`, por exemplo, e `console.log` certamente é útil. Algumas operações são também mais fáceis de se expressar de forma mais eficiente quando usamos efeitos colaterais, portanto a velocidade de computação pode ser uma boa razão para se evitar a "pureza".

Resumo

Este capítulo ensinou a você como escrever suas próprias funções. A palavra-chave `function`, quando usada como uma expressão, pode criar um valor de função. Quando usada como uma declaração, pode ser usada para declarar uma variável e dar a ela uma função como valor.

```
// Create a function value f
var f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}
```

Um aspecto chave para entender funções, é entender como os escopos locais funcionam. Parâmetros e variáveis declaradas dentro de uma função são locais àquela função, recriados toda vez que a função é invocada, e não são acessíveis do contexto externo à função. Funções declaradas dentro de outras têm acesso ao escopo local das funções mais externas que as envolvem.

Separar as tarefas que a sua aplicação executa em diferentes funções, é bastante útil. Você não terá que repetir o código e as funções fazem um programa mais legível, agrupando o código em pedaços conceituais, da mesma forma que os capítulos e as seções ajudam a organizar um texto.

Exercícios

Mínimo

O [capítulo anterior](#) introduziu a função `Math.min` que retorna o seu menor argumento. Nós podemos reproduzir essa funcionalidade agora. Escreva uma função `min` que recebe dois argumentos e retorna o menor deles.

```
// Your code here.

console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

Dica: Se estiver tendo problemas para colocar as chaves e os parênteses nos seus lugares corretos, para ter uma definição de uma função válida, comece copiando um dos exemplos desse capítulo e modificando-o. Uma função pode conter várias declarações de retorno (`return`).

Recursão

Nós vimos que o `%` (operador resto) pode ser usado para testar se um número é par ou ímpar, usando `% 2` para verificar se ele é divisível por dois. Abaixo, está uma outra maneira de definir se um número inteiro positivo é par ou ímpar:

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

Defina uma função recursiva `isEven` que satisfaça as condições descritas acima. A função deve aceitar um número como parâmetro e retornar um valor Booleano.

Teste-a com os valores 50 e 75. Observe como ela se comporta com o valor -1. Por quê? Você consegue pensar em uma maneira de arrumar isso?

```
// Your code here.  
  
console.log(isEven(50));  
// → true  
console.log(isEven(75));  
// → false  
console.log(isEven(-1));  
// → ??
```

Dica: Sua função será semelhante à função interna `find` do exemplo recursivo `findSolution` neste capítulo, com uma cadeia de declarações `if / else if / else` que testam qual dos três casos se aplica. O `else final`, correspondente ao terceiro caso, é responsável por fazer a chamada recursiva. Cada uma das ramificações deverá conter uma declaração de retorno ou retornar um valor específico.

Quando o argumento recebido for um número negativo, a função será chamada recursivamente várias vezes, passando para si mesma um número cada vez mais negativo, afastando-se cada vez mais de retornar um resultado. Ela, eventualmente, consumirá todo o espaço em memória da pilha de chamadas e abortar.

Contando feijões

Você pode acessar o N-ésimo caractere, ou letra, de uma `string` escrevendo `"string".charAt(N)`, similar a como você acessa seu tamanho com `"s".length`. O valor retornado será uma `string` contendo somente um caractere (por exemplo, `"b"`). O primeiro caractere está na posição zero, o que faz com que o último seja encontrado na posição `string.length - 1`. Em outras palavras, uma `string` com dois caracteres possui tamanho (`length`) dois, e suas respectivas posições são `0` e `1`.

Escreva uma função `countBs` que receba uma `string` como único argumento e retorna o número que indica quantos caracteres "B", em maiúsculo, estão presentes na `string`.

Em seguida, escreva uma função chamada `countChar` que se comporta de forma parecida com `countBs`, exceto que ela recebe um segundo argumento que indica o caractere que será contado (ao invés de contar somente o caractere "B" em maiúsculo). Reescreva `countBs` para fazer essa nova funcionalidade.

```
// Your code here.  
  
console.log(countBs("BBC"));  
// → 2  
console.log(countChar("kakkerlak", "k"));  
// → 4
```

Dica: Um laço de repetição em sua função fará com que todos os caracteres na `string` sejam verificados se usarmos um índice de zero até uma unidade abaixo que seu tamanho (`< string.length`). Se o caractere na posição atual for o mesmo que a função está procurando, ele incrementará uma unidade na variável de contagem (`counter`).

Quando o laço chegar ao seu fim, a variável `counter` deverá ser retornada.

Certifique-se de usar e criar variáveis locais à função, utilizando a palavra-chave `var` .

Estrutura de dados: Objetos e *Array*

Em duas ocasiões me perguntaram: "Ora, Sr. Babbage, se você colocar números errados em uma máquina, repostas certas irão sair?" [...] Certamente eu não sou capaz de compreender o tipo de confusão de ideias que poderia provocar tal questionamento.

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

Números, Booleanos e *strings* são os tijolos usados para construir as estruturas de dados. Entretanto, você não consegue fazer uma casa com um único tijolo. Objetos nos permitem agrupar valores (incluindo outros objetos) e, conseqüentemente, construir estruturas mais complexas.

Os programas que construímos até agora foram seriamente limitados devido ao fato de que estiveram operando apenas com tipos de dados simples. Esse capítulo irá adicionar uma compreensão básica sobre estrutura de dados para o seu *kit* de ferramentas. Ao final, você saberá o suficiente para começar a escrever programas úteis.

O capítulo irá trabalhar com um exemplo de programação mais ou menos realista, introduzindo conceitos a medida em que eles se aplicam ao problema em questão. O código de exemplo será, muitas vezes, construído em cima de funções e variáveis que foram apresentadas no início do texto.

O Esquilo-homem

De vez em quando, geralmente entre oito e dez da noite, Jacques se transforma em um pequeno roedor peludo com uma cauda espessa.

Por um lado, Jacques fica muito contente por não ter licantropia clássica. Transformar-se em um esquilo tende a causar menos problemas do que se transformar em um lobo. Ao invés de ter que se preocupar em comer acidentalmente o vizinho (isso seria bem estranho), ele tem que se preocupar em não ser comido pelo gato do vizinho. Após duas ocasiões em que ele acordou nu, desorientado e em cima de um galho fino na copa de uma árvore, ele resolveu trancar as portas e as janelas do seu quarto durante a noite e colocar algumas nozes no chão para manter-se ocupado.



Isso resolve os problemas do gato e da árvore. Mesmo assim, Jacques ainda sofre com sua condição. As ocorrências irregulares das transformações o faz suspeitar de que talvez possa ter alguma coisa que as ativam. Por um tempo, ele acreditava que isso só acontecia nos dias em que ele havia tocado em árvores. Por isso, ele parou de tocar de vez nas árvores e até parou de ficar perto delas, mas o problema persistiu.

Mudando para uma abordagem mais científica, Jacques pretende começar a manter um registro diário de tudo o que ele faz e se ele se transformou. Com essas informações, ele espera ser capaz de diminuir e limitar as condições que ativam as transformações.

A primeira coisa que ele deverá fazer é criar uma estrutura de dados para armazenar essas informações.

Conjuntos de dados

Para trabalhar com um pedaço de dados digitais, primeiramente precisamos encontrar uma maneira de representá-los na memória da nossa máquina. Vamos dizer que, como um exemplo simples, queremos representar a coleção de números: 2, 3, 5, 7 e 11.

Poderíamos ser criativos com *strings* (elas podem ter qualquer tamanho, assim podemos armazenar muitos dados nelas) e usar "2 3 5 7 11" como nossa representação. Entretanto, isso é estranho, pois você teria que, de alguma forma, extrair os dígitos e convertê-los em números para poder acessá-los.

Felizmente, o JavaScript fornece um tipo de dado específico para armazenar uma sequência de valores. Ele é chamado de *array* e é escrito como uma lista de valores separados por vírgulas e entre colchetes.

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

A notação para acessar elementos contidos em um *array* também usa colchetes. Um par de colchetes imediatamente após uma expressão, contendo outra expressão entre esses colchetes, irá procurar o elemento contido na expressão à esquerda que está na posição dada pelo *índice* determinado pela expressão entre colchetes.

Indexação de Arrays

O primeiro índice de um *array* é o número zero e não o número um. Portanto, o primeiro elemento pode ser acessado usando `listOfNumbers[0]`. Se você não tem experiência com programação, essa convenção pode levar um tempo para se acostumar. Mesmo assim, a contagem baseada em zero é uma tradição de longa data no mundo da tecnologia e, desde que seja seguida consistentemente (que é o caso no JavaScript), ela funciona bem.

Propriedades

Nós vimos, em exemplos anteriores, algumas expressões de aparência suspeita, como `myString.length` (para acessar o tamanho de uma *string*) e `Math.max` (função que retorna o valor máximo). Essas são expressões que acessam uma propriedade em algum valor. No primeiro caso, acessamos a propriedade `length` do valor contido em `myString`. No segundo, acessamos a propriedade chamada `max` no objeto `Math` (que é um conjunto de valores e funções relacionados à matemática).

Praticamente todos os valores no JavaScript possuem propriedades. As únicas exceções são `null` e `undefined`. Se você tentar acessar a propriedade em algum deles, você receberá um erro.

```
null.length;
// → TypeError: Cannot read property 'length' of null
```

As duas formas mais comuns de acessar propriedades no JavaScript são usando ponto e colchetes. Ambos `value.x` e `value[x]` acessam uma propriedade em `value`, mas não necessariamente a mesma propriedade. A diferença está em como o `x` é interpretado. Quando usamos o ponto, a parte após o ponto deve ser um nome de

variável válido e referente ao nome da propriedade em questão. Quando usamos colchetes, a expressão entre os colchetes é avaliada para obter o nome da propriedade. Enquanto que `value.x` acessa a propriedade chamada "x", `value[x]` tenta avaliar a expressão `x` e, então, usa o seu resultado como o nome da propriedade.

Portanto, se você sabe que a propriedade que você está interessado se chama "length", você usa `value.length`. Se você deseja extrair a propriedade cujo nome é o valor que está armazenado na variável `i`, você usa `value[i]`. Devido ao fato de que nomes de propriedades podem ser qualquer *string*, se você quiser acessar as propriedades "2" ou "John Doe", você deve usar os colchetes: `value[2]` OU `value["John Doe"]`, pois mesmo sabendo exatamente o nome da propriedade, "2" e "John Doe" não são nomes válidos de variáveis, sendo impossível acessá-los usando a notação com o ponto.

Métodos

Ambos os objetos *string* e *array* possuem, além da propriedade `length`, um número de propriedades que se referem à valores de função.

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

Toda *string* possui uma propriedade `toUpperCase`. Quando chamada, ela retornará uma cópia da string com todas as letras convertidas para maiúsculas. Existe também a propriedade `toLowerCase`, que você já pode imaginar o que faz.

Curiosamente, mesmo que a chamada para `toUpperCase` não passe nenhum argumento, de alguma forma a função tem acesso à *string* "Doh", que é o valor em que a propriedade foi chamada. Como isso funciona exatamente é descrito no [Capítulo 6](#).

As propriedades que contêm funções são geralmente chamadas de *métodos* do valor a que pertencem. Como por exemplo, `toUpperCase` é um método de uma *string*.

O exemplo a seguir demonstra alguns métodos que os objetos do tipo *array* possuem:

```
var mack = [];
mack.push("Mack");
mack.push("the", "Knife");
console.log(mack);
// → ["Mack", "the", "Knife"]
console.log(mack.join(" "));
// → Mack the Knife
console.log(mack.pop());
// → Knife
console.log(mack);
// → ["Mack", "the"]
```

O método `push` pode ser usado para adicionar valores ao final de um *array*. O método `pop` faz o contrário, remove o valor que está no final do *array* e o retorna. Um *array* de *strings* pode ser combinado em uma única *string* com o método `join`. O argumento passado para `join` determina o texto que será inserido entre cada elemento do *array*.

Objetos

Voltando ao esquilo-homem. Um conjunto de registros diários pode ser representado como um *array*. Entretanto, as entradas não são compostas apenas por um número ou uma *string*, pois precisam armazenar a lista de atividades e um valor booleano que indica se Jacques se transformou em um esquilo. Nós deveríamos, idealmente, agrupar

esses valores em um único valor e, em seguida, colocá-los em um *array* com os registros.

Valores do tipo *objeto* são coleções arbitrárias de propriedades, sendo que podemos adicionar ou remover essas propriedades da forma que desejarmos. Uma maneira de criar um objeto é usando a notação com chaves.

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

Dentro das chaves, podemos informar uma lista de propriedades separadas por vírgulas. Cada propriedade é escrita com um nome seguido de dois pontos e uma expressão que fornece o valor da propriedade. Espaços e quebras de linha não fazem diferença. Quando um objeto se estende por várias linhas, indentá-lo, como mostrado no exemplo anterior, melhora a legibilidade. Propriedades cujos nomes não são variáveis ou números válidos precisam estar entre aspas.

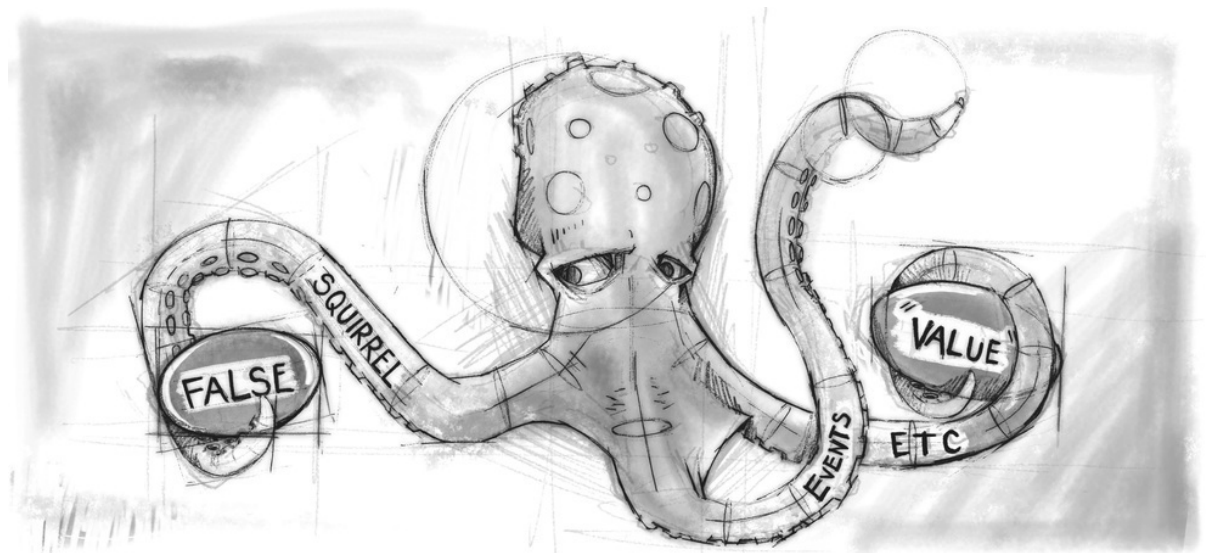
```
var descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

Isso significa que as chaves possuem dois significados no JavaScript. Quando usadas no início de uma declaração, elas definem o começo de um bloco de declarações. Em qualquer outro caso, elas descrevem um objeto. Felizmente, é praticamente inútil iniciar uma declaração com as chaves de um objeto e, em programas normais, não existe ambiguidade entre os dois casos de uso.

Tentar acessar uma propriedade que não existe irá produzir um valor `undefined`, o que acontece quando tentamos ler pela primeira vez a propriedade `wolf` no exemplo anterior.

É possível atribuir um valor a uma propriedade usando o operador `=`. Isso irá substituir o valor de uma propriedade, caso ela exista, ou criar uma nova propriedade no objeto se não existir.

Retornando brevemente ao nosso modelo de tentáculos para associações de variáveis, as associações de propriedades funcionam de forma similar. Elas *recebem* valores, mas outras variáveis e propriedades podem também estar associadas aos mesmos valores. Você pode pensar em objetos como polvos com um número qualquer de tentáculos, e cada tentáculo com um nome escrito nele.

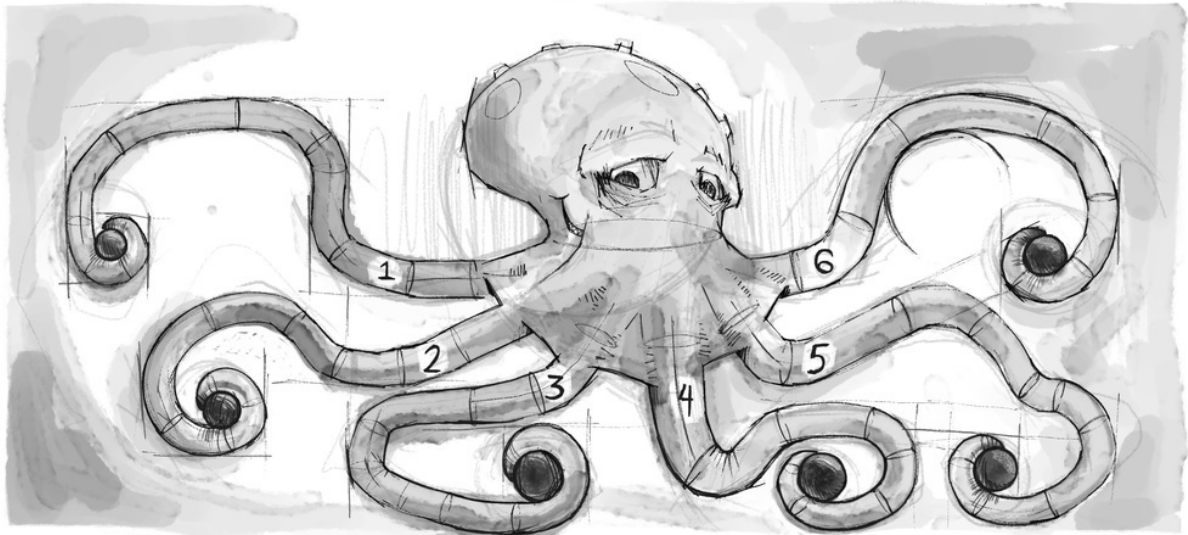


O operador `delete` corta um tentáculo de nosso polvo. Ele é um operador unário que, quando aplicado a uma propriedade, irá remover tal propriedade do objeto. Isso não é algo comum de se fazer, mas é possível.

```
var 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* ou a um objeto, retorna um valor booleano que indica se aquele objeto possui aquela propriedade. A diferença entre alterar uma propriedade para `undefined` e removê-la de fato, é que no primeiro caso, o objeto *continua com a propriedade* (ela simplesmente não tem um valor muito interessante), enquanto que no segundo caso, a propriedade não estará mais presente no objeto e o operador `in` retornará `false`.

Os *arrays* são, então, apenas um tipo especializado de objeto para armazenar sequências de coisas. Se você executar `typeof [1, 2]`, irá produzir `"object"`. Você pode interpretá-los como polvos com longos tentáculos de tamanhos semelhantes, ordenados em linha e rotulados com números.



Portanto, podemos representar o diário de Jacques como um *array* de objetos.

```
var 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},
  /* and so on... */
];
```

Mutabilidade

Nós iremos chegar na programação de fato muito em breve. Mas há, primeiramente, uma última parte teórica que precisamos entender.

Nós vimos que os valores de objetos podem ser modificados. Os tipos de valores discutidos nos capítulos anteriores, tais como números, *strings* e booleanos, são *imutáveis*. É impossível mudar o valor já existente desses tipos. Você pode, a partir deles, combiná-los e criar novos valores, mas quando você analisar um valor específico de *string*, ele será sempre o mesmo, sendo que o seu texto não pode ser alterado. Por exemplo, se você tiver referência a uma *string* que contém "cat", é impossível que outro código altere os caracteres dessa *string* para "rat".

Por outro lado, no caso de objetos, o conteúdo de um valor pode ser modificado quando alteramos suas propriedades.

Quando temos dois números, 120 e 120, podemos considerá-los exatamente o mesmo número, quer se refiram ou não aos mesmos bits físicos. Entretanto, no caso de objetos há uma diferença entre ter duas referências para o mesmo objeto e ter dois objetos diferentes que possuem as mesmas propriedades. Considere o código a seguir:

```
var object1 = {value: 10};
var object2 = object1;
var 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
```

As variáveis `object1` e `object2` estão associadas ao *mesmo objeto* e, por isso, alterar `object1` também altera o valor de `object2`. A variável `object3` aponta para um objeto diferente, o qual inicialmente contém as mesmas propriedades de `object1` e sua existência é totalmente separada.

Quando comparamos objetos, o operador `==` do JavaScript irá retornar `true` apenas se ambos os objetos possuem exatamente o mesmo valor. Comparar objetos diferentes irá retornar `false` mesmo se eles tiverem conteúdos idênticos. Não existe uma operação nativa no JavaScript de "*deep comparison*" (comparação "profunda"), onde se verifica o conteúdo de um objeto, mas é possível escrevê-la você mesmo (que será um dos [exercícios](#) ao final desse capítulo).

O log da licantropia

Jacques inicia seu interpretador de JavaScript e configura o ambiente que ele precisa para manter o seu diário.

```
var journal = [];
```

```
function addEntry(events, didITurnIntoASquirrel) {
  journal.push({
    events: events,
    squirrel: didITurnIntoASquirrel
  });
}
```





E então, todas as noites às dez ou as vezes na manhã seguinte após descer do topo de sua estante de livros, ele faz o registro do 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 ele tem dados suficientes, ele pretende calcular a correlação entre sua transformação em esquilo e cada um dos eventos do dia e espera aprender algo útil a partir dessas correlações.

A *correlação* é uma medida de dependência entre variáveis ("variáveis" no sentido estatístico e não no sentido do JavaScript). Ela é geralmente expressa em um coeficiente que varia de -1 a 1. Zero correlação significa que as variáveis não são relacionadas, enquanto que a correlação de um indica que as variáveis são perfeitamente relacionadas (se você conhece uma, você também conhece a outra). A correlação negativa de um também indica que as variáveis são perfeitamente relacionadas, mas são opostas (quando uma é verdadeira, a outra é falsa).

Para variáveis binárias (booleanos), o coeficiente *phi* (ϕ) fornece uma boa forma de medir a correlação e é relativamente fácil de ser calculado. Para calcular ϕ , precisamos de uma tabela *n* que contém o número de vezes que as diversas combinações das duas variáveis foram observadas. Por exemplo, podemos considerar o evento de "comer pizza" e colocá-lo nessa tabela da seguinte maneira:

 No pizza, no squirrel 76	 Pizza, no squirrel 9
 No pizza, squirrel 4	 Pizza, squirrel 1

ϕ pode ser calculado usando a seguinte fórmula, onde n se refere à tabela:

```


$$\phi = (n_{11}n_{00} - n_{10}n_{01}) / \sqrt{n_{1\bullet} \cdot n_{0\bullet} \cdot n_{\bullet 1} \cdot n_{\bullet 0}}$$

[TODO: Adicionar formatação correta da fórmula após converter em asciidoc]

```

A notação n_{01} indica o número de ocorrências nas quais a primeira variável (transformar-se em esquilo) é falsa (0) e a segunda variável (pizza) é verdadeira (1). Nesse exemplo, n_{01} é igual a 9.

O valor $n_{1\bullet}$ se refere à soma de todas as medidas nas quais a primeira variável é verdadeira, que no caso do exemplo da tabela é 5. Da mesma forma, $n_{\bullet 0}$ se refere à soma de todas as medidas nas quais a segunda variável é falsa.

Portanto, para a tabela de pizza, a parte de cima da linha (o dividendo) seria $1 \times 76 - 4 \times 9 = 40$, e a parte de baixo (o divisor) seria a raiz quadrada de $5 \times 85 \times 10 \times 80$, ou $\sqrt{340000}$. Esse cálculo resulta em $\phi \approx 0.069$, o que é um valor bem pequeno. Comer pizza parece não ter influência nas transformações.

Calculando a correlação

No JavaScript, podemos representar uma tabela dois por dois usando um *array* com quatro elementos (`[76, 9, 4, 1]`). Podemos também usar outras formas de representações, como por exemplo um *array* contendo dois *arrays* com dois elementos cada (`[[76, 9], [4, 1]]`), ou até mesmo um objeto com propriedades nomeadas de `"11"` e `"01"` . Entretanto, a maneira mais simples e que faz com que seja mais fácil acessar os dados é utilizando um *array* com quatro elementos. Nós iremos interpretar os índices do *array* como elementos binários de dois bits, onde o dígito a esquerda (mais significativo) se refere à variável do esquilo, e o dígito a direita (menos significativo) se refere à variável do evento. Por exemplo, o número binário `10` se refere ao caso no qual Jacques se tornou um esquilo, mas o evento não ocorreu (por exemplo "pizza"). Isso aconteceu quatro vezes, e já que o número binário `10` é equivalente ao número 2 na notação decimal, iremos armazenar esse valor no índice 2 do *array*.

Essa é a função que calcula o coeficiente ϕ de um *array* desse tipo:

```

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

```

Essa é simplesmente uma tradução direta da fórmula de ϕ para o JavaScript. `Math.sqrt` é a função que calcula a raiz quadrada, fornecida pelo objeto `Math` que é padrão do JavaScript. Temos que somar dois campos da tabela para encontrar valores como $n1$, pois a soma das linhas ou colunas não são armazenadas diretamente em nossa estrutura de dados.

Jacques manteve seu diário por três meses. O conjunto de dados resultante está disponível no ambiente de código desse capítulo, armazenado na variável `JOURNAL` e em um [arquivo](#) que pode ser baixado.

Para extrair uma tabela dois por dois de um evento específico desse diário, devemos usar um loop para percorrer todas as entradas e ir adicionando quantas vezes o evento ocorreu em relação às transformações de esquilo.

```

function hasEvent(event, entry) {
    return entry.events.indexOf(event) !== -1;
}

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

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

```

A função `hasEvent` testa se uma entrada contém ou não o evento em questão. Os *arrays* possuem um método `indexOf` que procura pelo valor informado no *array* (nesse exemplo o nome do evento), e retorna o índice onde ele foi encontrado ou -1 se não for. Portanto, se a chamada de `indexOf` não retornar -1, sabemos que o evento foi encontrado.

O corpo do loop presente na função `tableFor`, descobre qual caixa da tabela cada entrada do diário pertence, verificando se essa entrada contém o evento específico e se o evento ocorreu juntamente com um incidente de transformação em esquilo. O loop adiciona uma unidade no número contido no *array* que corresponde a essa caixa na tabela.

Agora temos as ferramentas necessárias para calcular correlações individuais. O único passo que falta é encontrar a correlação para cada tipo de evento que foi armazenado e verificar se algo se sobressai. Como podemos armazenar essas correlações assim que as calculamos?

Objetos como mapas

Uma maneira possível é armazenar todas as correlações em um *array*, usando objetos com as propriedades `name` (nome) e `value` (valor). Porém, isso faz com que o acesso às correlações de um evento seja bastante trabalhoso, pois você teria que percorrer por todo o *array* para achar o objeto com o `name` certo. Poderíamos encapsular esse processo de busca em uma função e, mesmo assim, iríamos escrever mais código e o computador iria trabalhar mais do que o necessário.

Uma maneira melhor seria usar as propriedades do objeto nomeadas de acordo com o tipo do evento. Podemos usar a notação de colchetes para acessar e ler as propriedades e, além disso, usar o operador `in` para testar se tal propriedade existe.

```
var map = {};  
function storePhi(event, phi) {  
  map[event] = phi;  
}  
  
storePhi("pizza", 0.069);  
storePhi("touched tree", -0.081);  
console.log("pizza" in map);  
// → true  
console.log(map["touched tree"]);  
// → -0.081
```

Um *map* é uma maneira de associar valores de um domínio (nesse caso nomes de eventos) com seus valores correspondentes em outro domínio (nesse caso coeficientes ϕ).

Existem alguns problemas que podem ser gerados usando objetos dessa forma, os quais serão discutidos no [capítulo 6](#). Por enquanto, não iremos nos preocupar com eles.

E se quiséssemos encontrar todos os eventos nos quais armazenamos um coeficiente? Diferentemente de um *array*, as propriedades não formam uma sequência previsível, impossibilitando o uso de um loop `for` normal. Entretanto, o JavaScript fornece uma construção de loop específica para percorrer as propriedades de um objeto. Esse loop é parecido com o loop `for` e se distingue pelo fato de utilizar a palavra `in`.

```
for (var event in map)  
  console.log("The correlation for '" + event +  
              "' is " + map[event]);  
// → The correlation for 'pizza' is 0.069  
// → The correlation for 'touched tree' is -0.081
```

A análise final

Para achar todos os tipos de eventos que estão presentes no conjunto de dados, nós simplesmente processamos cada entrada e percorremos por todos os eventos presentes usando um loop. Mantemos um objeto chamado `phis` que contém os coeficientes das correlações de todos os tipos de eventos que foram vistos até agora. A partir do momento em que encontramos um tipo que não está presente no objeto `phis`, calculamos o valor de sua correlação e então adicionamos ao objeto.

```
function gatherCorrelations(journal) {  
  var phis = {};  
  for (var entry = 0; entry < journal.length; entry++) {  
    var events = journal[entry].events;  
    for (var i = 0; i < events.length; i++) {  
      var event = events[i];  
      if (!(event in phis))  
        phis[event] = phi(tableFor(event, journal));  
    }  
  }  
  return phis;  
}  
  
var correlations = gatherCorrelations(JOURNAL);  
console.log(correlations.pizza);  
// → 0.068599434
```

Vamos ver o resultado.

```
for (var event in correlations)
  console.log(event + ": " + correlations[event]);
// → carrot: 0.0140970969
// → exercise: 0.0685994341
// → weekend: 0.1371988681
// → bread: -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

A grande maioria das correlações tendem a zero. Comer cenouras, pão ou pudim aparentemente não ativam a transformação em esquilo. Entretanto, elas parecem acontecer com mais frequência aos finais de semana. Vamos filtrar os resultados para mostrar apenas as correlações que são maiores do que 0.1 ou menores do que -0.1.

```
for (var event in correlations) {
  var correlation = correlations[event];
  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
```

A-ha! Existem dois fatores nos quais a correlação é claramente mais forte que a das outras. Comer amendoins tem um forte efeito positivo na chance de se transformar em um esquilo, enquanto que escovar os dentes tem um significativo efeito negativo.

Interessante. Vamos tentar uma coisa.

```
for (var i = 0; i < JOURNAL.length; i++) {
  var entry = JOURNAL[i];
  if (hasEvent("peanuts", entry) &&
      !hasEvent("brushed teeth", entry))
    entry.events.push("peanut teeth");
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

Está bem evidente! O fenômeno ocorre precisamente quando Jacques come amendoins e não escova os dentes. Se ele não fosse preguiçoso em relação à higiene bucal, ele não sequer teria reparado nesse problema que o aflige.

Sabendo disso, Jacques simplesmente para de comer amendoins e descobre que isso coloca um fim em suas transformações.

Tudo ficou bem com Jacques por um tempo. Entretanto, alguns anos depois, ele perdeu seu emprego e eventualmente foi forçado a trabalhar em um circo, onde suas performances como *O Incrível Homem-Esqilo* se baseavam em encher sua boca com pasta de amendoim antes de cada apresentação. Em um dia de sua pobre existência, Jacques não conseguiu se transformar de volta em sua forma humana e fugiu do circo, desapareceu pela floresta e nunca mais foi visto.

Estudo aprofundado de Arrays

Antes de finalizar esse capítulo, gostaria de introduzir alguns outros conceitos relacionados a objetos. Começaremos com alguns métodos normalmente úteis dos *arrays*.

Vimos no [início do capítulo](#) os métodos `push` e `pop`, que adicionam e removem elementos no final de um *array*. Os métodos correspondentes para adicionar e remover itens no início de um *array* são chamados `unshift` e `shift`.

```
var todoList = [];  
function rememberTo(task) {  
  todoList.push(task);  
}  
function whatIsNext() {  
  return todoList.shift();  
}  
function urgentlyRememberTo(task) {  
  todoList.unshift(task);  
}
```

O programa anterior gerencia uma lista de tarefas. Você pode adicionar tarefas no final da lista chamando `rememberTo("eat")` e, quando estiver preparado para realizar alguma tarefa, você chama `whatIsNext()` para acessar e remover o primeiro item da lista. A função `urgentlyRememberTo` também adiciona uma tarefa, porém, ao invés de adicionar ao final da lista, a adiciona no início.

O método `indexOf` tem um irmão chamado `lastIndexOf`, que começa a pesquisa de um dado elemento pelo final do *array* ao invés de começar pelo início.

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

Ambos `indexOf` e `lastIndexOf` recebem um segundo argumento opcional que indica onde iniciar a pesquisa.

Outro método fundamental é o `slice`, que recebe um índice de início e outro de parada, retornando um *array* que contém apenas os elementos presentes entre esses índices. O índice de início é inclusivo e o de parada 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 parada não é informado, o `slice` irá pegar todos os elementos após o índice de início. *Strings* também possuem o método `slice` com um comportamento similar.

O método `concat` pode ser usado para unir *arrays*, parecido com o que o operador `+` faz com as *strings*. O exemplo a seguir mostra ambos `concat` e `slice` em ação. Ele recebe um *array* e um índice como argumento, retornando um novo *array* que é uma cópia do *array* original, exceto pelo fato de que o elemento no índice informado foi 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"]
```

Strings e suas propriedades

Podemos ler propriedades como `length` e `toUpperCase` de *strings*. Porém, caso tente adicionar uma nova propriedade, ela não será adicionada.

```
var myString = "Fido";
myString.myProperty = "value";
console.log(myString.myProperty);
// → undefined
```

Valores do tipo *string*, *number* e *Boolean* não são objetos e, mesmo pelo fato da linguagem não reclamar quando tentamos adicionar novas propriedades neles, elas não são armazenadas. Esses valores são imutáveis e não podem ser alterados.

Mesmo assim, esses tipos possuem propriedades nativas. Toda *string* possui uma série de métodos. Provavelmente, alguns dos mais úteis são `slice` e `indexOf`, que são parecidos com os métodos de *array* que possuem o mesmo nome.

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

Uma diferença é que o `indexOf` das *strings* pode receber uma *string* contendo mais de um caractere, enquanto que o método correspondente no *array* procura apenas por um único elemento.

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

O método `trim` remove todos os espaços vazios (espaços, linhas, tabs e caracteres similares) do começo e do final de uma *string*.

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

Já vimos a propriedade `length` das *strings*. Para acessar caracteres individuais de uma *string*, podemos usar o método `charAt` ou simplesmente ler suas propriedades numéricas, da mesma forma que você faria em um *array*.

```
var string = "abc";
console.log(string.length);
// → 3
console.log(string.charAt(0));
// → a
console.log(string[1]);
// → b
```

O Objeto *Arguments*

Sempre que uma função é invocada, uma variável especial chamada `arguments` é adicionada ao ambiente no qual o corpo da função executa. Essa variável se refere a um objeto que contém todos os argumentos passados à função. Lembre-se de que no JavaScript você pode passar mais (ou menos) argumentos para uma função, independentemente do número de parâmetros que foi declarado.

```
function noArguments() {}
noArguments(1, 2, 3); // This is okay
function threeArguments(a, b, c) {}
threeArguments(); // And so is this
```

O objeto `arguments` possui a propriedade `length` que nos informa o número de argumentos que realmente foi passado à função. Além disso, contém uma propriedade para cada argumento, chamadas 0, 1, 2, etc.

Se isso soa muito parecido como um *array* para você, você está certo. Esse objeto é muito parecido com um *array*. Porém, ele não possui nenhum dos métodos de *array* (como `slice` ou `indexOf`), fazendo com que seja um pouco mais difícil de se usar do que um *array* de verdade.

```
function argumentCounter() {
  console.log("You gave me", arguments.length, "arguments.");
}
argumentCounter("Straw man", "Tautology", "Ad hominem");
// → You gave me 3 arguments.
```

Algumas funções podem receber qualquer número de argumentos, como no caso de `console.log`. Essas funções normalmente percorrem por todos os valores em seu objeto `arguments` e podem ser usadas para criar interfaces extremamente agradáveis. Por exemplo, lembre-se de como criamos as entradas no diário do Jacques.

```
addEntry(["work", "touched tree", "pizza", "running",
  "television"], false);
```

Devido ao fato de que essa função irá ser executada muitas vezes, poderíamos criar uma alternativa mais simples.

```
function addEntry(squirrel) {
  var entry = {events: [], squirrel: squirrel};
  for (var i = 1; i < arguments.length; i++)
    entry.events.push(arguments[i]);
  journal.push(entry);
}
addEntry(true, "work", "touched tree", "pizza",
  "running", "television");
```

Essa versão lê o primeiro argumento (`squirrel`) da forma normal e depois percorre o resto dos argumentos (o loop pula o primeiro argumento, iniciando no índice 1) juntando-os em um *array*.

O Objeto *Math*

Como vimos anteriormente, `Math` é uma caixa de ferramentas com funções relacionadas a números, tais como `Math.max` (máximo), `Math.min` (mínimo) e `Math.sqrt` (raiz quadrada).

O objeto `Math` é usado como um *container* para agrupar uma série de funcionalidades relacionadas. Existe apenas um único objeto `Math` e, na maioria das vezes, ele não é útil quando usado como um valor. Mais precisamente, ele fornece um *namespace* (espaço nominal) para que todas essas funções e valores não precisem ser declaradas como variáveis globais.

Possuir muitas variáveis globais "polui" o *namespace*. Quanto mais nomes são usados, mais prováveis são as chances de acidentalmente sobrescrever o valor de uma variável. Por exemplo, é provável que você queira chamar algo de `max` em um de seus programas. Sabendo que no JavaScript a função nativa `max` está contida de forma segura dentro do objeto `Math`, não precisamos nos preocupar em sobrescrevê-la.

Muitas linguagens irão parar você ou, ao menos, avisá-lo quando tentar definir uma variável com um nome que já está sendo usado. Como o JavaScript não faz isso, tenha cuidado.

De volta ao objeto `Math`, caso precise realizar cálculos trigonométricos, `Math` pode ajudá-lo. Ele contém `cos` (cosseno), `sin` (seno) e `tan` (tangente), tanto quanto suas funções inversas `acos`, `asin` e `atan` respectivamente. O número π (pi), ou pelo menos a aproximação que é possível ser representada através de um número no JavaScript, está disponível como `Math.PI`. (Existe uma tradição antiga na programação de escrever os nomes de valores constantes em caixa alta).

```
function randomPointOnCircle(radius) {  
  var 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 senos e cossenos não são muito familiares para você, não se preocupe. Quando eles forem usados no [Capítulo 13](#) desse livro, eu lhe explicarei.

O exemplo anterior usa `Math.random`. Essa é uma função que retorna um número "pseudo-aleatório" entre zero (incluído) e um (excluído) 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 os computadores sejam deterministas (sempre reagem da mesma maneira quando são usados os mesmos dados de entrada), é possível fazer com que eles produzam números que pareçam ser aleatórios. Para fazer isso, a máquina mantém um número (ou uma quantidade deles) armazenado em seu estado interno. Assim, toda vez que um número aleatório é requisitado, ela executa alguns cálculos complicados e deterministas usando esse estado interno e, então, retorna parte do resultado desses cálculos. A máquina também utiliza esses resultados para mudar o seu estado interno, fazendo com que o próximo número "aleatório" produzido seja diferente.

Se ao invés de um número fracionário, quisermos um número aleatório inteiro, podemos usar `Math.floor` (que arredonda o número para o menor valor inteiro mais próximo) no resultado de `Math.random`.

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

Multiplicar o número aleatório por dez resulta em um número que seja maior ou igual a zero e menor do que dez. Devido ao fato de que `Math.floor` arredonda o valor para baixo, essa expressão irá produzir, com chances iguais, qualquer número de zero a nove.

Também existem as funções `Math.ceil` (para arredondar o valor para o maior número inteiro mais próximo) e `Math.round` (para arredondar o valor para o número inteiro mais próximo).

O objeto global

O escopo global, que é o espaço no qual as variáveis globais residem, também pode ser abordado como um objeto no JavaScript. Cada variável global está presente como uma propriedade desse objeto. Nos navegadores, o objeto do escopo global é armazenado na variável `window`.

```
var myVar = 10;  
console.log("myVar" in window);
```



```
// → true
console.log(window.myVar);
// → 10
```

Resumo

Objetos e *arrays* (que são tipos específicos de objetos) fornecem maneiras de agrupar um conjunto de valores em um único valor. Conceitualmente, ao invés de tentar carregar e manter todas as coisas individualmente em nossos braços, eles nos permitem colocar e carregar todas as coisas relacionadas dentro de uma bolsa.

Com exceção de `null` e `undefined`, a maioria dos valores no JavaScript possuem propriedades e são acessados usando `value.propName` OU `value["propName"]`. Objetos tendem a usar nomes para suas propriedades e armazenam mais o menos uma quantidade fixa delas. Por outro lado, os *Arrays* normalmente contêm quantidades variáveis de valores conceitualmente iguais e usam números (iniciando do zero) como os nomes de suas propriedades.

Existem algumas propriedades com nomes específicos nos *arrays*, como `length` e uma série de métodos. Métodos são funções que são armazenadas em propriedades e, normalmente, atuam no valor nas quais elas são propriedade.

Objetos podem também ser usados como mapas, associando valores com seus nomes. O operador `in` pode ser usado para verificar se um objeto contém a propriedade com o nome informado. A mesma palavra-chave pode ser usada em um loop `for` (`for (var name in object)`) para percorrer todas as propriedades do objeto.

Exercícios

A soma de um intervalo

A [introdução](#) desse livro mencionou a seguinte maneira como uma boa alternativa para somar um intervalo de números:

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

Escreva uma função chamada `range` que recebe dois argumentos, `start` (início) e `end` (fim), e retorna um *array* contendo todos os números a partir do valor `start` até o valor `end` (incluindo-o).

Em seguida, escreva a função `sum` que recebe um *array* de números como argumento e retorna a soma desses números. Execute o programa anterior e veja se o resultado retornado é de fato 55.

Como exercício bônus, modifique a sua função `range` para aceitar um terceiro argumento opcional que indica o tamanho do "incremento" usado para construir o *array*. Se nenhum valor for atribuído ao tamanho do incremento, o *array* de elementos será percorrido em incrementos de um, correspondendo ao comportamento original. A chamada à função `range(1, 10, 2)` deve retornar `[1, 3, 5, 7, 9]`. Certifique-se de que funcione também com valores negativos, fazendo com que `range(5, 2, -1)` produza `[5, 4, 3, 2]`.

```
// Your code here.

console.log(range(1, 10));
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
console.log(sum(range(1, 10)));
// → 55
```

Dicas

A maneira mais fácil de construir um *array* é primeiramente inicializar uma variável usando `[]` (um novo *array* vazio) e, em seguida, chamar várias vezes o seu método `push` para adicionar os valores. Não se esqueça de retornar o *array* no final da função.

Devido ao fato de que o limite final é inclusivo, ao invés de usar um simples operador `<`, você deverá usar o operador `<=` para checar o final do seu loop.

Para verificar se o argumento opcional de incremento foi fornecido, você pode verificar o `arguments.length` ou comparar o valor do argumento com `undefined`. Caso não tenha sido informado, apenas configure o seu valor padrão (1) no início da função.

Fazer com que `range` entenda incrementos negativos é provavelmente mais fácil de ser feito escrevendo dois loops distintos, um para contar valores crescentes e outro para valores decrescentes. Isso se dá pelo fato de que, quando estamos contando valores decrescentes, o operador que compara e verifica se o loop terminou precisa ser `>=` ao invés de `<=`.

Pode ser útil usar um valor de incremento diferente do valor padrão (por exemplo -1) quando o valor final do intervalo for menor do que o valor de início. Dessa forma, ao invés de ficar preso em um loop infinito, `range(5, 2)` retorna algo relevante.

Invertendo um *array*

Os *arrays* possuem o método `reverse`, que modifica o *array* invertendo a ordem em que os elementos aparecem. Para esse exercício, escreva duas funções: `reverseArray` e `reverseArrayInPlace`. A primeira (`reverseArray`) recebe um *array* como argumento e produz um *novo array* que tem os mesmos elementos com ordem inversa. A segunda (`reverseArrayInPlace`) funciona da mesma forma que o método `reverse`, só que nesse caso, invertendo os elementos do próprio *array* que foi fornecido como argumento. Ambas as funções não devem usar o método padrão `reverse`.

Levando em consideração as notas sobre efeitos colaterais e funções puras do [capítulo anterior](#), qual versão você espera que seja útil em mais situações? Qual delas é mais eficiente?

```
// Your code here.

console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
var arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

Dicas

Existem duas maneiras óbvias de implementar `reverseArray`. A primeira é simplesmente iterar o *array* fornecido do início ao fim e usar o método `unshift` para inserir cada elemento no início do novo *array*. A segunda é iterar o *array* fornecido começando pelo fim e terminando no início, usando o método `push`. Iterar um *array* de trás para frente faz com que seja necessário usar uma notação `for` um pouco estranha (`var i = array.length - 1; i >= 0; i--`).

Inverter o *array* em questão (`reverseArrayInPlace`) é mais difícil. Você deve ter cuidado para não sobrescrever elementos que você precisará posteriormente. Usar `reverseArray` ou até mesmo copiar o *array* inteiro (`array.slice(0)`) é uma boa forma de se copiar um *array*) funciona, mas é considerado trapaça.

O truque é *inverter* o primeiro e o último elemento, depois o segundo e o penúltimo e assim por diante. Você pode fazer isso percorrendo até a metade do valor de `length` do *array* (use `Math.floor` para arredondar o valor para baixo, pois você não precisa lidar com o elemento do meio de um *array* com tamanho ímpar) e substituir o elemento na posição `i` com o elemento na posição `array.length - 1 - i`. Você pode usar uma variável local para armazenar

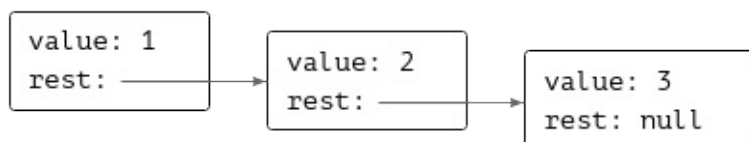
temporariamente um dos elementos, sobrescrever o seu valor com o valor do elemento espelhado (elemento que deseja substituir) e, por fim, colocar o valor da variável local no lugar onde o elemento espelhado estava originalmente.

A lista

Objetos tratados como agrupamentos genéricos de valores, podem ser usados para construir diversos tipos de estrutura de dados. Uma estrutura de dados comum é a *lista* (não se confunda com o *array*). A lista é um conjunto de objetos, sendo que o primeiro objeto contém uma referência para o segundo, o segundo para o terceiro, e assim por diante.

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

O resultado é uma cadeia de objetos conectados, como mostrado abaixo:



Uma das vantagens das listas é que elas podem compartilhar partes de sua estrutura. Por exemplo, se eu criasse dois novos valores `{value: 0, rest: list}` e `{value: -1, rest: list}` (sendo que `list` é uma referência à variável definida anteriormente), ambas serão listas independentes que compartilham a mesma estrutura que foi usada para criar os três últimos elementos. Além disso, a lista original ainda é uma lista válida com três elementos.

Escreva a função `arrayToList` que constrói uma estrutura de dados similar à estrutura anterior quando fornecido `[1, 2, 3]` como argumento e, escreva também, a função `listToArray` que produz um *array* a partir de uma lista. Além disso, implemente uma função auxiliar `prepend` que receberá um elemento e uma lista e será responsável por criar uma nova lista com esse novo elemento adicionado ao início da lista original e, por fim, crie a função `nth` que recebe uma lista e um número como argumentos e retorna o elemento que está na posição informada pelo número ou `undefined` caso não exista elemento em tal posição.

Caso não tenha feito ainda, implemente a versão recursiva da função `nth`.

```
// Your code here.

console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

Dicas

Construir uma lista é mais fácil de ser feito de trás para frente. Portanto, `arrayToList` poderia percorrer o `array` de trás para frente (veja o exercício anterior) e, para cada elemento, adicionar um objeto à lista. Você pode usar uma variável local para armazenar a parte da lista que foi criada e usar um padrão parecido com `list = {value: X, rest: list}` para adicionar um elemento.

Para percorrer uma lista (no caso de `listToArray` e `nth`), o seguinte loop `for` pode ser usado:

```
for (var node = list; node; node = node.rest) {}
```

Você consegue ver como isso funcionaria? A cada iteração do loop, `node` aponta para a próxima sublista e, por isso, o corpo da função pode acessar a propriedade `value` para pegar o elemento atual. Ao final de cada iteração, `node` é atualizado apontando para a próxima sublista. Quando seu valor é `null`, nós chegamos ao final da lista e o loop é finalizado.

A versão recursiva de `nth` irá, similarmente, olhar para uma parte ainda menor do *tail* (final) da lista e, ao mesmo tempo, fazer a contagem regressiva do índice até que chegue ao zero, significando que é o ponto no qual ela pode retornar a propriedade `value` do nó que está sendo verificado. Para acessar o elemento na posição zero de uma lista, você pode simplesmente acessar a propriedade `value` do seu nó *head* (inicial). Para acessar o elemento `N + 1`, você pega o `n`-ésimo elemento da lista que está contido na propriedade `rest` da lista em questão.

Comparação "profunda"

O operador `==` compara objetos pelas suas identidades. Entretanto, algumas vezes, você pode preferir comparar os valores das suas propriedades de fato.

Escreva a função `deepEqual` que recebe dois valores e retorna `true` apenas se os valores forem iguais ou se forem objetos que possuem propriedades e valores iguais quando comparados usando uma chamada recursiva de `deepEqual`.

Para saber se a comparação entre duas coisas deve ser feita pela identidade (use o operador `===` para isso) ou pela verificação de suas propriedades, você pode usar o operador `typeof`. Se ele produzir `"object"` para ambos os valores, você deverá fazer uma comparação "profunda". Entretanto, você deve levar em consideração uma exceção: devido a um acidente histórico, `typeof null` também produz `"object"`.

```
// Your code here.

var obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));
// → true
```

Dicas

O teste para saber se está lidando com um objeto real deverá ser parecido com `typeof x === "object" && x !== null`. Tome cuidado para comparar as propriedades apenas quando ambos argumentos forem objetos. Em todos os outros casos, você pode simplesmente retornar imediatamente o resultado da aplicação do operador `===`.

Use um loop `for/in` para percorrer todas as propriedades. Você precisa verificar se ambos os objetos possuem o mesmo conjunto de propriedades e se essas propriedades têm valores idênticos. O primeiro teste pode ser feito contando a quantidade de propriedades em cada objeto e retornar `false` se forem diferentes. Caso seja o mesmo, então, percorra todas as propriedades de um objeto e, para cada uma delas, verifique se o outro objeto também a possui. Os valores das propriedades são comparados usando uma chamada recursiva para `deepEqual`.

Para retornar o valor correto da função, é mais fácil retornar imediatamente `false` quando qualquer diferença for encontrada e retornar `true` apenas ao final da função.

Funções de ordem superior

"Tzu-li e Tzu-ssu estavam se gabando do tamanho dos seus últimos programas. 'Duzentas mil linhas sem contar os comentários!', disse Tzu-li. Tzu-ssu respondeu: 'Pssh, o meu já tem quase um milhão de linhas'. Mestre Yuan-Ma disse: 'Meu melhor programa tem quinhentas linhas'. Ouvindo isso, Tzu-li e Tzu-ssu ficaram esclarecidos."

— Master Yuan-Ma, The Book of Programming

"Existem duas maneiras de construir o design de um software: uma maneira é deixá-lo tão simples de tal forma em que obviamente não há deficiências, e a outra é torná-lo tão complicado que não haverá deficiências óbvias."

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

Um programa grande é um programa custoso, e não necessariamente devido ao tempo que leva para construir. Tamanho quase sempre envolve uma complexidade e complexidade confunde os programadores. Programadores confusos tendem a criar erros (bugs) no programa. Um programa grande tem a possibilidade de esconder bugs que são difíceis de serem encontrados.

Vamos rapidamente abordar dois exemplos que foram citados na introdução. O primeiro contém um total de 6 linhas.

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

O segundo necessita de duas funções externas e é escrito em apenas uma linha.

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

Qual é mais propenso a erros?

Se medirmos o tamanho das definições de `sum` e `range`, o segundo programa também será grande - até maior do que o primeiro. Mesmo assim, eu diria que ele é o mais provável a estar correto.

A razão dele possivelmente ser o mais correto, é que a solução é expressa em um vocabulário que corresponde ao problema que está sendo resolvido. Somar um intervalo de números não se trata de laços de repetição e contadores. Trata-se de intervalos e somas.

As definições desse vocabulário (as funções `sum` e `range`) ainda assim terão que lidar com laços de repetição, contadores e outros detalhes secundários. No entanto, devido ao fato de representarem conceitos mais simples, elas acabam sendo mais fáceis de se entender.

Abstração

No contexto da programação esse tipo de vocabulário é geralmente expressado pelo termo abstrações. Abstrações escondem detalhes e nos dá a habilidade de falar sobre problemas em alto nível (mais abstrato).

Isto é uma analogia que compara duas receitas de sopa de ervilha:

"Coloque 1 copo de ervilha por pessoa num recipiente. Adicione água até as ervilhas ficarem cobertas. Deixe as ervilhas na água por no mínimo 12 horas. Tire as ervilhas da água e coloque-as numa panela. Adicione 4 copos de água por pessoa. Cubra a panela e deixe-as cozinhando por duas horas. Pegue meia cebola por pessoa, corte em pedaços, adicione às ervilhas. Pegue um talo de aipo por pessoa, corte em pedaços e adicione às ervilhas. Pegue uma cenoura por pessoa, corte em pedaços! Adicione às ervilhas. Cozinhe por 10 minutos".

E a segunda receita:

"Para uma pessoa: 1 copo de ervilha, meia cebola, um talo de aipo e uma cenoura." Embeba as ervilhas por 12 horas, ferva por 2 horas em 4 copos de água (por pessoa). Pique e adicione os vegetais. Deixe cozinhar por mais 10 minutos".

A segunda é bem menor e mais fácil de interpretar. Mas ela necessita de um conhecimento maior sobre algumas palavras relacionadas à cozinhar como: embeber, ferva, pique e vegetais.

Quando programamos não podemos contar com todas as palavras do dicionário para expressar o que precisamos. Assim cairemos no primeiro padrão de receita - onde damos cada comando que o computador tem que realizar, um por um, ocultando os conceitos de alto níveis que se expressam.

Perceber quando um conceito implora para ser abstraído em uma nova palavra é um costume que tem de virar algo natural quando programamos.

Abstraindo Array transversal

Funções, como vimos anteriormente, são boas maneiras para se criar abstrações. Mas algumas vezes elas ficam aquém.

No [capítulo anterior](#), esse tipo de `loop` apareceu várias vezes:

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
  var current = array[i];
  console.log(current);
}
```

O que ele diz é: "Para cada elemento do array, registre no `console`". Mas utiliza um jeito redundante que envolve uma variável contadora, uma checagem do tamanho do `array` e a declaração de uma variável extra para pegar o elemento atual. Deixando de lado a monstruosidade do código, ele também nos dá espaço para possíveis erros: Podemos reusar a variável `i`, digitar errado `length` como `lenght`, confundir as variáveis `i` e `current` e por aí vai.

Então vamos tentar abstrair isso em uma nova função. Consegue pensar em alguma forma?

É trivial escrever uma função que passa sobre um `array` e chama `console.log` para cada elemento:

```
function logEach(array) {
  for (var i = 0; i < array.length; i++)
    console.log(array[i]);
}
```

Mas e se quisermos fazer algo diferente do que apenas registrar os elementos? Uma vez que "fazer alguma coisa" pode ser representado com uma função e as funções são apenas valores, podemos passar nossas ações como um valor para a função.

```
function forEach(array, action) {
```

```
for (var i = 0; i < array.length; i++)  
  action(array[i]);  
}  
  
forEach(["Wampeter", "Foma", "Granfalloon"], console.log);  
// → Wampeter  
// → Foma  
// → Granfalloon
```

Normalmente você não irá passar uma função predefinida para o `forEach`, mas ela será criada localmente dentro da função.

```
var numbers = [1, 2, 3, 4, 5], sum = 0;  
forEach(numbers, function(number) {  
  sum += number;  
});  
console.log(sum);  
// → 15
```

Isso parece muito com um `loop` clássico, com o seu corpo escrito como um bloco logo abaixo. No entanto o corpo está dentro do valor da função, bem como esta dentro dos parênteses da chamada de `forEach`. É por isso que precisamos fechar com chave e parêntese.

Nesse padrão, podemos simplificar o nome da variável (`number`) pelo elemento atual, ao invés de simplesmente ter que buscá-lo fora do `array` manualmente.

De fato, não precisamos definir um método `forEach`. Ele está disponível como um método padrão em `arrays`.

Quando um `array` é fornecido para o método agir sobre ele, o `forEach` espera apenas um argumento obrigatório: a função a ser executada para cada elemento.

Para ilustrar o quão útil isso é, vamos lembrar da função que vimos no [capítulo anterior](#), onde continha dois `arrays` transversais.

```
function gatherCorrelations(journal) {  
  var phis = {};  
  for (var entry = 0; entry < journal.length; entry++) {  
    var events = journal[entry].events;  
    for (var i = 0; i < events.length; i++) {  
      var event = events[i];  
      if (!(event in phis))  
        phis[event] = phi(tableFor(event, journal));  
    }  
  }  
  return phis;  
}
```

Trabalhando com `forEach` faz parecer levemente menor e bem menos confuso.

```
function gatherCorrelations(journal) {  
  var phis = {};  
  journal.forEach(function(entry) {  
    entry.events.forEach(function(event) {  
      if (!(event in phis))  
        phis[event] = phi(tableFor(event, journal));  
    });  
  });  
  return phis;  
}
```


Funções de ordem superior

Funções que operam em outras funções, seja ela apenas devolvendo argumentos, são chamadas de funções de ordem superior. Se você concorda com o fato de que as funções são valores normais, não há nada de notável sobre o fato de sua existência. O termo vem da matemática onde a distinção entre funções e outros valores é levado mais a sério.

Funções de ordem superior nos permitem abstrair as ações. Elas podem ser de diversas formas. Por exemplo, você pode ter funções que criam novas funções.

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

E você pode ter funções que alteram outras funções.

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
noisy(Boolean)(0);
// → calling with 0
// → called with 0 - got false
```

Você pode até escrever funções que fornecem novos tipos de fluxos de controles.

```
function unless(test, then) {
  if (!test) then();
}
function repeat(times, body) {
  for (var i = 0; i < times; i++) body(i);
}

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

As regras de escopo léxico que discutimos no [capítulo 3](#) trabalham a nosso favor quando usamos funções dessa maneira. No exemplo acima, a variável `n` é um parâmetro da função externa. Mas como as funções internas estão dentro do ambiente externo, podemos usar a variável `n`. Os "corpos" de tais funções internas podem acessar as variáveis que estão em torno delas. Eles podem desempenhar um papel similar aos blocos `{}` usados em `loops` e expressões condicionais. Uma diferença importante é que variáveis declaradas dentro das funções internas não podem ser acessadas fora da função. Isso geralmente é algo bom.

Passando argumentos

A função `noisy` declarada abaixo, envolve seu argumento em outra função, isso gera uma grave deficiência.

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
```

Se `f` receber mais de um parâmetro, ele recebe apenas o primeiro. Poderíamos acrescentar vários argumentos para a função interna (`arg1`, `arg2`, e assim por diante) e passar elas para `f`, mas mesmo assim isso não deixaria explícito quantos seriam suficientes. Essa solução limita algumas informações de `f` como por exemplo `arguments.length`. Sempre passaremos a mesma quantidade de argumentos, mas nunca saberemos a quantidade exata de argumentos que foi passada.

Para esse tipo de situação, funções em JavaScript possuem um método chamado `apply`. Você passa um `array` (ou um `array` como objeto) como argumento, e ele irá chamar a função com estes argumentos.

```
function transparentWrapping(f) {
  return function() {
    return f.apply(null, arguments);
  };
}
```

Essa função é inútil, mas nos mostra o padrão que estamos interessados, a função passa todos os argumentos dados para `f` e retorna, apenas estes argumentos, para `f`. Ela faz isso passando seus próprios argumentos para o objeto `apply`. O primeiro argumento do `apply`, estamos passando `null`, isto pode ser usado para simular uma chamada de método. Iremos voltar a ver isto novamente no [próximo capítulo](#).

JSON

Funções de ordem superior que aplicam uma função para os elementos de um `array` são bastante usadas em JavaScript. O método `forEach` é uma função mais primitiva. Existe outras variantes disponíveis como métodos em `arrays`. Para acostarmos com eles vamos brincar com um outro conjunto de dados.

Há alguns anos, alguém juntou um monte de arquivos e montou um livro sobre a história do nome da minha família (Haverbeke que significa Oatbrook). Eu abri na esperança de encontrar cavaleiros, piratas, e alquimistas... mas o livro acaba por ser principalmente de agricultores de Flamengos. Para minha diversão extraí uma informação sobre os meus antepassados e coloquei em um formato legível por um computador.

O arquivo que eu criei se parece mais ou menos assim:

```
[
  {"name": "Emma de Milliano", "sex": "f",
   "born": 1876, "died": 1956,
   "father": "Petrus de Milliano",
   "mother": "Sophia van Damme"},
  {"name": "Carolus Haverbeke", "sex": "m",
   "born": 1832, "died": 1905,
   "father": "Carel Haverbeke",
   "mother": "Maria van Brussel"},
  ... and so on
]
```

Este formato é chamado de `JSON` (pronuncia-se "Jason") que significa *JavaScript Object Notation*. `JSON` é amplamente utilizado como armazenamento de dados e formato de comunicação na *Web*.

`JSON` se escreve semelhantemente como `arrays` e objetos em JavaScript, mas com algumas restrições. Todos os nomes das propriedades devem ficar entre aspas duplas e apenas expressões de dados simples são permitidos, não é permitido chamadas de funções, variáveis ou qualquer coisa que envolva cálculo real. Comentários não são permitidos em `JSON`.

JavaScript nos fornece duas funções `JSON.stringify` e `JSON.parse`, que convertem dados para este formato. O primeiro recebe um valor em JavaScript e retorna uma string codificada em `JSON`. A segunda obtém uma `string` e converte-a para um valor que ele codifica.

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name":"X","born":1980}
console.log(JSON.parse(string).born);
// → 1980
```

O variável `ANCESTRY_FILE` está disponível na `sandbox` deste capítulo para download no site, onde está o conteúdo do meu arquivo `JSON` como uma `string`. Vamos decodificá-lo e ver quantas pessoas contém.

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

Filtrando um array

Para encontrar as pessoas no conjunto de dados dos ancestrais que eram jovens em 1924, a seguinte função pode ser útil. Ele filtra os elementos em uma matriz que não passa pelo teste.

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
}));
// → [{name: "Philibert Haverbeke", ...}, ...]
```

Este utiliza um argumento chamado de `test`, com um valor de função, para preencher uma lacuna na computação. A função `test` é chamada para cada elemento, e o seu valor de retorno determina se um elemento é incluído no `array` retornado.

Três pessoas no arquivo estavam vivas e jovens em 1924: meu avô, minha avó e minha tia-avó.

Observe como a função `filter`, em vez de excluir os elementos do `array`, constrói um novo com apenas os elementos que passaram no teste. Esta função é primitiva. Não modifica o `array` que foi dado.

Assim como `forEach`, `filter` é um método padrão de `arrays`. O exemplo define uma função só para mostrar o que ela faz internamente. A partir de agora vamos usá-lo assim:

```
console.log(ancestry.filter(function(person) {
```

```
    return person.father == "Carel Haverbeke";  
  }));  
  // → [{name: "Carolus Haverbeke", ...}]
```

Transformando com map

Digamos que temos um `array` de objetos que representam pessoas, produzido através do `array` de ancestrais de alguma forma. Mas queremos um `array` de nomes o que é mais fácil para ler.

O método `map` transforma um `array` aplicando uma função para todos os elementos e constrói um novo `array` a partir dos valores retornados. O novo `array` terá o mesmo tamanho do `array` enviado, mas seu conteúdo é mapeado para um novo formato através da função.

```
function map(array, transform) {  
  var mapped = [];  
  for (var i = 0; i < array.length; i++)  
    mapped.push(transform(array[i]));  
  return mapped;  
}  
  
var overNinety = ancestry.filter(function(person) {  
  return person.died - person.born > 90;  
});  
console.log(map(overNinety, function(person) {  
  return person.name;  
}));  
// → ["Clara Aernoudts", "Emile Haverbeke",  
//     "Maria Haverbeke"]
```

Curiosamente, as pessoas que viveram pelo menos 90 anos de idade são as mesmas três que vimos antes, as pessoas que eram jovens em 1920, passam a ser a geração mais recente no meu conjunto de dados. Eu acho que a medicina já percorreu um longo caminho.

Assim como `forEach` e `filter`, `map` também é um método padrão de `arrays`.

Resumindo com reduce

Outro padrão na computação em `arrays` é calcular todos elementos e transformá-los em apenas um. No nosso exemplo atual, a soma do nosso intervalo de números, é um exemplo disso. Outro exemplo seria encontrar uma pessoa com um ano de vida no conjunto de dados.

Uma operação de ordem superior que representa este padrão é chamada de *reduce* (diminui o tamanho do `array`). Você pode pensar nisso como dobrar a matriz, um elemento por vez. Quando somado os números, você inicia com o número zero e, para cada elemento, combina-o com a soma atual adicionando os dois.

Os parâmetros para a função `reduce` são, além do `array`, uma função para combinação e um valor inicial. Esta função é menos simples do que o `filter` e `map` por isso observe com muita atenção.

```
function reduce(array, combine, start) {  
  var current = start;  
  for (var i = 0; i < array.length; i++)  
    current = combine(current, array[i]);  
  return current;  
}  
  
console.log(reduce([1, 2, 3, 4], function(a, b) {  
  return a + b;  
}, 0));
```

```
// → 10
```

O `array` padrão do método `reduce` que corresponde a esta função tem uma maior comodidade. Se o seu `array` contém apenas um elemento, você não precisa enviar um valor inicial. O método irá pegar o primeiro elemento do `array` como valor inicial, começando a redução a partir do segundo.

Para usar o `reduce` e encontrar o meu mais antigo ancestral, podemos escrever algo parecido com isto:

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

Componibilidade

Considere como escreveríamos o exemplo anterior (encontrar a pessoa mais velha) sem funções de ordem superior. O código não ficaria tão ruim.

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; i++) {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

Existem mais variáveis, e o programa está com duas linhas a mais, mesmo assim continuou bem fácil de entender.

Funções de ordem superior são úteis quando você precisa compor funções. Como exemplo, vamos escrever um código que encontra a idade média para homens e mulheres no conjunto de dados.

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));
// → 61.67
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

(É um pouco bobo termos que definir `plus` como uma função, mas os operadores em JavaScript, diferentemente das funções, não são valores, então não podemos passar nenhum argumento.)

Ao invés de juntar toda a lógica em um `loop` gigante, ele está bem composto nos conceitos que interessamos como - determinando sexo, calculando a idade e a média dos números. Podemos aplicá-las uma de cada vez para obtermos o resultado que estamos procurando.

Escrever um código limpo é fabuloso. Infelizmente essa clareza tem um custo.

O Custo

No mundo dos códigos elegantes e lindos arco-íris, vive um monstro que estraga os prazeres chamado de ineficiência.

Um programa que processa um `array` é mais elegante expresso em uma sequência separada onde cada passo pode fazer algo com o `array` e produzir um novo `array`. Mas a construção de todos esses `arrays` intermediários é um pouco custoso.

Passar uma função para `forEach` e deixar que o método cuide da iteração para os nós é conveniente e fácil de ler. Mas chamadas de funções em JavaScript são custosas comparadas com os simples blocos de repetição.

E assim existem várias técnicas que ajudam a melhorar a clareza de um programa. Abstrações adiciona uma camada a mais entre as coisas cruas que o computador faz e o conceito que estamos trabalhando, sendo assim a máquina realiza mais trabalho. Esta não é uma lei de ferro, existem linguagens de programação que tem um suporte melhor para a construção de abstração sem adição de ineficiências, até mesmo em JavaScript, um programador experiente pode encontrar maneiras de escrever um código abstrato e rápido. Mas é um problema que é muito comum.

Existem várias técnicas que ajudam a esclarecer o código. Elas adicionam camadas entre as coisas cruas que o computador está fazendo com os conceitos que estamos trabalhando e faz com que a máquina trabalhe mais rápido. Isso não é uma lei inescapável -- existem linguagens de programação que possuem um melhor suporte para construir aplicações sem adicionar ineficiências e, ainda em JavaScript, um programador experiente pode encontrar jeitos de escrever códigos relativamente abstratos que ainda são rápidos, porém é um problema frequente.

Felizmente muitos computadores são extremamente rápidos. Se você estiver processando uma modesta coleção de dados ou fazendo alguma coisa que tem de acontecer apenas em uma escala de tempo humano (digamos, toda vez que o usuário clica em um botão), então não importa se você escreveu aquela solução maravilhosa que leva meio milissegundo ou uma super solução otimizada que leva um décimo de um milissegundo.

É útil saber quanto tempo mais ou menos leva um trecho de código para executar. Se vocês têm um `loop` dentro de um `loop` (diretamente, ou através de um `loop` externo chamando uma função que executa um `loop` interno), o código dentro do `loop` interno acaba rodando $N \times M$ vezes, onde N é o número de vezes que o `loop` de fora se repete e M é o número de vezes que o `loop` interno se repete dentro de cada interação do `loop` externo. Se esse `loop` interno tiver outro `loop` que realize P voltas, seu bloco rodará $M \times N \times P$ vezes e assim por diante. Isto pode adicionar muitas operações. Quando um programa é lento o problema muitas das vezes pode estar atribuída a apenas uma pequena parte do código que fica dentro de um `loop` interno.

O pai do pai do pai do pai

Meu avô, Philibert Haverbeke está incluído nos dados do arquivo. Começando com ele, eu posso traçar minha linhagem para descobrir qual é a pessoa mais velha no conjunto de dados, Pauwels van Haverbeke, é meu ancestral direto. E se ele for, gostaria de saber o quanto de DNA, teoricamente, que partilho com ele.

Para ser capaz de fazer uma busca pelo nome de um pai para um objeto real que representa uma pessoa, primeiramente precisamos construirmos um objeto que associa os nomes com as pessoas.

```
var byName = {};  
ancestry.forEach(function(person) {  
  byName[person.name] = person;  
});  
  
console.log(byName["Philibert Haverbeke"]);  
// → {name: "Philibert Haverbeke", ...}
```

Agora o problema não é totalmente simples como conseguir as propriedades do pai e ir contando quantos levam até chegar a Pauwels. Existem vários casos na árvore genealógica onde pessoas se casaram com seus primos de segundo grau (pequenos vilarejos têm essas coisas). Isso faz com que as ramificações da família se reencontrem em certos lugares, o que significa que eu compartilho mais de $1/2G$ do meu genes com essa pessoa, onde usaremos G como número de gerações entre Pauwels e mim. Esta fórmula vem a partir da ideia que de cada geração divide o conjunto de genes em dois.

Uma maneira razoável de pensar sobre este problema é olhar para ele como sendo um análogo de `reduce`, que condensa um `array` em um único valor, por valores que combinam várias vezes da esquerda para a direita. Neste caso nós também queremos condensar a nossa estrutura de dados para um único valor mas de uma forma que segue as linhas da família. O formato dos dados é a de uma árvore genealógica em vez de uma lista plana.

A maneira que nós queremos reduzir esta forma é calculando um valor para uma determinada pessoa, combinando com os valores de seus ancestrais. Isso pode ser feito de uma forma recursiva: se estamos interessados em uma pessoa A , temos que calcular os valores para os pais de A s, que por sua vez obriga-nos a calcular o valor para os avós de A s e assim por diante. A princípio isso iria exigir-nos a olhar para um número infinito de pessoas, já que o nosso conjunto de dados é finito, temos que parar em algum lugar. Vamos permitir um valor padrão para nossa função de redução, que será utilizado para pessoas que não estão em nossos dados. No nosso caso, esse valor é simplesmente zero, pressupondo de que as pessoas que não estão na lista não compartilham do mesmo DNA do ancestral que estamos olhando.

Dado uma pessoa, a função combina os valores a partir de dois pais de uma determinada pessoa, e o valor padrão, `reduceAncestors` condensa o valor a partir de uma árvore genealógica.

```
function reduceAncestors(person, f, defaultValue) {
  function valueFor(person) {
    if (person == null)
      return defaultValue;
    else
      return f(person, valueFor(byName[person.mother]),
               valueFor(byName[person.father]));
  }
  return valueFor(person);
}
```

A função interna (`valueFor`) lida com apenas uma pessoa. Através da magia da recursividade ela pode chamar a si mesma para lidar com o pai e com a mãe. Os resultados junto com o objeto da pessoa em si, são passados para `f` na qual devolve o valor real para essa pessoa.

Podemos então usar isso para calcular a quantidade de DNA que meu avô compartilhou com Pauwels van Haverbeke e depois dividir por quatro.

```
function sharedDNA(person, fromMother, fromFather) {
  if (person.name == "Pauwels van Haverbeke")
    return 1;
  else
    return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

A pessoa com o nome Pauwels van Haverbeke obviamente compartilhada 100 por cento de seu DNA com Pauwels van Haverbeke (não existem pessoas que compartilham o mesmo nome no conjunto de dados), então a função retorna 1 para ele. Todas as outras pessoas compartilham a média do montante que os seus pais compartilham.

Assim estatisticamente falando, eu compartilho cerca de 0,05 por cento do DNA de uma pessoa do século 16. Deve-se notar que este é só uma aproximação estatística e, não uma quantidade exata. Este é um número bastante pequeno mas dada a quantidade de material genético que carregamos (cerca de 3 bilhões de pares de bases), provavelmente ainda há algum aspecto na máquina biológica que se originou de Pauwels.

Nós também podemos calcular esse número sem depender de `reduceAncestors`. Mas separando a abordagem geral (condensação de uma árvore genealógica) a partir do caso específico (computação do DNA compartilhado) podemos melhorar a clareza do código permitindo reutilizar a parte abstrata do programa para outros casos. Por exemplo, o código a seguir encontra a porcentagem de antepassados conhecidos para uma determinada pessoa que viveu mais de 70 anos (por linhagem, para que as pessoas possam ser contadas várias vezes).

```
function countAncestors(person, test) {
  function combine(current, fromMother, fromFather) {
    var thisOneCounts = current != person && test(current);
    return fromMother + fromFather + (thisOneCounts ? 1 : 0);
  }
  return reduceAncestors(person, combine, 0);
}
function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person) {
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"]));
// → 0.129
```

Tais números não são levados muito a sério, uma vez que o nosso conjunto de dados contém uma coleção bastante arbitrária de pessoas. Mas o código ilustra o fato de que `reduceAncestors` dá-nos uma peça útil para trabalhar com o vocabulário da estrutura de dados de uma árvore genealógica.

Binding

O método `bind`, está presente em todas as funções, ele cria uma nova função que chama a função original mas com alguns argumentos já fixados.

O código a seguir mostra um exemplo de uso do `bind`. Ele define uma função `isInSet`, que nos diz se uma pessoa está em um determinado conjunto de `string`. Ao chamar `filter` a fim de selecionar os objetos pessoa cujos nomes estão em um conjunto específico. Nós podemos escrever uma expressão de função que faz a chamada para `isInSet` enviando nosso conjunto como primeiro argumento ou parcialmente aplicar a função `isInSet`.

```
var theSet = ["Carel Haverbeke", "Maria van Brussel",
             "Donald Duck"];
function isInSet(set, person) {
  return set.indexOf(person.name) > -1;
}

console.log(ancestry.filter(function(person) {
  return isInSet(theSet, person);
}));
// → [{name: "Maria van Brussel", ...},
//     {name: "Carel Haverbeke", ...}]
console.log(ancestry.filter(isInSet.bind(null, theSet)));
// → ... same result
```


A chamada usando `bind` retorna uma função que chama `isInSet` com `theset` sendo o primeiro argumento, seguido por todos os demais argumentos indicados pela função vinculada.

O primeiro argumento onde o exemplo passa `null`, é utilizado para as chamadas de método, semelhante ao primeiro argumento do `apply`. Eu vou descrever isso com mais detalhes no [próximo capítulo](#).

Sumário

A possibilidade de passar funções como argumento para outras funções não é apenas um artifício mas sim um aspecto muito útil em JavaScript. Ela nos permite escrever cálculos com intervalos como funções, e chamar estas funções para preencher estes intervalos, fornecendo os valores para função que descrevem os cálculos que faltam.

`Arrays` fornece uma grande quantidade de funções de ordem superior - `forEach` faz algo com cada elemento de um `array`, `filter` para construir um novo `array` com valores filtrados, `map` para construir um novo array onde cada elemento é colocado através de uma função e `reduce` para combinar todos os elementos de um `array` em um valor único.

Funções têm o método `apply` que pode ser usado para chamar um `array` especificando seus argumentos. Elas também possuem um método `bind` que é usado para criar uma versão parcial da função que foi aplicada.

Exercícios

Juntando

Use o método `reduce` juntamente com o método `concat` para juntar um `array` de `arrays` em um único `array` que tem todos os elementos de entrada do `array`.

```
var arrays = [[1, 2, 3], [4, 5], [6]];
// Your code here.
// → [1, 2, 3, 4, 5, 6]
```

Diferença de idade entre mãe e filho

Usando os dados de exemplo definidos neste capítulo, calcule a diferença de idade média entre mães e filhos (a idade da mãe quando a criança nasce). Você pode usar a função `average` definida anteriormente neste capítulo.

Note que nem todas as mães mencionadas no conjunto de dados estão presentes no `array`. O objeto `byName` facilita a busca por um objeto pessoa através do nome. Esse método pode ser útil aqui.

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

var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

// Your code here.

// → 31.2
```

Dica:

Como nem todos os elementos do `array` de ascendência produzem dados úteis (não podemos calcular a diferença de idade, a menos que saibamos a data de nascimento da mãe) teremos que aplicar de alguma maneira um filtro antes de chamarmos o `average`. Você pode fazer isso no primeiro passo, basta definir uma função `hasKnownMother` para a primeira filtragem. Alternativamente você pode começar a chamar o `map` e na função de mapeamento retornar a diferença de idade ou nulo se mãe for desconhecida. Em seguida você pode chamar o `filter` para remover os elementos nulos antes de passar o `array` para o método `average`.

O Histórico da expectativa de vida

Quando olhamos para todas as pessoas no nosso conjunto de dados que viveram mais de 90 anos, só a última geração dos dados que retornou. Vamos observar mais de perto esse fenômeno.

Calcule o resultado da idade média das pessoas no conjunto de dados definidos por século. Uma pessoa é atribuída a um século pegando o ano da sua morte, dividindo por 100 e arredondando para cima com `Math.ceil(person.died / 100)`.

```
function average(array) {  
  function plus(a, b) { return a + b; }  
  return array.reduce(plus) / array.length;  
}  
  
// Your code here.  
  
// → 16: 43.5  
//    17: 51.2  
//    18: 52.8  
//    19: 54.8  
//    20: 84.7  
//    21: 94
```

Para ganhar um ponto extra escreva uma função `groupBy` que abstrai a operação de agrupamento. Ele deve aceitar um `array` como argumento e uma função que calcula cada elemento do grupo de `array` e retorna um objeto que mapeia os nomes dos grupos de `arrays` e os membros do grupo.

Dica:

A essência desse exemplo encontra-se no agrupamento dos elementos em um conjunto por alguns aspectos - as divisões do `array` de ancestrais em pequenos `arrays` com os ancestrais de cada século.

Durante o processo de agrupamento, mantenha um objeto que associa os nomes dos séculos (números) com os `arrays` de objetos de pessoas ou idades. Já que não sabemos quais agrupamentos iremos encontrar, teremos que criá-los em tempo real. Depois de calcular o século para cada pessoa, vamos testar para saber se o século já existe. Se não existir adicione um `array` para ele. Em seguida adicione a pessoa (ou idade) no `array` de acordo com o século apropriado.

Finalmente um `loop for/in` pode ser usado para escrever a média de idades para cada século individualmente.

Todos e alguns

`Arrays` também vêm com os métodos padrões `every` (todos) e `some` (alguns). Ambos recebem uma função predicada que quando chamada com um `array` como argumento retorna `true` ou `false`. Assim como o operador `&&` retorna apenas `true` como valor quando as expressões de ambos os lados forem `true`; `every` retorna `true` quando a função predicada retorna `true` para cada elemento do `array`. Sendo assim, a função predicada `some` retorna quando algum elemento do `array` tiver um valor como `true`. Ele não processa mais elementos do que o necessário - por exemplo, se o predicado `some` encontrar o que precisa no primeiro elemento do `array` ele não percorrerá os outros elementos.

Escreva duas funções, que se comporte como esses métodos, `every` e `some`, exceto se eles receberem um `array` como seu primeiro argumento ao invés de um método.

```
// Your code here.  
  
console.log(every([NaN, NaN, NaN], isNaN));  
// → true  
console.log(every([NaN, NaN, 4], isNaN));  
// → false  
console.log(some([NaN, 3, 4], isNaN));  
// → true  
console.log(some([2, 3, 4], isNaN));  
// → false
```

Dica:

As funções podem seguir um padrão semelhante à definição de `forEach` que foi mostrado no início do capítulo, a única exceção é que eles devem retornar imediatamente (com o valor à direita) quando a função predicada retorna `true` ou `false`. Não se esqueça de colocar uma outra instrução de `return` após o `loop`; para que a função retorne um valor correto quando atingir o final do `array`.

A vida secreta dos objetos

"O problema com as linguagens orientadas a objeto é que elas têm tudo implícito no ambiente que elas carregam consigo. Você queria banana, mas o que você teve foi um gorila segurando a banana e toda a floresta." Joe Armstrong, entrevistado em Coders at Work

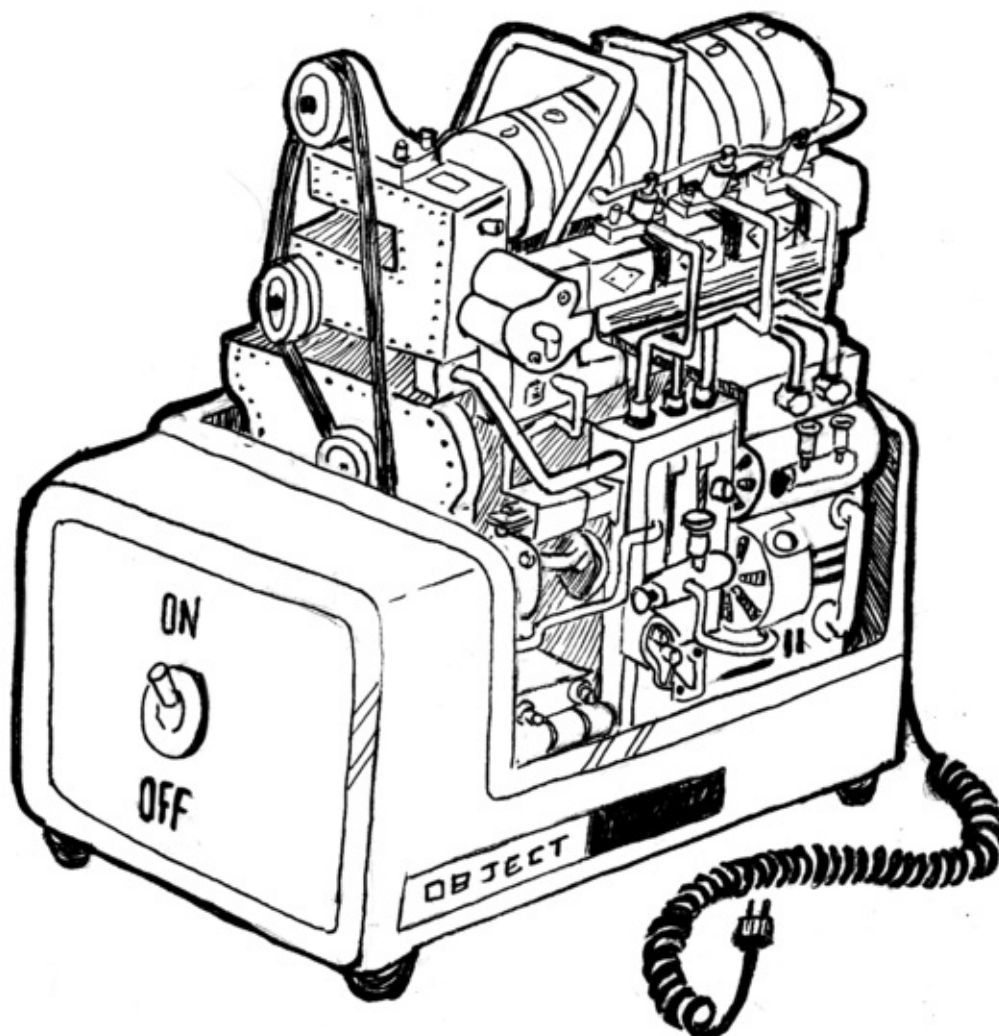
Quando um programador diz "objeto", isso é um termo carregado. Na minha profissão, objetos são a maneira de viver, o sujeito das guerras santas, e um jargão apaixonante que ainda não perdeu o seu poder.

Para um estrangeiro, isso provavelmente é um pouco confuso. Vamos começar com uma rápida história dos objetos como construtores da programação.

História

Essa história, como a maioria das histórias de programação, começa com um problema de complexidade. A teoria é de que a complexidade pode ser administrada separando-a em pequenos compartimentos isolados um do outro. Esses compartimentos acabaram ganhando o nome de *objetos*.

Um objeto é um escudo duro que esconde a complexidade grudenta dentro dele e nos apresenta pequenos conectores (como métodos) que apresentam uma interface para utilizarmos o objeto. A ideia é que a interface seja relativamente simples e toda as coisas complexas que vão dentro do objeto possam ser ignoradas enquanto se trabalha com ele.



Como exemplo, você pode imaginar um objeto que disponibiliza uma interface para uma determinada área na sua tela. Ele disponibiliza uma maneira de desenhar formas ou textos nessa área, mas esconde todos os detalhes de como essas formas são convertidos para os pixels que compõem a tela. Você teria um conjunto de métodos- `desenharCirculo` , por exemplo- e essas serão as únicas coisas que você precisa saber pra usar tal objeto.

Essas ideias foram trabalhadas inicialmente por volta dos anos 70 e 80 e, nos anos 90, foram trazidas à tona por uma enorme onda *hype*-a revolução da programação orientada a objetos. De repente, existia uma enorme tribo de pessoas declarando que objetos eram a maneira correta de programar-e que qualquer coisa que não envolvesse objetos era uma loucura ultrapassada.

Esse tipo de fanatismo produz um monte de bobagem impraticável, e desde então uma espécie de contra-revolução vem acontecendo. Em alguns círculos de desenvolvedores, os objetos têm uma péssima reputação hoje em dia.

Eu prefiro olhar para esse problema de um ângulo prático, e não ideológico. Existem vários conceitos úteis, dentre eles um dos mais importantes é o *encapsulamento* (distinguir complexidade interna e interface externa), que a cultura orientada a objetos tem popularizado. Vamos ver esses conceitos, pois eles valem a pena.

Esse capítulo descreve uma pegada mais excêntrica do JavaScript com foco nos objetos e na forma como eles se relacionam com algumas técnicas clássicas de orientação a objetos.

Métodos

Métodos são propriedades simples que comportam valores de funções. Isso é um método simples:

```
var coelho = {};  
coelho.diz = function(linha) {  
  console.log("O coelho diz '" + linha + "'");  
};  
  
coelho.diz("Estou vivo.");  
// → O coelho diz 'Estou vivo.'
```

Normalmente um método precisa fazer alguma coisa com o objeto pelo qual ele foi chamado. Quando uma função é chamada como um método-visualizada como uma propriedade e imediatamente chamada, como em `objeto.metodo()` -a variável especial `this` no seu conteúdo vai apontar para o objeto pelo qual foi chamada.

```
function speak(line) {  
  console.log("The " + this.type + " rabbit says '" +  
    line + "'");  
}  
var whiteRabbit = {type: "white", speak: speak};  
var fatRabbit = {type: "fat", speak: speak};  
  
whiteRabbit.speak("Oh my ears and whiskers, " +  
  "how late it's getting!");  
// → The white rabbit says 'Oh my ears and whiskers, how  
//   late it's getting!'  
fatRabbit.speak("I could sure use a carrot right now.");  
// → The fat rabbit says 'I could sure use a carrot  
//   right now.'
```

O código acima usa a palavra-chave `this` para dar a saída do tipo de coelho que está falando. Lembrando que ambos os métodos `apply` e `bind` podem usar o primeiro argumento para simular chamadas de métodos. Esse primeiro argumento, é na verdade, usado para passar um valor ao `this`.

Existe um método parecido ao `apply` chamado `call`. Ele também chama a função na qual ele é um método e aceita argumentos normalmente, ao invés de um array. Assim como `apply` e `bind`, o `call` pode ser passado com um valor específico no `this`.

```
speak.apply(fatRabbit, ["Burp!"]);  
// → The fat rabbit says 'Burp!'  
speak.call({type: "old"}, "Oh my.");  
// → The old rabbit says 'Oh my.'
```

Prototypes

Observe com atenção.

```
var empty = {};  
console.log(empty.toString());  
// → function toString(){...}  
console.log(empty.toString());  
// → [object Object]
```

Eu acabei de sacar uma propriedade de um objeto vazio. Mágica!

Só que não. Eu venho ocultando algumas informações sobre como os objetos funcionam no JavaScript. Além de sua lista de propriedades, quase todos os objetos também possuem um *protótipo*, ou *prototype*. Um *prototype* é outro objeto que é usado como fonte de *fallback* para as propriedades. Quando um objeto recebe uma chamada em uma

propriedade que ele não possui, seu *prototype* designado para aquela propriedade será buscado, e então o *prototype* daquele *prototype* e assim por diante.

Então quem é o *prototype* de um objeto vazio? É o ancestral de todos os *prototypes*, a entidade por trás de quase todos os objetos, `Object.prototype`.

```
console.log(Object.getPrototypeOf({}) ==
              Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

A função `Object.getPrototypeOf` retorna o *prototype* de um objeto como o esperado.

As relações dos objetos JavaScript formam uma estrutura em forma de árvore, e na raiz dessa estrutura se encontra o `Object.prototype`. Ele fornece alguns métodos que estão presentes em todos os objetos, como o `toString`, que converte um objeto para uma representação em *string*.

Muitos objetos não possuem o `Object.prototype` diretamente em seu *prototype*. Ao invés disso eles têm outro objeto que fornece suas propriedades padrão. Funções derivam do `Function.prototype`, e *arrays* derivam do `Array.prototype`.

```
console.log(Object.getPrototypeOf(isNaN) ==
              Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
              Array.prototype);
// → true
```

Por diversas vezes, o *prototype* de um objeto também terá um *prototype*, dessa forma ele ainda fornecerá indiretamente métodos como `toString`.

A função `Object.getPrototypeOf` obviamente retornarão o *prototype* de um objeto. Você pode usar `Object.create` para criar um objeto com um *prototype* específico.

```
var protoCoelho = {
  fala: function(linha) {
    console.log("O coelho " + this.tipo + " fala '" +
                linha + "'");
  }
};
var coelhoAssassino = Object.create(protoCoelho);
coelhoAssassino.tipo = "assassino";
coelhoAssassino.fala("SKREEEE!");
// → O coelho assassino fala 'SKREEEE!'
```

Construtores

A maneira mais conveniente de criar objetos que herdam algum *prototype* compartilhado é usar um construtor. No JavaScript, chamar uma função precedida pela palavra-chave `new` vai fazer com que ela seja tratada como um construtor. O construtor terá sua variável `this` atrelada a um objeto novo, e a menos que ele explicitamente retorne o valor de outro objeto, esse novo objeto será retornado a partir da chamada.

Um objeto criado com `new` é chamado de *instância* do construtor.

Aqui está um construtor simples para coelhos. É uma convenção iniciar o nome de um construtor com letra maiúscula para que seja fácil distingui-los das outras funções.

```
function Coelho(tipo) {
  this.tipo = tipo;
}

var coelhoAssassino = new Coelho("assassino");
var coelhoPreto = new Coelho("preto");
console.log(coelhoPreto.tipo);
// → preto
```

Construtores (todas as funções, na verdade) pegam automaticamente uma propriedade chamada `prototype`, que por padrão possui um objeto vazio que deriva do `Object.prototype`. Toda instância criada com esse construtor terá esse objeto assim como seu *prototype*. Então, para adicionar um método `fala` aos coelhos criados com o construtor `coelho`, nós podemos simplesmente fazer isso:

```
Coelho.prototype.fala = function(linha) {
  console.log("O coelho " + this.tipo + " fala '" +
    linha + "'");
};
coelhoPreto.fala("Doom...");
// → O coelho preto fala 'Doom...'
```

É importante notar a distinção entre a maneira que um *prototype* é associado a um construtor (por sua propriedade *prototype*) e a maneira que objetos *têm* um *prototype* (que pode ser obtido com `Object.getPrototypeOf`). O *prototype* propriamente dito de um construtor é `Function.prototype`, visto que os construtores são funções. Sua *propriedade prototype* será o *prototype* de instâncias criadas através dele mas não será seu *próprio prototype*.

Definindo uma tabela

Eu vou trabalhar sobre um exemplo ou pouco mais envolvido na tentativa de dar a você uma melhor ideia de polimorfismo, assim como de programação orientada a objetos em geral. O projeto é este: nós vamos escrever um programa que, dado um array de arrays de células de uma tabela, cria uma string que contém uma tabela bem formatada - significando que colunas são retas e linhas estão alinhadas. Algo dessa forma:

name	height	country
Kilimanjaro	5895	Tanzania
Everest	8848	Nepal
Mount Fuji	3776	Japan
Mont Blanc	4808	Italy/France
Vaalserberg	323	Netherlands
Denali	6168	United States
Popocatepetl	5465	Mexico

A forma que nosso sistema de construir tabelas vai funcionar é que a função construtora vai perguntar para cada célula quanto de altura e largura ela vai querer ter e então usar essa informação para determinar a largura das colunas e a altura das linhas. A função construtora vai então pedir para as células se desenharem no tamanho correto e montar o resultado dentro de uma string.

O programa de *layout* vai comunicar com os objetos células através de uma interface bem definida. Dessa forma, os tipos de células que o programa suporta não está definida antecipadamente. Nós podemos adicionar novas células de estilo depois — por exemplo, células sublinhadas para cabeçalho — e se eles suportarem nossa interface, isso vai simplesmente, funcionar, sem exigir alterações no layout do programa.

Esta é a interface:

- `minHeight()` retorna um número indicando a altura mínima que esta célula necessita (em linhas).

- `minWidth()` retorna um número indicando a largura mínima da célula (em caracteres).
- `draw(width, height)` retorna um array de tamanho `height`, que contém uma série de strings que contém `width` caracteres de tamanho. Isso representa o conteúdo da célula.

Irei fazer forte uso de métodos de ordem superior de array neste exemplo uma vez que isso é apropriado para essa abordagem.

A primeira parte do programa calcula matrizes de largura e altura mínima para uma grade de células.

```
function rowHeights(rows) {
  return rows.map(function(row) {
    return row.reduce(function(max, cell) {
      return Math.max(max, cell.minHeight());
    }, 0);
  });
}

function colWidths(rows) {
  return rows[0].map(function(_, i) {
    return rows.reduce(function(max, row) {
      return Math.max(max, row[i].minWidth());
    }, 0);
  });
}
```

Usar um nome de variável que se inicia com um *underscore* (`_`) ou consistir inteiramente em um simples *underscore* é uma forma de indicar (para leitores humanos) que este argumento não será usado.

A função `rowHeights` não deve ser tão difícil de ser seguida. Ela usa `reduce` para computar a altura máxima de um array de células e envolve isso em um `map` para fazer isso em todas as linhas no array `rows`.

As coisas são um pouco mais difíceis na função `colWidths` porque o array externo é um array de linhas, não de colunas. Não mencionei até agora que `map` (assim como `forEach`, `filter` e métodos de array similares) passam um segundo argumento à função fornecida: o índice do elemento atual. Mapeando os elementos da primeira linha e somente usando o segundo argumento da função de mapeamento, `colWidths` constrói um array com um elemento para cada índice da coluna. A chamada à `reduce` roda sobre o array `rows` exterior para cada índice e pega a largura da célula mais larga nesse índice.

Aqui está o código para desenhar a tabela:

```
function drawTable(rows) {
  var heights = rowHeights(rows);
  var widths = colWidths(rows);

  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }

  function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
  }

  return rows.map(drawRow).join("\n");
}
```

A função `drawTable` usa a função interna `drawRow` para desenhar todas as linhas e então as junta com caracteres *newline* (nova linha).

A função `drawRow` primeiro converte os objetos célula na linha em *blocos*, que são arrays representando o conteúdo das células, divididos por linha. Uma célula simples contendo apenas o número 3776 deve ser representada por um array com um único elemento como `["3776"]`, onde uma célula sublinhada deve conter duas linhas e ser representada pelo array `["name", "----"]`.

Os blocos para uma linha, que devem todos ter a mesma largura, devem aparecer próximos um ao outro na saída final. A segunda chamada a `map` em `drawRow` constrói essa saída linha por linha mapeando sobre as linhas do bloco mais à esquerda e, para cada uma delas, coletando uma linha que expande a tabela para sua largura máxima. Essas linhas são então juntadas com caracteres *newline* para fornecer a linha completa e ser o valor retornado de `drawRow`.

A função `drawLine` extrai linhas que devem aparecer próximas uma a outra a partir de um array de blocos e as junta com um caracter espaço para criar o espaço de um caracter entre as colunas da tabela.

Agora vamos escrever o construtor para células que contenham texto, implementando a interface para as células da tabela. O construtor divide a linha em um array de linhas usando o método string `split`, que corta uma string em cada ocorrência do seu argumento e retorna um array com as partes. O método `minWidth` encontra a linha com maior largura nesse array.

```
function repeat(string, times) {
  var result = "";
  for (var i = 0; i < times; i++)
    result += string;
  return result;
}

function TextCell(text) {
  this.text = text.split("\n");
}
TextCell.prototype.minWidth = function() {
  return this.text.reduce(function(width, line) {
    return Math.max(width, line.length);
  }, 0);
};
TextCell.prototype.minHeight = function() {
  return this.text.length;
};
TextCell.prototype.draw = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(line + repeat(" ", width - line.length));
  }
  return result;
};
```

O código usa uma função auxiliar chamada `repeat`, que constrói uma linha na qual o valor é um argumento `string` repetido `times` número de vezes. O método `draw` usa isso e adiciona "preenchimento" para as linhas assim todas vão ter o tamanho requerido.

Vamos testar tudo que construímos e criar um tabuleiro de damas 5 x 5.

```
var rows = [];
for (var i = 0; i < 5; i++) {
  var row = [];
  for (var j = 0; j < 5; j++) {
    if ((j + i) % 2 == 0)
      row.push(new TextCell("##"));
  }
}
```

```

        else
            row.push(new TextCell("  "));
        }
        rows.push(row);
    }
    console.log(drawTable(rows));
    // → ##    ##    ##
    //      ##    ##
    //    ##    ##    ##
    //      ##    ##
    //    ##    ##    ##

```

Funciona! Mas apesar de todas as células terem o mesmo tamanho, o código do layout da tabela não faz nada realmente interessante.

Os dados fonte para a tabela de montanhas que estamos tentando construir estão disponíveis na variável `MOUNTAINS` na *sandbox* e também [neste arquivo](#) em nosso website.

Vamos querer destacar a linha do topo, que contém o nome das colunas, sublinhando as células com uma série de caracteres *traço*. Sem problemas — nós simplesmente escrevemos um tipo de célula que manipula o sublinhado.

```

function UnderlinedCell(inner) {
    this.inner = inner;
};
UnderlinedCell.prototype.minWidth = function() {
    return this.inner.minWidth();
};
UnderlinedCell.prototype.minHeight = function() {
    return this.inner.minHeight() + 1;
};
UnderlinedCell.prototype.draw = function(width, height) {
    return this.inner.draw(width, height - 1)
        .concat([repeat("-", width)]);
};

```

Uma célula sublinhada *contém* outra célula. Ela reporta seu tamanho mínimo sendo o mesmo que da sua célula interna (chamando os métodos `minWidth` e `minHeight` desta célula) mas adicionando um à largura para contar o espaço usado pelo sublinhado.

Desenhar essa célula é bem simples - nós pegamos o conteúdo da célula interna e concatenamos uma simples linha preenchida com traços a ela.

Tendo um mecanismo de sublinhamento, nós podemos agora escrever uma função que constrói uma grade de células a partir do conjunto de dados.

```

function dataTable(data) {
    var keys = Object.keys(data[0]);
    var headers = keys.map(function(name) {
        return new UnderlinedCell(new TextCell(name));
    });
    var body = data.map(function(row) {
        return keys.map(function(name) {
            return new TextCell(String(row[name]));
        });
    });
    return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
// → name      height country
//    -----
//    Kilimanjaro 5895  Tanzania
//    ... etcetera

```

A função padrão `Object.keys` retorna um array com nomes de propriedades de um objeto. A linha do topo da tabela deve conter células sublinhadas que dão os nomes das colunas. Abaixo disso, os valores de todos os objetos no conjunto de dados aparecem como células normais - nós os extraímos mapeando sobre o array `keys` de modo que tenhamos certeza que a ordem das células é a mesma em todas as linhas.

A tabela resultante se assemelha ao exemplo mostrado anteriormente, exceto que ela não alinha os números à direita na coluna `height`. Vamos chegar nessa parte em um instante.

Getters and Setters

Quando especificamos uma interface, é possível incluir propriedades que não são métodos. Poderíamos ter definido `minHeight` e `minWidth` para simplesmente conter números. Mas isso teria exigido de nós computá-los no construtor, o que adicionaria código que não é estritamente relevante para *construção* do objeto. Isso pode causar problemas se, por exemplo, a célula interior de uma célula exterior mudou, onde nesse ponto o tamanho da célula sublinhada também deve mudar.

Isso tem levado algumas pessoas a adotarem um princípio de nunca incluírem propriedades *nonmethod* em interfaces. Ao invés de acessarem diretamente o valor da propriedade, eles usam métodos `getSomething` e `setSomething` para ler e escrever propriedades. Esta abordagem tem a parte negativa de que você irá acabar escrevendo - e lendo - muitos métodos adicionais.

Felizmente, o JavaScript fornece uma técnica que fornece o melhor de ambos os mundos. Nós podemos especificar propriedades que, do lado de fora, parecem propriedades normais mas secretamente tem métodos associados a elas.

```
var pile = {
  elements: ["eggshell", "orange peel", "worm"],
  get height() {
    return this.elements.length;
  },
  set height(value) {
    console.log("Ignoring attempt to set height to", value);
  }
};

console.log(pile.height);
// → 3
pile.height = 100;
// → Ignoring attempt to set height to 100
```

Em um objeto literal, a notação `get` ou `set` para propriedades permite que você especifique uma função a ser executada quando a propriedade for lida ou escrita. Você pode também adicionar tal propriedade em um objeto existente, por exemplo um protótipo, usando a função `Object.defineProperty` (que nós previamente usamos para criar propriedades não enumeráveis).

```
Object.defineProperty(TextCell.prototype, "heightProp", {
  get: function() { return this.text.length; }
});

var cell = new TextCell("no\\nway");
console.log(cell.heightProp);
// → 2
cell.heightProp = 100;
console.log(cell.heightProp);
// → 2
```

Você pode usar a propriedade similar `set`, no objeto passado à `defineProperty`, para especificar um método *setter*. Quando um *getter* é definido mas um *setter* não, escrever nessa propriedade é algo simplesmente ignorado.

Herança

Nós não estamos prontos com nosso exercício de layout de tabela. Ela deve ajudar na leitura de números alinhados à direita em colunas. Nós devemos criar outro tipo de célula como `TextCell`, mas ao invés de dar espaço nas linhas do lado direito, vamos espaçá-las do lado esquerdo que irá alinhá-las à direita.

Podemos simplesmente construir um novo construtor com todos os três métodos em seu protótipo. Mas protótipos podem ter seus próprios protótipos, e isso nos permite fazer algo inteligente.

```
function RTextCell(text) {
  TextCell.call(this, text);
}
RTextCell.prototype = Object.create(TextCell.prototype);
RTextCell.prototype.draw = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(repeat(" ", width - line.length) + line);
  }
  return result;
};
```

Nós reusamos o construtor e os métodos `minHeight` e `minWidth` de `TextCell`. Um `RTextCell` é agora basicamente equivalente a `TextCell`, exceto que seu método `draw` contém uma função diferente.

Este padrão é chamado *herança*. Isso nos permite construir tipos de dados levemente diferentes a partir de tipos de dados existentes com relativamente pouco esforço. Tipicamente, o novo construtor vai chamar o antigo construtor (usando o método `call` para ser capaz de dar a ele o novo objeto assim como o seu valor `this`). Uma vez que esse construtor tenha sido chamado, nós podemos assumir que todos os campos que o tipo do antigo objeto supostamente contém foram adicionados. Nós organizamos para que o protótipo do construtor derive do antigo protótipo, então as instâncias deste tipo também vão acesso às propriedades deste protótipo. Finalmente, nós podemos sobrescrever algumas das propriedades adicionando-as ao nosso novo protótipo.

Agora, se nós ajustarmos sutilmente a função `dataTable` para usar `RTextCell` para as células cujo valor é um número, vamos obter a tabela que estávamos buscando.

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      var value = row[name];
      // This was changed:
      if (typeof value == "number")
        return new RTextCell(String(value));
      else
        return new TextCell(String(value));
    });
  });
  return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
// → ... beautifully aligned table
```

Herança é uma parte fundamental da orientação a objetos tradicional, ao lado de encapsulamento e polimorfismo. Mas enquanto os dois últimos sejam agora geralmente considerados como ideias brilhantes, herança é algo controverso.

A principal razão para isso é que este tópico é geralmente confundido com polimorfismo, vendido como uma ferramenta mais poderosa do que realmente é, e subsequentemente usado em excesso de diversas horríveis formas. Onde encapsulamento e polimorfismo podem ser usados para *separar* pedaços de código de cada um, reduzindo o emaranhamento de todo o programa, herança fundamentalmente vincula os tipos, criando *mais* emaranhados.

Você pode ter polimorfismo sem herança, como nós vimos. Eu não vou dizer para você evitar herança completamente. Eu a uso regularmente em meus programas. Mas você deve vê-la como um leve truque desonesto que vai ajudá-lo a definir novos tipos com menos código, não como um grande princípio de organização de código. Uma forma mais apropriada de estender tipos é através da composição, como `UnderlinedCell` constrói em outra célula simplesmente armazenando-a em uma propriedade e um método posterior a chama nos seus próprios métodos.

O operador `instanceof`

Ocasionalmente é útil saber se um objeto foi derivado de um construtor em específico. Para isso, o JavaScript fornece um operador binário chamado `instanceof`.

```
console.log(new RTextCell("A") instanceof RTextCell);
// → true
console.log(new RTextCell("A") instanceof TextCell);
// → true
console.log(new TextCell("A") instanceof RTextCell);
// → false
console.log([1] instanceof Array);
// → true
```

O operador vai olhar através dos tipos herdados. Um `RTextCell` é uma instância de `TextCell` porque `RTextCell.prototype` deriva de `TextCell.prototype`. O operador pode ser aplicado a construtores padrão como `Array`. Praticamente todos os objetos são uma instância de `Object`.

Resumo

Então objetos são mais complicados do que inicialmente eu os retratei. Eles tem protótipos, que são outros objetos, e vão agir como se tivessem propriedades que eles não tem caso seu protótipo tenha essa propriedade. Objetos simples tem `Object.prototype` como seus protótipos.

Construtores, que são funções cujos nomes usualmente iniciam com uma letra maiúscula, podem ser usados com o operador `new` para criar objetos. O protótipo do novo objeto será o objeto encontrado na propriedade `prototype` da função construtora. Você pode fazer bom uso disso adicionando propriedades que todos os valores de um tipo compartilham em seus protótipos. O operador `instanceof` pode, dado um objeto e um construtor, dizer se o objeto é uma instância deste construtor.

Algo útil a se fazer com objetos é especificar uma interface para eles e dizer para todos que irão supostamente conversar com seu objeto a fazer isso somente por essa interface. O resto dos detalhes que constroem seu objeto estão agora *encapsulados*, escondidos atrás da interface.

Uma vez que você esteja conversando em termos de interfaces, quem diz que apenas um tipo de objeto pode implementar essa interface? Ter diferentes objetos expondo a mesma interface é chamado de *polimorfismo*. Isso é muito útil.

Quando implementando vários tipos que diferem apenas em alguns detalhes, pode ser útil simplesmente criar o protótipo do seu novo tipo derivando do protótipo do seu antigo tipo e ter seu novo construtor chamando o antigo. Isso lhe dá um tipo similar de objeto ao antigo mas que permite que você adicione ou sobrescreva propriedades quando necessário.

Exercícios

Um tipo de vetor

Escreva um construtor `Vector` que represente um vetor em duas dimensões do espaço. Ele recebe os parâmetros `x` e `y` (números), que deve salvar em propriedades de mesmo nome.

Dê ao protótipo de `Vector` dois métodos, `plus` e `minus`, que pegam outro vetor como parâmetro e retornam um novo vetor que tem a soma ou diferença dos valores `x` e `y` dos dois vetores (o vetor que está em `this` e o passado no parâmetro).

Adicione uma propriedade getter `length` ao protótipo que calcula o tamanho do vetor - isto é, a distância do ponto (x, y) até a origem $(0,0)$.

```
// Your code here.  
  
console.log(new Vector(1, 2).plus(new Vector(2, 3)));  
// → Vector{x: 3, y: 5}  
console.log(new Vector(1, 2).minus(new Vector(2, 3)));  
// → Vector{x: -1, y: -1}  
console.log(new Vector(3, 4).length);  
// → 5
```

Dicas

Sua solução pode seguir o padrão do construtor `Rabbit` deste capítulo de forma bem semelhante.

Adicionar uma propriedade getter ao construtor pode ser feita com a função `Object.defineProperty`. Para calcular a distância do ponto $(0, 0)$ até (x, y) você pode usar o teorema de Pitágoras, que diz que o quadrado da distância que estamos procurando é igual ao quadrado da coordenada x mais o quadrado da coordenada y. Assim, $\sqrt{x^2 + y^2}$ é o número que você quer, e `Math.sqrt` é o caminho para você calcular a raiz quadrada no JavaScript.

Outra célula

Implemente uma célula do tipo `StretchCell(inner, width, height)` que se adeque a [interface da célula da tabela](#) descrita anteriormente neste capítulo. Ela deve envolver outra célula (como `UnderlinedCell` faz) e assegurar que a célula resultante tem pelo menos a largura (`width`) e altura (`height`) especificada, mesmo se a célula interior for naturalmente menor.

```
// Your code here.  
  
var sc = new StretchCell(new TextCell("abc"), 1, 2);  
console.log(sc.minWidth());  
// → 3  
console.log(sc.minHeight());  
// → 2  
console.log(sc.draw(3, 2));  
// → ["abc", " "]
```

Dicas

Você vai ter que armazenar os 3 argumentos construtores na instância do objeto. Os métodos `minWidth` e `minHeight` devem chamar através dos métodos correspondentes na célula interna (`inner`), mas assegure-se que nenhum número menor que o tamanho dado é retornado (possivelmente usando `Math.max`).

Não se esqueça de adicionar um método `draw` que simplesmente encaminha a chamada para a célula interior.

Interface sequencial

Projete uma *interface* que abstraia interações sobre uma coleção de valores. Um objeto que fornece esta interface representa uma sequência, e a interface deve de alguma forma tornar possível para o código que usa este objeto iterar sobre uma sequência, olhando para o valor dos elementos de que ela é composta e tendo alguma forma de saber quando o fim da sequência foi atingido.

Quando você tiver especificado sua interface, tente escrever uma função `logFive` que pega um objeto sequencial e chama `console.log` para seus primeiros 5 elementos - ou menos, se a sequência tiver menos do que cinco elementos.

Então implemente um tipo de objeto `ArraySeq` que envolve um array e permite interação sobre o array usando a interface que você desenvolveu. Implemente outro tipo de objeto `RangeSeq` que itera sobre um intervalo de inteiros (recebendo os argumentos `from` e `to` em seu construtor).

```
// Your code here.

logFive(new ArraySeq([1, 2]));
// → 1
// → 2
logFive(new RangeSeq(100, 1000));
// → 100
// → 101
// → 102
// → 103
// → 104
```

Dicas

Uma forma de resolver isso é fornecendo objetos sequenciais *state*, que significa que suas propriedades são alteradas no seu processo de uso. Você pode armazenar um contador que indica quão longe o objeto sequenciais avançaram.

Sua interface vai precisar expor ao menos uma forma de pegar o próximo elemento e encontrar se a iteração já chegou no fim da sequência. É tentador fazer isso em um método, `next`, que retorna `null` ou `undefined` quando a sequência chegar ao fim. Mas agora você tem um problema quando a sequência realmente tiver `null`. Então um método separado (ou uma propriedade getter) para descobrir se o fim foi alcançado é provavelmente preferível.

Outra solução é evitar mudar o estado do objeto. Você pode expor um método para pegar o elemento atual (sem o auxílio de nenhum contador) e outro para pegar uma nova sequência que representa os elementos restantes depois do atual (ou um valor especial se o fim da sequência tiver sido atingido). Isso é bem elegante - um valor sequencial vai "permanecer ele mesmo" mesmo depois de ter sido usado e pode ser compartilhado com outro código sem a preocupação sobre o que pode acontecer com ele. Isso é, infelizmente, algo um pouco ineficiente numa linguagem como JavaScript porque envolve criar vários objetos durante a iteração.

Projeto - Vida eletrônica

[...] A questão da máquinas poder pensar [...] é tão relevante quanto a questão dos submarinos poderem nadar.

- Edsger Dijkstra, The Threats to Computing Science

Nos capítulo "Projeto" irei apresentar uma nova teoria por um breve momento e trabalhar através de um programa com você. A teoria é indispensável quando estamos aprendendo a programar mas deve ser acompanhada da leitura para entender os programas não triviais.

Nosso projeto neste capítulo é construir um ecossistema virtual, um mundo pequeno povoado com criaturas que se movem e luta pela sobrevivência.

Definição

Para tornar esta tarefa gerenciável, vamos simplificar radicalmente o conceito de um mundo. Ou seja, um mundo será uma `grid` bidimensional onde cada entidade ocupa um quadrado da `grid`. Em cada turno os bichos todos têm a chance de tomar alguma ação.

Utilizaremos o tempo e o espaço com um tamanho fixo como unidades. Os quadrados serão os espaços e as voltas o tempo. É claro que as aproximações serão imprecisas. Mas nossa simulação pretende ser divertida para que possamos livremente cortar as sobras.

Podemos definir um mundo com uma matriz de `Strings` que estabelece uma `grid` do mundo usando um personagem por metro quadrado.

```
var plan = ["#####",
  "#      #      #      o      ##",
  "#",
  "#      #####",
  "###      #      #      ##",
  "###      ##      #      #",
  "#      ###      #      #",
  "#      #####",
  "#      ##      o",
  "# o #      o      ### #",
  "#      #",
  "#####"];
```

Os caracteres `"#"` representam as paredes e rochas, e os personagens `"o"` representam os bichos. Os espaços como você já deve ter pensado é o espaço vazio.

Um plano de matriz pode ser usada para criar um objeto de mundo. Tal objeto mantém o controle do tamanho e do conteúdo do mundo. Ele tem um método `toString`, que converte o mundo de volta para uma sequência de impressão (similar ao plano que foi baseado) para que possamos ver o que está acontecendo lá dentro. O objeto do mundo também tem um método por sua vez que permite que todos os bichos podem darem uma volta e atualizar o mundo para terem suas ações.

Representando o espaço

A `grid` modela o mundo com uma largura e altura fixa. Os quadrados são identificados pelas suas coordenadas `x` e `y`. Nós usamos um tipo simples, `Vector`(como visto nos exercícios do capítulo anterior) para representar esses pares de coordenadas.

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

Em seguida, temos um tipo de objeto que é o modelo da `grid`. A `grid` é uma parte do mundo, e tornamos ela um objeto separado(que será uma propriedade de um objeto do mundo) para manter o objeto bem simples. O mundo deve preocupar-se com as coisas relacionadas com o mundo e a `grid` deve preocupar-se com as coisas relacionadas da `grid`.

Para armazenar um valor a `grid` temos várias opções. Podemos usar um `array` de `arrays` tendo duas propriedades de acessos para chegar a um quadrado específico como este:

```
var grid = ["top left", "top middle", "top right",
            "bottom left", "bottom middle", "bottom right"];
console.log(grid[1][2]);
// → bottom right
```

Ou podemos usar uma única matriz com largura x altura e decidir que o elemento `(x, y)` é encontrado na posição `x + (y * largura)` na matriz.

```
var grid = ["top left", "top middle", "top right",
            "bottom left", "bottom middle", "bottom right"];
console.log(grid[2 + (1 * 3)]);
// → bottom right
```

Uma vez que o acesso real a essa matriz esta envolto em métodos de tipo do objeto da `grid`, não importa o código que adotamos para abordagem. Eu escolhi a segunda representação pois torna muito mais fácil para criar a matriz. Ao chamar o construtor de `Array` com um único argumento, ele cria uma nova matriz vazia com o comprimento que foi passado de parâmetro.

Esse código define o objeto `grid`, com alguns métodos básicos:

```
function Grid(width, height) {
  this.space = new Array(width * height);
  this.width = width;
  this.height = height;
}
Grid.prototype.isInside = function(vector) {
  return vector.x >= 0 && vector.x < this.width &&
    vector.y >= 0 && vector.y < this.height;
};
Grid.prototype.get = function(vector) {
  return this.space[vector.x + this.width * vector.y];
};
Grid.prototype.set = function(vector, value) {
  this.space[vector.x + this.width * vector.y] = value;
};
```

Aqui esta um exemplo trivial do teste:

```
var grid = new Grid(5, 5);
console.log(grid.get(new Vector(1, 1)));
// → undefined
grid.set(new Vector(1, 1), "X");
console.log(grid.get(new Vector(1, 1)));
// → X
```

A interface da programação dos bichos

Antes de começarmos nosso construtor global, devemos especificar quais os objetos bichos que estarão vivendo em nosso mundo. Eu mencionei que o mundo vai especificar os bichos e as ações que eles terão. Isso funciona da seguinte forma: cada bicho é um objeto e tem um método de ação que quando chamado retorna uma ação. Uma ação é um objeto com uma propriedade de tipo, que dá nome a ação que o bicho terá, por exemplo `"move"`. A ação pode também conter informação extra de alguma direção que o bicho possa se mover.

Bichos são terrivelmente míopes e podem ver apenas os quadrados em torno da `grid`. Mas essa visão limitada pode ser útil ao decidir que ação tomar. Quando o método `act` é chamado o objeto de verificação permite que o bicho inspecione seus arredores. Nós vamos nomear oito quadrados vizinhos para ser as coordenadas: `"n"` para norte, `"ne"` para nordeste e assim por diante. Aqui está o objeto, vamos utilizar para mapear os nomes das direções para coordenar os `offsets`:

```
var directions = {
  "n": new Vector( 0, -1),
  "ne": new Vector( 1, -1),
  "e": new Vector( 1,  0),
  "se": new Vector( 1,  1),
  "s": new Vector( 0,  1),
  "sw": new Vector(-1,  1),
  "w": new Vector(-1,  0),
  "nw": new Vector(-1, -1)
};
```

O objeto de exibição tem um método que observa em qual direção o bicho esta indo e retorna um personagem por exemplo, um `"#"` quando há uma parede na direção ou um `" "` (espaço) quando não há nada. O objeto também fornece os métodos `find` e `findAll`. Ambos tomam um mapa de caráter como um argumento. O primeiro retorna a direção em que o personagem pode ser encontrado ao lado do bicho ou retorna nulo se não existir nenhum sentido. O segundo retorna um `array` contendo todas as direções possíveis para o personagem. Por exemplo, uma criatura sentada à esquerda(oeste) de um muro vai ter `["ne", "e", "se"]` ao chamar `findAll` passando o caractere `"#"` como argumento.

Aqui é um bicho simples e estúpido que segue apenas seu nariz até que ela atinja um obstáculo e depois salta para fora em uma direção aleatória:

```
function randomElement(array) {
  return array[Math.floor(Math.random() * array.length)];
}

var directionNames = "n ne e se s sw w nw".split(" ");

function BouncingCritter() {
  this.direction = randomElement(directionNames);
};

BouncingCritter.prototype.act = function(view) {
  if (view.look(this.direction) != " ")
    this.direction = view.find(" ") || "s";
  return {type: "move", direction: this.direction};
};
```

A função auxiliar `randomElement` simplesmente pega um elemento aleatório de uma matriz usando `Math.random` para obter um índice aleatório. Vamos usar isso de novo mais tarde porque a aleatoriedade pode ser útil em simulações.

Para escolher uma direção aleatória o construtor de `BouncingCritic` chama `randomElement` em uma matriz de nomes de direção. Nós também poderíamos termos usado `Object.keys` para obter essa matriz de direções que definimos anteriormente, mas não é garantido a ordem em que as propriedades serão listadas. Na maioria das situações os motores modernos de JavaScript retornam as propriedades na ordem em que foram definidos, mas eles não são obrigados a terem tais comportamentos.

O `"|| 's'"` no método de ação serve para impedir que `this.direction` obtenha um valor nulo para o bicho que está preso em um espaço vazio em torno dele (por exemplo, quando um canto está lotado de outros bichos).

O objeto do mundo

Agora podemos começar a fazer o objeto mundo. O construtor tem um plano (a matriz de `Strings` que representa a `grid` do mundo como descrito anteriormente) e uma legenda como argumentos. A legenda é um objeto que nos diz o que cada personagem no mapa significa. Ela contém um construtor para cada personagem, exceto para o caractere de espaço que sempre se refere como `null` sendo este o valor que vamos usar para representar o espaço vazio.

```
function elementFromChar(legend, ch) {
  if (ch == " ")
    return null;
  var element = new legend[ch]();
  element.originChar = ch;
  return element;
}

function World(map, legend) {
  var grid = new Grid(map[0].length, map.length);
  this.grid = grid;
  this.legend = legend;

  map.forEach(function(line, y) {
    for (var x = 0; x < line.length; x++)
      grid.set(new Vector(x, y),
        elementFromChar(legend, line[x]));
  });
}
```

Em `elementFromChar` primeiro criamos uma instância do tipo correto, observando o construtor do caráter aplicando um novo para ele. Em seguida é adicionado uma propriedade `originChar` tornando mais fácil de descobrir em qual personagem o elemento foi originalmente criado.

Precisamos da propriedade `originChar` quando implementarmos o método `toString` no mundo. Este método constrói uma sequência de mapeamento de estado atual do mundo através da realização de um ciclo de duas dimensões sobre os quadrados na `grid`.

```
function charFromElement(element) {
  if (element == null)
    return " ";
  else
    return element.originChar;
}

World.prototype.toString = function() {
  var output = "";
```

```

for (var y = 0; y < this.grid.height; y++) {
  for (var x = 0; x < this.grid.width; x++) {
    var element = this.grid.get(new Vector(x, y));
    output += charFromElement(element);
  }
  output += "\n";
}
return output;
};

```

A parede é um objeto simples que é usado apenas para ocupar espaço e não tem nenhum método de ação.

```
function Wall() {}
```

Vamos criar um objeto Mundo com base no plano passado no início do capítulo, em seguida iremos chamar `toString` sobre ele.

```

var world = new World(plan, {"#": Wall,
                             "o": BouncingCriticter});
console.log(world.toString());
// → #####
// #      #      #      o      ##
// #
// #      #####      #
// ##      #      #      ##      #
// ###      ##      #      #
// #      ###      #      #
// #      ####      #
// #      ##      o      #
// # o #      o      ### #
// #      #
// #####

```

This e seu escopo

O construtor do mundo contém uma chamada de `forEach`. Uma coisa interessante que podemos notar é que dentro da função do `forEach` não estamos mais no escopo da função do construtor. Cada chamada de função recebe o seu próprio escopo de modo que o escopo presente na função interna não se refere ao objeto externo recém-construído. Na verdade quando a função não é chamada como um método isso refere ao objeto global.

Isso significa que não podemos escrever `this.grid` para acessar nada de fora de dentro do `loop`. Podemos criar uma variável local na função exterior da `grid`, onde a função interna terá acesso a ela.

Isso é um erro de `design` no JavaScript. Felizmente a próxima versão da linguagem irá fornecer uma solução para este problema. Enquanto isso existem soluções alternativas. Um padrão comum é dizer `var auto = this` uma variável local que guarda sua referencia.

Outra solução é usar o método de `bind` que nos permite oferecer uma chamada explícita para o objeto.

```

var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }).bind(this);
  }
};
console.log(test.addPropTo([5]));
// → [15]

```

A função mapeia um `array` e retorna o valor do `prop` que esta dentro do objeto `test` somado ao resultado do valor de um elemento do `array`.

A maioria dos métodos que mapeiam matrizes tais como `forEach` e `map`, têm um segundo argumento opcional que pode ser usado para fornecer um escopo para dentro do bloco de interação (segundo argumento do iterador). Assim, você poderá expressar o exemplo anterior de uma forma um pouco mais simples.

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }, this); // ← no bind
  }
};
console.log(test.addPropTo([5]));
// → [15]
```

Isso funciona apenas para as funções de interações que suportam tal parâmetro de contexto. Quando algum método não suporta receber um contexto você irá precisar usar as outras abordagens.

Em nossas próprias funções de interações podemos apoiar tal parâmetro de contexto enviando um segundo argumento no bloco. Por exemplo, aqui no método `forEach` para o nosso tipo de `grid`, chamaremos uma determinada função para cada elemento da `grid` que não seja nulo ou indefinido:

```
Grid.prototype.forEach = function(f, context) {
  for (var y = 0; y < this.height; y++) {
    for (var x = 0; x < this.width; x++) {
      var value = this.space[x + y * this.width];
      if (value != null)
        f.call(context, value, new Vector(x, y));
    }
  }
};
```

Dando vida

O próximo passo é escrever um método para o objeto mundo que dá aos bichos a chance de movimento. Ele vai passar por cima da `grid` usando o método `forEach` que acabamos de definir a procura de objetos com um método `act`. Quando ele encontra um ele chama o método para obter uma ação e realiza a ação quando ela for válida. Por enquanto apenas as ações `"move"` serão compreendidas.

Existe um problema com esta abordagem. Você consegue identificar? Se deixarmos as criaturas se mover livremente, eles podem se mover para um quadrado que não existe, e nós vamos permitir que eles se mova novamente quando estiver dentro do quadrado vazio. Assim temos que ficar mantendo uma variedade de criaturas que já sumiram ao invés de apenas ignorarmos.

```
World.prototype.turn = function() {
  var acted = [];
  this.grid.forEach(function(critter, vector) {
    if (critter.act && acted.indexOf(critter) == -1) {
      acted.push(critter);
      this.letAct(critter, vector);
    }
  }, this);
};
```

Nós usamos o contexto como segundo parâmetro no método `forEach` para ser a referência da `grid` para conseguirmos acessar corretamente as funções internas. O método `letAct` contém a lógica real que permite que os bichos se movam.

```
World.prototype.letAct = function(critter, vector) {
  var action = critter.act(new View(this, vector));
  if (action && action.type == "move") {
    var dest = this.checkDestination(action, vector);
    if (dest && this.grid.get(dest) == null) {
      this.grid.set(vector, null);
      this.grid.set(dest, critter);
    }
  }
};

World.prototype.checkDestination = function(action, vector) {
  if (directions.hasOwnProperty(action.direction)) {
    var dest = vector.plus(directions[action.direction]);
    if (this.grid.isInside(dest))
      return dest;
  }
};
```

Em primeiro lugar, nós simplesmente pedimos para o bicho se mover, passando um objeto de exibição que tem informações sobre o mundo e a posição atual do bicho naquele mundo(vamos definir a tela em algum momento). O método retorna alguma tipo de ação.

Se o tipo de ação não é um `"move"` ele será ignorado. Se é `"move"` ele terá uma propriedade de direção que se refere a um sentido válido caso o `quadrado` na direção referida estiver vazio(`null`). Iremos definir o bicho para o `quadrado` de destino e ao se mover novamente vamos definir `null` para este quadrado visitado e armazenar o bicho na próximo `quadrado`.

Perceba que `letAct` não ignora coisas que não fazem sentidos como, se a propriedade direção é válida ou se a propriedade do tipo faz sentido. Este tipo de programação defensiva faz sentido em algumas situações. A principal razão para fazê-la é validar alguma fonte proveniente que não seja de controle(como alguma entrada de valores definidas por usuários), mas também pode ser útil para isolar outros subsistemas. Neste caso a intenção é que os bichos podem serem programados de forma não cuidadosa, eles não têm de verificar se suas ações de destinado faz sentido. Eles podem simplesmente solicitar uma ação e o mundo que vai permitir a ação.

Estes dois métodos não fazem a parte da interface externa de um objeto do mundo. Eles são um detalhe interno. Algumas línguas fornece maneiras de declarar explicitamente certos métodos e propriedades privadas e sinalizar um erro quando você tenta usá-los de fora do objeto. JavaScript não faz isso então você vai ter que confiar em alguma outra forma de comunicação para descrever o que faz ou não parte da interface de um objeto. Às vezes ele pode ajudar a usar um esquema de nomes para distinguir entre as propriedades externas e internas, por exemplo, prefixando todas as propriedades internas com um caractere sublinhado(`_`). Isso fará com que os usos acidentais de propriedades que não fazem parte da interface de um objeto fique mais fácil de detectar.

A única parte que falta para a tela se parece com isso:

```
function View(world, vector) {
  this.world = world;
  this.vector = vector;
}
View.prototype.look = function(dir) {
  var target = this.vector.plus(directions[dir]);
  if (this.world.grid.isInside(target))
    return charFromElement(this.world.grid.get(target));
  else
    return "#";
};
```

```
View.prototype.findAll = function(ch) {
  var found = [];
  for (var dir in directions)
    if (this.look(dir) == ch)
      found.push(dir);
  return found;
};
View.prototype.find = function(ch) {
  var found = this.findAll(ch);
  if (found.length == 0) return null;
  return randomElement(found);
};
```

O método observa e descobre se as coordenadas que estamos visitando está dentro da `grid` e se o personagem correspondente ao elemento. Para coordenadas fora da `grid` podemos simplesmente fingir que há uma paredes no modo que podemos definir um mundo que não é murado mas os bichos não poderão caminhar fora das bordas.

O movimento

Nós instanciamos o objeto mundo antes. Agora que nós adicionamos todos os métodos necessários, devemos fazer os movimentos dos elementos no mundo.

```
for (var i = 0; i < 5; i++) {
  world.turn();
  console.log(world.toString());
}
// → ... five turns of moving critters
```

Imprimir várias cópias do mundo é uma forma bastante desagradável para movimentar um mundo. É por isso que o `sandbox` oferece uma função `animateWorld` que executa uma animação, movendo o mundo com três voltas por segundo até que você aperte o botão de `stop`.

```
animateWorld(world);
// → ... life!
```

A implementação do `animateWorld` parece algo misterioso agora, mas depois que você ler os capítulos deste livro que discutem a integração JavaScript em navegadores web, ele não sera tão mágico.

Mais formas de vida

O destaque dramático do nosso mundo é quando duas criaturas saltam para fora. Você consegue pensar em outra forma interessante de comportamento?

O bicho que se move ao longo das paredes. Conceitualmente o bicho mantém a sua mão esquerda(pata, tentáculo ou o que for) para a parede e segue junto a ela. Este jeito acaba sendo não muito trivial de implementar.

Precisamos ser capazes de calcular as direções com a bússola. As direções são modelados por um conjunto de `String`, precisamos definir nossa própria operação(`dirPlus`) para calcular as direções relativas. Então `dirPlus("n", 1)` significa 45 graus no sentido horário para norte quando retornar "ne". Da mesma forma `dirPlus("s", -2)` significa 90 graus para a esquerda ao sul retornando leste.

```
function dirPlus(dir, n) {
  var index = directionNames.indexOf(dir);
  return directionNames[(index + n + 8) % 8];
}
```



```
function WallFollower() {
    this.dir = "s";
}

WallFollower.prototype.act = function(view) {
    var start = this.dir;
    if (view.look(dirPlus(this.dir, -3)) != " ")
        start = this.dir = dirPlus(this.dir, -2);
    while (view.look(this.dir) != " ") {
        this.dir = dirPlus(this.dir, 1);
        if (this.dir == start) break;
    }
    return {type: "move", direction: this.dir};
};
```

O método `act` só "varre" os arredores do bicho a partir do seu lado esquerdo no sentido horário até encontrar um quadrado vazio. Em seguida ele se move na direção do quadrado vazio.

O que complica é que um bicho pode acabar no meio de um espaço vazio, quer como a sua posição de partida ou como um resultado de caminhar em torno de um outro bicho. Se aplicarmos a abordagem que acabei de descrever no espaço vazio o bicho vai apenas virar à esquerda a cada passo correndo em círculos.

Portanto, há uma verificação extra(instrução `if`) no início da digitalização para a esquerda para analisar se o bicho acaba de passar algum tipo de obstáculo, no caso, se o espaço atrás e à esquerda do bicho não estiver vazio. Caso contrário, o bicho começa a digitalizar diretamente à frente de modo que ele vai andar em linha reta até um espaço vazio.

E finalmente há um teste comparando `this.dir` para começar após cada passagem do laço para se certificar de que o circuito não vai correr para sempre quando o bicho está no muro ou quando o mundo está lotado de outros bichos não podendo achar quadrados vazios.

Este pequeno mundo demonstra as criaturas na parede:

```
animateWorld(new World(
    [ "#####",
      "#   #   #",
      "# ~ ~ ~ #",
      "# ##  #",
      "# ## o####",
      "#      #",
      "#####"],
    { "#": Wall,
      "~": WallFollower,
      "o": BouncingCritter}
));
```

Uma simulação mais realista

Para tornar a vida em nosso mundo mais interessante vamos adicionar os conceitos de alimentação e reprodução. Cada coisa viva no mundo ganha uma nova propriedade, a energia, a qual é reduzida por realizar ações e aumenta comendo algumas coisas. Quando o bicho tem energia suficiente ele pode se reproduzir, gerando um novo bicho da mesma espécie. Para manter as coisas simples; os bichos em nosso mundo se reproduzem assexuadamente ou seja por si só.

Se bichos só se movimentar e comer uns aos outros o mundo em breve irá se sucumbir na lei da entropia crescente, ficando sem energia e tornando um deserto sem vida. Para evitar que isso aconteça(muito rapidamente pelo menos) adicionaremos plantas para o mundo. As plantas não se movem. Eles só usam a fotossíntese para crescer(ou seja aumentar a sua energia) e se reproduzir.

Para fazer este trabalho vamos precisar de um mundo com um método diferente de `letAct`. Poderíamos simplesmente substituir o protótipo global do método mas eu gostei muito da nossa simulação e gostaria que os novos bichos mantivesse o mesmo jeito do velho mundo.

Uma solução é usar herança. Criamos um novo construtor, `LifelikeWorld`, cujo seu protótipo é baseado no protótipo global, mas que substitui o método `letAct`. O novo método `letAct` delega o trabalho do que realmente deve executar uma ação para várias funções armazenados no objeto `actionTypes`.

```
function LifelikeWorld(map, legend) {
  World.call(this, map, legend);
}
LifelikeWorld.prototype = Object.create(World.prototype);

var actionTypes = Object.create(null);

LifelikeWorld.prototype.letAct = function(critter, vector) {
  var action = critter.act(new View(this, vector));
  var handled = action &&
    action.type in actionTypes &&
    actionTypes[action.type].call(this, critter,
                                  vector, action);

  if (!handled) {
    critter.energy -= 0.2;
    if (critter.energy <= 0)
      this.grid.set(vector, null);
  }
};
```

O novo método `letAct` verifica primeiro se uma ação foi devolvido, então se a função manipuladora para este tipo de ação existir, o resultado deste manipulador sera `true`, indicando que ele tratou com sucesso a ação. Observe que usamos uma chamada para dar o acesso ao manipulador do mundo, através de sua chamada. Observe que para dar o acesso ao manipulador no mundo, precisamos fazer uma chamada.

Se a ação não funcionou por algum motivo a ação padrão é que a criatura simplesmente espere. Perde um quinto de sua energia e se o seu nível de energia chega a zero ou abaixo a criatura morre e é removido da `grid`.

Manipuladores de ações

A ação mais simples que uma criatura pode executar é "crescer" e sera usado pelas plantas. Quando um objeto de ação como `{type: "grow"}` é devolvido o seguinte método de manipulação será chamado:

```
actionTypes.grow = function(critter) {
  critter.energy += 0.5;
  return true;
};
```

Crescer com sucesso acrescenta meio ponto no nível total da reserva de energia.

Analise o método para se mover

```
actionTypes.move = function(critter, vector, action) {
  var dest = this.checkDestination(action, vector);
  if (dest == null ||
      critter.energy <= 1 ||
      this.grid.get(dest) != null)
    return false;
  critter.energy -= 1;
  this.grid.set(vector, null);
  this.grid.set(dest, critter);
};
```

```
return true;
};
```

Esta ação verifica primeiro se o destino é válido usando o método `checkDestination`. Se não é válido, se o destino não está vazio ou se o bicho não tem energia necessária; o movimento retorna `false` para indicar que nenhuma ação foi feita. Caso contrário ele move o bicho e subtrai sua energia.

Além de movimentar, os bichos pode comer.

```
actionTypes.eat = function(critter, vector, action) {
    var dest = this.checkDestination(action, vector);
    var atDest = dest != null && this.grid.get(dest);
    if (!atDest || atDest.energy == null)
        return false;
    critter.energy += atDest.energy;
    this.grid.set(dest, null);
    return true;
};
```

Comer um outro bicho também envolve o fornecimento de um quadrado de destino válido. Desta vez o destino não pode estar vazio e deve conter algo com energia, por exemplo um bicho(mas não pode ser a parede pois elas não são comestíveis). Sendo assim a energia a partir da comida é transferido para o comedor e a vítima é retirada da `grid`.

E finalmente nós permitiremos que os nossos bichos se reproduzem.

```
actionTypes.reproduce = function(critter, vector, action) {
    var baby = elementFromChar(this.legend,
                               critter.originChar);
    var dest = this.checkDestination(action, vector);
    if (dest == null ||
        critter.energy <= 2 * baby.energy ||
        this.grid.get(dest) != null)
        return false;
    critter.energy -= 2 * baby.energy;
    this.grid.set(dest, baby);
    return true;
};
```

Reproduzir custa duas vezes mais o nível de energia de um bicho recém-nascido. Então primeiro criamos o bebê (hipoteticamente) usando `elementFromChar` no próprio caráter origem do bicho. Uma vez que temos um bebê podemos encontrar o seu nível de energia e testar se o pai tem energia suficiente para trazê-lo com sucesso no mundo. Também é exigido um destino válido(vazio).

Se tudo estiver bem o bebê é colocado sobre a `grid` (que já não é hipoteticamente), e a energia é subtraída do pai.

Populando o novo mundo

Agora temos um quadro para simular essas criaturas mais realistas. Poderíamos colocar os bichos do velho mundo para o novo, mas eles só iriam morrer, uma vez que não temos uma propriedade de energia. Então vamos fazer novos elementos. Primeiro vamos escrever uma planta que é uma forma de vida bastante simples.

```
function Plant() {
    this.energy = 3 + Math.random() * 4;
}
Plant.prototype.act = function(context) {
    if (this.energy > 15) {
        var space = context.find(" ");
```

```

    if (space)
        return {type: "reproduce", direction: space};
    }
    if (this.energy < 20)
        return {type: "grow"};
    };

```

As plantas começam com um nível de energia randomizados entre 3 e 7, isso é para que eles não se reproduzam todos no mesmo tempo. Quando a planta atinge nível 15 de energia e não há espaço vazio nas proximidades ela não se reproduz. Se uma planta não pode se reproduzir ele simplesmente cresce até atingir o nível 20 de energia.

Vamos agora definir um comedor de plantas.

```

function PlantEater() {
    this.energy = 20;
}
PlantEater.prototype.act = function(context) {
    var space = context.find(" ");
    if (this.energy > 60 && space)
        return {type: "reproduce", direction: space};
    var plant = context.find("*");
    if (plant)
        return {type: "eat", direction: plant};
    if (space)
        return {type: "move", direction: space};
    };

```

Vamos usar o caractere `*` para representar as plantas, quando algum bichos encontrar eles podem consumir como alimento.

Dando a vida

Agora faremos elementos suficientes para experimentar o nosso novo mundo. Imagine o seguinte mapa sendo um vale gramado com um rebanho de herbívoros em que há algumas pedras e vida vegetal exuberante em todos os lugares.

```

var valley = new LifelikeWorld(
    [ "#####",
      "#####",
      "##   **          **##",
      "##  ***          ** 0 ***",
      "#   ***    0   ##*  *##",
      "#      0      ##**  #",
      "#          ##**  #",
      "#  0      #*    #",
      "#*      **    0  #",
      "###*      ##*   0  **",
      "#####*#####",
      "#####"],
    { "#": Wall,
      "0": PlantEater,
      "*": Plant}
);

```

Vamos ver o que acontece ao executar.

```

animateWorld(valley);

```

Na maioria das vezes as plantas se multiplicam e expandem muito rapidamente, mas em seguida a abundância de alimento provoca uma explosão populacional dos herbívoros que saem para acabar com quase todas as plantas resultando em uma fome em massa dos bichos. Às vezes o ecossistema se recupera e começa outro ciclo. Em outros momentos uma das espécies desaparece completamente. Se é os herbívoros todo o espaço irá ser preenchido por plantas. Se é as plantas os bichos restantes morrem de fome e o vale se torna uma terra desolada. Olha que crueldade da natureza.

Exercícios

Estupidez artificial

Tendo os habitantes do nosso mundo se extinguindo após alguns minutos é uma espécie de deprimente. Para lidar com isso poderíamos tentar criar uma forma mais inteligente para o comedor de plantas.

Há vários problemas óbvios com os nossos herbívoros. Primeiro eles são terrivelmente ganancioso enchendo-se com todas as plantas que veem até que tenham dizimado a vida vegetal local. Em segundo lugar o seu movimento randomizado(lembrar-se que o método `view.find` retorna uma direção aleatória quando múltiplas direções combinar) faz com que eles fique em torno de si e acabe morrendo de fome se não acontecer de haver plantas nas proximidades. E finalmente eles se reproduzem muito rápido o que faz com que os ciclos entre abundância e fome se tornem bastante intensos.

Escrever um novo tipo de bicho que tenta abordar um ou mais desses pontos e substituí-lo para o tipo `PlantEater` no velho no mundo do vale. Veja como é que as tarifas estão. Ajuste um pouco mais se necessário.

```
// Your code here
function SmartPlantEater() {}

animateWorld(new LifelikeWorld(
  [ "#####",
    "#####",
    "##   **               *##",
    "#  *##*               ** 0 *##",
    "#   **   0   ##**   *##",
    "#     0   ##**   #",
    "#           ##**   #",
    "#  0       #*       #",
    "#*         **      0  #",
    "#**        ##**   0  *##",
    "#####   ##**   *##",
    "#####"],
  { "#": Wall,
    "0": SmartPlantEater,
    "*": Plant}
));
```

Dicas:

O problema adiverz podem ser atacados de diversas maneiras. Os bichos pode parar de comer quando atingem um certo nível de energia. Ou eles poderiam comer apenas a cada N voltas(mantendo um contador de voltas desde a sua última refeição em uma propriedade no objeto da criatura). Ou para certificar-se de que as plantas nunca seja extinta totalmente, os animais poderiam se recusar a comer uma planta a menos que tenha pelo menos uma outra planta próxima(usando o método `findAll` no `view`). Uma combinação desta ou alguma estratégia completamente diferente pode funcionar.

Podemos recuperar uma das estratégias do movimento dos bichos em nosso velho mundo para fazer os bichos se moverem de forma mais eficaz. Tanto o comportamento de saltar e o de seguir pela parede mostrou uma gama muito maior de movimento do que a de completamente aleatória.

Fazendo as criaturas mais lentas na reprodução pode ser trivial. Basta aumentar o nível de energia mínima em que se reproduzem. É claro que ao fazer isto o ecossistema ficara mais estável tornando-se também mais chato. Se você tem um rodado cheio de bichos imóveis mastigando um mar de plantas e nunca se reproduzindo torna o ecossistema muito estável. E ninguém quer ver isso.

Predators

Qualquer ecossistema sério tem uma cadeia alimentar mais do que um único link. Faça outro bicho que sobrevive comendo o bicho herbívoro. Você vai notar que a estabilidade é ainda mais difícil de conseguir, agora que há ciclos em vários níveis. Tente encontrar uma estratégia para tornar o ecossistema funcional sem problemas durante pelo menos um curto período.

Uma coisa que vai ajudar é fazer um mundo maior. Desta forma o crescimento da população local ou de bustos são menos propensos a acabar com uma espécie inteiramente e não há espaço para a população relativamente grande de presa necessária para sustentar uma população pequena de predadores.

```
// Your code here
function Tiger() {}

animateWorld(new LifelikeWorld(
  [
    "#####",
    "#          ***          ##",
    "# * @ ##          ##### 00  ##",
    "# * ##          0 0          ***  *",
    "#      ***          #####  *",
    "#      **** *          ***  **",
    "#* ** # * **          #####  **",
    "#* ** #          #          **",
    "#      ##          # 0 # ***  #####",
    "#*          @          #          * 0 #  ##",
    "#*          #          #####  **",
    "###          ***          ***  **",
    "#          0          @          0  #",
    "# *          ## ## ##          ###  *  #",
    "# **          #          *          ##### 0  #",
    "## ** 0 0 # #          *** ***          ##  **",
    "###          #          *****  *** #",
    "#####"],
  {
    "#": Wall,
    "@": Tiger,
    "0": SmartPlantEater, // from previous exercise
    "*": Plant
  }
));
```

Dicas:

Muitos dos mesmos truques que trabalhamos no exercício anterior também se aplicam aqui. Fazer os predadores grandes (lotes de energia) se reproduzirem lentamente é recomendado. Isso vai torná-los menos vulneráveis aos períodos de fome quando os herbívoros estiverem escassos.

Além de manter-se vivo, manter seu estoque de alimentos vivo é o objetivo principal de um predador. Encontrar uma forma de fazer predadores caçarem de forma mais agressiva quando há um grande número de herbívoros e caçarem mais lentamente quando a presa é rara. Os comedores de plantas se movimentam, o simples truque de comer um só quando os outros estão nas proximidades é improvável que funcione, raramente pode acontecer que seu predador morra de fome. Mas você poderia manter o controle de observações nas voltas anteriores; de alguma forma precisamos manter a estrutura de dados nos objetos dos predadores e teremos que basear o seu comportamento no que ele tem visto recentemente.

Bugs e manipulação de erros

“Debugar é duas vezes mais difícil do que escrever código. Portanto, se você escrever código da maneira mais inteligente possível, por definição, você não é inteligente o suficiente para debugá-lo.” — Brian Kernighan and P.J. Plauger, *The Elements of Programming Style*

Yuan-Ma havia escrito um pequeno programa onde utilizou muitas variáveis globais e atalhos que faziam a qualidade do seu código inferior. Lendo o programa, um estudante perguntou: “Você nos avisou para não usar essas técnicas e mesmo assim as encontro no seu programa. Como pode isso?”. O mestre respondeu: “Não há necessidade de se buscar uma mangueira de água quando a casa não está em chamas.” — Master Yuan-Ma, *The Book of Programming*

Programas são pensamentos “cristalizados”. Algumas vezes, esses pensamentos são confusos e erros podem ser inseridos quando convertemos pensamentos em código, resultando em um programa com falhas.

Falhas em um programa são normalmente chamadas de *bugs*, e podem ser causadas por erros inseridos pelo programador ou problemas em outros sistemas que a aplicação interage. Alguns *bugs* são imediatamente aparentes, enquanto outros são sutis e podem ficar escondidos em um sistema por anos.

Muitas vezes os problemas aparecem quando um programa executa de uma forma que o programador não considerou originalmente. Às vezes, tais situações são inevitáveis. Quando o usuário insere um dado inválido, isso faz com que a aplicação fique em uma situação difícil. Tais situações devem ser antecipadas e tratadas de alguma maneira.

Erros do programador

O nosso objetivo é simples quando se trata de erros do programador. Devemos encontrá-los e corrigi-los. Tais erros podem variar entre erros simples que faz o computador reclamar assim que ele tenta executar o programa ou erros sutis causado por uma compreensão errada da lógica do programa levando a resultados incorretos, podendo ser constante ou em apenas algumas condições específicas. Esse último tipo de erros pode levar semanas para ter um diagnóstico correto.

O nível de ajuda que as linguagens oferecem para encontrar os erros variam bastante. Isso não é nenhuma surpresa pois o JavaScript está no “quase não ajuda em nada” no final dessa escala. Algumas linguagens exigem os tipos de todas as suas variáveis e expressões antes mesmo de executar; isso dá a possibilidade do programa nos dizer imediatamente quando um tipo é usado de forma incorreta. JavaScript considera os tipos somente na execução do programa e mesmo assim ele permite que você faça algumas coisas visivelmente absurdas sem dar nenhum tipo de aviso como por exemplo: `x = true "macaco" *`.

Há algumas coisas que o JavaScript não se queixa. Mas escrever um programa que é sintaticamente incorreto faz com que ele nem execute e dispare um erro imediatamente. Existem outras coisas como, chamar algo que não é uma função ou procurar uma propriedade em um valor indefinido, isso causa um erro a ser relatado somente quando o programa entrar em execução e encontrar essa ação que não tem sentido.

Mas muitas das vezes um cálculo absurdo pode simplesmente produzir um `NaN` (não um número) ou um valor indefinido. O programa irá continuar alegremente convencido de que está fazendo algo correto. O erro vai se manifestar somente mais tarde, depois que o valor falso passou por várias funções. Não que isso venha desencadear um erro em tudo, mas isso pode silenciosamente causar uma série de saídas erradas. Encontrar a fonte de tais problemas são considerados difíceis.

O processo de encontrar erros (bugs) nos programas é chamado de depuração.

Modo estrito

JavaScript pode ser feito de uma forma mais rigorosa, permitindo que o modo seja estrito. Para obter esse modo basta inserir uma string `"use strict"` na parte superior de um arquivo ou no corpo de uma função. Veja o exemplo:

```
function canYouSpotTheProblem() {  
  "use strict";  
  for (counter = 0; counter < 10; counter++)  
    console.log("Happy happy");  
}  
  
canYouSpotTheProblem();  
// → ReferenceError: counter is not defined
```

Normalmente, quando você esquece de colocar `var` na frente de sua variável como acontece no exemplo, o JavaScript cria uma variável global para utilizá-la, no entanto no modo estrito um erro é relatado. Isto é muito útil. Porém deve-se notar que isso não funciona quando a variável em questão já existe como uma variável global, isso é apenas para atribuição ou criação.

Outra mudança no modo estrito é que esta ligação tem o valor `undefined` para funções que não são chamadas como métodos. Ao fazer tal chamada fora do modo estrito a referência do objeto é do escopo global. Então se você acidentalmente chamar um método ou um construtor incorretamente no modo estrito o JavaScript produzirá um erro assim que ele tentar ler algo com isso ao invés de seguir trabalhando normalmente com a criação e leitura de variáveis globais no objeto global.

Por exemplo, considere o seguinte código que chama um construtor sem a nova palavra-chave, na qual seu objeto não vai se referir a um objeto recém-construído:

```
function Person(name) { this.name = name; }  
var ferdinand = Person("Ferdinand"); // oops  
console.log(name);  
// → Ferdinand
```

Assim, a falsa chamada para `Person` foi bem sucedida, mas retornou um valor indefinido e criou uma variável global. No modo estrito, o resultado é diferente.

```
"use strict";  
function Person(name) { this.name = name; }  
// Oops, forgot 'new'  
var ferdinand = Person("Ferdinand");  
// → TypeError: Cannot set property 'name' of undefined
```

Somos imediatamente informados de que algo está errado. Isso é útil.

Existe mais coisas no modo estrito. Ele não permite dar a uma função vários parâmetros com o mesmo nome e remove totalmente certas características problemáticas da linguagem.

Em suma, colocando um `"use strict"` no topo do seu programa não irá causar frustrações mas vai ajudar a detectar problemas.

Testando

A linguagem não vai nos ajudar muito a encontrar erros, nós vamos ter que encontrá-los da maneira mais difícil: executando o programa e analisando se o comportamento está correto.

Fazer sempre testes manualmente é uma maneira insana de conduzir-se. Felizmente é possível muitas das vezes escrever um segundo programa que automatiza o teste do seu programa atual.

Como por exemplo, vamos construir um objeto `Vector` :

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}

Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

Vamos escrever um programa para verificar se a nossa implementação do objeto `Vector` funciona como o esperado. Então cada vez que mudarmos a implementação o programa de teste é executado, de modo que fiquemos razoavelmente confiantes de que nós não quebramos nada. Quando adicionarmos uma funcionalidade extra (por exemplo, um novo método) no objeto `Vector` , também devemos adicionar testes para o novo recurso.

```
function testVector() {
  var p1 = new Vector(10, 20);
  var p2 = new Vector(-10, 5);
  var p3 = p1.plus(p2);

  if (p1.x !== 10) return "fail: x property";
  if (p1.y !== 20) return "fail: y property";
  if (p2.x !== -10) return "fail: negative x property";
  if (p3.x !== 0) return "fail: x from plus";
  if (p3.y !== 25) return "fail: y from plus";
  return "everything ok";
}
console.log(testVector());
// → everything ok
```

Escrevendo testes como este tende a parecer um pouco repetitivo e um código estranho. Felizmente existem opções de software que ajudam a construir e executar coleções de testes (suites de teste), fornecendo uma linguagem (na forma de funções e métodos) adequada para expressar os testes e emitir informações informativas de quando um teste falhou. Isto é chamados de estruturas de teste.

Depuração

Você consegue perceber que há algo errado com o seu programa quando ele está se comportando mal ou produzindo erros; o próximo passo é descobrir qual é o problema.

Às vezes é óbvio. A mensagem de erro vai apontar para a linha específica; e se você olhar para a descrição do erro e para linha de código muitas vezes você irá entender o problema.

Mas nem sempre é assim. Às vezes a linha que desencadeou o problema é simplesmente o primeiro lugar onde um valor falso foi produzido e que em outros lugares foi usado de uma forma incorreta ou as vezes não há nenhuma mensagem de erro, apenas um resultado inválido. Se você tentou resolver os exercícios nos capítulos anteriores você provavelmente já experimentou tais situações.

O exemplo seguinte tenta converter um número inteiro para uma cadeia em qualquer base (decimal, binário, e assim por diante), para se livrar do último dígito escolhemos o último dígito repetidamente e em seguida dividimos. Mas a saída produzida sugere que ele tem um bug.

```
function numberToString(n, base) {
  var 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 viu o problema e fingiu por um momento que você não viu. Sabemos que o nosso programa não está funcionando corretamente e queremos descobrir o porquê.

Esta é a hora onde você deve resistir à tentação de começar a fazer mudanças aleatórias no código. Em vez disso pense, analise o que está acontecendo e chegue a uma teoria de por que isso pode estar acontecendo. Então faça observações adicionais para testar esta teoria ou se você ainda não tem uma teoria, faça observações adicionais que podem ajudá-lo.

Colocar algumas chamadas `console.log` estratégicas no programa é uma boa maneira de obter informações adicionais sobre o que o programa está fazendo. Neste caso queremos tomar os `n` valores de 13, 1 até 0. Vamos descrever o seu valor no início do loop.

```
13
1.3
0.13
0.013
...
1.5e-323
```

Certo. Dividindo 13 por 10 não produz um número inteiro. Em vez de `n /= base` o que nós realmente queremos é `n = Math.floor (n / base)` de modo que o número está devidamente deslocando-se para a direita.

Uma alternativa para o uso do `console.log` é usar os recursos de depuração do seu browser. Navegadores modernos vêm com a capacidade de definir um ponto de interrupção em uma linha específica de seu código. Isso fará com que a execução do programa faz uma pausa a cada vez que a linha com o ponto de interrupção é atingido. Isso permite que você inspecione os valores das variáveis nesse ponto. Eu não vou entrar em detalhes aqui pois depuradores diferem de navegador para navegador, mas vale a pena olhar as ferramentas de desenvolvimento do seu navegador e pesquisar na web para obter mais informações. Outra maneira de definir um ponto de interrupção é incluir uma declaração no depurador (que consiste em simplesmente em uma palavra-chave) em seu programa. Se as ferramentas de desenvolvedor do seu navegador estão ativos, o programa fará uma pausa sempre que ele atingir esta declaração e você será capaz de inspecionar o seu estado.

Propagação de erros

Infelizmente nem todos os problemas podem ser evitados pelo programador. Se o seu programa se comunica com o mundo externo de qualquer forma há uma chance da entrada de outros sistemas estarem inválidos ou a comunicação estar quebrada ou inacessível.

Programas simples ou programas que são executados somente sob a sua supervisão pode se dar ao luxo de simplesmente desistir quando esse problema ocorre. Você vai olhar para o problema e tentar novamente. Aplicações "reais" por outro lado espera que nunca falhe. Às vezes a maneira correta é tirar a má entrada rapidamente para que o programe continue funcionando. Em outros casos é melhor informar ao usuário o que deu de errado para depois desistir. Mas em qualquer situação o programa tem de fazer algo rapidamente em resposta ao problema.

Digamos que você tenha uma função `promptInteger` que pede para o usuário um número inteiro e retorna-o. O que ele deve retornar se as entradas do usuário for incorreta?

Uma opção é fazê-lo retornar um valor especial. Escolhas comuns são valores nulos e indefinido.

```
function promptNumber(question) {
  var result = Number(prompt(question, ""));
  if (isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"));
```

Isto é uma boa estratégia. Agora qualquer código que chamar a função `promptNumber` deve verificar se um número real foi lido, e na falha deve de alguma forma recuperar preenchendo um valor padrão ou retornando um valor especial para o seu chamador indicando que ele não conseguiu fazer o que foi solicitado.

Em muitas situações, principalmente quando os erros são comuns e o chamador deve explicitamente tê-las em conta, retornaremos um valor especial, é uma forma perfeita para indicar um erro. Mas essa maneira no entanto tem suas desvantagens. Em primeiro lugar, como a função pode retornar todos os tipos possíveis de valores? Para tal função é difícil encontrar um valor especial que pode ser distinguido a partir de um resultado válido.

O segundo problema é com o retorno de valores especiais, isso pode levar a um código muito confuso. Se um pedaço de código chama a função `promptNumber` 10 vezes, teremos que verificar 10 vezes se nulo foi devolvido. E se a sua resposta ao encontrar nulo é simplesmente retornar nulo, o chamador por sua vez tem que verificar assim por diante.

Exceções

Quando uma função não pode prosseguir normalmente, o que gostaríamos de fazermos é simplesmente parar o que esta sendo feito e saltar imediatamente de volta para o lugar onde devemos lidar com o problema. Isto é o que faz o tratamento de exceção.

As exceções são um mecanismo que torna possível parar o código que é executado com problema disparando (ou lançar) uma exceção que nada mais é que um simples valor. Levantando uma exceção lembra um pouco um retorno super carregado a partir de uma função: ele salta para fora não apenas da função atual mas também fora de todo o caminho de seus interlocutores para a primeira chamada que iniciou a execução atual. Isto é chamado de desenrolamento do `stack`. Você pode se lembrar das chamadas de função do `stack` que foi mencionado no Capítulo 3. Uma exceção é exibida no `stack` indicando todos os contextos de chamadas que ele encontrou.

Se as exceções tivessem um `stack` de uma forma ampliada não seria muito útil. Eles apenas fornecem uma nova maneira de explodir o seu programa. Seu poder reside no fato de que você pode definir "obstáculos" ao longo do seu `stack` para capturar a exceção. Depois você pode fazer alguma coisa com ele no ponto em que a exceção foi pega para que o programa continue em execução.

Aqui está um exemplo:

```
function promptDirection(question) {
  var 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 gerar uma exceção. Para tratar uma exceção basta envolver um pedaço de código em um bloco `try`, seguido pela palavra-chave `catch`. Quando o código no bloco `try` causa uma exceção a ser lançada o bloco `catch` é chamado. O nome da variável (entre parênteses) após captura será vinculado ao valor de exceção. Após o término do bloco `catch` ou do bloco `try` o controle prossegue sob toda a instrução `try/catch`.

Neste caso usaremos o construtor de erro para lançar o nosso valor de exceção. Este é um construtor JavaScript normal que cria um objeto com uma propriedade de mensagem. Em ambientes de JavaScript modernos instâncias deste construtor também coletam informações para o `stack` e sobre chamadas que existia quando a exceção foi criado, o chamado `stack` de rastreamento. Esta informação é armazenada na propriedade do `stack` e pode ser útil ao tentar depurar um problema: ela nos diz a função precisa de onde ocorreu o problema e que outras funções que levou até a chamada onde ocorreu a falha.

Note que se olharmos para função `promptDirection` podemos ignoramos completamente a possibilidade de que ela pode conter erros. Esta é a grande vantagem do tratamento de erros - manipulação de erro no código é necessário apenas no ponto em que o erro ocorre e no ponto onde ele é tratado. Essas funções no meio pode perder tudo sobre ela.

Bem, estamos quase lá.

Limpeza após exceções

Considere a seguinte situação: a função `withContext` quer ter certeza de que durante a sua execução, o contexto de nível superior da variável tem um valor de contexto específico. Depois que terminar ele restaura esta variável para o seu valor antigo.

```
var context = null;

function withContext(newContext, body) {
    var oldContext = context;
    context = newContext;
    var result = body();
    context = oldContext;
    return result;
}
```

Como que o `body` gera uma exceção? Nesse caso, a chamada para `withContext` será exibido no `stack` pela exceção, e o contexto nunca será definido de volta para o seu valor antigo.

O `try` tem mais uma declaração. Eles podem ser seguidos por um `finally` com ou sem o bloco `catch`. O bloco `finally` significa "não importa o que aconteça execute este código depois de tentar executar o código do bloco `try`". Se uma função tem de limpar alguma coisa, o código de limpeza geralmente deve ser colocado em um bloco `finally`.

```
function withContext(newContext, body) {
    var oldContext = context;
    context = newContext;
```

```
try {
  return body();
} finally {
  context = oldContext;
}
```

Note-se que não temos mais o resultado do `context` para armazenar (o que queremos voltar) em uma variável. Mesmo se sair diretamente do bloco `try` o último bloco será executado. Então podemos fazer isso de um jeito mais seguro:

```
try {
  withContext(5, function() {
    if (context < 10)
      throw new Error("Not enough context!");
  });
} catch (e) {
  console.log("Ignoring: " + e);
}
// → Ignoring: Error: Not enough context!

console.log(context);
// → null
```

Mesmo que a chamada da função `withContext` explodiu, `withContext` limpou corretamente a variável `context`.

Captura seletiva

Quando uma exceção percorre todo o caminho até o final do `stack` sem ser pego, ele é tratado pelo `environment`. Significa que isto é diferente entre os ambientes. Nos navegadores uma descrição do erro normalmente é escrita no console do JavaScript (alcançável através de "Ferramentas" do navegador no menu de "developer").

Erros passam muitas vezes como algo normal, isto acontece para erros do programador ou problemas que o browser não consegue manipular o erro. Uma exceção sem tratamento é uma forma razoável para indicar a um programa que ele esta quebrado e o console JavaScript em navegadores modernos terá que fornecer-lhe algumas informações no `stack` sobre quais foram as chamadas de funções quando o problema ocorreu.

Para problemas que se espera que aconteça durante o uso rotineiro chegando como uma exceção e que não seja tratada isso pode não ser uma resposta muito simpática.

Usos incorretos da linguagem como, a referência a uma variável inexistente, propriedade que tem null ou chamar algo que não é uma função também irá resultar em lançamentos de exceções. Essas exceções podem ser capturados como outra qualquer.

Quando um pedaço de código é inserido no bloco `catch`, todos nós sabemos que algo em nosso corpo `try` pode ou vai causar uma exceção. Mas nós não sabemos o que ou qual exceção que sera lançada.

O JavaScript (tem uma omissão gritante) não fornece suporte direto para a captura seletiva exceções: ou você manipula todos ou você trata de algum em específico. Isto torna muito fácil supor que a exceção que você recebe é o que você estava pensando quando escreveu o bloco `catch`.

Mas talvez não seja nenhuma das opções citadas. Alguma outra hipótese pode ser violada ou você pode ter introduzido um erro em algum lugar que está causando uma exceção. Aqui está um exemplo que tentei manter a chamada a função `promptDirection` até que ele receba uma resposta válida:

```
for (;;) {
  try {
    var dir = promptDirection("Where?"); // - typo!
```

```

    console.log("You chose ", dir);
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}

```

O `for (;;)` é a construção de um loop infinito de forma intencionalmente que não para sozinho. Nós quebramos o circuito de fora somente quando uma direção válida é fornecida. Mas a mal escrita do `promptDirection` resultará em um erro de "variável indefinida". O bloco `catch` ignora completamente o seu valor de exceção, supondo que ele sabe qual é o problema ele trata equivocadamente o erro de variável como uma indicação de má entrada. Isso não só causa um loop infinito mas também exibi uma mensagem de erro incorretamente sobre a variável que estamos usando.

Como regra geral não capturamos exceções a menos que tenha a finalidade de monitora-las em algum lugar, por exemplo através de softwares externos conectados à nossa aplicação que indica quando nossa aplicação está caída. E assim mesmo podemos pensar cuidadosamente sobre como você pode estar escondendo alguma informação.

E se quisermos pegar um tipo específico de exceção? Podemos fazer isso através da verificação no bloco `catch` para saber se a exceção que temos é a que queremos, daí então é so lançar a exceção novamente. Mas como que nós reconhecemos uma exceção?

Naturalmente nós poderíamos fazer uma comparação de mensagens de erros. Mas isso é uma forma instável de escrever código pois estaríamos utilizando informações que são destinadas ao consumo humano (a mensagem) para tomar uma decisão programática. Assim que alguém muda (ou traduz) a mensagem o código irá parar de funcionar.

Em vez disso, vamos definir um novo tipo de erro e usar `instanceof` para identificá-lo.

```

function InputError(message) {
  this.message = message;
  this.stack = (new Error()).stack;
}

InputError.prototype = Object.create(Error.prototype);
InputError.prototype.name = "InputError";

```

O `prototype` é feito para derivar-se de `Error.prototype` para que `instanceof Error` retorne `true` para objetos de `InputError`. Nome a propriedade também é dada para tipos de erro padrão (`Error`, `SyntaxError`, `ReferenceError` e assim por diante) para que também se tenha uma propriedade.

A atribuição da propriedade no `stack` tenta deixar o rastreamento do objeto pelo `stacktrace` um pouco mais útil, em plataformas que suportam a criação de um objeto de erro regular pode usar a propriedade de `stack` do objeto para si próprio.

Agora `promptDirection` pode lançar um erro.

```

function promptDirection(question) {
  var result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}

```

E o loop pode ser tratado com mais cuidado.

```

for (;;) {
  try {
    var 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 vai pegar apenas os casos de `InputError` e através disso deixa algumas exceções independentes. Se você introduzir um erro de digitação ou um erro de variável indefinida a aplicação nos avisará.

Asserções

As asserções são ferramentas que auxiliam na verificação da sanidade básica de erros do programador. Considere essa função auxiliar que afirma:

```
function AssertionFailed(message) {
    this.message = message;
}
AssertionFailed.prototype = Object.create(Error.prototype);

function assert(test, message) {
    if (!test)
        throw new AssertionFailed(message);
}

function lastElement(array) {
    assert(array.length > 0, "empty array in lastElement");
    return array[array.length - 1];
}
```

Isso fornece uma maneira compacta de fazer cumprir as expectativas solicitadas para quebrar um programa se a condição descrita não for válida. Por exemplo se a função `lastElement` que busca o último elemento de uma matriz voltar indefinida para matrizes vazias caso a declaração for omitida. Buscar o último elemento de uma matriz vazia não faz muito sentido por isso é quase certeza de que um erro de programação pode acontecer.

As afirmações são maneiras de certificar-se de que erros pode causar falhas e qual o ponto deste erro ao invés de valores sem sentido produzidos silenciosamente que pode acarretar problemas em uma parte do programa a qual não se tem nenhuma relação de onde ocorreu realmente.

Resumo

Erros e má entrada acontecem. Erros de programas precisam ser encontrados e corrigidos. Eles podem tornar-se mais fácil de perceber quando se tem uma suites de testes automatizadas e asserções adicionadas em seu programa.

Problemas causados por fatores fora do controle do programa devem geralmente serem tratados normalmente. Às vezes quando o problema pode ser tratado localmente, valores de retorno especiais é um caminho sensato para monitorá-los. Caso contrário as exceções são preferíveis.

Lançar uma exceção faz com que `stack` de chamadas se desencadeie o bloco `try/catch` até a parte inferior do `stack`. O valor da exceção será capturado pelo bloco `catch` onde podemos verificar se ele é realmente do tipo de exceção esperada e em seguida fazer algo com ela. Para lidar com o fluxo de controle imprevisível causado pelas exceções, o bloco `finally` pode ser utilizado para garantir que um pedaço de código seja sempre executado.

Exercícios

Tente outra vez...

Digamos que você tenha uma função `primitiveMultiply` que em 50 por cento dos casos multiplica dois números e em outros 50 por cento levanta uma exceção do tipo `MultiplicatorUnitFailure`. Escreva uma função que envolva esta função `MultiplicatorUnitFailure` e simplesmente tente até que uma chamada seja bem-sucedida retornando o resultado.

Certifique-se de lidar com apenas as exceções que você está tentando manipular.

```
function MultiplicatorUnitFailure() {}

function primitiveMultiply(a, b) {
  if (Math.random() < 0.5)
    return a * b;
  else
    throw new MultiplicatorUnitFailure();
}

function reliableMultiply(a, b) {
  // Coloque seu código aqui.
}

console.log(reliableMultiply(8, 8));
// → 64
```

Dica

A chamada de `primitiveMultiply` obviamente deve acontecer em um bloco `try`. O bloco `catch` fica responsável para relançar a exceção quando não é uma instância de `MultiplicatorUnitFailure` e garantir que a chamada é repetida quando ele é uma instância de `MultiplicatorUnitFailure`.

Para refazer o processo, você pode usar um `loop` que quebra somente quando a chamada for bem sucedida; veja os exemplos de recursão nos capítulos anteriores e faça o uso; espero que você não tenha uma grande séries de erros na função `primitiveMultiply` pois isso pode extrapolar o `stack` e entrar em loop infinito.

A caixa trancada

Considere o seguinte objeto:

```
var box = {
  locked: true,
  unlock: function() { this.locked = false; },

  lock: function() { this.locked = true; },

  _content: [],

  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};
```

Isto é uma caixa com um cadeado. Dentro dela tem um `array` mas você pode obtê-lo apenas quando a caixa for desbloqueada. Não é permitido acessar a propriedade `_content` diretamente.

Escreva uma função chamada `withBoxUnlocked` que assume o valor da função que é passada por argumento para abrir esta caixa. Execute a função e em seguida garanta que a caixa está bloqueada antes de voltar novamente; não importa se o argumento da função retornou normalmente ou emitiu uma exceção.

```
function withBoxUnlocked(body) {
  // Your code here.
}

withBoxUnlocked(function() {
  box.content.push("gold piece");
});

try {
  withBoxUnlocked(function() {
    throw new Error("Pirates on the horizon! Abort!");
  });
} catch (e) {
  console.log("Error raised:", e);
}

console.log(box.locked);
// → true
```

Para ganhar pontos extras, certifique-se de que chamou `withBoxUnlocked` quando a caixa já estava desbloqueada, pois a caixa deve sempre permanecer desbloqueada.

Dica:

Você provavelmente deve ter adivinhado que este exercício solicita o uso do bloco `finally`. Sua função deve ser destravar a caixa e em seguida chamar a função que vem de argumento dentro da função `withBoxUnlocked`. E no `finally` ele deve travar a caixa novamente.

Para certificar-se de que nós não bloqueamos a caixa quando ela já estava bloqueada verifique no início da função se a mesma verificação é válida para quando a caixa esta desbloqueada e para quando quisermos bloquear ela novamente.

Capítulo 9

Expressões Regulares

"Algumas pessoas, quando confrontadas com um problema, pensam "Eu sei, terei que usar expressões regulares." Agora elas têm dois problemas.

— Jamie Zawinski

"Yuan-Ma disse, 'Quando você serra contra o sentido da madeira, muita força será necessária. Quando você programa contra o sentido do problema, muito código será necessário'

— Mestre Yuan-Ma, The Book of Programming

A maneira como técnicas e convenções de programação sobrevivem e se disseminam, ocorrem de um modo caótico, evolucionário. Não é comum que a mais agradável e brilhante vença, mas sim aquelas que combinam bem com o trabalho e o nicho, por exemplo, sendo integradas com outra tecnologia de sucesso.

Neste capítulo, discutiremos uma dessas tecnologias, expressões regulares. Expressões regulares são um modo de descrever padrões em um conjunto de caracteres. Eles formam uma pequena linguagem à parte, que é incluída no JavaScript (assim como em várias outras linguagens de programação e ferramentas).

Expressões regulares são ao mesmo tempo, extremamente úteis e estranhas. Conhecê-las apropriadamente facilitará muito vários tipos de processamento de textos. Mas a sintaxe utilizada para descrevê-las é ridiculamente enigmática. Além disso, a interface do JavaScript para elas é um tanto quanto desajeitada.

Notação

Uma expressão regular é um objeto. Ele pode ser construído com o construtor `RegExp` ou escrito como um valor literal, encapsulando o padrão com o caractere barra (`/`).

```
var expReg1 = new RegExp("abc");  
var expReg2 = /abc/;
```

Este objeto representa um padrão, que no caso é uma letra "a" seguida de uma letra "b" e depois um "c".

Ao usar o construtor `RegExp`, o padrão é escrito como um texto normal, de modo que as regras normais se aplicam para barras invertidas. Na segunda notação, usamos barras para delimitar o padrão. Alguns outros caracteres, como sinais de interrogação (?) e sinais de soma (+), são usados como marcadores especiais em expressões regulares, e precisam ser precedidos por uma barra invertida, para representarem o caractere original e não o comando de expressão regular.

```
var umMaisum = /1 \+ 1/;
```

Saber exatamente quais caracteres devem ser escapados com uma barra invertida em uma expressão regular exige que você saiba todos os caracteres especiais e seus significados na sintaxe de expressões regulares. Por enquanto, pode não parecer fácil saber todos, então, se tiver dúvidas, escape todos os caracteres que não sejam letras e números ou um espaço em branco.

Testando por correspondências

Expressões regulares possuem vários métodos. O mais simples é `test`, onde dado um determinado texto, ele retorna um booleano que informa se o padrão fornecido na expressão foi encontrado nesse texto.

```
console.log( /abc/.test("abcde") );  
// → true  
console.log( /abc/.test("12345") );  
// → false
```

Uma expressão regular que contenha apenas caracteres simples, representa essa mesma sequência de caracteres. Se "abc" existe em qualquer lugar (não apenas no início) do texto testado, o resultado será verdadeiro.

Encontrando um conjunto de caracteres

Saber quando uma `_string_` contém "abc" pode muito bem ser feito usando a função `indexOf`. A diferença das expressões regulares é que elas permitem padrões mais complexos de busca.

Digamos que queremos achar qualquer número. Em uma expressão regular, colocar um conjunto de caracteres entre colchetes ("[]") faz com que a expressão ache qualquer dos caracteres dentro dos colchetes.

A expressão abaixo, acha todas as strings que contem um dígito numérico.

```
console.log( /[0123456789]/.test("ano 1992") );  
// → true  
console.log( /[0-9]/.test("ano 1992") );  
// → true
```

Dentro de colchetes, um hífen ("-") entre dois caracteres pode ser usado para indicar um conjunto entre dois caracteres. Uma vez que os códigos de caracteres Unicode de "0" a "9" contém todos os dígitos (códigos 48 a 57), `[0-9]` encontrará qualquer dígito.

Existem alguns grupos de caracteres de uso comum, que já possuem atalhos incluídos na sintaxe de expressões regulares. Dígitos são um dos conjuntos que você pode escrever usando um atalho, barra invertida seguida de um "d" minúsculo (`\d`), com o mesmo significado que `[0-9]`.

```
- \d    caracteres numéricos  
- \w    caracteres alfanuméricos ("letras")  
- \s    espaços em branco (espaço, tabs, quebras de linha e similares)  
- \D    caracteres que não são dígitos  
- \W    caracteres não alfanuméricos  
- \S    caracteres que não representam espaços  
- . (ponto)  todos os caracteres, exceto espaços
```

Para cada um dos atalhos de conjuntos de caracteres, existe uma variação em letra maiúscula que significa o exato oposto.

Então você pode registrar um formato de data e hora como "30/01/2003 15:20" com a seguinte expressão:

```
var dataHora = /\d\d\/\d\d\/\d\d\d\d \d\d:\d\d/;  
console.log( dataHora.test("30/01/2003 15:20") );  
// → true  
console.log( dataHora.test("30/jan/2003 15:20") );  
// → false
```

Parece confuso, certo? Muitas barras invertidas, sujando a expressão, que dificultam compreender qual o padrão procurado. Mas é assim mesmo o trabalho com expressões regulares.

Estes marcadores de categoria também podem ser usados dentro de colchetes, então `[d.]` significa qualquer dígito ou ponto.

Para "inverter" um conjunto de caracteres, buscar tudo menos o que você escreveu no padrão, um cento circunflexo ("`^`") é colocado no início do colchete de abertura.

```
var naoBinario = /^[^01]/;
console.log( naoBinario.test("01101") );
// → false
console.log( naoBinario.test("01201") );
// → true
```

Partes repetidas em um padrão

Já aprendemos a encontrar um dígito, mas o que realmente queremos é encontrar um número, uma sequência de um ou mais dígitos.

Quando se coloca um sinal de mais ("`+`") depois de algo em uma expressão regular, indicamos que pode existir mais de um. Então `/d+/` encontra um ou mais dígitos.

```
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 similar, mas também permite não encontrar o padrão. Então, algo com um asterisco depois não impede um padrão de ser achado, apenas retornando zero resultados.

Uma interrogação ("`?`") define uma parte do padrão de busca como "opcional", o que significa que ele pode ocorrer zero ou uma vez. Neste exemplo, é permitido que ocorra o caractere "u", mas o padrão também é encontrado quando ele está ausente.

```
var neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

Para permitir que um padrão ocorra um número definido de vezes, chaves ("`{}`") são usadas. Colocando `{4}` depois de um elemento do padrão, mostra que ele deve ocorrer 4 vezes, exatamente. Da mesma maneira, `{2,4}` é utilizado para definir que ele deve aparecer no mínimo 2 vezes e no máximo 4.

Segue outra versão do padrão mostrado acima, de data e hora. Ele permite, dias com um dígito, mês e hora como números e mais legível:

```
var dataHora = /\d{1,2}\d{1,2}\d{4} \d{1,2}:\d{2}/;
console.log( dataHora.test("30/1/2003 8:45") );
// → true
```

Também é possível deixar em aberto o número mínimo ou máximo de ocorrências, omitindo o número correspondente. Então `{,5}` significa que deve ocorrer de 0 até 5 vezes e `{5,}` significa que deve ocorrer cinco ou mais vezes.

Agrupando subexpressões

Para usar um operador como "*" ou "+" em mais de um caractere de uma vez, é necessário o uso de parênteses. Um pedaço de uma expressão regular que é delimitado por parênteses conta como uma única unidade, assim como os operadores aplicados a esse pedaço delimitado.

```
var cartoonCrying = /boo+(hoo+)/i;  
console.log( cartoonCrying.test("Boohooooohoooo") );  
// → true
```

O terceiro "+" se aplica a todo grupo (hoo+), encontrando uma ou mais sequências como essa.

O "i" no final da expressão do exemplo acima faz com que a expressão regular seja case-insensitive, permitindo-a encontrar a letra maiúscula "B" na `_string_dada`, mesmo que a descrição do padrão tenha sido feita em letras minúsculas.

Resultados e grupos

O método `test` é a maneira mais simples de encontrar correspondências de uma expressão regular. Ela apenas informa se foi encontrado algo e mais nada. Expressões regulares também possuem o método `exec` (executar), que irá retornar `null` quando nenhum resultado for encontrado, e um objeto com informações se encontrar.

```
var match = /\d+/.exec("one two 100");  
console.log(match);  
// → ["100"]  
console.log(match.index);  
// → 8
```

Valores `_string_` possuem um método que se comporta de maneira semelhante.

```
console.log("one two 100".match(/\d+/));  
// → ["100", index: 8, input: "one two 100"]
```

Um objeto retornado pelo método `exec` ou `match` possui um `index` de propriedades que informa aonde na `_string_` o resultado encontrado se inicia. Além disso, o objeto se parece (e de fato é) um array de strings, onde o primeiro elemento é a `_string_` que foi achada, no exemplo acima, a sequência de dígitos numéricos.

Quando uma expressão regular contém expressões agrupadas entre parênteses, o texto que corresponde a esses grupos também aparece no array. O primeiro elemento sempre é todo o resultado, seguido pelo resultado do primeiro grupo entre parênteses, depois o segundo grupo e assim em diante.

```
var textoCitado = /'([^']*)' /;  
console.log( textoCitado.exec("'ela disse adeus'") );  
// → ["'ela disse adeus'", "ela disse adeus", index: 0, input: "'ela disse adeus'"]
```

Quando um grupo não termina sendo achado (se por exemplo, possui um sinal de interrogação depois dele), seu valor no array de resultado será `undefined`. Do mesmo modo, quando um grupo é achado várias vezes, apenas o último resultado encontrado estará no array.

```
console.log(/bad(ly)?/.exec("bad"));  
// → ["bad", undefined]  
console.log(/(\d)+/.exec("123"));  
// → ["123", "3"]
```

Grupos podem ser muito úteis para extrair partes de uma string. Por exemplo, podemos não querer apenas verificar quando uma `_string_` contém uma data, mas também extraí-la, e construir um objeto que a representa. Se adicionarmos parênteses em volta do padrão de dígitos, poderemos selecionar a data no resultado da função `exec`.

Mas antes, um pequeno desvio.

O tipo *data*

O JavaScript possui um objeto padrão para representar datas, ou melhor, pontos no tempo. Ele é chamado *Date*. Se você simplesmente criar uma data usando *new*, terá a data e hora atual.

```
console.log( new Date() );  
// → Fri Feb 21 2014 09:39:31 GMT-0300 (BRT)
```

Também é possível criar um objeto para uma hora específica

```
console.log( new Date(2014, 6, 29) );  
// → Tue Jul 29 2014 00:00:00 GMT-0300 (BRT)  
console.log( new Date(1981, 6, 29, 18, 30, 50) );  
// → Wed Jul 29 1981 18:30:50 GMT-0300 (BRT)
```

O JavaScript utiliza uma convenção onde a numeração dos meses se inicia em zero (então Dezembro é 11), mas os dias iniciam-se em um. É bem confuso, então, tenha cuidado.

Os últimos quatro argumentos (horas, minutos, segundos e milissegundos) são opcionais, e assumem o valor de zero se não forem fornecidos.

Internamente, objetos do tipo data são armazenados como o número de milissegundos desde o início de 1970. Usar o método *getTime* em uma data retorna esse número, e ele é bem grande, como deve imaginar.

```
console.log( new Date(2014, 2, 21).getTime() );  
// → 1395370800000  
console.log( new Date( 1395370800000 ) );  
// → Fri Mar 21 2014 00:00:00 GMT-0300 (BRT)
```

Quando fornecemos apenas um argumento ao construtor do *Date*, ele é tratado como se fosse um número de milissegundos.

Objetos *Date* possuem métodos como *getFullYear* (*getYear* retorna apenas os inúteis dois últimos dígitos do ano), *getMonth*, *getDate*, *getHours*, *getMinutes* e *getSeconds* para extrair os componentes da data.

Então agora, ao colocar parênteses em volta das partes que nos interessam, podemos facilmente extrair uma data de uma *string*.

```
function buscaData(string) {  
  var dateTime = /(\d{1,2})\./(\d{1,2})\./(\d{4})/;  
  var match = dateTime.exec(string);  
  return new Date( Number(match[3]), Number(match[2] ), Number(match[1]) );  
}  
console.log( buscaData("21/1/2014") );  
// → Fri Feb 21 2014 00:00:00 GMT-0300 (BRT)
```

Limites de palavra e *string*

A função *buscaData* acima irá extrair facilmente a data de um texto como "100/1/30000", um resultado pode acontecer em qualquer lugar da *string* fornecida, então, nesse caso, vai encontrar no segundo caractere e terminar no último

Se quisermos nos assegurar que a busca seja em todo o texto, podemos adicionar os marcadores "^" e "\$". O primeiro acha o início da *string* fornecida e o segundo o final dela. Então `/^d+$/` encontra apenas em uma *string* feita de um ou mais dígitos, `/^!/` encontra qualquer *string* que começa com sinal de exclamação e `/x^/` não acha nada (o início de uma *string* não pode ser depois de um caractere).

Se, por outro lado, queremos ter certeza que a data inicia e termina no limite da palavra, usamos o marcador `\b`. Um limite de palavra é um ponto onde existe um caractere de um lado e um caractere que não seja de palavra de outro.

```
console.log( /cat/.test("concatenate") );
// → true
console.log( /\bcat\b/.test("concatenate") );
// → false
```

Note que esses marcadores de limite não cobrem nenhum caractere real, eles apenas asseguram que o padrão de busca irá achar algo na posição desejada, informada nos marcadores.

Alternativas

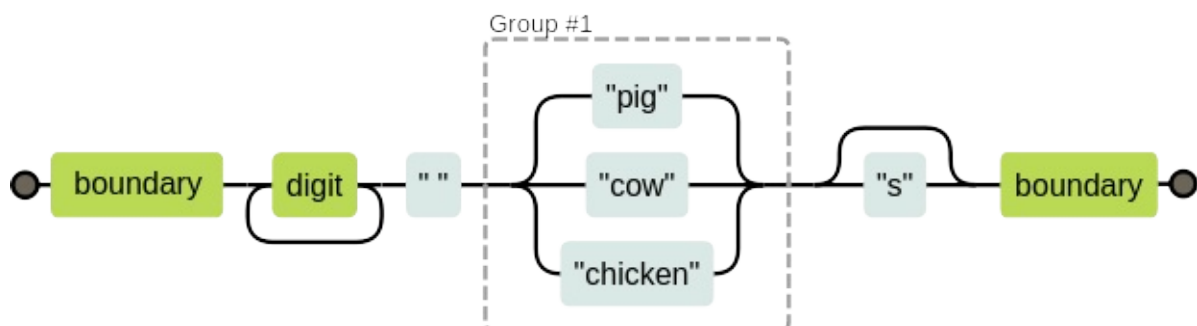
Agora, queremos saber se um pedaço do texto contém não apenas um número, mas um número seguido por uma das palavras "porco", "vaca", "galinha" ou seus plurais também.

Podemos escrever três expressões regulares, e testar cada uma, mas existe uma maneira mais simples. O caractere pipe ("|") indica uma opção entre o padrão à esquerda ou a direita. Então podemos fazer:

```
var contagemAnimal = /\b\d+ (porco|vaca|galinha)s?\b/;
console.log( contagemAnimal.test("15 porcos") );
// → true
console.log( contagemAnimal.test("15 porcosgalinhas") );
// → false
```

Parênteses podem ser usados para limitar a que parte do padrão que o pipe ("|") se aplica, e você pode colocar vários desses operadores lado a lado para expressar uma escolha entre mais de dois padrões.

O mecanismo de procura



Uma *string* corresponde à expressão se um caminho do início (esquerda) até o final (direita) do diagrama puder ser encontrado, com uma posição inicial e final correspondente, de modo que cada vez que passar em uma caixa, verificamos que a posição atual na sequência corresponde ao elemento descrito nela, e, para os elementos que correspondem caracteres reais (menos os limites de palavra), continue no fluxo das caixas.

Então se encontrarmos "the 3 pigs" existe uma correspondência entre as posições 4 (o dígito "3") e 10 (o final da string).

- Na posição 4, existe um limite de palavra, então passamos a primeira caixa
- Ainda na posição 4, encontramos um dígito, então ainda podemos passar a primeira caixa.
- Na posição 5, poderíamos voltar para antes da segunda caixa (dígitos), ou avançar através da caixa que contém um único caractere de espaço. Há um espaço aqui, não um dígito, por isso escolhemos o segundo caminho.
- Estamos agora na posição 6 (o início de "porcos") e na divisão entre três caminhos do diagrama. Nós não temos "vaca" ou "galinha" aqui, mas nós temos "porco", por isso tomamos esse caminho.
- Na posição 9, depois da divisão em três caminhos, poderíamos também ignorar o "s" e ir direto para o limite da palavra, ou achar o "s" primeiro. Existe um "s", não um limite de palavra, então passamos a caixa de "s".
- Estamos na posição 10 (final da string) e só podemos achar um limite de palavra. O fim de uma *string* conta como um limite de palavra, de modo que passamos a última caixa e achamos com sucesso a busca.

O modo como o mecanismo de expressões regulares do JavaScript trata uma busca em uma *string* é simples. Começa no início da *string* e tenta achar um resultado nela. Nesse caso, existe um limite de palavra aqui, então passamos pela primeira caixa, mas não existe um dígito, então ele falha na segunda caixa. Continua no segundo caractere da *string* e tenta novamente. E assim continua, até encontrar um resultado ou alcançar o fim da *string* e concluir que não encontrou nenhum resultado

Retrocedendo

A expressão regular `/b([01]+b|d+|[da-f]h)\b/` encontra um número binário seguido por um "b", um número decimal, sem um caractere de sufixo, ou um número hexadecimal (de base 16, com as letras "a" a "f" para os algarismos de 10 a 15), seguido por um "h". Este é o diagrama equivalente:

http://eloquentJavaScript.net/2nd_edition/preview/img/re_number.svg

Ao buscar esta expressão, muitas vezes o ramo superior será percorrido, mesmo que a entrada não contenha realmente um número binário. Quando busca a *string* "103", é apenas no "3" que torna-se claro que estamos no local errado. A expressão é buscada não apenas no ramo que se está executando.

É o que acontece se a expressão retroage. Quando entra em um ramo, ela guarda em que ponto aconteceu (nesse caso, no início da *string*, na primeira caixa do diagrama), então ela retrocede e tenta outro ramo do diagrama se o atual não encontra nenhum resultado. Então para a *string* "103", após encontrar o caractere "3", ela tentará o segundo ramo, teste de número decimal. E este, encontra um resultado.

Quando mais de um ramo encontra um resultado, o primeiro (na ordem em que foi escrito na expressão regular) será considerado.

Retroceder acontece também, de maneiras diferentes, quando buscamos por operadores repetidos. Se buscamos `/^x/` em "abcxe", a parte "." tentará achar toda a *string*. Depois, tentará achar apenas o que for seguido de um "x", e não existe um "x" no final da *string*. Então ela tentará achar desconsiderando um caractere, e outro, e outro. Quando acha o "x", sinaliza um resultado com sucesso, da posição 0 até 4.

É possível escrever expressões regulares que fazem muitos retrocessos. O Problema ocorre quando um padrão encontra um pedaço da *string* de entrada de muitas maneiras. Por exemplo, se confundimos e escrevemos nossa expressão regular para achar binários e números assim `/([01]+)+b/`.

http://eloquentJavaScript.net/2nd_edition/preview/img/re_slow.svg

Ela tentará achar séries de zeros sem um "b" após elas, depois irá percorrer o circuito interno até passar por todos os dígitos. Quando perceber que não existe nenhum "b", retorna uma posição e passa pelo caminho de fora mais uma vez, e de novo, retrocedendo até o circuito interno mais uma vez. Continuará tentando todas as rotas possíveis

através destes dois *loops*, em todos os caracteres. Para *strings* mais longas o resultado demorará praticamente para sempre.

O método *replace*

Strings possuem o método *replace*, que pode ser usado para substituir partes da *string* com outra *string*

```
console.log("papa".replace("p", "m"));  
// → mapa
```

O primeiro argumento também pode ser uma expressão regular, que na primeira ocorrência de correspondência será substituída.

```
console.log("Borobudur".replace(/ou/, "a"));  
// → Barobudur  
console.log("Borobudur".replace(/ou/g, "a"));  
// → Barabadar
```

Quando a opção "g" ("global") é adicionada à expressão, todas as ocorrências serão substituídas, não só a primeira.

Seria melhor se essa opção fosse feita através de outro argumento, em vez de usar a opção própria de uma expressão regular. (Este é um exemplo de falha na sintaxe do JavaScript)

A verdadeira utilidade do uso de expressões regulares com o método *replace* é a opção de fazer referências aos grupos achados através da expressão. Por exemplo, se temos uma *string* longa com nomes de pessoas, uma por linha, no formato "Sobrenome, Nome" e queremos trocar essa ordem e remover a vírgula, para obter o formato "Nome Sobrenome", podemos usar o seguinte código:

```
console.log("Hopper, Grace\nMcCarthy, John\nRitchie, Dennis".replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));  
// → Grace Hopper  
//   John McCarthy  
//   Dennis Ritchie
```

O "\$1" e "\$2" na *string* de substituição referem-se as partes entre parênteses no padrão. "\$1" será substituído pelo texto achado no primeiro grupo entre parênteses e "\$2" pelo segundo, e assim em diante, até "\$9".

Também é possível passar uma função, em vez de uma *string* no segundo argumento do método *replace*. Para cada substituição, a função será chamada com os grupos achados (assim como o padrão) como argumentos, e o valor retornado pela função será inserido na nova *string*.

Segue um exemplo simples:

```
var s = "the cia and fbi";  
console.log(s.replace(/\b(fbi|cia)\b/g, function(str) {  
    return str.toUpperCase();  
}));  
// → the CIA and FBI
```

E outro exemplo:

```
var stock = "1 lemon, 2 cabbages, and 101 eggs";  
function minusOne(match, amount, unit) {  
    amount = Number(amount) - 1;  
    if (amount == 1) // only one left, remove the 's'  
        unit = unit.slice(0, unit.length - 1);  
    else if (amount == 0)  
        unit = unit.slice(0, unit.length - 1);  
    return amount + unit;  
}
```

```

    amount = "no";
    return amount + " " + unit;
  }
  console.log(stock.replace(/(\d+) (\w+)/g, minusOne));
  // → no lemon, 1 cabbage, and 100 eggs

```

Ele pega a *string*, acha todas as ocorrências de um número seguido por uma palavra alfanumérica e retorna uma nova *string* onde cada achado é diminuído em um.

O grupo `(\d+)` finaliza o argumento da função e o `(\w+)` limita a unidade. A função converte o valor em um número, desde que achado, `\d+` faz ajustes caso exista apenas um ou zero esquerda.

Quantificador / Greed

É simples usar o método *replace* para escrever uma função que remove todos os comentários de um pedaço de código JavaScript. Veja uma primeira tentativa

```

function stripComments(code) {
  return code.replace(/\/\//.*|\/\/*[\w\W]*\*\/\//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 `[wW]` é uma maneira (feia) de encontrar qualquer caractere. Lembre-se que um ponto não encontra um caractere de quebra de linha / linha nova. Comentários podem conter mais de uma linha, então não podemos usar um ponto aqui. Achar algo que seja ou não um caractere de palavra, irá encontrar todos os caracteres possíveis.

Mas o resultado do último exemplo parece errado. Porque?

A parte `."` da expressão, como foi escrita na seção "Retrocedendo", acima, encontrará primeiro tudo que puder e depois, se falhar, volta atrás e tenta mais uma vez a partir daí. Nesse caso, primeiro procuramos no resto da *string* e depois continuamos a partir daí. Encontrará uma ocorrência de `"!"` depois volta quatro caracteres e acha um resultado. Isto não era o que desejávamos, queríamos um comentário de uma linha, para não ir até o final do código e encontrar o final do último comentário.

Existem duas variações de operadores de repetição em expressões regulares (`'+'`, `'*'`, e `'{ }'`). Por padrão, eles quantificam, significa que eles encontram o que podem e retrocedem a partir daí. Se você colocar uma interrogação depois deles, eles se tornam *non_greedy*, e começam encontrando o menor grupo possível e o resto que não contenha o grupo menor.

E é exatamente o que queremos nesse caso. Com o asterisco encontramos os grupos menores que tenham `"*/"` no fechamento, encontramos um bloco de comentários e nada mais.

```

function stripComments(code) {
  return code.replace(/\/\//.*|\/\/*[\w\W]*?\/\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1

```

Criando objetos RegExp dinamicamente

Existem casos onde você pode não saber o padrão exato que você precisa quando escreve seu código. Digamos que você queira buscar o nome de um usuário em um pedaço de texto e colocá-lo entre caracteres "_" para destacá-lo. O nome será fornecido apenas quando o programa estiver sendo executado, então não podemos usar a notação de barras para criar nosso padrão.

Mas podemos construir uma *string* e usar o construtor *RegExp* para isso. Por exemplo:

```
var name = "harry";
var text = "Harry is a suspicious character.";
var regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Harry_ is a suspicious character.
```

Ao criar os marcos de limite "\b", usamos duas barras invertidas, porque estamos escrevendo-os em uma *string* normal, não uma expressão regular com barras. As opções (global e case-insensitive) para a expressão regular podem ser inseridas como segundo argumento para o construtor *RegExp*.

Mas e se o nome for "dea+hl[]rd" porque o usuário é um adolescente nerd? Isso irá gerar uma falsa expressão regular, por conter caracteres comando, que irá gerar um resultado estranho

Para contornar isso, adicionamos contrabarras antes de qualquer caractere que não confiamos. Adicionar contrabarras antes de qualquer caractere alfabético é uma má idéia, porque coisas como "\b" ou "\n" possuem significado para uma expressão regular. Mas escapar tudo que não for alfanumérico ou espaço é seguro.

```
var name = "dea+hl[]rd";
var text = "This dea+hl[]rd guy is quite annoying.";
var escaped = name.replace(/[\^w\s]/g, "\\$&");
var regexp = new RegExp("\\b(" + escaped + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → This _dea+hl[]rd_ guy is quite annoying.
```

O marcador "\$&" na *string* de substituição age como se fosse "\$1", mas será substituído em todos os resultados ao invés do grupo encontrado.

O método *search*

O método *indexOf* em *strings* não pode ser invocado com uma expressão regular. Mas existe um outro método, *search*, que espera como argumento uma expressão regular, e como o *indexOf*, retorna o índice do primeiro resultado encontrado ou -1 se não encontra.

```
console.log(" word".search(/S/));
// → 2
console.log(" ".search(/S/));
// → -1
```

Infelizmente, não existe um modo de indicar onde a busca deve começar, com um índice (como o segundo argumento de *indexOf*), o que seria muito útil.

A propriedade *lastIndex*

O método *exec* também não possui um modo conveniente de iniciar a busca a partir de uma determinada posição. Mas ele fornece um método não muito prático.

Expressões regulares possuem propriedades (como *source* que contém a *string* que originou a expressão). Uma dessas propriedades, *lastIndex*, controla, em algumas circunstâncias, onde a busca começará.

Essas circunstâncias são que a expressão regular precisa ter a opção "global" (g) habilitada e precisa ser no método *exec*. Novamente, deveria ser da mesma maneira que permitir um argumento extra para o método *exec*, mas coesão não é uma característica que define a sintaxe de expressões regulares em JavaScript

```
var pattern = /y/g;
pattern.lastIndex = 3;
var match = pattern.exec("xyzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

A propriedade *lastIndex* é atualizada ao ser executada após encontrar algo. Quando não encontra nada, *lastIndex* é definida como zero, que também é o valor quando uma nova expressão é construída.

Quando usada uma expressão regular global para múltiplas chamadas ao método *exec*, esta mudança da propriedade *lastIndex* pode causar problemas, sua expressão pode iniciar por acidente em um índice deixado na ultima vez que foi executada.

Outro efeito interessante da opção global é que ela muda a maneira como o método *match* funciona em uma *string*. Quando chamada com uma expressão global, em vez de retornar um array semelhante ao retornado pelo *exec*, *match* encontrará todos os resultados do padrão na *string* e retornará um array contendo todas as *strings* encontradas.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Então tenha cuidado com expressões regulares globais. Os casos em que são necessárias - chamadas para substituir e lugares onde você deseja usar explicitamente *lastIndex* - normalmente são os únicos lugares onde você deseja utilizá-las.

Um padrão comum é buscar todas as ocorrências de um padrão em uma *string*, com acesso a todos os grupos encontrados e ao índice onde foram encontrados, usando *lastIndex* e *exec*.

```
var input = "A text with 3 numbers in it... 42 and 88.";
var re = /\b(\d+)\b/g;
var match;
while (match = re.exec(input))
    console.log("Found", match[1], "at", match.index);
// → Found 3 at 12
//   Found 42 at 31
//   Found 88 at 38
```

Usa-se o fato que o valor de uma expressão de definição (‘=’) é o valor assinalado. Então usando-se `match = re.exec(input)` como a condição no bloco `while`, podemos buscar no início de cada iteração.

Analizando um arquivo .ini

Agora vamos ver um problema real que pede por uma expressão regular. Imagine que estamos escrevendo um programa que coleta informação automaticamente da internet dos nossos inimigos. (Não vamos escrever um programa aqui, apenas a parte que lê o arquivo de configuração, desculpe desapontá-los). Este arquivo tem a seguinte aparência:

```

searchengine=http://www.google.com/search?q=$1
spitefulness=9.7

; comments are preceded by a semicolon...
; these are sections, concerning individual enemies
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[gargamel]
fullname=Gargamel
type=evil sorcerer
outputdir=/home/marijn/enemies/gargamel

```

As regras exatas desse formato (que é um formato muito usado, chamado arquivo .ini) são as seguintes:

- Linhas em branco e linhas iniciadas com ponto e vírgula são ignoradas.
- Linhas entre colchetes "[" iniciam uma nova seção.
- Linhas contendo um identificador alfanumérico seguido por um caractere = adicionam uma configuração à seção atual.
- Qualquer outra coisa é inválida.

Nossa tarefa é converter uma *string* como essa em um *array* de objetos, cada uma com um nome e um *array* de pares nome/valor. Precisaremos de um objeto para cada seção e outro para as configurações de seção.

Já que o formato precisa ser processado linha a linha, dividir em linhas separadas é um bom começo. Usamos o método *split* antes para isso, *string.split("\n")*. Entretanto alguns sistemas operacionais não usam apenas um caractere de nova linha para separar linhas, mas um caractere de retorno seguido por um de nova linha ("\r\n").

Desse modo o método *split*, em uma expressão regular com */r?\n/* permite separar os dois modos, com "\n" e "\r\n" entre linhas.

```

function parseINI(texto) {
  var categorias = [];
  function novaCategoria(nome) {
    var categ = {nome: nome, fields: []};
    categorias.push(categ);
    return categ;
  }
  var categoriaAtual = novaCategoria("TOP");

  texto.split(/\r?\n/).forEach(function(linha) {
    var encontrados;
    if (/^s*(;.*)?$/.test(linha))
      return;
    else if (encontrados = linha.encontrados(/^\[([.]*)\]$/)
      categoriaAtual = novaCategoria(encontrados[1]);
    else if (encontrados = linha.encontrados(/^(\w+)=([.]*)$/)
      categoriaAtual.fields.push({nome: encontrados[1],
                                value: encontrados[2]});
    else
      throw new Error("Linha '" + linha + "' is invalid.");
  });

  return categorias;
}

```

O código percorre cada linha no arquivo. Ele mantém um objeto "categoria atual", e quando encontra um diretiva normal, adiciona ela ao objeto. Quando encontra uma linha que inicia uma nova categoria, ela troca a categoria atual pela nova, para adicionar as diretivas seguintes. Finalmente, retorna um *array* contendo todas as categorias que encontrou.

Observe o uso recorrente de `^` e `$` para certificar-se que a expressão busca em toda a linha, não apenas em parte dela. Esquecer isso é um erro comum, que resulta um código que funciona mas retorna resultados estranhos para algumas entradas.

A expressão `^\s(:)?$/` pode ser usada para testar linhas que podem ser ignoradas. Entende como funciona? A parte entre parênteses irá encontrar comentários e o `?` depois certificará que também encontrará linhas apenas com espaços em branco.

O padrão `if (encontrados = texto.match(...))` é parecido com o truque que foi usado como definição do `while` antes. Geralmente não temos certeza se a expressão encontrará algo. Mas você só deseja fazer algo com o resultado se ele não for nulo, então você precisa testar ele antes. Para não quebrar a agradável sequência de `ifs` podemos definir o resultado a uma variável para o teste, e fazer a busca e testes em uma única linha.

Caracteres internacionais

Devido a uma implementação inicial simplista e o fato que esta abordagem simplista mais tarde foi gravada em pedra como comportamento padrão, expressões regulares do JavaScript são um pouco estúpidas sobre caracteres que não parecem na língua inglesa. Por exemplo, "caracteres palavra", nesse contexto, atualmente significam apenas os 26 caracteres do alfabeto latino. Coisas como "é" ou "ß", que definitivamente são caracteres de palavras, não encontrarão resultados com `\w` (e serão encontradas com o marcador de letras maiúsculas `\W`).

Devido a um estranho acidente histórico, `\s` (espaço em branco) é diferente, e irá encontrar todos os caracteres que o padrão Unicode considera como espaço em branco, como espaços sem quebra ou o separador de vogais do alfabeto Mongol.

Algumas implementações de expressões regulares em outras linguagens de programação possuem uma sintaxe para buscar conjuntos específicos de caracteres Unicode, como todas as maiúsculas, todos de pontuação, caracteres de controle ou semelhantes. Existem planos para adicionar esse suporte ao JavaScript, mas infelizmente parece que isso não acontecerá tão cedo.

Uma ou mais ocorrências do padrão

Expressões regulares são objetos que representam padrões em *strings*. Eles usam sua própria sintaxe para expressar esses padrões.

```
/abc/      Sequência de caracteres
/[abc]/    Qualquer caractere do conjunto
/[^abc]/   Qualquer caractere que não seja do conjunto
/[0-9]/    Qualquer caractere no intervalo de caracteres
/x+/       Uma ou mais ocorrências do padrão
/x+?/     Uma ou mais ocorrências do padrão, não obrigatório
/x*/       Zero ou mais ocorrências
/x?/       Zero ou uma ocorrência
/x{2,4}/   Entre duas e quatro ocorrências
/(abc)+/   Agrupamento
/a|b|c/    Padrões alternativos
/\d/       Caracteres dígitos
/\w/       Caracteres alfanuméricos ("caracteres palavra")
/\s/       caracteres espaço em branco
/./        Todos caracteres exceto quebras de linha
/\b/       Limite de palavra
/^/        Início da entrada
/$/        Final da Entrada
```

Uma expressão regular possui um método *test* para testar quando um padrão é encontrado em uma *string*, um método *exec* que quando encontra um resultado retorna um *array* com todos os grupos encontrados e uma propriedade *index* que indica onde o resultado inicia.

Strings possuem um método *match* para testá-las contra uma expressão regular e um método *search* para buscar por um resultado. O método *replace* pode substituir resultados encontrados por um padrão. Como alternativa, uma função pode ser passada para montar o texto que será substituído de acordo com que foi achado.

Expressões regulares podem ter opções configuradas (*flags*), que são escritas após o fechamento da barra. A opção "i" faz a busca sem se importar se é maiúscula ou minúscula, a opção "g" faz a busca global, que, entre outras coisas, faz o método *replace* substituir todas as ocorrências, em vez de só a primeira.

O construtor *RegExp* pode ser usado para criar uma expressão regular dinâmica a partir de uma *string*.

Expressões regulares são uma ferramenta precisa mas com um manuseio estranho. Elas simplificarão muito algumas tarefas simples, mas rapidamente se tornarão inviáveis quando aplicadas a tarefas mais complexas. Saber quando usá-las é útil. Parte do conhecimento de saber **quando** usá-las é o conhecimento de saber **como** usá-las e quando desistir do seu uso e procurar uma abordagem mais simples.

Exercícios

É quase inevitável que, no decorrer do trabalho, você irá ficar confuso e frustrado por algum comportamento estranho de uma expressão regular. O que ajuda às vezes é colocar a sua expressão em uma ferramenta online como debuggex.com, para ver se a visualização corresponde à sua intenção inicial, e rapidamente ver como ela responde à várias *strings* diferentes.

Regexp golf

"Golf de Código" é um termo usado para o jogo de tentar escrever um programa com o menor número de caracteres possível. Parecido, o regexp golf é a prática de escrever pequenas expressões regulares para achar um determinado padrão (e apenas esse padrão).

Escreva uma expressão regular que testa quando qualquer das *sub-strings* dadas ocorre em um texto. A expressão regular deverá achar apenas *strings* contendo uma das *sub-strings* dadas. Não se preocupe com limites de palavras a não ser que seja explicitamente pedido. Quando a sua expressão funcionar, veja se consegue fazê-la menor.

```
"car" e "cat"
"pop" e "prop"
"ferret", "ferry", e "ferrari"
Qualquer palavra terminando em "ious"
Um espaço em branco seguido por um ponto, vírgula, dois-pontos, ou ponto-e-vírgula
Uma palavra com mais de seis letras
Uma palavra sem a letra "e"
```

Consulte a tabela no capítulo Sumário para achar algo rapidamente. Teste cada solução encontrada com alguns testes com *strings*.

```
// Fill in the regular expressions

verify(/.../,
  ["my car", "bad cats"],
  ["camper", "high art"]);

verify(/.../,
  ["pop culture", "mad props"],
  ["plop"]);
```



```

verify(/.../,
  ["ferret", "ferry", "ferrari"],
  ["ferrum", "transfer A"]);

verify(/.../,
  ["how delicious", "spacious room"],
  ["ruinous", "consciousness"]);

verify(/.../,
  ["bad punctuation ."],
  ["escape the dot"]);

verify(/.../,
  ["hottentottententen"],
  ["no", "hotten totten tenten"]);

verify(/.../,
  ["red platypus", "wobbling nest"],
  ["earth bed", "learning ape"]);

function verify(regex, yes, no) {
  // Ignore unfinished tests
  if (regex.source == "...") return;
  yes.forEach(function(s) {
    if (!regex.test(s))
      console.log("Failure to match '" + s + "'");
  });
  no.forEach(function(s) {
    if (regex.test(s))
      console.log("Unexpected match for '" + s + "'");
  });
}

```

Estilo de aspas

Imagine que você escreveu um texto e usou aspas simples por toda parte. Agora você deseja substituir todas que realmente possuem algum texto com aspas duplas, mas não as usadas em contrações de texto com `_aren't`).

Pense em um padrão que faça distinção entre esses dois usos de aspas e faça uma chamada que substitua apenas nos lugares apropriados.

```

var text = "I'm the cook," he said, "it's my job.";
// Altere esta chamada
console.log(text.replace(/"/, "B"));
// → "I'm the cook," he said, "it's my job."

```

Dicas

A solução mais óbvia é substituir apenas as aspas que não estão cercadas de caracteres de palavra. A primeira expressão vem à mente é `/\W\W/`, mas é preciso cuidado para lidar com o início da *string* corretamente. Isso pode ser feito usando os marcadores `"^"` e `"$"`, como em `/(W|^)(W|$)/`.

Novamente números

Séries de dígitos podem ser usados pela agradável expressão regular `/d+/`.

Escreva uma expressão que encontre (apenas) números no estilo JavaScript. Isso significa que precisa suportar um sinal de menor ou maior, opcional, na frente do número, um ponto decimal e a notação exponencial —`5e-3` ou `1E10` —, novamente com o sinal opcional na frente dele.

```
// Preencha esta expressão regular
var number = /^...$/;

// Tests:
["1", "-1", "+15", "1.55", ".5", "5.", "1.3e2", "1E-4",
 "1e+12"].forEach(function(s) {
  if (!number.test(s))
    console.log("Falhou em achar '" + s + "'");
});
["1a", "+-1", "1.2.3", "1+1", "1e4.5", ".5.", "1f5",
 "."].forEach(function(s) {
  if (number.test(s))
    console.log("Aceitou erroneamente '" + s + "'");
});
```

Dicas

Primeiro, não esqueça da barra invertida em frente ao ponto.

Achar o sinal opcional na frente do número, como na frente do exponencial, pode ser feito com `[+-]?` ou `(+|-|)` (mais, menos ou nada).

A parte mais complicada deste exercício provavelmente é a dificuldade de achar `"5."` e `".5"` sem achar também o `"."`. Para isso, achamos que a melhor solução é usar o operador `"|"` para separar os dois casos, um ou mais dígitos opcionalmente seguidos por um ponto e zero ou mais dígitos, ou um ponto seguido por um ou mais dígitos.

Finalmente, fazer o `"e"` *case-insensitive*, ou adicional a opção `"i"` à expressão regular ou usar `"[eE]"`.

Capítulo 10

Módulos

Um programador iniciante escreve seus programas como uma formiga constrói seu formigueiro, um pedaço de cada vez, sem pensar na estrutura maior. Seus programas irão parecer como areia solta. Eles podem durar um tempo, mas se crescem demais, desmoronam.

Percebendo esse problema, o programador começará a gastar muito tempo pensando sobre a estrutura. Seus programas serão rigidamente estruturados, como esculturas em pedra. Eles são sólidos, mas quando precisam mudar, devem ser quebrados.

O programador experiente sabe quando aplicar uma estrutura e quando deixar as coisas mais simples. Seus programas são como argila, sólidos mas ainda maleáveis.

—Master Yuan-Ma, The Book of Programming

Todo programa possui uma forma. Em menor escala essa forma é determinada pela divisão em funções e os blocos dentro destas funções. Programadores têm muita liberdade na forma que dão aos seus programas. É determinado mais pelo bom (ou mau) gosto, do que pela funcionalidade planejada.

Quando olhamos um programa grande em seu todo, funções individuais começam a se misturar e seria bom possuir uma unidade maior de organização.

Módulos dividem programas em blocos de código, que por algum critério pertencem a uma mesma unidade. Este capítulo explora alguns dos benefícios que estes agrupamentos fornecem e mostra algumas técnicas para construção de módulos em JavaScript.

Organização

Existem algumas razões porque autores dividem seus livros em capítulos e seções. Elas facilitam para o leitor entender como o livro foi feito ou achar uma parte específica em que está interessado. Elas também ajudam o autor, dando um foco claro para cada seção.

Os benefícios de dividir um programa em vários arquivos ou módulos são semelhantes, ajudam as pessoas que não estão familiarizadas com o código a achar o que elas buscam, e ajudam o programador a colocar coisas semelhantes juntas.

Alguns programas são organizados seguindo o modelo de um texto tradicional, com uma ordem bem definida que encoraja o leitor a percorrer o programa, e muito falatório (comentários) fornecendo uma descrição coerente do código. Isso faz o programa muito menos intimidador (ler código desconhecido é geralmente intimidador). Mas existe um lado ruim que é a maior quantidade de trabalho a fazer e dificulta um pouco as alterações, porque os comentários tendem a ser mais interligados do que o código em si.

Como regra geral, organização tem um custo, e é nos estágios iniciais do projeto, quando não sabemos com certeza aonde vamos e que tipo de módulos o programa precisará. Eu defendo uma estrutura minimalista, com pouca estrutura. Apenas coloque tudo em um simples arquivo até que o código esteja estabilizado. Dessa maneira, você não estará se sobrecarregando pensando em organização enquanto tem pouca informação, não perderá tempo fazendo e desfazendo coisas, e não irá acidentalmente travar-se em uma estrutura que não serve realmente para seu programa.

Namespaces

A maioria das linguagens modernas de programação têm um nível de escopo entre "global" (todos podem ver) e "local" (só esta função pode ver isto). JavaScript não. Assim, por padrão, tudo o que precisa ser visível fora do pequeno escopo da função atual é visível em todos os lugares.

Poluição de Namespace, o problema de um monte de código não relacionado ter que compartilhar um único conjunto de nomes de variáveis globais, foi mencionado no capítulo 4, onde o objeto `Math` foi dado como um exemplo de um objeto que age como uma espécie de módulo por um agrupamento série de funcionalidades relacionadas com a matemática.

Embora JavaScript não possua a criação de módulos nativamente, objetos podem ser usados para criar sub-namespaces publicamente acessíveis, e funções podem ser usadas para criar um namespace privado dentro de um módulo. Vou demonstrar algumas técnicas que nos permitirão construir módulos namespace isolados bem convenientes.

Reuso

Em um projeto "flat" (plano), não é claro quais partes do código são necessárias para se usar uma função em particular. Se, no meu programa para espionar inimigos (*spying on enemies*), eu escrever uma função para ler os arquivos de configuração, e agora eu uso essa função novamente em outro projeto, eu devo ir e copiar as partes do programa antigo que são relevantes para a funcionalidade que eu preciso, e colá-las no meu novo programa. Então, se eu encontrar um erro nesse código, eu vou consertar isso neste programa que eu estava trabalhando no momento, e esquecer de também consertar no outro programa.

Uma vez que você tenha muitos pedaços de código compartilhados e duplicados, você vai se encontrar perdendo uma grande quantidade de tempo e energia organizá-los e mantê-los atualizados.

Quando partes de funcionalidades que são independentes são colocadas em arquivos e módulos separados, elas podem ser rastreadas mais facilmente, atualizadas quando uma nova versão for criada, ou até mesmo compartilhadas, tendo várias partes do código que desejam usá-las carregando o mesmo arquivo.

Essa ideia fica ainda mais poderosa quando as relações entre os módulos - onde outros módulos cada módulo depende - são explicitamente especificados. Você pode então automatizar o processo de instalação e atualização de módulos externos.

E, levando isso ainda mais longe, imagine um serviço online que rastreia e distribui centenas de milhares destes módulos, permitindo a você buscar pela funcionalidade que deseja, e, uma vez que você a encontre, configure-a no seu projeto para ser baixada automaticamente.

Este serviço existe. É chamado NPM ([npmjs.org](https://www.npmjs.org)). NPM consiste em um banco de dados online de módulos, e uma ferramenta para download e atualização dos módulos que seu programa depende. Ele cresceu com o Node.js, o ambiente JavaScript *browser-less* (que não depende do navegador), discutido no capítulo 20, mas também pode ser usado quando programando para o navegador.

Desacoplamento

Outro importante papel dos módulos é os de isolar partes de código um do outro, da mesma forma que as interfaces dos objetos no capítulo 6 fazem. Um módulo bem desenvolvido fornece uma interface para uso de códigos externos, e mesmo que o módulo continue sendo trabalhado (bugs consertados, funcionalidades adicionadas) a interface existente permanece estável, assim outros módulos podem usar uma nova e melhorada versão sem qualquer alteração neles mesmos.

Note que uma interface estável não significa que novos elementos não são adicionados. Isso apenas significa que elementos existentes não serão removidos ou seus significados não serão alterados.

Construir a interface de um módulo que permite que este cresça sem quebras na antiga interface significa encontrar um balanço entre expor a menor quantidade de conceitos internos ao mundo exterior quanto possível, e ainda assim criar uma "linguagem" exposta pela interface que seja poderosa e flexível o suficiente para ser aplicada em uma vasta variedade de situações.

Para interfaces que expõem um único e focado conceito, como um arquivo leitor de configuração, isso é natural. Para as outras interfaces, como um componente editor de texto, onde código externo precisa acessar vários conceitos diferentes, isso requer cuidado no projeto.

Funções como namespaces

Funções são o único construtor em JavaScript que criam um novo escopo. Então se nós desejamos que nossos módulos tenham um escopo próprio, teremos que colocá-los em funções de alguma forma.

Considere este módulo trivial que associa nomes com o número dos dias da semana retornado pelo método `getDay` de um objeto *date*.

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"];
function dayName(number) {
    return names[number];
}

console.log(dayName(1));
// → Monday
```

A função `dayName` é parte desta interface, mas a variável `names` não. Nós preferimos não deixá-la no escopo global.

Podemos fazer isso:

```
var dayName = function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
    return function(number) {
        return names[number];
    };
}();

console.log(dayName(3));
// → Wednesday
```

Agora `names` é uma variável local dentro de uma função (anônima). Esta função é criada e chamada imediatamente, e seu valor retornado (a função `dayName`) é armazenada em uma variável. Podemos ter páginas e mais páginas de código nessa função, criando centenas de variáveis locais. Elas serão todas internas ao módulo, visíveis ao próprio módulo, mas não visível a códigos externos.

Um padrão similar é usado para isolar inteiramente código do mundo exterior. O módulo abaixo tem algum efeito, mas não fornece qualquer valor para outros módulos usarem.

```
(function() {
    function square(x) { return x * x; }
    var hundred = 100;

    console.log(square(hundred));
})();
// → 10000
```

Este código simplesmente imprime o quadrado de cem (no mundo real, este poderia ser um módulo que adiciona um método a algum prototype, ou configura algum *widget* em uma página da web). Ele encapsula seu código em uma função para, novamente, prevenir que as variáveis que ele usa internamente estejam no escopo global.

Por que a função namespace está encapsulada em uma par de parênteses? Isso tem relação com um truque da sintaxe JavaScript. Se uma expressão começa com a palavra-chave `function`, ela é uma expressão de função. Entretanto, se uma declaração inicia com esta palavra-chave, será uma declaração de função, que requer um nome e não pode ser chamada imediatamente. Mesmo que uma declaração comece com uma expressão, a segunda regra tem precedência, e se os parênteses extras foram esquecidos no exemplo acima, isso irá produzir um erro de sintaxe. Você pode imaginá-los como um truque para forçar a linguagem a entender que nós queremos escrever uma expressão.

Objetos como namespaces

Agora imagine que o módulo dia-da-semana (*day-of-the-week*) precise fornecer não uma, mas duas funções, porque nós adicionamos uma função `dayNumber` que vai de um nome para um número. Nós podemos mais simplesmente retornar a função, mas devemos encapsular as duas funções em um objeto.

```
var weekDay = function() {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"];

  return {
    name: function(number) { return names[number]; },
    number: function(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

Para módulos maiores, juntar todos os módulos exportados em um objeto no fim da função se torna algo incômodo, e geralmente requer que façamos algo repetido. Isso pode ser melhorado declarando um objeto, usualmente nomeado `exports`, e adicionando propriedades a este objeto sempre que nós definirmos algo que precise ser exportado. Este objeto pode então ser retornado, ou aceito como um parâmetro armazenado em algum lugar pelo código exterior ao módulo.

```
(function(exports) {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"];

  exports.name = function(number) {
    return names[number];
  };
  exports.number = function(name) {
    return names.indexOf(name);
  };
})(window.weekDay = {});

console.log(weekDay.name(weekDay.number("Saturday")));
// → Saturday
```

Removendo do escopo global

O padrão acima é usado normalmente em módulos JavaScript criados para o navegador. Eles requerem um simples e conhecido nome global, e encapsular seu código em uma função para ter seu namespace privado próprio.

Ainda existe um problema quando múltiplos módulos reivindicam o mesmo nome, ou quando você quer, por qualquer motivo, carregar duas versões do mesmo módulo de forma conjunta.

Com um pequeno encanamento, nós podemos criar um sistema que permite que aos módulos requererem diretamente por interfaces de objetos de outros módulos que eles precisem de acessar, sem precisarmos usar o escopo global. Isso resolve os problemas mencionados acima e tem um benefício adicional de ser explícito sobre suas dependências, tornando difícil usar acidentalmente algum módulo sem declarar que você precisa dele.

Nosso objetivo é uma função 'require' que, quando dado o nome de um módulo, vai carregar esse arquivo (do disco ou da web, dependendo da plataforma que estivermos rodando), e retornar o valor apropriado da interface.

Para isso nós precisamos de pelo menos duas coisas. Primeiramente, nós vamos imaginar que temos uma função `readFile` (que não está presente por padrão no JavaScript), que retorna o conteúdo do arquivo com um nome fornecido. Existem formas de acessar a web com JavaScript no navegador, e acessar o disco rígido com outras plataformas JavaScript, mas elas são mais envolvidas. Por agora, nós apenas pretendemos desta simples função.

Em segundo lugar, nós precisamos de ser capazes, quando tivermos uma string contendo o código (lida do arquivo), de realmente executar o código como um programa JavaScript.

Avaliando dados como código

Existem várias formas de se pegar dados (uma `string` de código) e rodá-los no contexto do programa atual.

A mais óbvia maneira é o operador padrão especial `eval`, que vai executar a string de código no escopo atual. Isso usualmente é uma ideia muito ruim, porque quebra algumas propriedades que escopos normalmente tem (ser isolado do mundo externo é a mais notável).

```
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
```

A melhor forma de converter dados dentro do programa é usar uma função construtora. Ela recebe como argumentos uma lista de nomes de argumentos separados por vírgula, e então uma string contendo o corpo da função.

```
var plusOne = new Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

Isso é precisamente o que precisamos - podemos encapsular o código para um módulo em uma função, com este escopo de função se tornando nosso escopo de módulo.

Require

Se a nova função construtora, usada pelo nosso módulo de carregamento, encapsula o código em uma função de qualquer forma, nós podemos omitir a função *namespace* encapsuladora atual dos arquivos. Nós também vamos fazer `exports` um argumento à função módulo, então o módulo não precisará de declarar isso. Isso remove um monte de barulho supérfluo do nosso módulo de exemplo:

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday"];
```

```
exports.name = function(number) {  
    return names[number];  
};  
exports.number = function(name) {  
    return names.indexOf(name);  
};
```

Essa é uma implementação mínima de `require` :

```
function require(name) {  
    var code = new Function("exports", readFile(name));  
    var exports = {};  
    code(exports);  
    return exports;  
}  
  
console.log(require("weekDay").name(1));  
// → Monday
```

Quando usando este sistema, um módulo tipicamente começa com pequenas declarações de variáveis que carregam os módulos que ele precisa.

```
var weekDay = require("weekDay");  
var today = require("today");  
  
console.log(weekDay.name(today.dayNumber()));
```

A implementação de `require` acima tem diversos problemas. Primeiro, ela vai carregar e rodar um módulo todas as vezes que este for "require-d" (requisitado), então se diversos módulos têm a mesma dependência, ou uma chamada `require` é colocada dentro de uma função que vai ser chamada múltiplas vezes, tempo e energia serão desperdiçados.

Isso pode ser resolvido armazenando os módulos que já tenham sido carregados em um objeto, e simplesmente retornando o valor existente se eles forem carregados novamente.

O segundo problema é que não é possível para um módulo expor diretamente um valor simples. Por exemplo, um módulo pode querer exportar apenas o construtor do tipo do objeto que ele define. Por agora, isso não pode ser feito, porque `require` sempre vai usar o objeto `exports` que ele cria como o valor exportado.

A solução tradicional para isso é fornecer outra variável, `module`, que é um objeto que tem a propriedade `exports`. Essa propriedade inicialmente aponta para o objeto vazio criado por `require`, mas pode ser sobrescrita com outro valor para exportar algo a mais.

```
function require(name) {  
    if (name in require.cache)  
        return require.cache[name];  
  
    var code = new Function("exports, module", readFile(name));  
    var exports = {}, module = {exports: exports};  
    code(exports, module);  
  
    require.cache[name] = module.exports;  
    return module.exports;  
}  
require.cache = Object.create(null);
```

Agora temos um sistema de módulo que usa uma simples variável global (`require`) para permitir que módulos encontrem e usem um ao outro sem ter que ir para o escopo global.

Este estilo de sistema de módulos é chamado "Módulos CommonJS", após o pseudo-padrão que o implementou pela primeira vez. Ele também é feito dentro do Node.js. Implementações reais fazem bem mais do que o exemplo que eu mostrei. Mais importante, eles tem uma forma muito mais inteligente de ir de um nome de módulo para uma parte de código real, permitindo ambos caminhos relativos e nomes de módulos registrados "globalmente".

Carregando módulos lentamente

Embora seja possível usar a técnica acima para carregar JavaScript no navegador, isso é um pouco complicado. A razão para isso é que ler um arquivo (módulo) na web é muito mais lento que ler este mesmo arquivo do seu disco rígido. JavaScript no navegador é obrigado a se comportar de tal forma que, enquanto um script esteja rodando, nada mais pode acontecer no site que ele está rodando. Isso significa que se todas as chamadas `require` carregarem algo em algum servidor web distante, a página vai ficar congelada por um doloroso longo período durante sua inicialização.

Existem maneiras de se trabalhar isso, por exemplo, rodando outro programa (como o Browserify) em seu programa antes, que irá concatenar todas as dependências olhando todas as chamadas `require`, e colocando-as em juntas em um grande arquivo.

Outra solução é encapsular seu módulo em uma função, carregar os módulos que ela depende em segundo plano, e apenas rodas essa função quando todas suas dependências forem carregadas. Isso é o que o sistema de módulos AMD ("Asynchronous Module Definition") faz.

Nosso programa trivial com dependências, em AMD, se parece com isso:

```
define(["weekDay", "today"], function(weekDay, today) {
    console.log(weekDay.name(today.dayNumber()));
});
```

A função `define` é o conceito central nessa abordagem. Ela primeiro recebe um array com nomes de módulos, e então uma função que recebe um argumento para cada dependência. Ela vai carregar as dependências (se elas ainda não tiverem sido carregadas) em segundo plano, permitindo que a página continue a trabalhar em quanto está esperando. Uma vez que todas as dependências estejam carregadas, ela vai carregar a função que foi passada, com as interfaces das dependências como argumentos.

Os módulos que são carregados dessa forma devem conter uma chamada a `define`. O valor usado para sua interface é qualquer valor retornado pela função que é o segundo argumento passado nessa chamada. Aqui está o módulo `weekDay` de novo.

```
define([], function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
    return {
        name: function(number) { return names[number]; },
        number: function(name) { return names.indexOf(name); }
    };
});
```

Para mostrar uma simples implementação de `define`, vamos supor que também temos uma função

`backgroundReadFile`, que pega o nome do arquivo e uma função, e vai chamar a função com o conteúdo do arquivo assim que este for carregado.

```
function define(depNames, moduleFunction) {
    var deps = [], myMod = define.currentModule;

    depNames.forEach(function(name) {
```

```

    if (name in define.cache) {
        var depMod = define.cache[name];
    } else {
        var depMod = {exports: null,
                      loaded: false,
                      onLoad: []};
        define.cache[name] = depMod;
        backgroundReadFile(name, function(code) {
            define.currentModule = depMod;
            new Function("", code)();
        });
    }
    deps.push(depMod);
    if (!depMod.loaded)
        depMod.onLoad.push(runIfDepsLoaded);
});

function runIfDepsLoaded() {
    if (!deps.every(function(m) { return m.loaded; }))
        return;

    var args = deps.map(function(m) { return m.exports; });
    var exports = moduleFunction.apply(null, args);
    if (myMod) {
        myMod.exports = exports;
        myMod.loaded = true;
        myMod.onLoad.every(function(f) { f(); });
    }
}
runIfDepsLoaded();
}
define.cache = Object.create(null);

```

Isso é muito mais difícil de seguir que a função `require`. Sua execução não segue um caminho simples e previsível. Ao invés disso, múltiplas operações são definidas para acontecerem em algum tempo não especificado no futuro (quando o módulo for carregado), que obscurece a forma que o código é executado.

O maior problema que este código lida é coletar os valores das interfaces das dependências do módulo. Para rastrear os módulos, e seus estados, um objeto é criado para cada módulo que é carregado por `define`. Este objeto armazena o valor exportado pelo módulo, um booleano indicando se o módulo já foi completamente carregado e um array de funções para ser chamado quando o módulo tiver sido carregado.

Um `cache` é usado para prevenir o carregamento de módulos múltiplas vezes, assim como fizemos para o `require`. Quando `define` é chamada, nós primeiro construímos um array de módulos de objetos que representam as dependências deste módulo. Se o nome da dependência corresponde com o nome de um módulo *cacheado*, nós usamos o objeto existente. Caso contrário, nós criamos um novo objeto (com o valor de `loaded` igual a `false`) e armazenamos isso em `cache`. Nós também começamos a carregar o módulo, usando a função `backgroundReadFile`. Uma vez que o arquivo tenha sido carregado, seu conteúdo é rodado usando o construtor `Function`.

É assumido que este arquivo também contenha uma (única) chamada a `define`. A propriedade `define.currentModule` é usada para informar a esta chamada sobre o módulo objeto que está sendo carregado atualmente, dessa forma podemos atualizá-lo umas vez e terminar o carregamento.

Isso é manipulado na função `runIfDepsLoaded`, que é chamada uma vez imediatamente (no caso de não ser necessário carregar nenhuma dependência) e uma vez para cada dependência que termina seu carregamento. Quando todas as dependências estão lá, nós chamamos `moduleFunction`, passando para ela os valores exportados apropriados. Se existe um módulo objeto, o valor retornado da função é armazenado, o objeto é marcado como carregado (*loaded*), e as funções em seu array `onLoad` são chamadas. Isso vai notificar qualquer módulo que esteja esperando que suas dependências sejam carregadas completamente.

Uma implementação real do AMD é, novamente, bem mais inteligente em relação a resolução dos nomes e suas URLs, e genericamente mais robusta. O projeto RequireJS (<http://requirejs.org>) fornece uma implementação popular deste estilo que carregamento de módulos.

Projeto de interfaces

Projetar interfaces para módulos e tipos de objeto é um dos aspectos sutis da programação. Qualquer pedaço não trivial de funcionalidade pode ser modelada de formas diferentes. Encontrar um caminho que funciona bem requer perspicácia e previdência.

A melhor forma de aprender o valor de um bom projeto de interface é usar várias interfaces, algumas boas, algumas horríveis. Experiência vai ensinar a você o que funciona e o que não funciona. Nunca assuma que uma interface dolorosa de se usar é "da forma que ela deve ser". Conserte-a, ou encapsule-a em uma nova interface de forma que funcione melhor para você.

Previsibilidade

Se programadores podem prever a forma que a interface vai funcionar, eles (ou você) não vão ser desviados frequentemente pela necessidade de checar como trabalhar com esta interface. Portanto, tente seguir convenções (por exemplo, quando se trata da capitalização de nomes). Quando existe outro módulo ou parte do ambiente padrão JavaScript que faz algo similar ao que você está implementando, é uma boa ideia fazer sua interface se assemelhar a interface existente. Dessa forma, as pessoas que conhecem a interface existente vão se sentir em casa.

Outra área que previsibilidade é importante é no comportamento do seu código. Pode ser tentador "empilhar inteligência" com a justificativa que isso torna a interface fácil de ser utilizada. Por exemplo, aceitando todos os diferentes tipos e combinações de argumentos, e fazendo "a coisa certa" para todos eles, ou fornecendo dezenas de diferentes funções especializadas por "conveniência" que fornecem pequenas alterações do sabor da funcionalidade do seu módulo. Isso pode tornar o código construído em cima da sua interface um pouco menor, mas isso vai também tornar o código muito mais difícil para as pessoas manterem um modelo mental do comportamento do módulo em suas cabeças.

"Componibilidade"

Em suas interfaces, tente usar as estruturas de dados mais simples que funcionem e crie funções que façam algo simples e claro - sempre que possível, crie funções puras (veja capítulo 3).

Por exemplo, não é comum para módulos fornecerem suas próprias coleções de objetos similares a arrays, com sua própria interface para contar e extrair elementos. Tais objetos não terão os métodos `map` e `forEach`, e qualquer função existente que espere um array real não será capaz de trabalhar com estas coleções. Este é um exemplo de componibilidade (*composability*) ruim - o módulo não pode ser facilmente composto com outro código.

Outro exemplo seria um módulo verificação ortográfica de texto, que podemos necessitar se quisermos escrever um editor de texto. O verificador pode ser construído para funcionar diretamente em qualquer tipo complexo de estrutura de dados que o editor usa, e chamar funções internas diretamente no editor para que o usuário possa escolher entre as sugestões de ortografia. Se formos por esse caminho, o módulo não poderá ser usado com outros programas. De outra forma, se nós definirmos a interface do verificador ortográfico para que possamos passar simples strings e retornar a possível localização do erro, juntamente com um array de correções sugeridas, nós teremos uma interface que pode ser composta com outros sistemas, porque strings e arrays estarão sempre disponíveis.

Interfaces em camadas

Quando projetando uma interface para uma complexa parte de funcionalidade - digo, enviar email - você geralmente se depara com um dilema. Em uma mão, você não quer sobrecarregar o usuário da sua interface com detalhes. Ele não deve estudar sua interface por 20 minutos antes de ser capaz de enviar um email. Na outra mão, você não quer esconder todos os detalhes - quando pessoas precisam fazer coisas complicadas com seu módulo, eles também devem ser capazes.

Normalmente a solução é oferecer duas interfaces: uma de "baixo nível" detalhada para situações complexas e uma de "alto nível" simples para uso rotineiro. A segunda pode ser construída de forma simples utilizando as ferramentas fornecidas pela primeira camada. No módulo de email, a interface de alto nível pode simplesmente ser uma função que recebe uma mensagem, um endereço de remetente, um endereço de destinatário e envia o email. A interface de baixo nível deve permitir um controle completo sobre os cabeçalhos do email, anexos, envio de email HTML, e por aí vai.

Resumo

Módulos fornecem estrutura para programas grandes, separando o código em diferentes arquivos e *namespaces*. Dando a estes módulos interfaces bem definidas os tornam fáceis de se utilizar, reusando-os em contextos diferentes, e continuando os usando mesmo quando evoluem.

Mesmo que a linguagem JavaScript não auxilie muito quando se trata de módulos, as flexíveis funções e objetos que ela fornece fazem que seja possível definir úteis sistemas de módulo. Escopo de função pode ser utilizado como namespace interno para o módulo, e objetos podem ser usados para armazenar blocos de valores exportados.

Existem duas abordagens populares para tais módulos. Uma é chamada "Módulos CommonJS", e funciona em torno da função `require` que busca um módulo pelo seu nome e retorna sua interface. A outra abordagem é chamada "AMD", e usa a função assíncrona `define` que recebe um array de nome de módulos e uma função, e depois de carregar os módulos, roda a função com suas interfaces e argumentos.

Exercícios

Nomes dos meses

Escreva um simples módulo similar ao módulo `weekDay`, que pode converter os números dos meses (*zero-based*, assim como o tipo `Date`) para nomes, e nomes para números. Dê a este módulo seu próprio namespace, pois ele vai precisar de um array interno com o nome dos meses, mas use JavaScript puro, sem nenhum sistema de carregamento de módulos.

```
// Your code here.

console.log(month.name(2));
// → March
console.log(month.number("November"));
// → 10
```

Ele vai seguir o módulo `weekDay` praticamente por inteiro. Uma função anônima, chamada imediatamente, encapsula a variável que contém o array de nomes, assim como as duas funções que precisam ser exportadas. As funções são colocadas em um objeto. A interface de objeto retornada é armazenada na variável `month`.

Dependências circulares

Um assunto complicado na gestão de dependências é o de dependências circulares, onde módulo A depende do módulo B, e B também depende do módulo A. Muitos sistemas simplesmente proíbem isso. CommonJS permite uma forma limitada disso, onde isso funciona se os módulos não trocarem seus objetos exportados por padrão com outro valor, e somente começam a acessar a interface um do outro após terem finalizados seus carregamentos.

Você pode pensar em algo que dê suporte para essa funcionalidade ser implementada? Olhe anteriormente a definição de `require`, e considere o quê você deve fazer para permitir isso.

O segredo é adicionar o objeto `exports` criado por um módulo para requisitar o cache antes de rodar o módulo de fato. Isso significa que o módulo não teria tido ainda uma chance de sobrescrever `module.exports`, então não sabemos se ele deseja exportar outro valor. Depois de carregar, o objeto cache é sobrescrito com `module.exports`, que pode ser um valor diferente.

Mas se, no curso de carregar o módulo, um segundo módulo é carregado e solicita o primeiro módulo, seu objeto `exports` padrão, ainda vazio até este ponto, vai estar no cache, e o segundo módulo vai receber uma referência dele. Se ele não tentar fazer nada com o objeto até que o segundo módulo tenha terminado seu carregamento, as coisas vão funcionar.

Um retorno a vida eletrônica

Esperando que o capítulo 7 ainda esteja um pouco fresco em sua mente, pense novamente no sistema projetado neste capítulo e elabore uma separação em módulo para o código. Para refrescar sua memória, essas são as funções e tipos definidos naquele capítulo, em ordem de aparição.

- Point
- Grid
- directions
- randomElement
- BouncingCritic
- elementFromChar
- World
- charFromElement
- Wall
- View
- directionNames
- WallFollower
- dirPlus
- LifeLikeWorld
- Plant
- PlantEater
- SmartPlantEater
- Tiger

Não exagere em criar muitos módulos. Um livro que começa um novo capítulo para cada página provavelmente vai te deixar nervoso, por todo espaço perdido com os títulos. De forma similar, ter que abrir dez arquivos para ler um pequeno projeto não é útil. Vise por três ou cinco módulos.

Você pode escolher ter algumas funções internas ao módulo, e então inacessíveis a outros módulos.

Não existe uma única solução correta aqui. Organização de módulos é meramente uma questão de gosto.

Aqui está o que eu fiz. Coloquei parenteses em torno de funções internas.

- Module "grid"
 - Point

- Grid
- directions
- Module "world"
 - (randomElement)
 - (elementFromChar)
 - (charFromElement)
 - View
 - World
 - LifeLikeWorld
 - directions [re-exported]
- Module "simple_ecosystem"
 - (randomElement) [duplicated]
 - (directionNames)
 - (dirPlus)
 - Wall
 - BouncingCritter
 - WallFollower
- Module "ecosystem"
 - Wall [duplicated]
 - Plant
 - PlantEater
 - SmartPlantEater
 - Tiger

Eu reexportei o array `directions` do módulo `grid` para `world`, então módulos criados com eles (`ecosystems`) não precisam de saber ou se preocupar da existência do módulo `grid`.

Eu também dupliquei dois valores minúsculos e genéricos (`randomElement` e `wall`) pois eles são usados como detalhes internos em contextos diferentes, e não pertencem nas interfaces destes módulos.

Linguagem de programação

"O avaliador que determina qual o significado das expressões em uma linguagem de programação é apenas mais um programa."

Hal Abelson e Gerald Sussman, Estrutura e Interpretação de Programas de Computador

"Quando um estudante perguntou ao mestre sobre a natureza do ciclo de dados e controle, Yuan-Ma respondeu: 'Pense em um compilador compilando a si mesmo.'"

Mestre Yuan-Ma, O Livro de Programação

Construir sua própria linguagem de programação é surpreendentemente fácil(desde que você não seja ambicioso demais) e bastante esclarecedor.

A principal coisa que eu quero mostrar neste capítulo é que não há mágica envolvida na construção de sua própria linguagem. Eu sempre senti que algumas invenções humanas eram imensamente inteligentes e complicadas que eu nunca seria capaz de compreendê-las. Mas com um pouco de leitura e ajustes; tais coisas muitas vezes acabam por ser muito simples.

Iremos construir uma linguagem de programação chamada **Egg**. Vai ser uma pequena e simples linguagem mas poderosa o suficiente para expressar qualquer computação que você possa imaginar. Ela também permite abstração simples baseadas em funções.

Parsing

A parte imediatamente mais visível de uma linguagem de programação é sua sintaxe ou notação. Um analisador é um programa que lê um pedaço de texto e produz uma estrutura de dados que refletem a estrutura do programa contida nesse texto. Se o texto não faz um programa válido o analisador deve reclamar e apontar o erro.

Nossa linguagem terá uma sintaxe simples e uniforme. Tudo em **Egg** é uma expressão. Uma expressão pode ser uma variável, um `Number`, uma `String`, ou uma aplicação. As aplicações são usadas para chamadas de função, mas também para construções como `if` ou `while`.

Para manter o analisador simples, `String` em **Egg** não suportam qualquer coisa como escapes e uma sequência simplesmente de caracteres que não são aspas duplas envolvidas em aspas duplas. Um número é uma sequência de dígitos. Os nomes das variáveis podem consistir de qualquer caractere que não seja um espaço em branco e não tem um significado especial na sintaxe.

As aplicações serão escritas da forma como é em JavaScript; colocando parênteses após uma expressão e com uma série 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 coisas que são operadores de JavaScript(como `>`) nesta linguagem serão apenas variáveis normais aplicadas apenas como outras funções. E uma vez que a sintaxe também não tem o conceito de um bloco precisamos construir um representador fazendo várias coisas em sequência.

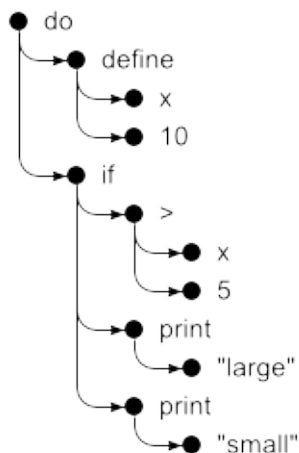
A estrutura de dados que o analisador irá usar para descrever um programa será composto de objetos de expressões cada um dos quais tem uma propriedade de tipo que indica o tipo de expressão que é; e as outras propriedades para descreverem o seu conteúdo.

Expressões do tipo **"value"** representam `Strings`, `literals` ou `Numbers`. O valor da propriedade contém o valor da cadeia ou o número que ele representa. Expressões do tipo **"word"** são usados para identificadores(nomes). Esses objetos têm uma propriedade que contém o nome do identificador de uma `String`. Por fim as expressões **"apply"** representam algo que é uma aplicação. Eles têm uma propriedade de operador que se refere à expressão que são aplicáveis e têm uma propriedade de `args` que refere-se a um conjunto de expressões de argumento.

A parte `>(x, 5)` do programa anterior seria representado assim:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

Essa estrutura de dados é chamado de árvore de sintaxe. Se você imaginar os objetos como pontos de ligações entre eles e com linhas entre esses pontos, ele tem uma forma treelike. O fato de que as expressões contêm outras expressões que por sua vez pode conter mais expressões é semelhante à maneira como dividir ramos e dividir novamente.



Compare isso com o analisador que escrevemos para o formato de arquivo de configuração no capítulo 9 que tinha uma estrutura simples: dividir a entrada em linhas e tratar essas linhas uma de cada vez. Havia apenas algumas formas simples de mostrar que uma linha foi permitida.

Aqui temos de encontrar uma abordagem diferente. As expressões não são separados em linhas e elas têm uma estrutura recursiva. Expressões aplicadas contêm outras expressões.

Felizmente, este problema pode ser resolvido com elegância escrevendo uma função analisadora que é recursiva de uma forma que reflete a natureza recursiva da linguagem.

Nós definimos uma função `parseExpression` que recebe uma string como entrada e retorna um objeto que contém a estrutura de dados para a expressão no início da cadeia, depois é feito a junção com a parte da cadeia da esquerda para analisar esta expressão. Ao analisar essa `subexpressions` (o argumento para um aplicativo, por exemplo) esta função pode ser chamado novamente dando origem a expressão argumento bem como o texto nos mostra. Este texto pode por sua vez contém mais argumentos ou pode ser o parêntese de fechamento, que da termino a lista de argumentos.

Esta é a primeira parte do analisador:

```
function parseExpression(program) {
  program = skipSpace(program);
  var 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) {
  var first = string.search(/\s/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Temos que remover os espaços em brancos repetidos no início de qualquer sequência do programa pois o **Egg** permite qualquer quantidade de espaço em branco entre os seus elementos inseridos. Quem tem essa funcionalidade é a da função `skipSpace`.

Depois de pular qualquer espaço à esquerda `parseExpression` usa três expressões regulares para detectar os três elementos simples(atômicas) que **Egg** suporta: `String`, `Number` e `words`. O analisador constrói um tipo diferente de estrutura de dados dependendo de sua correspondência. Se a entrada não coincide com uma destas três formas não será considerado uma expressão válida e o analisador gerará um erro. `SyntaxError` é um tipo de erro padrão de objeto que é gerado quando é feita uma tentativa de executar um programa em JavaScript inválido.

Podemos cortar algumas partes que nós comparamos a partir da sequência e passar isso juntamente com o objeto para a expressão do `parseApply` que irá verificar se a expressão é uma aplicação. Se assim for ele 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] != ")") {
    var 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 carácter no programa não é um parêntese de abertura, este não é aplicável, e `parseApply` simplesmente retorna que a expressão foi proferida.

Caso contrário ele ignora o parêntese de abertura e cria o objeto na árvore de sintaxe para essa expressão aplicável. Em seguida ele chama recursivamente `parseExpression` para analisar cada argumento até o parêntese de fechamento ser encontrado. A recursividade é indireta através da função `parseApply` e `parseExpression` chamando uns aos outros.

Uma expressão de aplicação pode ser aplicado em si própria (como em `multiplier(2)(1)`); `parseApply` deve analisar um pedido depois chamar-se novamente para verificar se existe outro par de parênteses.

Isso é tudo que precisamos para o analisador do **Egg**. Nós vamos envolvê-lo em uma função de análise conveniente que verifica se ele chegou ao fim da cadeia de entrada após o análise da expressão (um programa de **Egg** é uma única expressão) e que nos dá estrutura de dados do programa.

```
function parse(program) {
  var result = parseExpression(program);
  if (skipSpace(result.rest).length > 0)
    throw new SyntaxError("Unexpected text after program");
  return result.expr;
}

console.log(parse("(+ (a, 10)"));
// → {type: "apply",
//   operator: {type: "word", name: "+"},
//   args: [{type: "word", name: "a"},
//         {type: "value", value: 10}]}
```

Funcionou! Ele não nos dá informação muito útil quando há falhas e não armazena a linha e coluna na qual cada expressão começa, o que pode ser útil ao relatar erros mais tarde mas é bom o suficiente para nossos propósitos.

O avaliador

O que podemos fazer com uma árvore de sintaxe de um programa? Executá-lo é claro! E é isso que o avaliador faz. Você entrega-lhe uma árvore de sintaxe e um objeto do `environment` que associa nomes com os valores, e ele irá avaliar a expressão que a árvore representa e retornar o valor que esta produz.

```
function evaluate(expr, env) {
  switch(expr.type) {
    case "value":
      return expr.value;

    case "word":
      if (expr.name in env)
        return env[expr.name];
      else
        throw new ReferenceError("Undefined variable: " +
                                expr.name);

    case "apply":
      if (expr.operator.type == "word" &&
          expr.operator.name in specialForms)
        return specialForms[expr.operator.name](expr.args,
                                                  env);

      var op = evaluate(expr.operator, env);
      if (typeof op != "function")
        throw new TypeError("Applying a non-function.");
      return op.apply(null, expr.args.map(function(arg) {
        return evaluate(arg, env);
      }));
  }
}

var specialForms = Object.create(null);
```

O avaliador possui código para cada um dos tipos de expressão. A expressão de valor literal simplesmente produz o seu valor(por exemplo, a expressão 100 apenas avalia para o número 100). Para uma variável é preciso verificar se ele está realmente definido no `environment atual`, se estiver, buscar o valor da variável.

As aplicações são mais envolvidas. Se eles são de uma forma especial, nós não avaliamos nada e simplesmente passamos as expressões como argumento junto com o `environment` para a função que lida com essa forma. Se for uma chamada normal nós avaliamos o operador verificamos se ele é uma função e chamamos com o resultado da avaliação dos argumentos.

Iremos usar os valores de uma função simples em JavaScript para representar os valores de função em **Egg**. Voltaremos a falar sobre isso mais tarde quando o `specialForm` chamado `fun` estiver definido.

A estrutura recursiva de um avaliador se assemelha à estrutura de um analisador. Ambos espelham a estrutura da própria linguagem. Além disso, seria possível integrar o analisador com o avaliador e avaliar durante a análise, mas dividindo-se desta forma torna o programa mais legível.

Isso é tudo que precisamos para interpretar Egg. É simples assim. Mas sem definir algumas formas especiais e adicionar alguns valores úteis para o `environment` você não pode fazer nada com essa linguagem ainda.

Formas especiais

O objecto `specialForms` é utilizado para definir sintaxe especial em **Egg**. Ele associa palavras com funções que avaliam essas formas especiais. Atualmente ele está vazio. Vamos adicionar algumas formas.

```
specialForms["if"] = function(args, env) {
  if (args.length !== 3)
    throw new SyntaxError("Bad number of args to if");

  if (evaluate(args[0], env) !== false)
    return evaluate(args[1], env);
  else
    return evaluate(args[2], env);
};
```

Egg - `if` espera exatamente três argumentos. Ele irá avaliar o primeiro, se o resultado não é o valor falso ele irá avaliar a segunda. Caso contrário a terceira fica avaliada. Esta é a forma mais semelhante ao ternário do JavaScript `?:` estes operadores tem o mesmo significado de `if/else` em JavaScript. Isso é uma expressão e não uma indicação que produz um valor, ou seja, o resultado do segundo ou terceiro argumento.

Egg difere de JavaScript na forma de como ele lida com o valor de um condição como o valor do `if`. Ele não vai tratar as coisas como zero ou cadeia vazia como falsa, somente valores precisos são falsos.

A razão especial é que nós preciso representar o `if` como uma forma especial, ao invés de uma função regular onde todos os argumentos para funções são avaliadas antes que a função seja chamada, ao passo que se deve avaliar apenas seu segundo ou terceiro argumento, dependendo do valor do primeiro.

A forma `while` é semelhante.

```
specialForms["while"] = function(args, env) {
  if (args.length !== 2)
    throw new SyntaxError("Bad number of args to while");

  while (evaluate(args[0], env) !== false)
    evaluate(args[1], env);

  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
  return false;
};
```

```
};
```

Outro bloco na construção básico é fazer que executa todos os seus argumentos de cima para baixo. O seu valor é o valor produzido pelo último argumento.

```
specialForms["do"] = function(args, env) {
  var value = false;
  args.forEach(function(arg) {
    value = evaluate(arg, env);
  });
  return value;
};
```

Para ser capaz de criar variáveis e dar-lhes novos valores, vamos criar um `specialForms` chamado `define`. Ele espera uma palavra como primeiro argumento de uma expressão que produz o valor a ser atribuído a essa palavra que sera seu segundo argumento. Vamos definir sendo tudo uma expressão e ela deve retornar um valor. Vamos fazê-lo retornar o valor que foi atribuído(igual ao operador `=` de JavaScript).

```
specialForms["define"] = function(args, env) {
  if (args.length !== 2 || args[0].type !== "word")
    throw new SyntaxError("Bad use of define");
  var value = evaluate(args[1], env);
  env[args[0].name] = value;
  return value;
};
```

Ambiente

O `environment` aceita avaliar um objeto com propriedades cujos nomes correspondem aos nomes de variáveis e cujos valores correspondem aos valores dessas variáveis. Vamos definir um objeto no `environment` para representar o escopo global.

Para ser capaz de usar `if` que acabamos de definir teremos de ter acesso aos valores `booleanos`. Uma vez que existem apenas dois valores `booleanos` nós não precisamos de sintaxe especial para eles. Nós simplesmente vamos ligar duas variáveis em `topEnv` para os valores verdadeiros e falsos e daí então usá-los.

```
var topEnv = Object.create(null);

topEnv["true"] = true;
topEnv["false"] = false;
```

Agora podemos avaliar uma expressão simples que nega um valor `booleano`.

```
var prog = parse("if(true, false, true)");
console.log(evaluate(prog, topEnv));
// → false
```

Para suprir os operadores aritméticos e comparações básicas vamos adicionar alguns valores para função de `environment`. No interesse de manter um código pequeno vamos utilizar uma nova função para sintetizar um monte de funções de operador em um loop ao invés de definir todos eles individualmente.

```
["+", "-", "*", "/", "==", "<", ">"].forEach(function(op) {
  topEnv[op] = new Function("a, b", "return a " + op + " b;");
});
```

É muito útil fazer uma maneira para que valores de saída sejam visualizados, por isso vamos colocar alguns `console.log` na função e executá-lo para imprimir.

```
topEnv["print"] = function(value) {
  console.log(value);
  return value;
};
```

Isso já nos proporcionou uma ferramenta elementar e suficiente para escrever programas simples. A seguinte função `run` fornece uma maneira conveniente de escrever e executá-los. Ele cria um `environment` em tempo real, analisa e avalia as `String` que damos como um programa único.

```
function run() {
  var env = Object.create(topEnv);
  var program = Array.prototype.slice
    .call(arguments, 0).join("\n");
  return evaluate(parse(program), env);
}
```

O uso de `Array.prototype.slice.call` é um truque para transformar um objeto de matriz como argumentos em uma matriz real; de modo que podemos chamar e juntar cada pedaço. No exemplo abaixo iremos percorrer todos os argumentos dados e tratar cada linha do programa.

```
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 escrito em **Egg**. É evidente que é mais feio do que um programa em JavaScript, mas não é tão ruim para uma linguagem implementada em menos de 150 linhas de código.

Funções

A linguagem de programação sem funções é uma linguagem de programação pobre.

Felizmente, não é difícil para adicionar `fun` a nossa linguagem, que vai tratar todos os argumentos antes do último como nomes de argumentos da função e seu último argumento como corpo da função.

```
specialForms["fun"] = function(args, env) {
  if (!args.length)
    throw new SyntaxError("Functions need a body");
  function name(expr) {
    if (expr.type !== "word")
      throw new SyntaxError("Arg names must be words");
    return expr.name;
  }
  var argNames = args.slice(0, args.length - 1).map(name);
  var body = args[args.length - 1];

  return function() {
    if (arguments.length !== argNames.length)
      throw new TypeError("Wrong number of arguments");
    var localEnv = Object.create(env);
    for (var i = 0; i < arguments.length; i++)
```

```

    localEnv[argNames[i]] = arguments[i];
    return evaluate(body, localEnv);
  };
};

```

Funções em **Egg** tem seu próprio `environment` local assim como em JavaScript. Usamos `Object.create` para fazer um novo objeto que tem acesso às variáveis do ambiente externo(`prototype`) mas que também pode conter novas variáveis sem modificar esse escopo exterior.

A função criada pela `especialForm` `fun` cria em ambito local e adiciona as variáveis de argumento para isso. Em seguida ele avalia o corpo da função neste ambiente 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 nós construímos foi um intérprete. Durante a avaliação ele age diretamente sobre a representação do programa produzido pelo analisador.

A compilação é o processo de adicionar mais um passo entre a análise e a execução de um programa; que transforma o programa em algo que possa ser avaliado de forma mais eficiente fazendo o trabalho tanto quanto possível com antecedência. Por exemplo, em línguas bem desenhadas, é óbvio para cada uso de uma variável ele verifica qual esta se referindo sem realmente executar o programa. Isso pode ser usado para evitar a procura de uma variável pelo nome sempre que é acessado ou buscado diretamente de algum local pré-determinado da memória.

Tradicionalmente, compilação envolve a conversão do programa para código de máquina no formato `raw` que o processador de um computador pode executar. Qualquer processo que converte um programa de uma representação diferente pode ser encarado como compilação.

Seria possível escrever uma estratégia de avaliação alternativa para **Egg**, aquele que primeiro converte o programa para um programa JavaScript utilizando a nova função para chamar o compilador JavaScript, e em seguida executar o resultado. Sendo feito assim **Egg** executaria muito mais rápido e continuaria bastante simples de implementar.

Se você está interessado e disposto neste assunto gaste algum tempo com isso, encorajo-vos a tentar implementar um compilador nos exercícios.

Cheating

Quando definimos `if` e `while`, você provavelmente percebeu que eles eram invólucros triviais em torno do próprio JavaScript. Da mesma forma, os valores em **Egg** são antigos valores de JavaScript.

Se você comparar a execução de **Egg** que foi construída em alto nível utilizando a ajuda de JavaScript com a quantidade de trabalho e complexidade necessários para construir uma linguagem de programação utilizando diretamente a funcionalidade `raw` fornecido por uma máquina essa diferença é enorme. Independentemente disso este é apenas um exemplo; espero ter lhe dado uma impressão de que maneira as linguagens de programação trabalham.

E quando se trata de conseguir fazer algo, o `cheating` é o jeito mais eficaz de fazer tudo sozinho. Embora a linguagem que brincamos neste capítulo não faz nada de melhor que o JavaScript possui, existem situações em que a escrever pequenas línguas ajuda no entendimento verdadeiro do trabalho.

Essa língua não possui semelhanças com uma linguagem típica de programação. Se o JavaScript não vêm equipado com expressões regulares você pode escrever seu próprio analisador e avaliador para tal sub linguagem.

Ou imagine que você está construindo um dinossauro robótico gigante e precisa programar o seu comportamento. JavaScript pode não ser a forma mais eficaz de fazer isso. Você pode optar por uma linguagem que se parece com isso:

```
behavior walk
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

Isto é o que geralmente é chamado de linguagem de domínio específica, uma linguagem adaptada para expressar um estreito conhecimento de um domínio. Essa linguagem pode ser mais expressiva do que uma linguagem de um propósito geral. Isto porque ela é projetada para expressar exatamente as coisas que precisam serem expressadas no seu domínio e nada mais.

Exercícios

Arrays

Adicionar suporte para `array` em **Egg** construindo as três funções em `topEnv` do escopo: `array(...)` vai ser a construção de uma matriz contendo os argumentos como valores, `length(array)` para obter o comprimento de um `array` e `element(array, n)` buscar `n` elementos de uma matriz.

```
// Modify these definitions...

topEnv["array"] = "...";

topEnv["length"] = "...";

topEnv["element"] = "...";

run("do(define(sum, fun(array, ",
  "    do(define(i, 0), ",
  "        define(sum, 0), ",
  "        while(<(i, length(array)), ",
  "            do(define(sum, +(sum, element(array, i))), ",
  "                define(i, +(i, 1)))), ",
  "            sum))), ",
  "    print(sum(array(1, 2, 3))))");

// - 6
```

Dica:

A maneira mais fácil de fazer isso é representar as matrizes de **Egg** através de matrizes do JavaScript.

Os valores adicionados ao `enviroment` no `topEnv` deve ser uma funções. `Array.prototype.slice` ; pode ser utilizado para converter um `array` em um `object` de argumentos numa matriz regular.

Resolução

Closures

A maneira que definimos o `fun` é permitido que as funções em **Egg** se chamem em ambiente circundante, permitindo o corpo da função utilizar valores locais que eram visíveis no momento que a função foi definida, assim como as funções em JavaScript fazem.

O programa a seguir ilustra isso: função `f` retorna uma função que adiciona o seu argumento ao argumento de `f`, o que significa que ele precisa de acesso ao escopo local dentro de `f` para ser capaz de utilizar a variável.

```
run("do(define(f, fun(a, fun(b, +(a, b)))),",
    "print(f(4)(5)))");
// → 9
```

Volte para a definição da forma `fun` e explique qual o mecanismo feito para que isso funcione.

Dica:

Mais uma vez, estamos cavalcando sobre um mecanismo de JavaScript para obter a função equivalente em **Egg**. Formas especiais são passados para o `enviroment` local de modo que eles possam ser avaliados pelas suas sub-formas do `enviroment` . A função retornada por `fun` se fecha sobre o argumento `env` dada a sua função de inclusão e usa isso para criar `enviroment` local da função quando é chamado.

Isto significa que o `prototype` do `enviroment` local será o `enviroment` em que a função foi criado, o que faz com que seja possível ter acesso as variáveis de `enviroment` da função. Isso é tudo o que há para implementar e finalizar(embora para compilá-lo de uma forma que é realmente eficiente, você precisa de um pouco mais de trabalho).

Comentários

Seria bom se pudéssemos escrever comentários no **Egg**. Por exemplo, sempre que encontrar um cardinal (`"#"`), poderíamos tratar o resto da linha como um comentário e ignorá-lo, semelhante que Javascript faz com o `"//"`.

Não temos de fazer quaisquer grandes mudanças para que o analisador suporte isto. Nós podemos simplesmente mudar o `skipSpace` para ignorar comentários assim como é feito com os espaços em branco; para que todos os pontos onde `skipSpace` é chamado agora também irá ignorar comentários. Vamos fazer essa alteração:

```
// This is the old skipSpace. Modify it...
function skipSpace(string) {
  var first = string.search(/\s/);
  if (first == -1) return "";
  return string.slice(first);
}

console.log(parse("# hello\nx"));
// → {type: "word", name: "x"}

console.log(parse("a # one\n  # two\n()"));
// → {type: "apply",
```



```
// operator: {type: "word", name: "a"},
// args: []}
```

Dica:

Certifique-se de que sua solução é válida com vários comentários em uma linha e principalmente com espaço em branco entre ou depois deles.

Uma expressão regular é a maneira mais fácil de resolver isso. Faça algo que corresponda "espaços em branco ou um comentário, uma ou mais vezes". Use o método `exec` ou `match` para olhar para o comprimento do primeiro elemento na matriz retornada(desde de o início) para saber quantos caracteres precisa para cortar.

Resolução

Corrigindo o escopo

Atualmente, a única maneira de atribuir uma variável um valor é utilizando o método `define`. Esta construção atua tanto como uma forma para definir novas variáveis e dar um novo valor para existentes.

Isto causa um problema de ambiguidade. Quando você tenta dar uma variável um novo valor que não esta local, você vai acabar definindo uma variável local com o mesmo nome em seu lugar(Algumas línguas funcionam assim por design, mas eu sempre achei uma maneira estranha de lidar com escopo).

Adicionar um `specialForm` similar ao `define` dara a variável um novo valor ou a atualização da variável em um escopo exterior se ele ainda não existir no âmbito interno. Se a variável não é definida em tudo lançar um `ReferenceError` (que é outro tipo de erro padrão).

A técnica de representar escopos como simples objetos tornou as coisas convenientes, até agora, e vai ficar um pouco no seu caminho neste momento. Você pode querer usar a função `Object.getPrototypeOf` que retorna os protótipos de um objeto. Lembre-se também que os escopos não derivam de `Object.prototype`, por isso, se você quiser chamar `hasOwnProperty` sobre eles,você terá que usar esta expressão não muito elegante:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

Este método(`hasOwnProperty`) busca o protótipo do objeto e depois chama-o em um objeto do escopo.

```
specialForms["set"] = function(args, env) {
  // Your code here.
};

run("do(define(x, 4),",
    "  define(setx, fun(val, set(x, val))),",
    "  setx(50),",
    "  print(x))");
// → 50
run("set(quux, true)");
// → Some kind of ReferenceError
```

Dica:

Você vai ter que percorrer um escopo de cada vez usando `Object.getPrototypeOf` ate ir ao escopo externo. Para cada um dos escopos use `hasOwnProperty` para descobrir se a variável indicado pela propriedade `name` do primeiro argumento definida existe nesse escopo. Se isso acontecer defina-o como o resultado da avaliação do segundo argumento, e em seguida retorne esse valor.

Se o escopo mais externo é atingido(`Object.getPrototypeOf` retornando `null`) e não encontramos a variável, isto significa que não existe; então um erro deve ser acionado.

Resolução

JavaScript e o Navegador

"O navegador é um ambiente realmente hostil de programação."

- Douglas Crockford, The JavaScript Programming Language (video lecture)

A próxima parte deste livro vai falar sobre os navegadores web. Sem os navegadores, não existiria JavaScript. E mesmo se existisse, ninguém daria atenção a ele.

A tecnologia web, desde de o início, é descentralizada não apenas tecnicamente mas também na maneira que se evolui. Vários fornecedores de navegador tem adicionado funcionalidades *ad-hoc* e muita das vezes tem sido de maneiras mal pensadas, que acabam sendo adotadas por outros e finalmente viram um padrão.

Isso é igualmente a uma benção e uma maldição. Por outro lado, isso reforça a não existência de uma partição central controlando um sistema mas o mesmo vem sendo melhorado por várias partes trabalhando com pouca colaboração (ou, ocasionalmente com franca hostilidade). Sendo assim a forma casual que a Web foi desenvolvida significa que o sistema resultante não é exatamente um brilhante exemplo interno de consistência. De fato, algumas partes são completamente bagunçadas e confusas.

Redes e a Internet

Redes de computador existem desde 1950. Se você colocar cabos entre dois ou mais computadores e permitir que eles enviem dados um para o outro por estes cabos, você pode fazer todo tipo de coisas maravilhosas.

Se conectando duas máquinas no mesmo prédio permite que nós façamos coisas incríveis, conectando máquinas por todo o planeta deve ser ainda melhor. A tecnologia para começar a implementação desta visão foi desenvolvida em meados de 1980, e a rede resultante é chamada de *Internet*. Ela tem vivido desde a sua promessa.

Um computador pode usar essa rede para enviar bits para outro computador. Para qualquer comunicação efetiva nascida desse envio de bits, os computadores em ambas as pontas devem conhecer qual a representação de cada bit. O significado de cada sequência de bits depende inteiramente do tipo de coisa que está tentando se expressar e o mecanismo de codificação usado.

Um *protocolo de rede* descreve um estilo de comunicação em uma rede. Existem protocolos para mandar email, para receber email, para transferir arquivos, e até mesmo para controlar computadores que foram infectados por softwares maliciosos.

Por exemplo, um simples protocolo de chat deve consistir em um computador enviando os bits que representam o texto "CHAT?" para outra máquina, e o outro respondendo "OK!" para confirmar que o protocolo foi entendido. Eles podem então proceder e enviar um para o outro `strings` de texto, ler o texto enviado um para o outro pela rede, e mostrar o que eles receberam nas suas telas.

A maioria dos protocolos são feitos em cima de outros protocolos. Nosso exemplo de protocolo de chat considera a rede como um tipo de dispositivo de *stream*, no qual você pode enviar bits e recebê-los com destino correto e na ordem correta. Assegurar essas coisas atualmente é um problema técnico bastante difícil.

O TCP (Protocolo de Controle de Transmissão) é um protocolo que resolve este problema. Todos os aparelhos conectados na Internet "falam" ele, e a maioria da comunicação na Internet é feita através dele.

Uma conexão TCP funciona da seguinte maneira: um computador deve estar esperando, ou *ouvindo*, outros computadores que irão começar a falar com ele. Para ser capaz de escutar por diferentes tipos de comunicação ao mesmo tempo em uma única máquina, cada *ouvinte* tem um número (chamado de **porta**) associado a ele. A maioria


```
<!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>
```

As *tags*, definidas entre os sinais de menor e maior que (< e >), fornecem informações sobre a estrutura do documento. O conteúdo restante é apenas texto puro.

O documento começa com `<!doctype html>`, que diz ao navegador para interpretá-lo como HTML *moderno* (HTML5), ao invés de outras versões que foram usadas no passado.

Documentos HTML possuem um `head` (cabeça) e um `body` (corpo). O `head` contém informações *sobre* o documento, o `body` contém o documento em si. Neste caso, nós primeiro declaramos que o título do documento é "My home page" e em seguida, declaramos o `body` contendo um cabeçalho (`<h1>`, que significa "heading 1" - As *tags* de `<h2>` a `<h6>` produzem cabeçalhos menores) e dois parágrafos (`<p>`).

Tags aparecem em diversas formas. Um elemento, como o `<body>`, um parágrafo ou um link, começa com uma *tag* de abertura como em `<p>` e termina com uma *tag* de fechamento como em `</p>`. Algumas *tags* de abertura, como aquela para o link (`<a>`), contém informações extra na forma de pares `nome="valor"`. Estes são chamados de *atributos*. Nesse caso, o destino do link é indicado pelo atributo `href="http://eloquentjavascript.net"`, onde `href` significa "hypertext reference" (referência de hipertexto).

Alguns tipos de *tags* não englobam conteúdo e assim não necessitam de uma *tag* de fechamento. Um exemplo seria ``, que irá mostrar a imagem encontrada na URL informada no atributo `src`.

Para sermos capazes de incluir os sinais de menor e maior no texto de um documento, mesmo esses possuindo um significado especial em HTML, teremos que introduzir mais uma nova forma de notação especial. Uma *tag* de abertura simples é escrita como `<` ("less than" - menor que), e uma *tag* de fechamento é escrita como `>` ("greater than" - maior que). Em HTML, o caractere `&` (o sinal de "E comercial") seguido por uma palavra e um ponto e vírgula é chamado de "entidade" (*entity*), e será substituída pelo caractere que representa.

Essa notação é parecida com a forma que as barras invertidas são utilizadas nas *strings* em JavaScript. Uma vez que esse mecanismo dá ao caractere `&` um significado especial, este tem que ser representado como `&`. Dentro de um atributo, que é definido entre aspas duplas, a entidade `"` pode ser usada para representar um caractere de aspas duplas.

O HTML é interpretado de uma forma notavelmente tolerante a erros. Se uma *tag* é omitida, o navegador irá inseri-la. A forma com que isto é feito foi padronizada, você pode confiar em todos os navegadores modernos para realizar tal tarefa.

O documento a seguir será tratado exatamente como o mostrado anteriormente:

```
<!doctype html>

<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>` foram retiradas. O navegador sabe que a *tag* `<title>` pertence ao `head` , e que `<h1>` pertence ao `body` . Além disso, eu não especifiquei o final dos parágrafos, o fato de começar um novo parágrafo ou fechar o documento irá implicitamente fechá-los. As aspas que envolviam o destino do link também foram retiradas.

Esse livro geralmente vai omitir as *tags* `<html>` , `<head>` e `<body>` dos exemplos para mantê-los curtos e ordenados. Mas eu irei fechar as *tags* e incluir aspas nos valores de atributos.

Eu geralmente também irei omitir o *doctype*. Isso não deve ser interpretado como um incentivo a omitir declarações de *doctype*. Os navegadores frequentemente irão fazer coisas ridículas quando você esquece delas. Você deve considerá-las implicitamente presentes nos exemplos, mesmo quando elas não forem mostradas no texto.

HTML e JavaScript

No contexto desse livro, a *tag* mais importante do HTML é `<script>` . Essa *tag* nos permite incluir trechos de JavaScript em um documento.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

Esse *script* será executado assim que a *tag* `<script>` for encontrada enquanto o navegador interpreta o HTML. A página mostrada acima irá exibir uma mensagem de alerta quando aberta.

Incluir programas grandes diretamente no documento HTML é impraticável. A *tag* `<script>` pode receber um atributo `src` a fim de buscar um arquivo de *script* (um arquivo de texto contendo um programa em JavaScript) a partir de uma URL.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

O arquivo `code/hello.js` incluído aqui contém o mesmo simples programa, `alert("hello!")` . Quando uma página HTML referencia outras URLs como parte de si, por exemplo um arquivo de imagem ou um *script*, os navegadores irão buscá-los imediatamente e incluí-los na página.

Uma *tag* de *script* deve sempre ser fechada com `</script>` , mesmo quando fizer referência para um arquivo externo e não contenha nenhum código. Se você esquecer disso, o restante da página será interpretado como parte de um *script* .

Alguns atributos podem conter um programa JavaScript. A *tag* `<button>` mostrada abaixo (que aparece como um botão na página) possui um atributo `onclick` , cujo conteúdo será executado sempre que o botão for clicado.

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

Perceba que eu tive que usar aspas simples para a *string* do atributo `onclick` porque aspas duplas já estão sendo usadas para envolver o valor do atributo. Eu também poderia ter usado `"` , mas isso tornaria o programa difícil de ler.

Na caixa de areia

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 são bem intencionadas. Executar programas de pessoas que tenham más intenções é como ter seu computador infectado por vírus, seus dados roubados e suas contas

hackeadas.

Contudo, a atração da *Web* é que você pode navegar sem necessariamente confiar nas páginas que visita. Esse é o motivo pelo qual os navegadores limitam severamente as funções que um programa JavaScript pode fazer: eles não podem bisbilhotar os arquivos do seu computador ou modificar qualquer coisa que não esteja relacionada a página em que foi incorporado.

O isolamento de um ambiente de programação dessa maneira é chamado de *sandboxing*, a ideia é que o programa é inofensivo "brincando" em uma "caixa de areia". Mas você deve imaginar esse tipo específico de caixas de areia como tendo sobre si uma gaiola de grossas barras de aço, o que as torna um pouco diferentes das caixas de areia típicas de *playgrounds*.

A parte difícil do *sandboxing* é permitir que os programas tenham espaço suficiente para serem úteis e ao mesmo tempo impedi-los de fazer qualquer coisa perigosa. Várias funcionalidades úteis, como se comunicar com outros servidores ou ler o conteúdo da área de transferência, podem ser usadas para tarefas problemáticas ou invasivas à privacidade.

De vez em quando, alguém aparece com uma nova forma de burlar as limitações de um navegador e fazer algo prejudicial, variando de vazamentos de alguma pequena informação pessoal até assumir o controle total da máquina onde o navegador está sendo executado. Os desenvolvedores de navegadores respondem "tapando o buraco", e tudo está bem novamente —até que o próximo problema seja descoberto e divulgado, ao invés de ser secretamente explorado por algum governo ou máfia.

Compatibilidade e a guerra dos navegadores

No início da web, um navegador chamado Mosaic dominava o mercado. Depois de alguns anos, quem desequilibrou a balança foi o Netscape, que por sua vez, foi derrotado pelo Internet Explorer da Microsoft. Nos momentos em que um único navegador era dominante, seus desenvolvedores se sentiam no direito de criar, unilateralmente, novas funcionalidades para a web. Como a maior parte dos usuários usava o mesmo navegador, os sites simplesmente começaram a usar esses recursos —sem se importarem com os outros navegadores.

Essa foi a idade das trevas da compatibilidade, frequentemente chamada de "guerra dos navegadores". Os desenvolvedores web não tiveram uma web unificada, mas sim duas ou três plataformas incompatíveis. Para piorar as coisas, os navegadores usados por volta de 2003 eram cheios de *bugs*, e, é claro que esses *bugs* foram diferentes para cada navegador. A vida era difícil para aqueles que escreviam páginas web.

O Mozilla Firefox, uma ramificação sem fins lucrativos do Netscape, desafiou a hegemonia do Internet Explorer no final dos anos 2000. A Microsoft não estava particularmente interessada em se manter competitiva nessa época, o Firefox levou uma parcela do mercado para longe do IE. Pela mesma época, a Google introduziu seu navegador Chrome, e o navegador Safari da Apple ganhou popularidade, levando-nos a uma situação onde existiam quatro grandes "competidores" nesse seguimento ao invés de um.

Os novos navegadores possuíam uma postura mais séria sobre a questão dos padrões e de melhores práticas de engenharia, diminuindo as incompatibilidades e *bugs*. A Microsoft, vendo sua cota de mercado se esfalar, começou a adotar essas atitudes. Se você está começando a aprender sobre desenvolvimento web hoje, considere-se com sorte. As últimas versões da maior parte dos navegadores se comportam de uma maneira uniforme e possuem relativamente menos *bugs*.

Ainda não dá para dizer que a situação é perfeita. Algumas pessoas que usam a web estão, por motivo de inércia ou políticas corporativas, presas a navegadores antigos. Enquanto esses navegadores não "morrerem" completamente, desenvolver sites que funcionem para eles vai exigir uma grande quantidade de conhecimento "misterioso" sobre suas peculiaridades e defeitos. Este livro não tratará dessas peculiaridades. Pelo contrário, tem como objetivo apresentar um estilo de programação moderno e sensato.

O Modelo de Objeto de Documentos (DOM)

Quando você abre uma página web em seu navegador, ele resgata o texto em HTML da página e o interpreta, de maneira semelhante ao que nosso interpretador do [Capítulo 11](#) fazia. O navegador constrói um modelo da estrutura do documento e depois usa esse modelo para desenhar a página na tela.

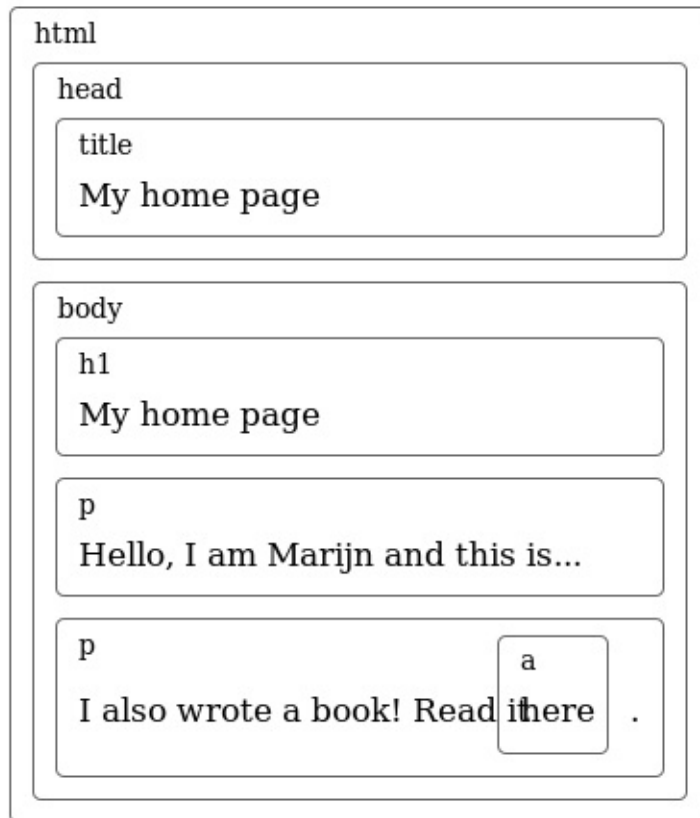
Um dos "brinquedos" que um programa em JavaScript possui disponível em sua caixa de ferramentas é essa representação do documento. Você pode lê-la e também alterá-la. Essa representação age como uma estrutura viva de dados: quando modificada, a página na tela é atualizada para refletir as mudanças.

Estrutura do Documento

Você pode imaginar um documento HTML como um conjunto de caixas aninhadas. Tags como `e` encapsulam outras tags, as quais, por sua vez, contêm outras tags ou texto. Aqui está o documento de exemplo do último capítulo:

```
<html>
  <head>
    <title>Minha home page</title>
  </head>
  <body>
    <h1>Minha home page</h1>
    <p>Olá, eu sou Marijn e essa é minha home page.</p>
    <p>Eu também escrevi um livro! leia-o
      <a href="http://eloquentjavascript.net">aqui</a>.</p>
  </body>
</html>
```

Essa página tem a seguinte estrutura:



A estrutura de dados que o navegador usa para representar o documento segue este formato. Para cada caixa há um objeto, com o qual podemos interagir para descobrir coisas como: qual tag HTML ele representa e quais caixas e textos ele contém. Essa representação é chamada de Modelo de Objeto de Documentos, também apelidada de DOM (do inglês *Document Object Model*).

A variável global `document` nos dá acesso à esses objetos. Sua propriedade `documentElement` se refere ao objeto que representa a tag `<html>`. Essa propriedade também nos fornece as propriedades `head` e `body`, alocando objetos para esses elementos.

Árvores

Relembre-se da sintaxe das árvores do [Capítulo 11](#) por um momento. A estrutura delas é incrivelmente similar a estrutura de um documento do navegador. Cada nó pode se referir a outros nós, "filhos", os quais podem ter, por sua vez, seus próprios "filhos". Esse formato é típico de estruturas aninhadas, nas quais os elementos podem conter subelementos que são similares à eles mesmos.

Nós chamamos uma estrutura de dados de uma *árvore* quando ela possui uma estrutura de galhos, sem ciclos (um nó não deve conter ele mesmo, direta ou indiretamente) e possui uma única, bem definida, raiz. No caso do DOM, `document.documentElement` representa a raiz.

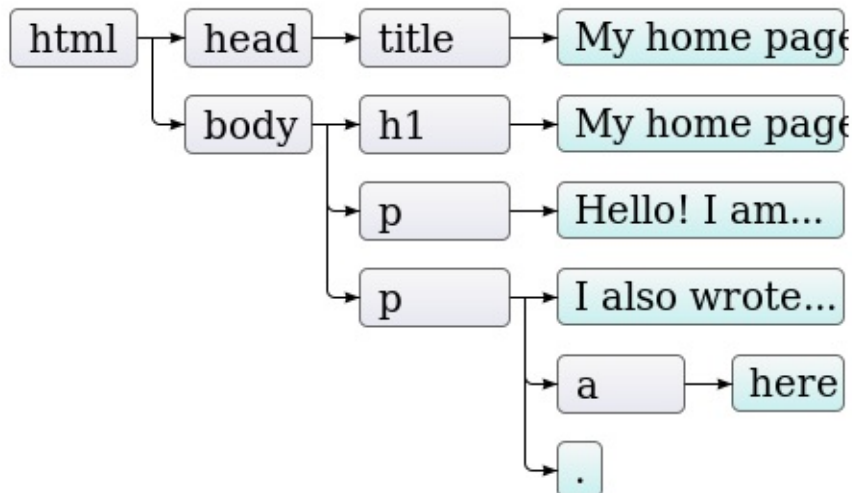
Árvores aparecem muito em Ciências da Computação. Além de representar estruturas recursivas como documentos HTML ou programas, elas também são comumente usadas para manter conjuntos ordenados de dados, pois elementos podem ser tipicamente encontrados ou inseridos de maneira mais eficiente em uma árvore ordenada do que em um conjunto (ou "array") plano ordenado.

Uma árvore típica possui diferentes tipos de nós. A árvore de sintaxe para a [Egg Language](#) continha variáveis, valores e nós de aplicação. Nós de aplicação sempre têm filhos, diferentemente das variáveis e valores, que eram *folhas*, ou seja, nós sem filhos.

O mesmo vale para o DOM. Nós de elementos comuns, os quais representam tags HTML, determinam a estrutura do documento. Esses podem possuir nós filhos. Um exemplo de um desses nós é o `document.body`. Alguns desses nós filhos podem ser folhas, assim como fragmentos de texto ou comentários (os quais são escritos entre `<!--` e `-->` em HTML).

Cada objeto que é um nó do DOM tem a propriedade `nodeType`, a qual contém um código numérico que identifica o tipo do nó. Elementos comuns têm valor 1, o qual também é definido como a propriedade constante `document.ELEMENT_NODE`. Nós de texto, representando um intervalo de texto no documento, possuem o valor 3 (`document.TEXT_NODE`). Comentários têm valor 8 (`document.COMMENT_NODE`).

Sendo assim, outra maneira de visualizar a árvore do nosso documento é:



Na imagem acima, as folhas são os nós de texto e as setas indicam a relação de pai e filho entre os nós.

O Padrão

Usar estranhos códigos numéricos para representar tipos de nós não é algo muito ao estilo JavaScript de se fazer. Mais tarde neste capítulo, veremos que outras partes da interface DOM também se sentem estranhas, *não pertencentes*. A razão para isso é que o DOM não foi concebido apenas para uso com o JavaScript, ao invés disso, ele tenta definir uma interface com uma linguagem neutra, a qual pode ser usada por outros sistemas—não somente HTML, mas também XML, o qual é um formato genérico de dados com um sintaxe semelhante ao HTML.

Padrões são, geralmente, úteis, mas nesse caso, a vantagem (consistência entre diferentes linguagens), não é tão convincente. Possuir uma interface que é corretamente integrada com a linguagem que você está usando vai fazer você economizar mais tempo do que uma interface familiar entre diferentes linguagens.

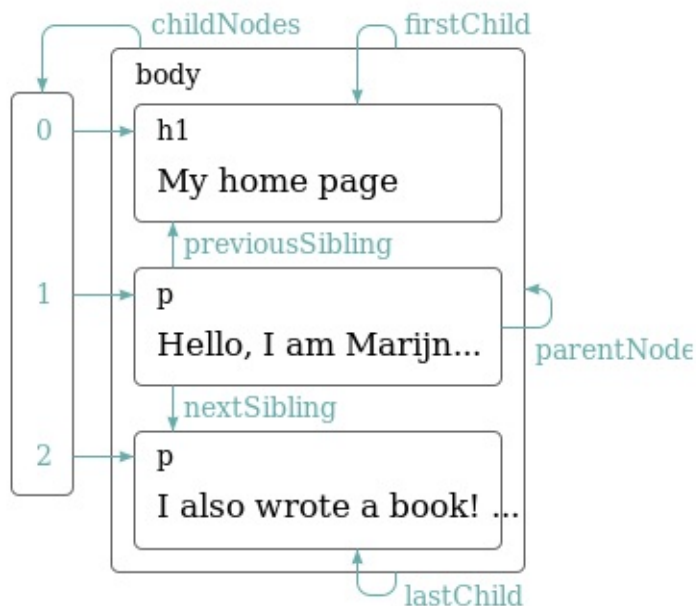
Como um exemplo dessa integração *pobre*, considere a propriedade `childNodes` que os nós de elementos DOM possuem. Essa propriedade carrega um objeto parecido com um array, com uma propriedade `length` e propriedades identificadas por números para acessar os nós filhos. Mas ele é uma instância do tipo `NodeList`, não um array real, logo ele não possui métodos como `slice` e `forEach`.

Além disso existem outros problemas que são simplesmente ocasionados por um design falho. Por exemplo: não há nenhuma maneira de criar um novo nó e imediatamente adicionar nós filhos ou atributos à ele. Ao invés disso, você precisa primeiro criá-lo, depois adicionar os filhos, um por um, e só então definir os atributos um à um usando *side effects*. Códigos que interagem muito com o DOM tendem à ficar muito longos, repetitivos e feios.

Porém nenhuma dessas falhas é fatal, pois JavaScript nos permite criar nossas próprias abstrações. É fácil escrever algumas funções auxiliares que permitem que você expresse as operações que quer fazer de maneira mais curta. Na verdade, muitas *libraries* dedicadas à programação em browsers já vêm com essas ferramentas.

Movendo-se Através da Árvore

Os nós DOM contêm uma variedade de ligações para outros nós próximos. O diagrama a seguir tenta ilustrá-los:



Ainda que o diagrama mostre apenas uma ligação de cada tipo, todo nó possui uma propriedade `parentNode` que aponta para o nó que o contém (seu nó pai). Dessa mesma maneira, todo nó de um elemento (nó do tipo 1) possui a propriedade `childNodes` que aponta para um objeto parecido com um array, o qual contém seus nós filhos.

Em teoria, você pode se mover para qualquer lugar na árvore usando apenas essas ligações entre nós pais e nós filhos, porém JavaScript também te dá acesso às outras ligações muito convenientes. As propriedades `firstChild` e `lastChild` apontam para o primeiro e último elemento filho, respectivamente, ou então possuem o valor `null` (nulo) para nós sem filhos. Similarmente, `previousSibling` e `nextSibling` apontam para os nós adjacentes, que são nós com o mesmo pai que aparecem imediatamente antes ou depois do nó em questão. No caso de usarmos a propriedade `previousSibling` para um primeiro nó filho, ela irá possuir um valor nulo, o mesmo acontece se usarmos a propriedade `nextSibling` para o último nó filho.

Quando lidamos com uma estrutura de dados aninhada como essa, funções recursivas são geralmente muito úteis. A função abaixo escaneia um documento procurando por nós de texto contendo uma determinada string e retorna `true` quando encontrar um.

```
function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (var i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string))
        return true;
    }
    return false;
  } else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

A propriedade `nodeValue` de um nó de texto se refere à string de texto que ele representa.

Encontrando Elementos

Navegar por essas ligações entre pais, filhos e irmãos pode até ser útil, assim como no exemplo da função acima, a qual passa por todo o documento, mas, se quisermos encontrar um nó específico no documento, é uma má ideia começarmos a busca em `document.body` e seguirmos cegamente um caminho preestabelecido de ligações até finalmente achá-lo. Fazer isso irá formular pressuposições no nosso programa sobre a estrutura precisa do documento—uma estrutura que pode mudar depois. Outro fator complicante é que os nós de texto são criados até mesmo por espaço em branco entre nós. A tag `body` que usamos no começo deste capítulo, por exemplo, não tem apenas três filhos (um `<h1>` e dois `<p>` 's), na verdade ela tem sete: esses três e ainda o espaço antes, depois e entre eles.

Então se quisermos obter o atributo `href` do link naquele documento, nós não queremos dizer algo como "pegue o segundo filho do sexto filho do corpo do documento". Seria muito melhor se pudéssemos dizer "pegue o primeiro link nesse documento". E nós podemos.

```
var link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Todos os nós possuem um método `getElementsByTagName`, o qual coleta todos os elementos com o nome passado que são descendentes (filhos diretos ou indiretos) do nó dado e retorna-os no formato de um objeto parecido com um array.

Para encontrar um nó único específico, você pode dar à ele um atributo `id` e usar o método

```
document.getElementById .
```

```
<p>Minha avestruz Gertrude:</p>
<p></p>

<script>
  var ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

Um terceiro método, similar a esse, é o `getElementsByClassName`, o qual, assim como `getElementsByTagName`, faz uma busca entre os conteúdos de um nó elemento e retorna todos os elementos que possuem a string passada no seu atributo `class`.

Alterando o Documento

Quase tudo na estrutura de dados DOM pode ser alterado. Nós de elementos possuem uma variedade de métodos que podem ser usados para mudar seu conteúdo. O método `removeChild` remove um dado nó filho do documento. Para adicionar um filho, nós podemos usar o método `appendChild`, o qual coloca nó filho no fim da lista de filhos, ou até o método `insertBefore`, que insere um nó passado como primeiro argumento antes do nó passado como segundo argumento.

```
<p>Um</p>
<p>Dois</p>
<p>Três</p>

<script>
  var paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

Um nó pode existir no documento em apenas um lugar. Sendo assim, inserir o parágrafo "Três" na frente do parágrafo "Um" vai apenas removê-lo do fim do documento e depois inseri-lo na frente, resultando em "Três/Um/Dois". Todas as operações que inserem um nó em algum lugar irão, como efeito colateral, fazer com que ele seja removido de sua posição atual (caso ele tenha uma).

O método `replaceChild` é usado para substituir um nó filho por outro. Ele aceita como argumentos dois nós: um novo nó e o nó a ser substituído. O nó substituído deverá ser um filho do elemento com o qual o método é chamado. Note que ambos `replaceChild` e `insertBefore` esperam o seu *novo* nó como primeiro argumento.

Criando Nós

No exemplo seguinte, nós queremos escrever um script que substitua todas as imagens (tags ``) no documento pelo texto que elas possuem no seu atributo `alt`, o qual especifica uma alternativa textual para representação da imagem.

Isso envolve não só remover as imagens, mas adicionar um novo nó de texto para substituí-las. Para isso, nós usamos o método `document.createTextNode`.

```
<p>The  in the
  .</p>

<p><button onclick="replaceImages()">Substituir</button></p>

<script>
  function replaceImages() {
    var images = document.body.getElementsByTagName("img");
    for (var i = images.length - 1; i >= 0; i--) {
      var image = images[i];
      if (image.alt) {
        var text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Dada uma string, o método `createTextNode` nos dá um nó do DOM de tipo 3 (um nó de texto), que podemos inserir no nosso documento para que seja mostrado na tela.

O *loop* (ou repetição) que percorre as imagens começa no fim da lista de nós. Isso é necessário porque a lista de nós retornada por um método como `getElementsByTagName` (ou uma propriedade como `childNodes`) é *viva*—isto é, atualizada em tempo real conforme o documento muda. Se nós começássemos pelo início do documento, remover a primeira imagem faria com que a lista perdesse seu primeiro elemento, então na segunda vez que o *loop* se repetisse, quando `i` é um, ele iria parar, pois o comprimento da coleção agora também é um.

Se você quiser um conjunto sólido de nós, em oposição a um conjunto em tempo real, você pode converter o conjunto para um *array* de verdade, chamando o método `slice`.

```
var arrayish = {0: "um", 1: "dois", length: 2};
var real = Array.prototype.slice.call(arrayish, 0);
real.forEach(function(elt) { console.log(elt); });
// → um
//   dois
```

Para criar nós comuns de elementos (tipo 1), você pode usar o método `document.createElement`. Esse método pega o nome de uma tag e retorna um novo nó vazio do tipo fornecido.

O exemplo à seguir define uma função `elt`, a qual cria um nó de elemento e trata o resto dos argumentos como filhos para aquele nó. Essa função é depois usada para adicionar uma simples atribuição para uma citação (em inglês, *quote*).

```
<blockquote id="quote">
  Nenhum livro pode ser terminado. Enquanto trabalhos nele
  nós aprendemos apenas o suficiente para considerá-lo imaturo
  no momento em que damos as costas a ele.
</blockquote>

<script>
  function elt(type) {
    var node = document.createElement(type);
    for (var i = 1; i < arguments.length; i++) {
      var child = arguments[i];
      if (typeof child == "string")
        child = document.createTextNode(child);
      node.appendChild(child);
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "-",
      elt("strong", "Karl Popper"),
      ", prefácio da segunda edição de ",
      elt("em", "A Sociedade Aberta e Seus Inimigos"),
      ", 1950"));
</script>
```

Atributos

Alguns atributos de elementos, como `href` para links, podem ser acessados através de uma propriedade com o mesmo nome do objeto DOM do elemento. Esse é o caso para um conjunto limitado de atributos padrões comumente usados.

HTML permite que você defina qualquer atributo que você queira em nós. Isso pode ser útil, pois pode permitir que você guarde informações extras em um documento. Se você quiser fazer seus próprios nomes de atributos, porém, esses atributos não estarão presentes como uma propriedade no nó do elemento. Ao invés disso, você terá que usar os métodos `getAttribute` e `setAttribute` para trabalhar com eles.

```
<p data-classified="secret">O código de lançamento é 00000000.</p>
<p data-classified="unclassified">Eu tenho dois pés.</p>

<script>
  var paras = document.getElementsByTagName("p");
  Array.prototype.forEach.call(paras, function(para) {
    if (para.getAttribute("data-classified") == "secret")
      para.parentNode.removeChild(para);
  });
</script>
```

Eu recomendo prefixar os nomes dos atributos inventados com `data-`, para certificar-se que eles não irão entrar em conflito com outros atributos.

Como um simples exemplo, nós iremos escrever um "destacador de sintaxe" que procura por tags `<pre>` (tag para "pré-formatado", usado para código ou similares em texto plano) com um atributo `data-language` e tenta destacar as palavras chaves para aquela linguagem.

```
function highlightCode(node, keywords) {
  var text = node.textContent;
  node.textContent = ""; // Limpa o nó.

  var match, pos = 0;
  while (match = keywords.exec(text)) {
    var before = text.slice(pos, match.index);
    node.appendChild(document.createTextNode(before));
    var strong = document.createElement("strong");
    strong.appendChild(document.createTextNode(match[0]));
    node.appendChild(strong);
    pos = keywords.lastIndex;
  }
  var after = text.slice(pos);
  node.appendChild(document.createTextNode(after));
}
```

A função `highlightCode` pega um nó `<pre>` e uma expressão regular (com a opção "global" ligada) que identifica as palavras reservadas da linguagem de programação que o elemento contém.

A propriedade `textContent` é usada para pegar todo o texto dentro do nó e depois é definida para uma string vazia, a qual tem o efeito de esvaziar o nó. Nós fazemos um *loop* por todas as ocorrências das palavras chaves da linguagem, e fazemos o texto entre essas ocorrências como nós normais de texto e cercamos as palavras chaves com a tag `<bold>`, fazendo com que elas fiquem em negrito.

Nós podemos sublinhar automaticamente todos os códigos de programas na página fazendo um *looping* entre todos os elementos `<pre>` que possuem o atributo `data-language` e então chamando a função `highlightCode` em cada um e depois aplicando uma expressão regular adequada para a linguagem que se quer destacar.

```
var languages = {
  javascript: /\b(function|return|var)\b/g /* ... etc */
};

function highlightAllCode() {
  var pres = document.body.getElementsByTagName("pre");
  for (var i = 0; i < pres.length; i++) {
    var pre = pres[i];
    var lang = pre.getAttribute("data-language");
    if (languages.hasOwnProperty(lang))
      highlightCode(pre, languages[lang]);
  }
}
```

Por exemplo:

```
<p>Aqui está, a função identidade:</p>
<pre data-language="javascript">
function id(x) { return x; }
</pre>

<script>highlightAllCode();</script>
```

Existe um atributo comumente usado, `class`, o qual é uma palavra reservada na linguagem JavaScript. Por razões históricas—algumas implementações antigas de JavaScript não conseguiam lidar nomes de propriedades que correspondiam a palavras chave ou palavras reservadas da linguagem—a propriedade usada para acessar esse atributo é chamada de `className`. Você também pode acessá-la pelo seu nome real, `"class"`, usando os métodos `getAttribute` e `setAttribute`.

Layout

Você provavelmente notou que tipos diferentes de elementos são dispostos de maneiras diferentes. Alguns, como parágrafos (`<p>`) ou cabeçalhos (`<h1>`), ocupam toda a largura do documento e são mostrados em linhas separadas. Esses são chamados de elementos *bloco*. Outros, como links (`<a>`) ou o elemento `` , usado no exemplo acima, são mostrados na mesma linha, juntamente com o texto que os cerca. Esses elementos são chamados elementos *inline* (em linha).

Para qualquer documento, navegadores são capazes de computar um layout, o qual dá para cada elemento um tamanho e uma posição baseando-se em seu tipo e conteúdo. Esse layout é depois usado para desenhar o documento na tela.

O tamanho e posição de um elemento pode ser acessado através de JavaScript. As propriedades `offsetWidth` e `offsetHeight` irão fornecer à você o espaço que o elemento ocupa em *pixels*. Um *pixel* é a unidade básica de medida em um navegador e tipicamente corresponde ao menor ponto que sua tela pode mostrar. Do mesmo modo, `clientWidth` e `clientHeight` irão fornecer o espaço *dentro* do elemento, ignorando a largura da borda.

```
<p style="border: 3px solid red">
  Estou encaixotado em
</p>

<script>
  var para = document.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

A maneira mais efetiva de encontrar a posição precisa de um elemento na tela é o método `getBoundingClientRect` . Ele retorna um objeto com as propriedades `top` (topo), `bottom` (baixo), `left` (esquerda) e `right` (direita), que correspondem às posições dos pixels em relação ao canto esquerdo da tela. Se você quiser que eles sejam relativos ao documento como um todo, você deverá adicionar a posição atual de rolagem, encontrada à partir das variáveis globais `pageXOffset` e `pageYOffset` .

Organizar um documento, fazer seu *layout*, pode ser muito trabalhoso. Para ganhar velocidade, os motores dos navegadores não fazem uma reorganização do documento imediatamente a cada vez que ele muda, ao invés disso eles esperam o máximo que podem. Quando um programa JavaScript que mudou o documento termina de rodar, o navegador irá ter que computar um novo *layout* para poder mostrar o documento alterado na tela. Quando um programa pede pela posição ou tamanho de algo, lendo propriedades como `offsetHeight` ou chamando `getBoundingClientRect` , prover a ele uma informação correta também requer computar um *layout*.

Um programa que repetidamente alterna entre ler informações sobre a organização (*layout*) do DOM e alterá-lo, força muitas reorganizações e conseqüentemente compromete o desempenho. O código à seguir mostra um exemplo disso. Ele contém dois programas diferentes que constroem uma linha de "X" caracteres com 2000 pixels de comprimento e mede quanto tempo cada um leva.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    var start = Date.now(); // Tempo atual milissegundos
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", function() {
    var target = document.getElementById("one");
    while (target.offsetWidth < 2000)
      target.appendChild(document.createTextNode("X"));
  });
  // → naive levou 32 ms
```

```
time("clever", function() {
  var target = document.getElementById("two");
  target.appendChild(document.createTextNode("XXXXX"));
  var total = Math.ceil(2000 / (target.offsetWidth / 5));
  for (var i = 5; i < total; i++)
    target.appendChild(document.createTextNode("X"));
});
// → clever levou 1 ms
</script>
```

Estilizando

Nós vimos que diferentes elementos HTML irão apresentar diferentes comportamentos. Alguns são mostrados como blocos, outros são mostrados em linha. Alguns adicionam um tipo de estilo, como `` fazendo seu conteúdo ficar em negrito e `<a>` fazendo seu conteúdo azul e sublinhando-o.

A maneira que uma tag `` mostra uma imagem, e a maneira uma tag `<a>` faz com que um link seja acessado quando é clicado, estão intimamente ligadas ao tipo do elemento. Mas o estilo padrão associado à um elemento, assim como a cor de texto ou sublinhado, pode ser mudado por nós. Veja o exemplo abaixo usando a propriedade `style`.

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Link verde</a></p>
```

Um atributo `style` pode conter um ou mais declarações, as quais são propriedades (como por exemplo `color`) seguidas por dois pontos e um valor (assim como `green`). Caso existam múltiplas declarações, elas deverão ser separadas por pontos e vírgulas. Por exemplo, `"color: red;border:none"`

Existem muitos aspectos que podem ser influenciados através dessa estilização. Por exemplo, a propriedade `display` controla quando um elemento é mostrado como um bloco ou em linha.

```
Esse texto é mostrado <strong>em linha</strong>,
<strong style="display: block">como um bloco</strong>, e
<strong style="display: none">não é mostrado</strong>.
```

A tag `block` vai acabar em sua própria linha, pois elementos em blocos não são mostrados em linha com texto ao seu redor. A última tag não é mostrada, `display: none` faz com que um elemento seja exibido na tela. Essa é uma maneira de esconder elementos e é comumente preferível à removê-los por completo do documento, tornando mais fácil revelá-los mais tarde.

Código JavaScript pode manipular diretamente o estilo de um elemento através da propriedade `style` do nó. Essa propriedade carrega um objeto que possui todas as propriedades possíveis para o atributo `style`. Os valores dessas propriedades são strings, os quais nós podemos escrever para mudar um aspecto em particular do estilo do elemento.

```
<p id="para" style="color: purple">
  Texto bonito
</p>

<script>
  var para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Alguns nomes de propriedades de estilo contêm traços, como `font-family`. Devido ao fato desses nomes de propriedades serem estranhos para serem trabalhados em JavaScript (você teria que escrever `style["font-family"]`), os nomes de propriedades no objeto `style`, nestes casos, têm seus traços removidos e a letra após eles é tornada maiúscula (`style.fontFamily`).

Estilos em Cascata

O sistema de estilos para HTML é chamado de CSS, que é uma abreviação para *Cascading Style Sheets* (Folhas de Estilo em Cascata, em português). Uma folha de estilos é um conjunto de regras de como estilizar os elementos no documento. Ela pode ser fornecida dentro de uma tag `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: grey;
  }
</style>
<p>Agora <strong>textos com tag strong</strong> são itálicos e cinza.</p>
```

A palavra *cascata* no nome refere-se ao fato de que múltiplas regras são combinadas para produzir o estilo final de um elemento, aplicando-se em "cascata". No exemplo acima, o estilo padrão para as tags ``, o qual dá à eles `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 tem um nível de preferência maior e vence. Então se a regra na tag `<style>` incluísse `font-weight: normal`, conflitando com a regra `font-weight` padrão, o texto seria normal e não em negrito. Estilos em um atributo `style` aplicados diretamente ao nó possuem maior preferência e sempre vencem.

É possível selecionar outras coisas além de nomes de tags em regras CSS. Uma regra para `.abc` aplica-se para todos os elementos com "abc" no seu atributo classe. Uma regra para `#xyz` aplica-se para o elemento com um atributo `id` de "xyz" (o qual deve ser único dentro do documento).

```
.subtle {
  color: grey;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* Elementos p, com classes a e b, e id main */
p.a.b#main {
  margin-bottom: 20px;
}
```

A regra de preferência favorecendo a regra mais recentemente definida é válida somente quando as regras possuem a mesma especificidade. A especificidade de uma regra é uma medida de o quão precisamente ela descreve os elementos que seleciona, sendo determinada por um número e um tipo (tag, classe ou id) de um aspecto do elemento que requer. Por exemplo, `p.a` é mais específico que apenas um `p` ou apenas um `.a`, então uma regra composta como essa teria preferência.

A notação `p > a {...}` aplica os estilos passados para todas as tags `<a>` que são filhos diretos de tags `<p>`. Do mesmo modo, `p a {...}` aplica-se à todas as tags `<a>` dentro de tags `<p>`, sejam elas filhos diretos ou indiretos.

Seletores de Busca

Nós não iremos usar muitas folhas de estilo neste livro. Ainda assim, entendê-las é crucial para programar no navegador, explicar todas as propriedades que elas suportam de maneira correta e a interação entre essas propriedades levaria dois ou três livros somente para isso.

A razão principal pela qual eu introduzi a sintaxe de *seletores*—a notação usada em folhas de estilo para definir a qual elemento um conjunto de regras se aplica—é que nós podemos usar essa mesma mini linguagem para definir uma maneira eficaz de encontrar elementos do DOM.

O método `querySelectorAll`, que é definido em tanto no objeto `document` quanto nos nós de elementos, leva apenas uma string seletora e retorna um objeto parecido um array, contendo todos os elementos que encontra.

```
<p>Se você sair por aí caçando
  <span class="animal">coelhos</span></p>
<p>E você souber que vai cair</p>
<p>Diga à eles que <span class="character">enquanto fumava narguilé,
  <span class="animal">uma lagarta</span></span></p>
<p>Lhe deu a ordem</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")); // Filhos diretos de <p>
  // → 1
</script>
```

Diferentemente de métodos como `getElementsByTagName`, o objeto retornado por `querySelectorAll` não é "vivo", ou seja, atualizado em tempo real. Ele não irá mudar quando você mudar o documento.

O método `querySelector` (sem a parte `All`) funciona de maneira similar. Ele é útil para quando você quiser um único e específico elemento. Ele irá retornar apenas o primeiro elemento coincidente com a busca ou `null` se nenhum elemento cumprir os critérios de busca.

Posicionando e Animando

A propriedade de estilo `position` influencia o layout de uma maneira muito poderosa.

Por padrão, essa propriedade tem o valor `static`, significando que o elemento fica em seu lugar "absoluto", estático. Quando essa propriedade é definida como `relative`, o elemento ainda ocupa espaço no documento, mas agora as propriedades `top` e `left` podem ser usadas para movê-lo em relação ao seu lugar original. Quando `position` é definida como `absolute` o elemento é removido do fluxo normal do documento—isso é, não ocupa mais espaço e pode sobrepor outros elementos—e suas propriedades `top` e `left` podem ser usadas para posiciiná-lo de maneira absoluta em relação ao canto superior esquerdo do elemento fechado mais próximo cuja propriedade `position` não é estática. Se não houver tal elemento, ele é posicionado em relação ao documento.

Nós podemos usar essa técnica para criar uma animação. O documento abaixo mostra uma foto de um gato que flutua seguindo a forma de uma elipse:

```
<p style="text-align: center">
  
</p>
<script>
  var cat = document.querySelector("img");
```

```

var angle = 0, lastTime = null;
function animate(time) {
  if (lastTime != null)
    angle += (time - lastTime) * 0.001;
  lastTime = time;
  cat.style.top = (Math.sin(angle) * 20) + "px";
  cat.style.left = (Math.cos(angle) * 200) + "px";
  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
</script>

```

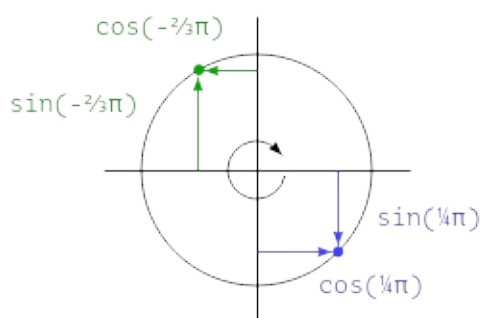
Primeiro a imagem é centralizada na página e dada uma posição do tipo `relative`. Nós vamos atualizar repetidamente as propriedades de estilo `top` e `left` dessa imagem para movê-la.

O script usa `requestAnimationFrame` para agendar a função `animate` para rodar sempre que o navegador estiver preparado para redesenhar na tela. A função `animate` por sua vez, chama `requestAnimationFrame` para agendar a próxima atualização do elemento. Quando a janela ou aba do navegador está ativa, isso irá fazer com que as atualizações ocorram em uma taxa de cerca de 60 por segundo, o que tende a produzir uma animação com um bom aspecto.

Se nós apenas atualizássemos o DOM em um loop, a página iria congelar e nada seria mostrado na tela. Navegadores não fazem atualizações do que mostram enquanto um programa JavaScript está rodando e também não permitem qualquer interação com a página durante esse período. Esse é o motivo pelo qual nós precisamos da função `requestAnimationFrame` —ela permite que o navegador saiba que nós estamos satisfeitos por enquanto e que ele pode ir em frente e fazer as coisas que os navegadores geralmente fazem, como atualizar a tela e responder às ações do usuário.

Nossa função de animação recebe como argumento o tempo atual, o qual é comparado com o tempo recebido anteriormente (nesse caso, a variável `lastTime`) para ter certeza que o movimento do gato por milissegundo é estável, e então a animação se move suavemente. Se ela se movesse uma porcentagem fixa à cada passo, o movimento iria sofrer atraso se, por exemplo, outra tarefa que exige muito processamento no mesmo computador acabasse impedindo com que a função fosse executada por uma fração de segundo.

`Math.cos` (cosseno) e `Math.sin` (seno) são úteis para achar pontos que se localizam em um círculo ao redor de um ponto (0,0) com o raio de uma unidade. Ambas as funções interpretam seu argumento como a posição nesse círculo, com 0 significando o ponto na extrema direita do círculo, indo em sentido horário até 2π (cerca de 6.28) nos levou ao redor de todo o círculo. `Math.cos` informa a coordenada x (no plano cartesiano) do ponto que corresponde à dada posição no círculo, enquanto `Math.sin` informa a coordenada y. Posições (ou ângulos) maiores que 2π ou abaixo de 0 são válidos—a rotação se repete, de modo que $a+2\pi$ refere-se ao mesmo ângulo que a .



A animação do gato mantém um contador, `angle`, para o ângulo atual da animação, e incrementa-o proporcionalmente ao tempo decorrido a cada vez que a função `animate` é chamada. Ela pode usar esse ângulo para computar a posição atual do elemento de imagem. A propriedade de estilo `top` é computada com `Math.sin` e multiplicada por 20, que é o raio vertical do nosso círculo. O estilo `left` é baseado em `Math.cos` e multiplicado por 200, de maneira que o círculo é muito mais largo do que alto, resultando em uma rotação elíptica.

Note que os estilos geralmente precisam de *unidades*. Nesse caso, nós temos que inserir " px " para o número com o intuito de dizer ao navegador que nós estamos contando em pixels (não em centímetros, "ems" ou outras unidades). Isso é algo fácil de esquecer. Usar números sem unidades vai resultar em uma regra de estilo ignorada—exceto se o número for 0, que sempre significa a mesma coisa, não importando a unidade.

Resumo

Programas JavaScript podem inspecionar e interferir com o documento atual cujo navegador está mostrando através de uma estrutura de dados chamada DOM. Essa estrutura representa o modelo do documento feito pelo navegador e um programa JavaScript pode modificá-la para mudar o documento que está sendo mostrado.

O DOM é organizado como uma árvore, na qual elementos são organizados hierarquicamente de acordo com a estrutura do documento. Os objetos representando elementos possuem propriedades como `parentNode` e `childNodes`, que podem ser usadas para navegar pela árvore.

A maneira com que um documento é mostrada pode ser influenciada através da *estilização*, tanto anexando estilos diretamente à um nó ou definindo regras que aplicam-se à certos nós. Existem muitas propriedades de estilo diferentes, assim como `color` ou `display`. JavaScript pode manipular o estilo de um elemento diretamente através de sua propriedade `style`.

Exercícios

Construa uma Tabela

Nós construímos tabelas de texto plano no [Capítulo 6](#). O HTML faz com que mostrar tabelas seja um pouco mais fácil. Uma tabela em HTML é construída com a seguinte estrutura de tags:

```
<table>
  <tr>
    <th>nome</th>
    <th>altura</th>
    <th>país</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

Para cada *sequência* (linha), a tag `<table>` contém uma tag `<tr>`. Dentro dessa tag nós podemos colocar elementos célula: ou células de cabeçalho (`<th>`) ou células comuns (`<td>`).

A mesma fonte de dados usada no [Capítulo 6](#) está disponível novamente na variável `MOUNTAINS`, disponível em nossa *sandbox* e também [disponível para download](#) na nossa lista de conjunto de dados no [website\(eloquentjavascript.net/code\)](http://eloquentjavascript.net/code).

Escreva uma função `buildTable` que, dado um array de objetos com um mesmo conjunto de propriedades, construa uma estrutura DOM representando uma tabela. A tabela deve ter uma sequência (linha) de cabeçalho com os nomes das propriedades dentro de elementos `<th>` e uma linha subsequente por objeto no array, com seus valores das propriedades em elementos `<td>`.

A função `Object.keys`, a qual retorna um array contendo os nomes das propriedades que um objeto possui, provavelmente vai ser útil nesse caso.

Uma vez que você fez a parte básica funcionar, alinhe as células que contêm números à direita usando a propriedade `style.textAlign` com o valor `"right"`.

```
<style>
/* Define uma visualização mais limpa para tabelas */
table { border-collapse: collapse; }
td, th { border: 1px solid black; padding: 3px 8px; }
th { text-align: left; }
</style>

<script>
function buildTable(data) {
  // Seu código aqui.
}

document.body.appendChild(buildTable(MOUNTAINS));
</script>
```

Dicas

Use `document.createElement` para criar novos nós de elementos, `document.createTextNode` para criar nós de texto e o método `appendChild` para colocar nós dentro de outros nós.

Você deve fazer um loop através das palavras chaves uma vez para preencher a linha do topo e depois novamente para cada objeto no array para construir linhas com os dados.

Não esqueça de retornar o elemento `<table>` que engloba tudo isso no fim da função.

Elementos por Nome de Tags

O método `getElementsByTagName` retorna todos os elementos filhos com um dado nome de `tag`. Implemente sua própria versão disso, como uma função normal, a qual recebe um nó e uma string (com o nome da tag) como argumentos e retorna um array contendo todos os nós de elementos descendentes com a dada tag.

Para encontrar o nome de tag de um elemento, use sua propriedade `tagName`. Mas note que isso irá retornar o mesmo nome de tag todo em caixa alta. Use os métodos `toLowerCase` ou `toUpperCase` para compensar isso.

```
<h1>Cabeçalho com um elemento <span>span</span>.</h1>
<p>Um parágrafo com <span>um</span>, <span>dois</span>
spans.</p>

<script>
function byTagName(node, tagName) {
  // Seu código aqui.
}

console.log(byTagName(document.body, "h1").length);
// → 1
console.log(byTagName(document.body, "span").length);
// → 3
var para = document.querySelector("p");
console.log(byTagName(para, "span").length);
// → 2
</script>
```

Dicas

A solução é expressada mais facilmente com uma função recursiva, similar à função `talksAbout` definida anteriormente neste capítulo.

Você pode chamar `byTagName` usando ela mesma, ou seja, de maneira recursiva, concatenando os arrays resultantes para produzir uma saída. Uma aproximação mais eficiente à esse problema envolve definir uma função interior à qual chama a si mesma recursivamente, a qual tem acesso a qualquer posição do array definida na função exterior, nas quais ela pode adicionar os elementos que encontrar. Não esqueça de chamar a função interior através da função exterior uma vez.

A função recursiva deve checar o tipo de nó. Aqui nós estamos interessados apenas no nó tipo 1 (`document.ELEMENT_NODE`). Para tais nós, nós deveremos fazer um loop sobre seus filhos e, para cada filho, ver se ele cumpre nossos critérios de busca e também fazer uma chamada recursiva para inspecionar, por sua vez, seus próprios filhos.

O Chapéu do Gato

Extenda a animação do gato definida anteriormente de modo que tanto o gato quando seu chapéu (``) façam a mesma órbita em lados diferentes da elipse.

Ou faça o chapéu circular ao redor do gato. Você pode ainda alterar a animação de outra maneira que julgar interessante.

Para tornar mais fácil a tarefa de posicionar múltiplos objetos, é provavelmente uma boa idéia optar por posicionamento absoluto. Isso significa que as propriedades `top` e `left` são contadas relativamente ao topo esquerdo do documento. Para evitar usar coordenadas negativas, você pode simplesmente adicionar um número fixo de pixels para os valores das posições.

```



<script>
  var cat = document.querySelector("#cat");
  var hat = document.querySelector("#hat");
  // Seu código aqui.
</script>
```


Manipulando eventos

Você tem o poder sobre sua mente e não sobre eventos externos. Perceba isso e você encontrara resistência.

Marcus Aurelius, *Meditations*

Alguns programas funcionam com entradas direta do usuário, tais como a interação de mouse e teclado. O tempo e a ordem de tal entrada não pode ser previsto com antecedência. Isso requer uma abordagem diferente para controlar o fluxo do que utilizamos até agora.

Os manipuladores de eventos

Imaginem uma interface onde a única maneira de descobrir se uma tecla está sendo pressionada é ler o estado atual dessa tecla. Para ser capaz de reagir às pressões de teclas você teria que ler constantemente o estado da tecla antes que ela fique liberado novamente. Seria perigoso executar outros cálculos demoradas pois você poderia perder alguma tecla.

É assim que tal atributo foi tratado em máquinas primitivas. A um passo para o hardware, o sistema operacional deve notificar qual a tecla pressionada e colocá-lo em uma fila. Um programa pode então verificar periodicamente a fila para novos eventos e reagir ao encontrar.

É claro que ha sempre uma responsabilidade de verificar a fila e executá-las várias vezes, isso é necessário porque ha uma latência entre a pressão da tecla e a leitura da fila pelo programa com isso o software pode sentir que não esta tendo resposta. Esta abordagem é chamada de `polling`. A maioria dos programadores tentam evitar essa abordagem sempre que possível.

A melhor mecanismo para o sistema subjacente é dar ao nosso código a chance de reagir a eventos que ocorrerem. Os browsers podem fazerem isto por que nos permite registrar funções como manipuladores para eventos específicos.

```
<p>Click this document to activate the handler.</p>
<script>
  addEventListener("click", function() {
    console.log("You clicked!");
  });
</script>
```

A função `addEventListener` registra seu segundo argumento sempre que o evento descrito por seu primeiro argumento é chamado.

Eventos e nós do DOM

Cada navegador tem seu manipulador de eventos registrado em um contexto. Quando você chamar `addEventListener` como mostramos anteriormente você estara chamando um método em todo `window` no navegador o escopo global é equivalente ao objeto `window`. Cada elemento DOM tem seu próprio método `addEventListener` que permite ouvir eventos especificamente para cada elemento.

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  var button = document.querySelector("button");
  button.addEventListener("click", function() {
    console.log("Button clicked.");
  });
```

```
});  
</script>
```

O exemplo atribuiu um manipulador para um nó de botão. Assim quando existir um clique no botão o manipulador será executado, enquanto no resto do documento não.

Dar a um nó um atributo `onclick` tem um efeito similar. Mas um nó tem apenas um atributo `onclick` para que você possa registrar apenas um manipulador por nó para que você não substitua acidentalmente um manipulador que já foi registrado. O método `addEventListener` permite que você adicione vários manipuladores.

O método `removeEventListener` quando chamado com argumentos é semelhante ao `addEventListener`, mas ele remove o manipulador que foi registrado.

```
<button>Act-once button</button>  
<script>  
  var button = document.querySelector("button");  
  function once() {  
    console.log("Done.");  
    button.removeEventListener("click", once);  
  }  
  button.addEventListener("click", once);  
</script>
```

Ele é capaz de cancelar um registro de manipulador de uma função, precisamos dar-lhe um nome para que possamos utilizar tanto para `addEventListener` quanto para `removeEventListener`.

Os objetos de evento

Embora tenhamos ignorado os exemplos anteriores as funções manipuladoras de eventos são passados via argumento e chamamos de objeto de evento. Este objeto nos dá informações adicionais sobre o evento. Por exemplo, se queremos saber qual botão do mouse que foi pressionado podemos observar as propriedades do objeto de evento.

```
<button>Click me any way you want</button>  
<script>  
  var button = document.querySelector("button");  
  button.addEventListener("mousedown", function(event) {  
    if (event.which == 1)  
      console.log("Left button");  
    else if (event.which == 2)  
      console.log("Middle button");  
    else if (event.which == 3)  
      console.log("Right button");  
  });  
</script>
```

As informações armazenadas em um objeto de evento são diferentes dependendo do tipo de evento. Vamos discutir vários tipos mais adiante neste capítulo. Propriedade de tipo do objeto sempre detém uma cadeia que identifica o evento (por exemplo, `"click"` ou `"mousedown"`).

Propagação

Os manipuladores de eventos registrados em nós também receberão alguns eventos que ocorrem nos filhos. Se um botão dentro de um parágrafo é clicado manipuladores de eventos no parágrafo também vai receber o evento

`click`.

Mas se tanto o parágrafo e o botão tem um manipulador o manipulador mais específico é o do botão e será chamado primeiro. O evento foi feito para propagar para o exterior a partir do nó onde aconteceu até o nó pai do nó raiz do documento. Finalmente depois de todos os manipuladores registrados em um nó específico tiveram sua vez manipuladores registrados em todo `window` tem a chance de responder ao evento.

A qualquer momento um manipulador de eventos pode chamar o método `stopPropagation` para evitar que os manipuladores mais acima possam receber o evento. Isso pode ser útil quando por exemplo, se você tem um botão dentro de outro elemento clicável e você não quer o clique no botão aconteça se houver algum compartimento de clique no elemento exterior.

O exemplo a seguir registra manipuladores `"mousedown"` em ambos no botão e no parágrafo e em torno dele. Quando clicado com o botão direito do mouse o manipulador do botão chama `stopPropagation`, o que impedirá o manipulador no parágrafo de executar. Quando o botão é clicado com outro botão do mouse os dois manipuladores são executados.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  var para = document.querySelector("p");
  var button = document.querySelector("button");
  para.addEventListener("mousedown", function() {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", function(event) {
    console.log("Handler for button.");
    if (event.which == 3)
      event.stopPropagation();
  });
</script>
```

A maioria dos objetos de evento tem uma propriedade de destino que se refere ao nó onde eles se originaram. Você pode usar essa propriedade para garantir que você não está lidando com algo que acidentalmente propagou a partir de um nó que você não queira lidar.

Também é possível usar uma propriedade de destino para lançar uma ampla rede para um tipo específico de evento. Por exemplo, se você tem um nó que contém uma longa lista de botões, pode ser mais conveniente registrar um único manipulador de clique para o nó do exterior e que ele use a propriedade de destino para descobrir se um botão foi clicado, ao invés de se registrar manipuladores individuais sobre todos os botões.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", function(event) {
    if (event.target.nodeName == "BUTTON")
      console.log("Clicked", event.target.textContent);
  });
</script>
```

Ações padrão

Muitos eventos têm sua ação padrão que lhes estão associados. Se você clicar em um link você será levado para outra página. Se você pressionar a seta para baixo o navegador vai rolar a página para baixo. Se você clicar com o botão direito você terá um menu e assim por diante.

Para a maioria dos tipos de eventos, os manipuladores de eventos de JavaScript são chamados antes do comportamento padrão. Se o condutor não quer que o comportamento normal aconteça pode simplesmente chamar o método `preventDefault` no objeto de evento.

Isso pode ser usado para implementar seus próprios atalhos de teclado ou menus. Ele também pode ser utilizado para interferir como um comportamento desagradavelmente que os utilizadores não esperam. Por exemplo aqui está um link que não podem ser clicável:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  var link = document.querySelector("a");
  link.addEventListener("click", function(event) {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Tente não fazer tais coisas, a menos que você tem uma boa razão para isso. Para as pessoas que usam sua página isso pode ser desagradável quando o comportamento que eles esperam são quebrados.

Dependendo do navegador alguns eventos não podem ser interceptados. No Chrome por exemplo, os atalhos de teclado para fechar a aba atual (Ctrl- W ou Command-W) não pode ser manipulado por JavaScript.

Evento de tecla

Quando uma tecla do teclado é pressionado, o seu browser dispara um evento `"keydown"` quando ele é liberado um evento de `"keyup"` é emitido.

```
<p>This page turns violet when you hold the V key.</p>
<script>
  addEventListener("keydown", function(event) {
    if (event.keyCode == 86)
      document.body.style.background = "violet";
  });
  addEventListener("keyup", function(event) {
    if (event.keyCode == 86)
      document.body.style.background = "";
  });
</script>
```

O evento `"keydown"` é acionado não só quando a tecla fisicamente é empurrada para baixo. Quando uma tecla é pressionada e mantida o evento é disparado novamente toda vez que se repete a tecla. Por exemplo se você quiser aumentar a aceleração de um personagem do jogo quando uma tecla de seta é pressionado e diminuído somente quando a tecla é liberada você tem que ter cuidado para não aumentá-lo novamente toda vez que se repete a tecla ou vai acabar com os valores involuntariamente enormes.

O exemplo anterior nos atentou para a propriedade `keyCode` do objeto de evento. Isto é como você pode identificar qual tecla está sendo pressionada ou solta. Infelizmente não é sempre óbvio traduzir o código numérico para uma tecla.

Para as teclas de letras e números, o código da tecla associado será o código de caracteres Unicode associado as letras maiúsculas ou número impresso na tecla. O método `charCodeAt` que pertence a propriedade `String` nos dá uma maneira de encontrar este código.

```
console.log("Violet".charCodeAt(0));
// → 86
console.log("1".charCodeAt(0));
// → 49
```

Outras teclas têm códigos previsíveis. A melhor maneira de encontrar os códigos que você precisa é geralmente experimentar o registro de um manipulador de eventos de tecla que registra os códigos de chave que ela recebe quando pressionado a tecla que você está interessado.

Teclas modificadoras como Shift, Ctrl, Alt e Command(do Mac) geram eventos de teclas apenas como teclas normais. Mas quando se olha para as combinações de teclas, você também pode descobrir se essas teclas são pressionadas verificando as propriedades de eventos `shiftKey`, `ctrlKey`, `altKey` e `metaKey` tanto para teclado quanto para mouse.

```
<p>Press Ctrl-Space to continue.</p>
<script>
  addEventListener("keydown", function(event) {
    if (event.keyCode == 32 && event.ctrlKey)
      console.log("Continuing!");
  });
</script>
```

Os eventos de `"keydown"` e `"keyup"` dão informações sobre a tecla física que está sendo pressionado. Mas e se você está interessado no texto que está sendo digitado? Conseguir o texto a partir de códigos de tecla é estranho. Em vez disso existe um outro evento `"keypress"` que é acionado logo após `"keydown"` (repetido junto com `"keydown"` quando a tecla é solta) mas apenas para as teclas que produzem entrada de caracteres. A propriedade `charCode` no objeto do evento contém um código que pode ser interpretado como um código de caracteres Unicode. Podemos usar a função `String.fromCharCode` para transformar esse código em uma verdadeira cadeia de caracteres simples.

```
<p>Focus this page and type something.</p>
<script>
  addEventListener("keypress", function(event) {
    console.log(String.fromCharCode(event.charCode));
  });
</script>
```

O nó `DOM` onde um evento de tecla se origina depende do elemento que tem o foco quando a tecla for pressionada. Nós normais não podem ter o foco(a menos que você de um atributo `tabindex`) o foco ocorre normalmente para os nós links, botões e campos de formulário. Voltaremos a formar campos no Capítulo 18. Quando nada em particular tem foco `document.body` é o um dos principais eventos dos destinos principais.

Evento de mouse

Pressionar o botão do mouse também provoca uma série de eventos para ser emitido. O `"mousedown"` e `"mouseup"` são semelhantes aos `"keydown"` e `"keyup"` e são acionados quando o botão é pressionado e liberado. Estes irão acontecer no DOM que estão abaixo do ponteiro do mouse quando o evento ocorrer.

Após o evento de `"mouseup"` um evento `"click"` é acionado no nó mais específico que continha tanto ao pressionar e liberar o botão. Por exemplo se eu pressionar o botão do mouse em um parágrafo e em seguida, movo o ponteiro para outro parágrafo e solto o botão o evento de `"click"` acontecerá em ambos parágrafos.

Se dois cliques acontecem juntos um evento de `"dblclick"` (clique duplo) é emitido também após o segundo evento de clique.

Para obter informações precisas sobre o local onde aconteceu um evento do mouse você pode olhar para as suas propriedades `pageX` e `pageY`, que contêm as coordenadas do evento(em pixels) em relação ao canto superior esquerdo do documento.

A seguir veja a implementação de um programa de desenho primitivo. Toda vez que você clique no documento ele acrescenta um ponto sob o ponteiro do mouse. Veja o Capítulo 19 um exemplo de programa de desenho menos primitivo.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  addEventListener("click", function(event) {
    var 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>
```

As propriedades `clientX` e `clientY` são semelhantes aos `pageX` e `pageY` mas em relação à parte do documento que está sendo rolado. Estes podem ser úteis quando se compara a coordena do mouse com as coordenadas retornados por `getBoundingClientRect` que também retorna coordenadas relativas da `viewport`.

Movimento do mouse

Toda vez que o ponteiro do mouse se move, um eventos de `"mousemove"` é disparado. Este evento pode ser usado para controlar a posição do mouse. Uma situação comum em que isso é útil é ao implementar algum tipo de funcionalidade de arrastar o mouse.

O exemplo a seguir exibe um programa com uma barra e configura os manipuladores de eventos para que ao arrastar para a esquerda ou direita a barra se torna mais estreita ou mais ampla:

```
<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  var lastX; // Tracks the last observed mouse X position
  var rect = document.querySelector("div");
  rect.addEventListener("mousedown", function(event) {
    if (event.which == 1) {
      lastX = event.pageX;
      addEventListener("mousemove", moved);
      event.preventDefault(); // Prevent selection
    }
  });

  function moved(event) {
    if (event.which != 1) {
      removeEventListener("mousemove", moved);
    } else {
      var dist = event.pageX - lastX;
      var newWidth = Math.max(10, rect.offsetWidth + dist);
      rect.style.width = newWidth + "px";
      lastX = event.pageX;
    }
  }
}
```

```
}
</script>
```

Note que o controlador `"mousemove"` é registrado no `window`. Mesmo que o mouse vai para fora da barra durante o redimensionamento nós ainda queremos atualizar seu tamanho e parar de arrastar somente quando o mouse é liberado.

Sempre que o ponteiro do mouse entra ou sai de um nó um evento de `"mouseover"` ou `"mouseout"` é disparado. Esses dois eventos podem ser usados entre outras coisas, para criar efeitos de foco, mostrando um denominado algo quando o mouse está sobre um determinado elemento.

Infelizmente não é tão simples de ativar a criação de um tal efeito com `"mouseover"` e acabar com ela em `"mouseout"`. Quando o mouse se move a partir de um nó em um dos seus filhos `"mouseout"` é acionado no nó pai, embora o mouse não chegou a deixar extensão do nó. Para piorar as coisas esses eventos se propagam assim como outros eventos, portanto você também receberá eventos `"mouseout"` quando o mouse deixa um dos nós filhos do nó em que o manipulador é registrado.

Para contornar este problema, podemos usar a propriedade `relatedTarget` dos objetos de eventos criados por esses eventos. Ele garante que no caso de `"mouseover"` o elemento que o ponteiro do mouse passou antes e no caso do `"mouseout"` o elemento que o ponteiro do mouse ira passar. Nós queremos mudar o nosso efeito `hover` apenas quando o `relatedTarget` está fora do nosso nó de destino. Neste caso é que este evento realmente representa um cruzamento de fora para dentro do nó(ou ao contrário).

```
<p>Hover over this <strong>paragraph</strong>.</p>
<script>
  var para = document.querySelector("p");
  function isInside(node, target) {
    for (; node != null; node = node.parentNode)
      if (node == target) return true;
  }
  para.addEventListener("mouseover", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "red";
  });
  para.addEventListener("mouseout", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "";
  });
</script>
```

A função `isInside` percorre os links pai do nó até ele atingir o topo do documento(quando nulo) ou encontrar o pai que está procurando.

Devo acrescentar que um efeito `hover` como isso pode ser muito mais facilmente alcançado utilizando o pseudo selector em CSS `:hover` como o exemplo a seguir mostra. Mas quando o seu efeito `hover` envolve fazer algo mais complexo do que apenas mudar um estilo no nó de destino, você deve usar o truque com os eventos de `"mouseover"` e `"mouseout"`.

```
<style>
  p:hover { color: red }
</style>
<p>Hover over this <strong>paragraph</strong>.</p>
```

Evento de rolagem

Sempre que um elemento é rolado um evento de `"scroll"` é disparado sobre ele. Isto tem vários usos como saber o que o usuário está olhando(para desativar animações fora da tela ou o envio de relatórios de espionagem para o seu quartel general) ou apresentar alguma indicação de progresso(por destacar parte de uma tabela de conteúdo ou que mostra um número de página).

O exemplo a seguir desenha uma barra de progresso no canto superior direito do documento e atualiza enchendo quando é rolada para baixo:

```
<style>
.progress {
  border: 1px solid blue;
  width: 100px;
  position: fixed;
  top: 10px; right: 10px;
}
.progress > div {
  height: 12px;
  background: blue;
  width: 0%;
}
body {
  height: 2000px;
}
</style>
<div class="progress"><div></div></div>
<p>Scroll me...</p>
<script>
var bar = document.querySelector(".progress div");
addEventListener("scroll", function() {
  var max = document.body.scrollHeight - innerHeight;
  var percent = (pageYOffset / max) * 100;
  bar.style.width = percent + "%";
});
</script>
```

Um elemento com uma posição fixa é muito parecido com um elemento de posição absoluta, mas ambos impedem a rolagem junto com o resto do documento. O efeito é fazer com que nossa barra de progresso pare no canto. Dentro dele existe outro elemento que é redimensionada para indicar o progresso atual. Usamos `%` em vez de `px` como unidade, definimos a largura de modo que quando o elemento é dimensionado em relação ao conjunto da barra.

A variável `innerHeight` nos dá a altura de `window`, devemos subtrair do total altura de sua rolagem para não ter rolagem quando você chegar no final do documento.(Há também uma `innerWidth` que acompanha o `innerHeight`.) Ao dividir `pageYOffset` a posição de rolagem atual menos posição de deslocamento máximo multiplicando por 100 obtemos o percentual da barra de progresso.

Chamando `preventDefault` em um evento de rolagem não impede a rolagem de acontecer. Na verdade o manipulador de eventos é chamado apenas após da rolagem ocorrer.

Evento de foco

Quando um elemento entra em foco o navegador dispara um evento de `"focus"` nele. Quando se perde o foco um eventos de `"blur"` é disparado.

Ao contrário dos eventos discutidos anteriormente, esses dois eventos não se propagam. Um manipulador em um elemento pai não é notificado quando um filho ganha ou perde o foco do elemento.

O exemplo a seguir exibe um texto de ajuda para o campo de texto que possui o foco no momento:

```
<p>Name: <input type="text" data-help="Your full name"></p>
```



```
<p>Age: <input type="text" data-help="Age in years"></p>
<p id="help"></p>
<script>
  var help = document.querySelector("#help");
  var fields = document.querySelectorAll("input");
  for (var i = 0; i < fields.length; i++) {
    fields[i].addEventListener("focus", function(event) {
      var text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    fields[i].addEventListener("blur", function(event) {
      help.textContent = "";
    });
  }
</script>
```

O objeto `window` recebe os eventos de `"focus"` e `"blur"` quando o usuário move-se para outra aba ou janela do navegador a qual o documento esta sendo mostrado.

Evento de load

Quando uma página termina de carregar o evento `"load"` é disparado no `window` e no objeto `body` da página. Isso é muitas vezes usado para programar ações de inicialização que exigem que todo o documento tenha sido construído.

Lembre-se que o conteúdo de tags `<script>` é executado imediatamente quando o tag é encontrada. As vezes a tag `<script>` é processada antes do carregamento total da página e ela necessita de algum conteúdo que ainda não foi carregado.

Elementos como imagens e tags de script carregam arquivo externo e tem um evento de `"load"` para indica que os arquivos que eles fazem referência foram carregados. Eles são como os eventos de `focus` e não se propagam.

Quando uma página é fechada ou navegação é colocado em segundo plano um evento de `"beforeunload"` é acionado. O uso principal deste evento é para evitar que o usuário perca o trabalho acidentalmente por fechar um documento. Prevenir que a página seja fechada não é feito com o método `preventDefault`. Ele é feito através do envio de uma `string` a partir do manipulador. A sequência será usado em uma caixa de diálogo que pergunta ao usuário se ele quer permanecer na página ou deixá-la. Este mecanismo garante que um usuário seja capaz de deixar a página, mesmo se estiver executado um script malicioso que prefere mantê-los para sempre, a fim de forçá-los a olhar para alguns anúncios que leva alguns segundos.

Cronograma do Script de execução

Há várias coisas que podem causar a inicialização da execução de um script. A leitura de um tag `<script>` é um exemplo disto. Um disparo de eventos é outra. No capítulo 13 discutimos a função `requestAnimationFrame` que agenda uma função a ser chamada antes de redesenhar a próxima página. Essa é mais uma forma em que um script pode começar a correr.

É importante entender que disparo de eventos podem ocorrer a qualquer momento, quando há dois scripts em um único documento eles nunca iram correr no mesmo tempo. Se um script já está em execução os manipuladores de eventos e o pedaço de código programado em outras formas teram de esperar por sua vez. Esta é a razão pela qual um documento irá congelar quando um script é executado por um longo tempo. O navegador não pode reagir aos cliques e outros eventos dentro do documento porque ele não pode executar manipuladores de eventos até que o script atual termine sua execução.

Alguns ambientes de programação permitem que múltiplas `threads` de execução se propaguem ao mesmo tempo.

Fazer várias coisas ao mesmo tempo torna um programa mais rápido. Mas quando você tem várias ações tocando nas mesmas partes do sistema, ao mesmo tempo torna-se de uma amplitude muito difícil.

O fato de que os programas de JavaScript fazem apenas uma coisa de cada vez torna a nossa vida mais fácil. Para os casos em que você precisar realmente fazer várias coisas ao mesmo tempo sem o congelamento da página, os navegadores fornecem algo chamado de `web workers`. Um `web workers` é um ambiente isolado do JavaScript que funciona ao lado do principal programa de um documento e pode se comunicar com ele apenas por envio e recebimento de mensagens.

Suponha que temos o seguinte código em um arquivo chamado `code/squareworker.js`:

```
addEventListener("message", function(event) {
  postMessage(event.data * event.data);
});
```

Imagine que esta multiplicação de números seja pesado e com uma computação de longa duração e queremos performance então colocamos em uma `thread` em segundo plano. Este código gera um `worker` que envia algumas mensagens e produz respostas.

```
var squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", function(event) {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

A função `postMessage` envia uma mensagem o que causa um evento de `"message"` disparado ao receptor. O roteiro que criou o `worker` envia e recebe mensagens através do objeto `Worker`, ao passo que as conversações de `worker` para o script que o criou é enviado e ouvido diretamente sobre o seu âmbito global não compartilhada-se do mesmo roteiro original.

Definindo temporizadores

A função `requestAnimationFrame` é similar à `setTimeout`. Ele agenda outra função a ser chamado mais tarde. Mas em vez de chamar a função na próximo redesenho ele espera por uma determinada quantidade de milissegundos. Esta página muda de azul para amarelo depois de dois segundos:

```
<script>
  document.body.style.background = "blue";
  setTimeout(function() {
    document.body.style.background = "yellow";
  }, 2000);
</script>
</script>
```

Às vezes você precisa cancelar uma função que você programou. Isto é feito através do armazenamento do valor devolvido por `setTimeout` e logo em seguida chamando `clearTimeout`.

```
var bombTimer = setTimeout(function() {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

A função `cancelAnimationFrame` funciona da mesma forma que `clearTimeout` chamando um valor retornado pelo `requestAnimationFrame` que irá cancelar esse `frame` (supondo que ele já não tenha sido chamado).

Um conjunto de funções semelhante são `setInterval` e `clearInterval` são usados para definir `timers` que devem repetir a cada X milissegundos.

```
var ticks = 0;
var clock = setInterval(function() {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Debouncing

Alguns tipos de eventos têm o potencial para disparar rapidamente muitas vezes em uma linha(os eventos `"mousemove"` e `"scroll"` por exemplo). Ao manusear tais eventos, você deve ter cuidado para não fazer nada muito demorado ou seu manipulador vai ocupar tanto tempo que a interação com o documento passa a ficar lento e instável.

Se você precisa fazer algo não trivial em tal manipulador você pode usar `setTimeout` para se certificar de que você não esteja fazendo isso com muita frequência. Isto é geralmente chamado de `debouncing` de evento. Há várias abordagens ligeiramente diferentes para isso.

No primeiro exemplo, queremos fazer algo quando o usuário digitar alguma coisa mas não quero imediatamente, para todos os eventos de tecla. Quando ele esta digitando rapidamente nós só queremos esperar até que uma pausa é feita. Em vez de realizar uma ação imediatamente no manipulador de eventos vamos definir um tempo limite em seu lugar. Nós também limpamos o tempo limite anterior(se houver), de modo que, quando ocorrer os eventos juntos(mais perto do que o nosso tempo de espera) o tempo de espera do evento anterior será cancelado.

```
<textarea>Type something here...</textarea>
<script>
  var textarea = document.querySelector("textarea");
  var timeout;
  textarea.addEventListener("keydown", function() {
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      console.log("You stopped typing.");
    }, 500);
  });
</script>
```

Dando um valor indefinido para `clearTimeout` ou chamando-o em um tempo limite que já tenha demitido, ele não terá efeito. Assim não temos que ter cuidado sobre quando chamá-lo simplesmente fazemos para todos os eventos.

Podemos usar um padrão ligeiramente diferente se quisermos de respostas no espaço de modo que eles fiquem separados por pelo menos um determinado período de tempo, mas quero remove-los durante uma série de eventos e não depois. Por exemplo, podemos querer responder a eventos de `"mousemove"`, mostrando as coordenadas atuais do mouse, mas apenas a cada 250 milissegundos.

```
<script>
  function displayCoords(event) {
    document.body.textContent =
      "Mouse at " + event.pageX + ", " + event.pageY;
  }
```

```
var scheduled = false, lastEvent;
addEventListener("mousemove", function(event) {
    lastEvent = event;
    if (!scheduled) {
        scheduled = true;
        setTimeout(function() {
            scheduled = false;
            displayCoords(lastEvent);
        }, 250);
    }
});
</script>
```

Sumário

Os manipuladores de eventos tornam possível detectar e reagir sobre eventos que não têm controle direto. O método `addEventListener` é usado para registrar esse manipulador.

Cada evento tem um tipo(`"keydown"` , `"focus"` , e assim por diante) que o identifica. A maioria dos eventos são chamados em um elementos DOM específicos e então propagam aos ancestrais desse elemento, permitindo que manipuladores associados a esses elementos possam lidar com eles.

Quando um manipulador de eventos é chamado, é passado um objeto de evento com informações adicionais sobre o mesmo. Este objeto também tem métodos que nos permitem parar a propagação(`stopPropagation`) ou evitar a manipulação padrão do navegador do evento(`preventDefault`).

Pressionando uma tecla, eventos de `"keydown"` , `"keypress"` e `"keyup"` são disparados. Pressionar um botão do mouse, eventos de `"mousedown"` , `"mouseup"` e `"click"` são disparados. Movendo o mouse, eventos de `"mousemove"` , `"mouseenter"` e `"mouseout"` são disparados.

A rolagem pode ser detectado com o evento de `"scroll"` , e quando a mudança de foco este eventos podem ser detectadas com o `"focus"` e `"blur"` . Quando o documento termina de carregar, um evento de `"load"` é disparado no `window` .

Apenas um pedaço de programa JavaScript pode ser executado por vez. Manipuladores de eventos e outros scripts programados tem que esperar até outros scripts terminarem antes de chegar a sua vez.

Exercícios

Censores de teclado

Entre 1928 e 2013, uma lei Turca proibiu o uso das letras Q, W, X em documentos oficiais. Isso foi parte de uma iniciativa mais ampla para reprimir culturas Kurdish, essas casos ocorreram na língua utilizada por pessoas Kurdish mas não para os turcos de Istambul.

Neste exercício você esta fazendo uma coisas ridículas com a tecnologia, eu estou pedindo para você programar um campo de texto(uma tag `<input type="text">`) onde essas letras não pode ser digitada.

(Não se preocupe em copiar e colar algum exemplo.)

```
<input type="text">
<script>
    var field = document.querySelector("input");
    // Your code here.
</script>
```

Dica

A solução para este exercício que envolve o impedindo do comportamento padrão dos eventos de teclas. Você pode lidar com qualquer evento `"keypress"` ou `"keydown"`. Se um dos dois tiver `preventDefault` chamado sobre ele, a tecla não aparece.

Identificar a letra digitada requer olhar o código de acesso ou propriedade `charCode` e comparar isso com os códigos para as letras que você deseja filtrar. Em `"keydown"` você não precisa se preocupar com letras maiúsculas e minúsculas, uma vez que precisa somente identificar somente a tecla pressionada. Se você decidir lidar com `"keypress"` que identifica o caráter real digitado você tem que ter certeza que você testou para ambos os casos. Uma maneira de fazer isso seria esta:

```
/[qwx]/i.test(String.fromCharCode(event.charCode))
```

Solução

Trilha do mouse

Nos primeiros dias de JavaScript que era a hora de home pages berrantes com lotes de imagens animadas, as pessoas viram algumas maneiras verdadeiramente inspiradoras para usar a linguagem.

Uma delas foi a "trilha do mouse" a série de imagens que viriam a seguir o ponteiro do mouse quando você muda o cursor através da página.

Neste exercício, eu quero que você implemente um rastro de mouse. Use posicionadores absolutamente ao elemento `<div>` com um tamanho fixo e com uma cor de fundo(consulte o código na seção `"mouseclick"` para um exemplo). Crie um grupo de tais elementos e quando o mouse se mover exibir a esteira do ponteiro do mouse de alguma forma.

Existem várias abordagens possíveis aqui. Você pode fazer a sua solução simples ou complexa; como você quiser. Uma solução simples para começar é manter um número fixo de elementos da fuga e percorrê-las, movendo-se o próximo a posição atual do rato cada vez que um evento `"mousemove"` ocorrer.

```
<style>
  .trail { /* className for the trail elements */
    position: absolute;
    height: 6px; width: 6px;
    border-radius: 3px;
    background: teal;
  }
  body {
    height: 300px;
  }
</style>

<script>
  // Your code here.
</script>
```

Dica

Para criar os elementos o melhor é fazer um loop e anexá-las ao documento para poder exibir. Para ser capaz de poder acessar mais tarde para alterar a sua posição e armazenar os elementos da fuga em uma matriz.

Ciclismo através deles pode ser feito mantendo uma variável de contador e adicionando 1 a ela toda vez que o evento de `"mousemove"` é disparado. O operador `resto(% 10)` pode então ser usado para obter um índice de matriz válida para escolher o elemento que você deseja posicionar durante um determinado evento.

Outro efeito interessante pode ser alcançado por um sistema de modelagem física simples. Use o evento `"mousemove"` apenas para atualizar um par de variáveis que rastreiam a posição do mouse. Em seguida, use `requestAnimationFrame` para simular os elementos de rastros sendo atraídos para a posição do ponteiro do mouse. Em cada passo de animação atualizar a sua posição com base na sua posição relativa para o ponteiro do mouse(opcionalmente programe uma velocidade que é armazenado para cada elemento). Descobrir uma boa maneira de fazer isso é com você.

Solução

Tab

A interface com abas é um padrão comum de design. Ele permite que você selecione um painel de interface escolhendo entre uma série de abas que se destaca acima de um outro elemento.

Neste exercício você vai implementar uma interface simples com abas. Escreva uma função `asTabs` que leva um nó do DOM e cria uma interface com abas mostrando os elementos filho desse nó. Você deverá inserir uma lista de elementos `<button>` na parte superior do nó e para cada elemento filho devera conter o texto recuperado do atributo `tabname` de cada botão. Todos exceto um dos filhos originais devem ser escondidos(dando um estilo de `display: none`) atualmente os nó disponíveis podem ser selecionados com um click nos botões.

Quando funcionar você devera mudar o estilo do botão ativo.

```
<div id="wrapper">
  <div data-tabname="one">Tab one</div>
  <div data-tabname="two">Tab two</div>
  <div data-tabname="three">Tab three</div>
</div>
<script>
  function asTabs(node) {
    // Your code here.
  }
  asTabs(document.querySelector("#wrapper"));
</script>
```

Dica

Uma armadilha que você provavelmente vai encontrar é que não podera usar diretamente propriedade `childNodes` do nó como uma coleção de nós na tabulação. Por um lado quando você adiciona os botões eles também se tornam nós filhos e acabam neste objeto porque é em tempo de execução. Por outro lado os nós de texto criados para o espaço em branco entre os nós também estão lá e não deve obter os seus próprios guias.

Para contornar isso vamos começar a construir uma matriz real de todos os filhos do `wrapper` que têm um `nodeType` igual a 1.

Ao registrar manipuladores de eventos sobre os botões as funções de manipulador vai precisar saber qual separador do elemento está associada ao botão. Se eles são criados em um circuito normal você pode acessar a variável de índice do ciclo de dentro da função mas não vai dar-lhe o número correto pois essa variável terá posteriormente sido alterada pelo loop.

Uma solução simples é usar o método `forEach` para criar as funções de manipulador de dentro da função passada. O índice de loop que é passado como um segundo argumento para essa função, será uma variável local normal e que não serão substituídos por novas iterações.

Solução

Plataforma de jogo

Toda realidade é um jogo.

Iain Banks, *The Player of Games*

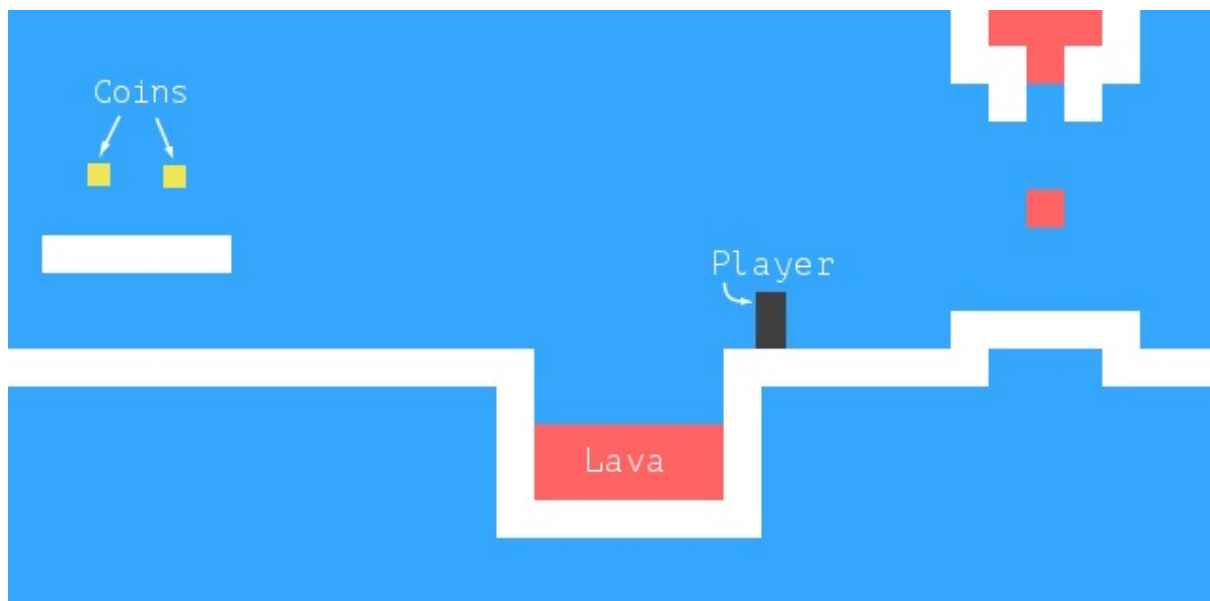
Meu fascínio inicial com computadores foi como o de muitas crianças, originado por jogos de computadores. Fui convocado para um pequeno mundo simulado por computadores onde eu poderia manipular as histórias (mais ou menos) que iam se desenrolando, mais, eu suponho, por causa da maneira que eu poderia projetar a minha imaginação neles do que pelas possibilidades que eles realmente ofereciam.

Eu não desejo uma carreira na programação de jogos a ninguém. Assim como a indústria da música, a discrepância entre os muitos jovens ansiosos que querem trabalhar nela e a demanda real para essas pessoas cria um ambiente não muito saudável. Mas escrever jogos para se divertir é muito legal.

Este capítulo vai falar sobre a implementação de um jogo de plataforma simples. Jogos de Plataforma (ou jogos de "saltar e correr") são os jogos que esperam o jogador para mover uma figura através de um mundo que muitas vezes é bidimensional e visto de lado, onde pode ter a possibilidade de muitos saltos para se mover sobre as coisas.

O jogo

Nosso jogo será mais ou menos baseado em **Dark blue** por Thomas Palef. Eu escolhi este jogo porque é divertido, minimalista e pode ser construído sem muito código. Observe:



A caixa escura representa o jogador, cuja a tarefa é coletar as caixas amarelas (moedas), evitando o material vermelho (lava). Um nível (*level*) é concluído quando todas as moedas forem recolhidas.

O jogador pode movimentar o personagem com as setas do teclado para a esquerda, para a direita, ou pular com a seta para cima. *Jumping* é uma especialidade deste personagem do jogo. Ela pode atingir várias vezes sua própria altura e é capaz de mudar de direção em pleno ar. Isto pode não ser inteiramente realista mas ajuda a dar ao jogador a sensação de estar no controle do avatar na tela.

O jogo consiste em um fundo fixo como uma grade e com os elementos que se deslocam, sobrepostos ao fundo. Cada campo na grade pode estar vazio, sólido ou ser uma lava. Os elementos móveis são os jogadores, moedas e alguns pedaços de lava. Ao contrário da simulação de vida artificial no Capítulo 7, as posições destes elementos não

estão limitadas a grade - suas coordenadas podem ser fracionadas, permitindo movimentos suaves.

A tecnologia

Nós vamos usar o DOM e o navegador para exibir o jogo e iremos ler a entrada do usuário por manipulação de eventos de teclas.

O código de triagem e manipulação com o teclado é apenas uma pequena parte do trabalho que precisamos fazer para construir este jogo. A parte do desenho é simples, uma vez que tudo parece colorido: criamos elementos no DOM e usamos `styling` para dar-lhes uma cor de fundo, tamanho e posição.

Podemos representar o fundo como uma tabela, uma vez que é uma grade imutável de quadrados. Os elementos de movimento livre podem ser cobertos em cima disso, utilizando-se posicionamentos absolutos.

Em jogos e outros programas, que têm que animar gráficos e responder à entrada do usuário sem demora notável, a eficiência é importante. O DOM não foi originalmente projetado para gráficos de alto desempenho, mas é o melhor que podemos esperar. Você viu algumas animações no capítulo 13. Em uma máquina moderna um jogo simples como este tem um bom desempenho mesmo se não estivermos pensando em otimização.

No próximo capítulo vamos explorar uma outra tecnologia do navegador que é a tag `<canvas>`, onde é proporcionado uma forma mais tradicional para desenhar gráficos, trabalhando em termos de formas e pixels em vez de elementos no DOM.

Níveis

No Capítulo 7 usamos matrizes de sequências para descrever uma grade bidimensional. Nós podemos fazer o mesmo aqui. Ele nos permitirá projetar `Level` sem antes construir um editor de `Level`.

Um `Level` simples ficaria assim:

```
var simpleLevelPlan = [
  "
  "
  " x          = x
  " x      o o  x
  " x @      xxxxx x
  " xxxxx      x
  "      x!!!!!!!!x
  "      xxxxxxxxxxxx
  "
];
```

Tanto a grade (*grid*) fixa e os elementos móveis são inclusos no plano. Os caracteres `x` representam paredes, os caracteres de espaço são para o `espaço vazio` e os `!` representam algo fixo, seções de lava que não se mechem.

O `@` define o local onde o jogador começa. Todo `o` é uma moeda e o sinal de igual `=` representa um bloco de lava que se move para trás e para a frente na horizontal. Note que a grade para essas regras será definida para conter o espaço vazio, e outra estrutura de dados é usada para rastrear a posição de tais elementos em movimento.

Vamos apoiar dois outros tipos de lava em movimento: O personagem *pipe* (`|`) para blocos que se deslocam verticalmente e `v` por gotejamento de lava verticalmente. Lava que não salta para trás e nem para a frente só se move para baixo pulando de volta à sua posição inicial quando atinge o chão.

Um jogo inteiro é composto por vários `Levels` que o jogador deve completar. Um `Level` é concluído quando todas as moedas forem recolhidas. Se o jogador toca a lava o `Level` atual é restaurado à sua posição inicial e o jogador pode tentar novamente.

A leitura de um level

O construtor a seguir cria um objeto de `Level`. Seu argumento deve ser uma matriz de sequências que define o `Level`.

```
function Level(plan) {
  this.width = plan[0].length;
  this.height = plan.length;
  this.grid = [];
  this.actors = [];

  for (var y = 0; y < this.height; y++) {
    var line = plan[y], gridLine = [];
    for (var x = 0; x < this.width; x++) {
      var ch = line[x], fieldType = null;
      var Actor = actorChars[ch];
      if (Actor)
        this.actors.push(new Actor(new Vector(x, y), ch));
      else if (ch == "x")
        fieldType = "wall";
      else if (ch == "!")
        fieldType = "lava";
      gridLine.push(fieldType);
    }
    this.grid.push(gridLine);
  }

  this.player = this.actors.filter(function(actor) {
    return actor.type == "player";
  })[0];
  this.status = this.finishDelay = null;
}
```

Para deixar o código pequeno, não verificamos entradas erradas. Ele assume que você sempre entrega um plano de *level* adequado, completo, com a posição de início do jogador e com outros itens essenciais.

Um *level* armazena a sua largura e altura juntamente com duas matrizes, uma para a grade e um para os agentes que são os elementos dinâmicos. A grade é representada como uma matriz de matrizes onde cada uma das séries internas representam uma linha horizontal, e cada quadrado contém algo ou é nulo; para as casas vazias, ou uma string, indicaremos o tipo do quadrado ("muro" ou "lava").

A matriz contém objetos que rastreiam a posição atual e estado dos elementos dinâmicos no *level*. Cada um deles deverá ter uma propriedade para indicar sua posição (as coordenadas do seu canto superior esquerdo), uma propriedade `size` dando o seu tamanho, e uma propriedade `type` que mantém uma cadeia que identifica o elemento ("lava", "dinheiro" ou "jogador").

Depois de construir a `grid` (grade), usaremos o método de filtro para encontrar o objeto jogador que nós armazenamos em uma propriedade do `level`. A propriedade `status` controla se o jogador ganhou ou perdeu. Quando isto acontece, `finishDelay` é usado para manter o `Level` ativo durante um curto período de tempo, de modo que uma animação simples pode ser mostrada (repor imediatamente ou avançar o `Level` ficaria mais fácil). Este método pode ser usado para descobrir se um `Level` foi concluído.

```
Level.prototype.isFinished = function() {
  return this.status != null && this.finishDelay < 0;
};
```

Atores

Para armazenar a posição e o tamanho de um ator vamos voltar para o nosso tipo `Vector` que agrupa uma coordenada `x` e `y` para coordenar um objeto.

```
function Vector(x, y) {
  this.x = x; this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
Vector.prototype.times = function(factor) {
  return new Vector(this.x * factor, this.y * factor);
};
```

O método de escalas temporais de um vetor nos passa uma determinada quantidade. Isso será útil para quando precisarmos de multiplicar um vetor de velocidade por um intervalo de tempo, para obter a distância percorrida durante esse tempo.

Na seção anterior, o objeto `actorChars` foi usado pelo construtor `Level` para associar personagens com as funções do construtor. O objeto parece com isso:

```
var actorChars = {
  "@": Player,
  "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

Três personagens estão sendo mapeados para o objeto `Lava`. O construtor `Level` passa o caractere fonte do ator como o segundo argumento para o construtor, e o construtor de `Lava` usa isso para ajustar o seu comportamento (saltando horizontalmente, saltando verticalmente ou gotejando).

O tipo do jogador é construído da seguinte forma. A velocidade esta sendo armazenada com velocidade atual, que vai ajudar a simular movimento e gravidade.

```
function Player(pos) {
  this.pos = pos.plus(new Vector(0, -0.5));
  this.size = new Vector(0.8, 1.5);
  this.speed = new Vector(0, 0);
}
Player.prototype.type = "player";
```

Como um jogador tem a altura de um quadrado e meio, a sua posição inicial está sendo definida para ser a metade de um quadrado acima da posição em que o `@` personagem apareceu. Desta forma a sua parte inferior fica alinhada com a parte inferior do quadrado que apareceu.

Ao construir um objeto `Lava` dinamicamente é preciso inicializar o objeto de uma forma diferente, dependendo do personagem que se baseia. `Lava Dinâmica` se move longitudinalmente em sua velocidade dada até atingir um obstáculo. Nesse ponto, se ele tem uma propriedade `repeatPos` ele vai pular de volta à sua posição inicial (gotejamento). Se isso não acontecer, ele irá inverter a sua velocidade e continuar no outro sentido (pular). O construtor só configura as propriedades necessárias. O método que faz o movimento real será escrito mais tarde.

```
function Lava(pos, ch) {
  this.pos = pos;
  this.size = new Vector(1, 1);
  if (ch == "=") {
    this.speed = new Vector(2, 0);
  } else if (ch == "|") {
    this.speed = new Vector(0, 2);
  } else if (ch == "v") {
    this.speed = new Vector(0, 3);
  }
```

```

        this.repeatPos = pos;
    }
}
Lava.prototype.type = "lava";

```

`Coin` são atores simples. A maioria dos blocos simplesmente esperam em seus lugares. Mas para animar o jogo eles recebem um pouco de "oscilação", um ligeiro movimento vertical de vai e volta. Para controlar isto, um objeto `Coin` armazena uma posição da base, bem como uma propriedade que controla a oscilação da fase do movimento no salto. Juntas essas propriedades determinam a posição real da moeda (armazenada na propriedade `pos`).

```

function Coin(pos) {
    this.basePos = this.pos = pos.plus(new Vector(0.2, 0.1));
    this.size = new Vector(0.6, 0.6);
    this.wobble = Math.random() * Math.PI * 2;
}
Coin.prototype.type = "coin";

```

No capítulo 13 vimos que `Math.sin` nos dá a coordenada y de um ponto em um círculo. Isso é para coordenar um vai e vem em forma de onda suave à medida que avançamos o círculo, fazendo a função `seno` se tornar útil para a modelagem de um movimento ondulatório.

Para evitar uma situação em que todas as moedas se movam para cima ou para baixo de forma síncrona, a fase inicial de cada moeda é aleatória. A fase da onda de `Math.Sin`, a largura de uma onda produzida, é de 2π . Multiplicamos o valor retornado pelo `Math.random` por esse número para dar a posição inicial de uma moeda de forma aleatória.

Agora escrevemos todas as peças necessárias para representar o `Level` nesse estado.

```

var simpleLevel = new Level(simpleLevelPlan);
console.log(simpleLevel.width, "by", simpleLevel.height);
// → 22 by 9

```

A tarefa a seguir deve exibir tais *levels* na tela, e assim modelar o tempo do movimento entre deles.

Tarefa de encapsulamento

A maior parte do código neste capítulo não ira se preocupar com o encapsulamento. Isto tem duas razões. Em primeiro lugar o encapsulamento exige esforço extra. Em programas maiores isso requer conceitos adicionais de interfaces a serem introduzidas. Como só há código para você enviar ao leitor que esta jogando com seus olhos vidrados, fiz um esforço para manter o programa pequeno.

Em segundo lugar, os vários elementos neste jogo estão tão ligados que se o comportamento de um deles mudar, é improvável que qualquer um dos outros seriam capazes de ficar na mesma ordem. As interfaces e os elementos acabam codificando uma série de suposições sobre a forma de como o jogo funciona. Isso os torna muito menos eficazes - sempre que você altera uma parte do sistema, você ainda tem que se preocupar com a forma como ela afeta as outras partes, isto porque suas interfaces não cobrem a nova situação.

Alguns pontos de corte que existem em um sistema são as separações através de interfaces rigorosas, mas em outros casos não. Tentar encapsular algo que não é um limite adequado é uma maneira de desperdiçar uma grande quantidade de energia. Quando você está cometendo este erro, normalmente você vai perceber que suas interfaces estarão ficando desajeitadamente grandes e detalhadas, e que elas precisam ser modificadas muitas vezes, durante a evolução do programa.

Há uma coisa que vamos encapsular neste capítulo que é o subsistema de desenho. A razão para isso é que nós vamos mostrar o mesmo jogo de uma maneira diferente no próximo capítulo. Ao colocar o desenho atrás de uma interface, podemos simplesmente carregar o mesmo programa de jogo lá e ligar um novo módulo para exibição.

Desenho

O encapsulamento do código de desenho é feito através da definição de um objeto de exibição de um determinado `Level`. O tipo de exibição que definimos neste capítulo é chamado de `DOMDisplay`, e usaremos elementos simples do DOM para mostrar o `Level`.

Nós estaremos usando uma folha de estilo para definir as cores reais e outras propriedades fixas dos elementos que farão parte do jogo. Também seria possível atribuir diretamente as propriedades de estilo dos elementos quando os criamos, mas queremos produzir programas mais detalhados.

A seguinte função auxiliar fornece uma maneira curta para criar um elemento e dar-lhe uma classe:

```
function elt(name, className) {
  var elt = document.createElement(name);
  if (className) elt.className = className;
  return elt;
}
```

O modo de exibição é criado dando-lhe um elemento pai a que se deve acrescentar e um objeto de `Level`.

```
function DOMDisplay(parent, level) {
  this.wrap = parent.appendChild(elt("div", "game"));
  this.level = level;

  this.wrap.appendChild(this.drawBackground());
  this.actorLayer = null;
  this.drawFrame();
}
```

Levando em consideração o fato de que `appendChild` retorna o elemento ao criar o conteúdo do elemento, então podemos armazená-lo na suas propriedade com apenas uma única instrução.

O fundo do `Level`, que nunca muda, é desenhado apenas uma vez. Os atores são redesenhados toda vez que o `display` for atualizado. A propriedade `actorLayer` será utilizada para controlar o elemento que contém os agentes, de modo que elas possam ser facilmente removidas e substituídas.

Nossas coordenadas e tamanhos são rastreadas em unidades relativas ao tamanho do `grid`, onde o tamanho ou distância de 1 significa uma unidade do `grid`. Ao definir os tamanhos de pixel vamos ter que escalar essas coordenadas, tudo no jogo seria ridiculamente pequeno em um único pixel por metro quadrado. A variável de escala indica o número de pixels que uma única unidade ocupa na tela.

```
var scale = 20;

DOMDisplay.prototype.drawBackground = function() {
  var table = elt("table", "background");
  table.style.width = this.level.width * scale + "px";
  this.level.grid.forEach(function(row) {
    var rowElt = table.appendChild(elt("tr"));
    rowElt.style.height = scale + "px";
    row.forEach(function(type) {
      rowElt.appendChild(elt("td", type));
    });
  });
  return table;
}
```

```
};
```

Como mencionado anteriormente o fundo é desenhado com um elemento `<table>`. Este corresponde à estrutura da propriedade `grid` onde cada linha é transformada em uma linha da tabela (elemento `<tr>`). As cordas na grade são usadas como nomes de classe para a célula da tabela (elemento `<td>`). O seguinte CSS ajuda a olhar o resultado do quadro como o fundo que queremos:

```
.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; }
```

Alguns deles (`table-layout`, `border-spacing` e `padding`) são simplesmente usados para suprimir o comportamento padrão indesejado. Nós não queremos que o layout da tabela dependa do conteúdo de suas células, e nós não queremos espaço entre as células da tabela ou `padding` dentro deles.

A regra de `background` define a cor de fundo. No CSS é permitido as cores serem especificadas tanto com palavras (`write`) tanto com um formato como RGB (`R`, `G`, `B`) onde os componentes são vermelho, verde e azul ou separados em três números de 0 a 255. Assim em `rgb(52, 166, 251)`, o componente vermelho é de 52 o verde é 166 e azul é 251. Como o componente azul é maior, a cor resultante será azulada. Você pode ver que na regra das `lavas` o primeiro número (vermelho) é o maior.

Chamamos a cada ator criado por um elemento no DOM, e para ele definimos sua posição e o tamanho desse elemento com base nas propriedades do ator. Os valores devem ser multiplicados por escala e convertidos para unidades de pixels do jogo.

```
DOMDisplay.prototype.drawActors = function() {
  var wrap = elt("div");
  this.level.actors.forEach(function(actor) {
    var rect = wrap.appendChild(elt("div",
                                   "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 wrap;
};
```

Para dar mais classe ao elemento separamos os nomes de classe com espaços. No código CSS abaixo mostramos a classe `actor` que nos dá os atores com sua posição absoluta. O seu nome é o tipo usado como uma classe extra para dar-lhes uma cor diferente. Não temos que definir a classe de lava novamente porque vamos reutilizar a classe para os quadradinhos de lava que definimos anteriormente.

```
.actor { position: absolute; }
.coin { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64); }
```

Quando se atualiza a exibição, o método `drawFrame` que foi passado primeiro remove os velhos gráficos do ator, se houver algum, e em seguida redesenha-os em suas novas posições. Pode ser tentador tentar reutilizar os elementos DOM para os atores, mas para fazer esse trabalho seria preciso uma grande quantidade de fluxo de informação adicional entre o código de exibição e o código de simulação. Precisaríamos associar os atores com os elementos do DOM e o código de desenho, a remoção dos elementos é feita quando seus atores desaparecem. Uma vez que normalmente não teremos bastante atores no jogo, redesenhar todos eles não custa caro.

```
DOMDisplay.prototype.drawFrame = function() {
  if (this.actorLayer)
    this.wrap.removeChild(this.actorLayer);
  this.actorLayer = this.wrap.appendChild(this.drawActors());
  this.wrap.className = "game " + (this.level.status || "");
  this.scrollPlayerIntoView();
};
```

Ao adicionar o estado atual do `Level` com um nome de classe para o `wrapper` podemos denominar que o ator do jogador está ligeiramente diferente quando o jogo está ganho ou perdido, para isso basta adicionar uma regra no CSS que tem efeito apenas quando o jogador tem um elemento ancestral com uma determinada classe.

```
.lost .player {
  background: rgb(160, 64, 64);
}

.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

Depois de tocar em lava a cor do jogador ficara vermelho escuro escaldante. Quando a última moeda for coletada nós usamos duas caixas brancas com sombras borradas, um para o canto superior esquerdo e outro para o canto superior direito, para criar um efeito de halo branco.

Não podemos assumir que os `Level` sempre se encaixem na janela de exibição. É por isso que a chamada `scrollPlayerIntoView` é necessária e garante que se o `Level` está saindo do visor nós podemos rolar o `viewport` para garantir que o jogador está perto de seu centro. O seguinte CSS dá ao elemento DOM o embrulho do jogo com um tamanho máximo e garante que qualquer coisa que não se destaca da caixa do elemento não é visível. Também damos ao elemento exterior uma posição relativa, de modo que os atores estão posicionados no seu interior em relação ao canto superior esquerdo do `Level`.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

No método `scrollPlayerIntoView` encontramos a posição do jogador e atualizamos a posição de rolagem do elemento conforme seu envolvimento. Vamos mudar a posição de rolagem através da manipulação das propriedades desses elementos com os eventos de `scrollLeft` e `scrollTop` para quando o jogador estiver muito perto do canto.

```
DOMDisplay.prototype.scrollPlayerIntoView = function() {
  var width = this.wrap.clientWidth;
  var height = this.wrap.clientHeight;
  var margin = width / 3;

  // The viewport
  var left = this.wrap.scrollLeft, right = left + width;
  var top = this.wrap.scrollTop, bottom = top + height;

  var player = this.level.player;
  var center = player.pos.plus(player.size.times(0.5))
    .times(scale);

  if (center.x < left + margin)
    this.wrap.scrollLeft = center.x - margin;
  else if (center.x > right - margin)
    this.wrap.scrollLeft = center.x + margin - width;
  if (center.y < top + margin)
```

```
this.wrap.scrollTop = center.y - margin;  
else if (center.y > bottom - margin)  
    this.wrap.scrollTop = center.y + margin - height;  
};
```

A forma de como o centro do jogador é encontrado mostra como os métodos em nosso tipo `Vector` permite calcular os objetos a serem escritos de forma legível. Para encontrar o centro do ator nós adicionamos a sua posição (o canto superior esquerdo) e a metade do seu tamanho. Esse é o centro em coordenadas de `Level` mas precisamos dele em coordenadas de pixel, por isso em seguida vamos multiplicar o vetor resultante de nossa escala de exibição.

Em seguida uma série de verificações são feitas para a posição do jogador dentro e fora do intervalo permitido. Note-se que, as vezes, isto irá definir as coordenadas absolutas de rolagem, abaixo de zero ou fora da área de rolagem do elemento. Isso é bom pois o DOM vai ser obrigado a ter valores verdadeiros. Definir `scrollLeft` para `-10` fará com que ele torne `0`.

Teria sido um pouco mais simples tentar deslocarmos o jogador para o centro da janela. Mas isso cria um efeito bastante chocante. Como você está pulando a visão vai mudar constantemente de cima e para baixo. É mais agradável ter uma área "neutra" no meio da tela onde você pode se mover sem causar qualquer rolagem.

Finalmente, vamos precisar de algo para limpar um `Level` para ser usado quando o jogo se move para o próximo `Level` ou redefine um `Level`.

```
DOMDisplay.prototype.clear = function() {  
    this.wrap.parentNode.removeChild(this.wrap);  
};
```

Estamos agora em condições de apresentar o nosso melhor `Level` atualmente.

```
<link rel="stylesheet" href="css/game.css">  
  
<script>  
    var simpleLevel = new Level(simpleLevelPlan);  
    var display = new DOMDisplay(document.body, simpleLevel);  
</script>
```

A tag `<link>` quando usado com `rel="stylesheet"` torna-se uma maneira de carregar um arquivo CSS em uma página. O arquivo `game.css` contém os estilos necessários para o nosso jogo.

Movimento e colisão

Agora estamos no ponto em que podemos começar a adicionar movimento, que é um aspecto mais interessante do jogo. A abordagem básica tomada pela maioria dos jogos como este consiste em dividir o tempo em pequenos passos, e para cada etapa movemos os atores por uma distância correspondente a sua velocidade (distância percorrida por segundo), multiplicada pelo tamanho do passo em tempo (em segundos).

Isto é fácil. A parte difícil é lidar com as interações entre os elementos. Quando o jogador atinge uma parede ou o chão ele não devem simplesmente se mover através deles. O jogo deve notar quando um determinado movimento faz com que um objeto bata sobre outro objeto e responder adequadamente. Para paredes o movimento deve ser interrompido. As moedas devem serem recolhidas e assim por diante.

Resolver este problema para o caso geral é uma grande tarefa. Você pode encontrar as bibliotecas, geralmente chamadas de motores de física, que simulam a interação entre os objetos físicos em duas ou três dimensões. Nós vamos ter uma abordagem mais modesta neste capítulo, apenas manipularemos as colisões entre objetos retangulares e manusearemos de uma forma bastante simplista.

Antes de mover o jogador ou um bloco de lava, testamos se o movimento iria levá-los para dentro de uma parte não vazia de fundo. Se isso acontecer, nós simplesmente cancelamos o movimento por completo. A resposta a tal colisão depende do tipo de ator - o jogador vai parar, enquanto um bloco de lava se recupera.

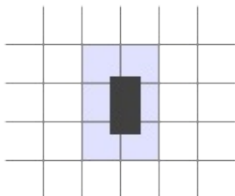
Essa abordagem requer alguns passos para termos uma forma reduzida, uma vez que o objeto que está em movimento para antes dos objetos se tocarem. Se os intervalos de tempo (os movimentos dos passos) são muito grandes, o jogador iria acabar em uma distância perceptível acima do solo. A outra abordagem é indiscutivelmente melhor mas é mais complicada, que seria encontrar o local exato da colisão e se mudar para lá. Tomaremos uma abordagem simples de esconder os seus problemas, garantindo que a animação prossiga em pequenos passos.

Este método nos diz se um retângulo (especificado por uma posição e um tamanho) coincide com qualquer espaço não vazio na `grid` de fundo:

```
Level.prototype.obstacleAt = function(pos, size) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  if (xStart < 0 || xEnd > this.width || yStart < 0)
    return "wall";
  if (yEnd > this.height)
    return "lava";
  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      var fieldType = this.grid[y][x];
      if (fieldType) return fieldType;
    }
  }
};
```

Este método calcula o conjunto de quadrados que o `body` se sobrepõe usando `Math.floor` e `Math.ceil` nas coordenadas do `body`. Lembre-se que as unidades de tamanho dos quadrados são 1 por 1. Arredondando os lados de uma caixa de cima para baixo temos o quadrado da gama de fundo que tem os toques nas caixas.



Se o corpo se sobressai do `Level`, sempre retornaremos `"wall"` para os lados e na parte superior e `"lava"` para o fundo. Isso garante que o jogador morra ao cair para fora do mundo. Quando o corpo está totalmente no interior da `grid`, nosso loop sobre o bloco de quadrados encontra as coordenadas por arredondamento e retorna o conteúdo da primeira `nonempty`.

Colisões entre o jogador e outros atores dinâmicos (moedas, lava em movimento) são tratadas depois que o jogador se mudou. Quando o movimento do jogador coincide com o de outro ator, se for uma moeda é feito o efeito de recolha ou se for lava o efeito de morte é ativado.

Este método analisa o conjunto de atores, procurando um ator que se sobrepõe a um dado como um argumento:

```
Level.prototype.actorAt = function(actor) {
  for (var i = 0; i < this.actors.length; i++) {
    var other = this.actors[i];
    if (other !== actor &&
        actor.pos.x + actor.size.x > other.pos.x &&
        actor.pos.x < other.pos.x + other.size.x &&
        actor.pos.y + actor.size.y > other.pos.y &&
        actor.pos.y < other.pos.y + other.size.y) {
      return other;
    }
  }
};
```

```

        actor.pos.y < other.pos.y + other.size.y)
        return other;
    }
};

```

Atores e ações

O método `animate` do tipo `Level` dá a todos os atores do `level` a chance de se mover. Seu argumento `step` traz o tempo do passo em segundos. O objeto `key` contém informações sobre as teclas que o jogador pressionou.

```

var maxStep = 0.05;

Level.prototype.animate = function(step, keys) {
    if (this.status != null)
        this.finishDelay -= step;

    while (step > 0) {
        var thisStep = Math.min(step, maxStep);
        this.actors.forEach(function(actor) {
            actor.act(thisStep, this, keys);
        }, this);
        step -= thisStep;
    }
};

```

Quando a propriedade `status` do `level` tem um valor não nulo (que é o caso de quando o jogador ganhou ou perdeu), devemos contar para baixo a propriedade `finishDelay` que controla o tempo entre o ponto onde o jogador ganhou ou perdeu e o ponto onde nós paramos de mostrar o `Level`.

O `loop while` corta o passo de tempo onde estamos animando em pedaços pequenos. Ele garante que nenhum passo maior do que `maxStep` é tomado. Por exemplo um passo de 0,12 segundo iria ser cortado em dois passos de 0,05 segundos e um passo de 0,02.

Objetos do ator tem um método `act` que toma como argumentos o tempo do passo, o objeto do `level` que contém as chaves de objeto. Aqui está um exemplo para o tipo de ator (Lava) que ignora as teclas de objeto:

```

Lava.prototype.act = function(step, level) {
    var newPos = this.pos.plus(this.speed.times(step));
    if (!level.obstacleAt(newPos, this.size))
        this.pos = newPos;
    else if (this.repeatPos)
        this.pos = this.repeatPos;
    else
        this.speed = this.speed.times(-1);
};

```

Ele calcula uma nova posição através da adição do produto do tempo do passo e a sua velocidade atual para sua antiga posição. Se nenhum bloco de obstáculos tem uma nova posição ele se move para lá. Se houver um obstáculo, o comportamento depende do tipo da lava: lava e bloco de gotejamento tem uma propriedade `repeatPos` para ele poder saltar para trás quando bater em algo. Saltando, a lava simplesmente inverte sua velocidade (multiplica por -1) a fim de começar a se mover em outra direção.

As moedas usam seu método `act` para se mover. Elas ignoram colisões uma vez que estão simplesmente oscilando em torno de seu próprio quadrado, e colisões com o jogador serão tratadas pelo método `act` do jogador.

```

var wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.act = function(step) {

```

```

    this.wobble += step * wobbleSpeed;
    var wobblePos = Math.sin(this.wobble) * wobbleDist;
    this.pos = this.basePos.plus(new Vector(0, wobblePos));
  };

```

A propriedade `wobble` é atualizada para controlar o tempo e em seguida utilizada como um argumento para `math.sin` para criar uma onda que é usada para calcular sua nova posição.

Isso deixa o próprio jogador. O movimento do jogador é tratado separadamente para cada eixo, porque bater no chão não deve impedir o movimento horizontal, e bater na parede não deve parar a queda ou o movimento de saltar. Este método implementa a parte horizontal:

```

var playerXSpeed = 7;

Player.prototype.moveX = function(step, level, keys) {
  this.speed.x = 0;
  if (keys.left) this.speed.x -= playerXSpeed;
  if (keys.right) this.speed.x += playerXSpeed;

  var motion = new Vector(this.speed.x * step, 0);
  var newPos = this.pos.plus(motion);
  var obstacle = level.obstacleAt(newPos, this.size);
  if (obstacle)
    level.playerTouched(obstacle);
  else
    this.pos = newPos;
};

```

O movimento é calculado com base no estado das teclas de seta esquerda e direita. Quando um movimento faz com que o jogador bata em alguma coisa é o método `playerTouched` que é chamado no `level` que lida com coisas como morrer na lava ou coletar moedas. Caso contrário o objeto atualiza a sua posição.

Movimento vertical funciona de forma semelhante, mas tem que simular salto e gravidade.

```

var gravity = 30;
var jumpSpeed = 17;

Player.prototype.moveY = function(step, level, keys) {
  this.speed.y += step * gravity;
  var motion = new Vector(0, this.speed.y * step);
  var newPos = this.pos.plus(motion);
  var obstacle = level.obstacleAt(newPos, this.size);
  if (obstacle) {
    level.playerTouched(obstacle);
    if (keys.up && this.speed.y > 0)
      this.speed.y = -jumpSpeed;
    else
      this.speed.y = 0;
  } else {
    this.pos = newPos;
  }
};

```

No início do método o jogador é acelerado verticalmente para ter em conta a gravidade. Ao saltar a velocidade da gravidade é praticamente igual a todas as outras constantes neste jogo que foram criadas por tentativa e erro. Eu testei vários valores até encontrar uma combinação agradável.

Em seguida é feito uma verificação para identificar se há obstáculos novamente. Se bater em um obstáculo há dois resultados possíveis. Quando a seta para cima é pressionada e estamos nos movendo para baixo (ou seja, a coisa que bater é abaixo de nós) a velocidade é definida como um valor relativamente grande e negativo. Isso faz com que o jogador salte. Se esse não for o caso, nós simplesmente esbarramos em alguma coisa e a velocidade é zerada.

O método atual parece com isso:

```
Player.prototype.act = function(step, level, keys) {
  this.moveX(step, level, keys);
  this.moveY(step, level, keys);

  var otherActor = level.actorAt(this);
  if (otherActor)
    level.playerTouched(otherActor.type, otherActor);

  // Losing animation
  if (level.status == "lost") {
    this.pos.y += step;
    this.size.y -= step;
  }
};
```

Depois de se mover o método verifica os outros atores que o jogador está colidindo e é chamado o `playerTouched` novamente quando encontra um. Desta vez ele passa o objeto ator como segundo argumento, isto é, porque se o outro ator é uma moeda, `playerTouched` precisa saber qual moeda está sendo coletada.

Finalmente quando o jogador morre (toca lava), montamos uma pequena animação que faz com que ele se "encolha" ou "afunde" reduzindo a altura do objeto jogador.

E aqui é o método que manipula as colisões entre o jogador e outros objetos:

```
Level.prototype.playerTouched = function(type, actor) {
  if (type == "lava" && this.status == null) {
    this.status = "lost";
    this.finishDelay = 1;
  } else if (type == "coin") {
    this.actors = this.actors.filter(function(other) {
      return other != actor;
    });
    if (!this.actors.some(function(actor) {
      return actor.type == "coin";
    })) {
      this.status = "won";
      this.finishDelay = 1;
    }
  }
};
```

Quando a lava é tocada, o status do jogo é definido como `"lost"`. Quando uma moeda é tocada essa moeda é removida do conjunto de atores e se fosse a última, o estado do jogo é definido como `"won"`.

Isso nos dá a opção do `Level` de ser animado. Tudo o que está faltando agora é o código que aciona a animação.

Rastreamento de teclas

Para um jogo como este nós não queremos que as teclas tenham efeito apenas quando pressionadas. Pelo contrário, queremos que o seu efeito (movimentar a figura do jogador) continue movendo o jogador enquanto as teclas estiverem pressionadas.

Precisamos criar um manipulador de teclas que armazena o estado atual da esquerda, direita e cima das teclas de seta. Nós também queremos chamar `preventDefault` para essas teclas para não dar rolagem da página.

A função a seguir, quando dado um objeto com o código da tecla e com o nome de propriedade como valores, vai retornar um objeto que rastreia a posição atual dessas teclas. Ele registra manipuladores de eventos para `"keydown"` e `"keyup"` e, quando o código de tecla no evento está presente no conjunto de códigos que está sendo rastreado, é

executada a atualização do objeto.

```
var arrowCodes = {37: "left", 38: "up", 39: "right"};

function trackKeys(codes) {
  var pressed = Object.create(null);
  function handler(event) {
    if (codes.hasOwnProperty(event.keyCode)) {
      var down = event.type == "keydown";
      pressed[codes[event.keyCode]] = down;
      event.preventDefault();
    }
  }
  addEventListener("keydown", handler);
  addEventListener("keyup", handler);
  return pressed;
}
```

Note como o mesmo manipulador da função é usado para ambos os tipos de eventos. Ele olha para a propriedade `type` do objeto de evento para determinar se o estado da tecla deve ser atualizado para `true` ("keydown") ou `false` ("keyup").

Executar o jogo

A função `requestAnimationFrame` que vimos no capítulo 13 fornece uma boa maneira de animar um jogo. Mas sua interface é bastante primitiva para usá-la, o que nos obriga a ficar controlando sua última chamada para executar a função `requestAnimationFrame` novamente após cada frame.

Vamos definir uma função auxiliar que envolve as partes chatas em uma interface conveniente e nos permitir simplesmente chamar `runAnimation` dando-lhe uma função que espera uma diferença de tempo como um argumento e desenhá-la em um quadro único. Quando a função de armação retorna o valor falso a animação para.

```
function runAnimation(frameFunc) {
  var lastTime = null;
  function frame(time) {
    var stop = false;
    if (lastTime != null) {
      var timeStep = Math.min(time - lastTime, 100) / 1000;
      stop = frameFunc(timeStep) === false;
    }
    lastTime = time;
    if (!stop)
      requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

Temos que definir um passo de quadros máximo de 100 milissegundos(um décimo de segundo). Quando a aba ou janela do navegador com a página estiver oculto as chamadas `requestAnimationFrame` será suspenso até que a aba ou janela é mostrado novamente. Neste caso a diferença entre `lasttime` será todo o tempo em que a página estiver oculta. Avançando o jogo, que em uma única etapa vai parecer fácil mas podemos ter um monte de trabalho (lembre-se o tempo-splitting no método de animação).

A função também converte os passos de tempo para segundos, que são uma quantidade mais fácil de pensar do que milissegundos.

A função de execução do `Level` toma um objeto do `Level` no construtor de uma exposição e opcionalmente uma função. Ele exibe o `Level` (em `document.body`) e permite que o usuário peça por ele. Quando o `Level` está terminado (perda ou ganho), `Level` de execução, limpa o visor, para a animação e caso a função `andThen` for dada, chama essa função com o status do `Level`.

```
var arrows = trackKeys(arrowCodes);

function runLevel(level, Display, andThen) {
  var display = new Display(document.body, level);
  runAnimation(function(step) {
    level.animate(step, arrows);
    display.drawFrame(step);
    if (level.isFinished()) {
      display.clear();
      if (andThen)
        andThen(level.status);
      return false;
    }
  });
}
```

Um jogo é uma sequência de `Level`. Sempre que o jogador morre o `Level` atual é reiniciado. Quando um `Level` é concluído vamos passar para o próximo `Level`. Isso pode ser expresso pela seguinte função o que leva um conjunto de planos de `Level` (arrays de strings) e um construtor de exibição:

```
function runGame(plans, Display) {
  function startLevel(n) {
    runLevel(new Level(plans[n]), Display, function(status) {
      if (status == "lost")
        startLevel(n);
      else if (n < plans.length - 1)
        startLevel(n + 1);
      else
        console.log("You win!");
    });
  }
  startLevel(0);
}
```

Estas funções mostram um estilo peculiar de programação. Ambos `runAnimation` e `Level` de execução são funções de ordem superior, mas não são no estilo que vimos no capítulo 5. O argumento da função é usado para organizar as coisas para acontecer em algum momento no futuro e nenhuma das funções retorna alguma coisa útil. A sua tarefa é de certa forma, agendar ações. Envolvendo estas ações em funções nos dá uma maneira de armazená-las com um valor de modo que eles podem ser chamados no momento certo.

Este estilo de programação geralmente é chamado de programação assíncrona. Manipulação de eventos também é um exemplo deste estilo, vamos ver muito mais do que quando se trabalha com tarefas que podem levar uma quantidade arbitrária de tempo, como solicitações de rede no capítulo 17 e entrada e saída em geral no Capítulo 20.

Há um conjunto de planos de `Level` disponíveis na variável `GAME_LEVELS`. Esta página alimenta `runGame`, começando um jogo real:

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Veja se você pode vencer. Aqui eu espero vários `Level` construídos.

Exercício

Fim do Jogo

É tradicional para jogos de plataforma ter o início do jogador com um número limitado de vidas e subtrair uma vida cada vez que ele morre. Quando o jogador está sem vidas, o jogo será reiniciado desde o início. Ajuste `runGame` para implementar as três vidas ao iniciar.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // The old runGame function. Modify it...
  function runGame(plans, Display) {
    function startLevel(n) {
      runLevel(new Level(plans[n]), Display, function(status) {
        if (status == "lost")
          startLevel(n);
        else if (n < plans.length - 1)
          startLevel(n + 1);
        else
          console.log("You win!");
      });
    }
    startLevel(0);
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

Dica

A solução mais óbvia seria, tornar a vida uma variável que vive em `runGame` e é portanto visível para o encerramento do `startLevel`.

Uma outra abordagem que se encaixa com o espírito do resto da função seria, adicionar um segundo parâmetro para o `startLevel` que dá o número de vidas. Quando todo o estado de um sistema é armazenado nos argumentos para uma função, chamar essa função fornece uma maneira elegante de fazer a transição para um novo estado.

Em qualquer caso, quando o `Level` está perdido deverá agora existir duas transições de estado possíveis. Se esse for a última vida vamos voltar ao `Level` zero com o montante inicial de vidas. Se não vamos repetir o `Level` atual com menos uma vida restante.

Pausar o jogo

Faça o possível para fazer uma pausa(suspenso) e retomar o jogo pressionando a tecla Esc.

Isso pode ser feito alterando a execução função do `Level` para usar outro manipulador de eventos de teclado e interromper ou retomar a animação sempre que a tecla `Esc` é pressionada.

A interface `runAnimation` não pode se responsabilizar por isso à primeira vista, mas basta você reorganizar a maneira que `RUNLEVEL` é chamado.

Quando você tem que trabalhar não há outra coisa que você pode tentar. O caminho que temos vindo a registrar manipuladores de eventos de teclas é um pouco problemático. O objeto `keys` é uma variável global e seus manipuladores de eventos são mantidas ao redor mesmo quando nenhum jogo está sendo executado. Pode-se dizer

que isso pode vazar para fora do nosso sistema. Estender `trackKeys` nos dá uma maneira de fornecer o cancelamento do registro e de seus manipuladores e em seguida mudar a execução do `Level` para registrar seus tratadores quando começa e cancela o registro novamente quando ele for concluído.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // The old runLevel function. Modify this...
  function runLevel(level, Display, andThen) {
    var display = new Display(document.body, level);
    runAnimation(function(step) {
      level.animate(step, arrows);
      display.drawFrame(step);
      if (level.isFinished()) {
        display.clear();
        if (andThen)
          andThen(level.status);
        return false;
      }
    });
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

Dicas

Uma animação pode ser interrompida retornando um valor `false` na função dada ao `runAnimation`. Ele pode ser continuado chamando `runAnimation` novamente.

Para comunicar que a animação deve ser interrompido a função passada para `runAnimation` deve retornar `false`; você pode usar uma variável que tanto o manipulador de eventos e a função tenha acesso.

Quando encontrar uma maneira de cancelar o registro dos manipuladores registrados por `trackKeys` lembre-se que o mesmo valor função exata que foi passado para `addEventListener` deve ser passado para `removeEventListener` para remover com êxito um manipulador. Assim o valor da função manipuladora criada em `trackKeys` deverá estar disponível para o código que cancela os manipuladores.

Você pode adicionar uma propriedade para o objeto retornado por `trackKeys` contendo um ou outro valor da função ou um método que manipula ou remove o registro diretamente.

Desenhando no canvas

Desenhar é enganar.

M.C. Escher, citado por Bruno Ernst em The Magic Mirror of M.C. Escher.

Os Browsers permitem de várias maneiras de mostrarem gráficos. A maneira mais simples é usar um estilos para posição e cor de elementos regulares do DOM. Isso pode ser impraticável, como ficou claro no jogo do capítulo anterior. Podemos adicionar parcialmente uma transparência no fundo das imagens e ainda girar ou inclinar algum usando o estilo de `transform`.

Mas estaríamos usando o DOM para algo que não foi originalmente projetado. Algumas tarefas, tais como desenhar uma linha entre pontos arbitrários são extremamente difíceis de fazer com elementos regulares em HTML.

Existem duas alternativas. O primeiro é baseado em DOM mas utiliza Scalable Vector Graphics(`svg`) ao invés de elementos HTML. Pense em SVG como um dialeto para descrever documentos que se concentra em formas ao invés de texto. Você pode embutir um documento SVG em um documento HTML ou você pode incluí-lo através de uma tag ``.

A segunda alternativa é chamado de `canvas`. A tela é um único elemento DOM que encapsula uma imagem. Ele fornece uma interface de programação para desenhar formas para o espaço ocupado pelo nó. A principal diferença entre um `canvas` e uma imagem de `svg`, é que em `svg` a descrição original das formas é preservada de modo que eles podem ser movidos ou redimensionados em qualquer momento. O `canvas` por outro lado, converte as formas para pixels(pontos coloridos em um rastro), logo eles são desenhados e não guardam informações do que estes pixels representam. A única maneira de mover uma forma em `canvas` é limpar a tela(ou a parte da tela em torno) e redesenhar uma forma em uma nova posição.

SVG

Este livro não vai entrar no assunto `svg` em detalhes, mas vou explicar brevemente como ele funciona. No final do capítulo eu vou voltar para os `trade-offs` que você deve considerar ao decidir qual mecanismo de desenho é adequado para uma determinada aplicação.

Este é um documento HTML com uma imagem SVG simples:

```
<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>
```

O atributo `xmlns` muda um elemento(e seus filhos) a um namespace diferente de XML. Este namespace é identificado por um URL, especificando o dialeto que estamos falando no momento. As `tags` `<circle>` e `<rect>` que não existem em HTML não têm um significado em SVG para desenhar formas usando o estilo e posição especificada para seus atributos.

Essas `tags` criam elementos no DOM assim como as `tags` em HTML. Por exemplo, isso muda a cor para ciano do elemento `<circle>`:

```
var circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

O elemento canvas

Gráfico em *canvas* pode ser desenhado com a tag `<canvas>`. Você pode dar a um elemento a largura e altura em pixel para determinar o seu tamanho.

A nova tela esta vazia, o que significa que é totalmente transparente e portanto simplesmente mostra-se com um espaço vazio no documento.

A tag `<canvas>` destina-se a apoiar os diferentes estilos de desenho. Para ter acesso a uma verdadeira interface de desenho primeiro precisamos criar um contexto que é um objeto, cujos métodos fornecem a interface de desenho. Atualmente existem dois estilos de desenho amplamente suportados: "2d" para gráficos bidimensionais e "WebGL" para gráficos tridimensionais através da interface `OpenGL`.

Este livro não vai discutir *WebGL*. Nós estaremos as duas dimensões. Mas se você estiver interessado em gráficos tridimensionais eu encorajo-vos a olhar para *WebGL*, que fornece uma interface muito direta com o hardware com gráfico moderno e permite que você processe cenas eficientemente complicadas utilizando JavaScript.

Um contexto é criado através do método `getContext` sobre o elemento `<canvas>`.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  var canvas = document.querySelector("canvas");
  var context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

Depois de criar o objeto de contexto, o exemplo desenha um retângulo vermelho de 100 pixels de largura e 50 pixels de altura em relação ao seu canto superior esquerdo nas coordenadas (10,10).

Assim como em HTML e SVG o sistema que a tela usa puts(0,0) no canto superior esquerdo de coordenadas, e o eixo y positivo vai para baixo. Então (10,10) é de 10 pixels abaixo e a direita do canto superior esquerdo.

Preenchimento e traçado

Na interface uma forma pode ser cheia ou seja, sua área é dada uma determinada cor padrão; ou pode ser riscada o que significa que uma linha é desenhada ao longo de sua borda. A mesma terminologia é utilizada por SVG.

O método `fillRect` preenche um retângulo. É preciso ter as coordenadas `x` e `y` do canto superior esquerdo do retângulo, em seguida a sua largura e a sua altura. Um método semelhante `strokeRect` desenha o contorno de um retângulo.

Nenhum dos métodos tem parâmetros. A cor do preenchimento e a espessura do traçado não são determinados por argumento do método(como você espera), mas sim pelas propriedades do contexto do objecto.

As definições de `fillStyle` pode alterar o jeito que as formas são preenchidas. Ele pode ser definido como uma `string` que especifica uma cor de qualquer modo que é compreendido por CSS.

A propriedade `strokeStyle` funciona de forma semelhante, mas determina a cor usada para uma linha. A largura da linha é determinada pela propriedade `lineWidth` que pode conter qualquer número positivo.

```
<canvas></canvas>
<script>
  var 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>
```

Quando nenhuma largura ou altura é especificado como atributo, como no exemplo anterior um elemento de tela adquire uma largura padrão de 300 pixels e altura de 150 pixels.

Paths

Um `path` é uma sequência de linhas. A interface de uma tela 2D tem uma abordagem peculiar de descrever esse `path`. Isso é feito inteiramente através dos efeitos colaterais. Os `paths` não constituem valores que podem ser armazenados ou repassados. Se você deseja fazer algo com um `path`, você faz uma sequência de chamadas de método para descrever sua forma.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (var y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

Este exemplo cria um `path` com um número de segmentos de linha horizontal e faz traços usando o método `stroke`. Cada segmento criado com `lineTo` começa na posição atual do `path`. Esta posição é normalmente o fim do último segmento a não ser que `moveTo` seja chamado. Nesse caso, o próximo segmento começara na posição passada para `moveTo`.

Ao preencher um `path` (usando o método `fill`) cada forma é preenchido separadamente. Um `path` pode conter várias formas, cada movimento com `moveTo` inicia um novo. Mas o `path` tem de ser fechado(ou seja o seu início e fim devem ficar na mesma posição) antes de ser preenchido. Se o `path` não estiver fechado a linha é adicionada a partir de sua extremidade para o começo da forma delimitada pelo `path` como completado e preenchido.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

Este exemplo estabelece um triângulo cheio. Note que apenas dois dos lados do triângulo são explicitamente desenhados. A terceira é a partir do canto inferior direito ate o topo; é implícito e não estará lá quando você traçar o `path`.

Você também pode usar o método `closePath` para fechar explicitamente um `path` através da adição de um segmento da linha atual de volta ao início do `path`. Este segmento é desenhado traçando o `path`.

Curvas

Um `path` também pode conter linhas com curvas. Estes infelizmente é um pouco mais complexo do que desenhar linhas retas. O método `quadraticCurveTo` desenha uma curva ate um ponto considerado. Para determinar a curvatura da linha é dado no método um ponto de controle e um ponto de destino. Imagine o seguinte, ponto de controle é uma atração a linha, o que da a ela sua curvatura. A linha não passa pelo ponto de controle. Ao contrário disso a direção da linha nos seus pontos de início e fim fica alinhado, com a linha puxando para o ponto de controle. O exemplo a seguir ilustra isso:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

Nós desenharemos uma curva quadrática a partir da esquerda para a direita com (60,10) no ponto de controle e depois colocamos dois segmentos da linha passando por esse ponto de controle de volta para o início da linha. O resultado lembra um pouco uma insígnia do Star Trek. Você pode ver o efeito do ponto de controle: as linhas que saem dos cantos inferiores começam na direção do ponto de controle e em seguida se curva em direção a seu alvo.

O método `bezierCurve` desenha um tipo semelhante de uma curva. Em vez de um único ponto de controle este tem dois, um para cada um dos pontos das extremidades da linha. Aqui é um esboço semelhante para ilustrar o comportamento de uma tal curva:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

Os dois pontos de controle especificam a direção em ambas as extremidades da curva. Quanto mais eles estão longe de seu ponto correspondente, maior a curva que vai nesse sentido.

Tais curvas pode ser difícil de trabalhar, nem sempre é evidente encontrar a forma dos pontos de controle que proporcionam a forma que você está procurando. Às vezes você pode calcular, e às vezes você apenas tem que encontrar um valor apropriado por tentativa e erro.

Fragmentos `arcs` de um círculo são mais fáceis de se trabalhar. O método `arcTo` não leva menos de cinco argumentos. Os quatro primeiros agem um pouco como os argumentos para `quadraticCurveTo`. O primeiro par fornece uma espécie de ponto de controle e o segundo par dá o destino a linha. O quinto argumento fornece o raio do arco. O método vai conceitualmente projetar um canto da linha que vai para o ponto de controle e em seguida volta ao ponto de destino para que ele faça parte de um círculo com o raio dado. O método `arcTo` chega então a uma parte arredondada bem como uma linha a partir da posição de partida até o início de uma parte arredondada.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 10);
  // control=(90,10) goal=(90,90) radius=20
  cx.arcTo(90, 10, 90, 90, 20);
  cx.moveTo(10, 10);
  // control=(90,10) goal=(90,90) radius=80
  cx.arcTo(90, 10, 90, 90, 80);
  cx.stroke();
</script>
```

```
</script>
```

O método `arcTo` não vai desenhar a linha a partir da parte final do arredondamento para a posição do objetivo, embora a palavra no seu nome sugere o que ele faz. Você pode acompanhar com uma chamada de `lineTo` com o mesmo objetivo de coordena e acrescentar uma parte da linha.

Para desenhar um círculo você poderia usar quatro chamadas para `arcTo` (cada um que giram 90 graus). Mas o método `arcTo` fornece uma maneira mais simples. É preciso um par de coordenadas para o centro do arco, um raio e em seguida um ângulo de início e fim.

Esses dois últimos parâmetros tornam possível desenhar apenas uma parte do círculo. Os ângulos são medidos em radianos não em graus. Isso significa que um círculo completo tem um ângulo de 2π ou `2 * Math.PI` que é de cerca de `6,28`. O ângulo começa a contar a partir do ponto da direita do centro do círculo e vai a partir do sentido horário. Você pode usar um começo de `0` e um fim maior do que 2π (digamos 7) para desenhar um círculo completo.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to 1/2π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

A imagem resultante contém uma linha no círculo(primeira chamada de `arc`) a esquerda do quarto do círculo(segunda chamada). Como outros métodos estão ligados ao desenho de um `path`, uma linha traçada é ligado ao segmento do arco anterior por padrão. Se você quiser evitar isso teria que chamar `moveTo` ou iniciar um novo `path`.

Desenho de um gráfico de pizza

Imagine que você acabou de conseguir um emprego na EconomiCorp Inc. e sua primeira missão é desenhar um gráfico de pizza dos resultados da pesquisa de satisfação do cliente.

A variável dos resultados contém uma matriz de objetos que representam as respostas da pesquisa.

```
var 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"}
];
```

Para desenhar um gráfico de pizza, traçamos um número de fatias, cada um é composto por um arco e um par de linhas para o centro desse arco. Podemos calcular o ângulo ocupado por cada arco dividindo um círculo completo(2π) pelo número total de respostas, em seguida multiplicamos esse número(o ângulo por resposta) pelo número de pessoas que fizeram determinadas escolhas.

```
<canvas width="200" height="200"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var total = results.reduce(function(sum, choice) {
    return sum + choice.count;
  }, 0);
  // Start at the top
```

```

var currentAngle = -0.5 * Math.PI;
results.forEach(function(result) {
    var sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
});
</script>

```

Mas um gráfico que não nos diz o que significa não é útil. Nós precisamos de uma maneira para desenhar o texto na tela.

Texto

Um contexto de desenho em canvas 2D fornece os métodos `fillText` e `strokeText`. Este último pode ser útil para delinear as letras mas geralmente `fillText` é o que você precisa. Ele vai encher o texto com a cor atual de `fillColor`.

```

<canvas></canvas>
<script>
    var cx = document.querySelector("canvas").getContext("2d");
    cx.font = "28px Georgia";
    cx.fillStyle = "fuchsia";
    cx.fillText("I can draw text, too!", 10, 50);
</script>

```

Você pode especificar o tamanho, estilo e tipo da letra do texto com a propriedade `font`. Este exemplo apenas dá um tamanho de fonte e nome da família. Você pode adicionar o itálico ou negrito para o início de uma sequência de caracteres.

Os dois últimos argumentos para `fillText` (e `strokeText`) fornecem a posição em que a fonte é desenhado. Por padrão a posição do início da linha indica a base alfabética do texto, que é a linha que as letras ficam não tendo partes penduradas; em letras como `j` ou `p` você pode mudar a posição horizontal definindo a propriedade `textAlign` para `end` ou `center` ou posicionamento vertical definindo `textBaseline` para `top`, `middle` ou `bottom`.

Vamos voltar ao nosso gráfico de pizza para corrigir o problema de rotular as fatias nos exercícios no final do capítulo.

Imagens

Na computação gráfica uma distinção é feita frequentemente entre gráficos vetoriais e bitmap. O primeiro é como iremos fazer neste capítulo; a especificação de uma imagem dando uma descrição lógica de formas. Os gráficos de bitmap não especificam formas reais, mas sim trabalham com dados de pixel (rastros de pontos coloridos).

O método `drawImage` nos permite desenhar dados de pixel em `canvas`. Estes dados de pixel pode ter origem a partir de uma tag `` ou `<canvas>`, e nem todos são visíveis no documento atual. O exemplo a seguir cria um elemento `` e carrega um arquivo de imagem nele. Mas não é iniciado imediatamente; a elaboração desta imagem não ocorreu porque o browser ainda não buscou por isso. Para lidar com tal situação registramos um manipulador de eventos(`"load"`) para fazer o desenho depois que a imagem for carregada.

```

<canvas></canvas>

```

```
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", function() {
    for (var x = 10; x < 200; x += 30)
      cx.drawImage(img, x, 10);
  });
</script>
```

Por padrão, `drawImage` vai desenhar a imagem em seu tamanho original. Você também pode dar-lhe dois argumentos adicionais para ditar uma largura e altura diferentes.

`drawImage` recebe nove argumentos, ele pode ser utilizado para desenhar apenas um fragmento de uma imagem. Do segundo ao quinto argumento indicam o retângulo(x, y, largura e altura) na imagem de origem que deve ser copiado, do sexto ao nono argumentos indica o retângulo(na tela) em que deve ser copiado.

Isso pode ser usado para embalar várias sprites(elementos de imagem) em um único arquivo de imagem, em seguida desenhar apenas a parte que você precisa. Por exemplo, nós temos esta imagem contendo uma personagem do jogo em várias poses:



Ao alternar a pose que traçamos, podemos mostrar uma animação que simula o movimento de andar do personagem.

Para animar a imagem em uma tela o método `clearRect` é útil. Assemelha-se a `fillRect` mas ao invés de colorir o retângulo, torna-se transparente removendo os pixels previamente desenhados.

Sabemos que a cada sprite são sub-imagens de 24 pixels de largura por 30 pixels de altura. O código a seguir carrega as imagens, e em seguida define um intervalo(temporizador de repetição) para desenhar os quadros seguintes:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var img = document.createElement("img");
  img.src = "img/player.png";
  var spriteW = 24, spriteH = 30;
  img.addEventListener("load", function() {
    var cycle = 0;
    setInterval(function() {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
        // source rectangle
        cycle * spriteW, 0, spriteW, spriteH,
        // destination rectangle
        0, 0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

A variável `cycle` mapeia nossa posição na animação. A cada quadro ele é incrementado e em seguida cortado de volta para o intervalo de 0 a 7 usando o operador restante. Esta variável é usada para calcular a coordenada `x` que o sprite tem para a pose atual da imagem.

Transformações

Mas e se queremos que o nosso personagem ande para a esquerda em vez de para a direita? Poderíamos acrescentar um outro conjunto de sprites, é claro. Mas também podemos instruir a tela para desenhar a imagem de outra maneira.

Chamar o método `scale` fará com que qualquer coisa desenhada depois possa ser escalado. Este método tem dois parâmetros, um para definir uma escala horizontal e um para definir uma escala vertical.

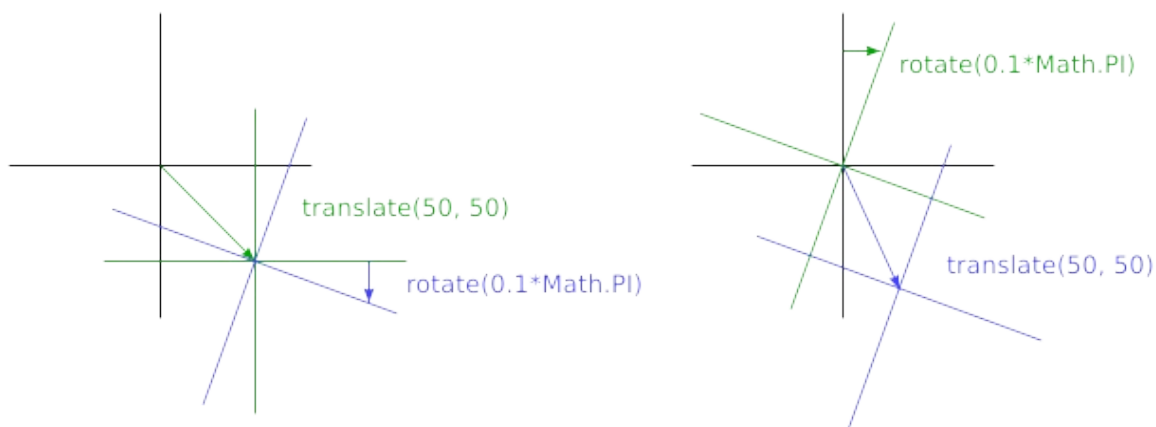
```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

`scaling` fará tudo sobre a imagem desenhada incluindo: a largura da linha a ser esticado ou espremido, conforme especificado. Dimensionamento por um valor negativo vai inverter a imagem ao redor. A inversão acontece em torno do ponto(0,0); o que significa que tudo irá virar a direção do sistema de coordenadas. Quando uma escala horizontal de -1 é aplicada, a forma desenhada em x na posição 100 vai acabar na posição -100.

Então para transformar uma imagem em torno não podemos simplesmente adicionar `cx.scale (-1, 1)` antes da chamada `drawImage` pois irá mover a nossa imagem fora da tela onde não será mais possível vê-la. Você pode ajustar as coordenadas dadas a `drawImage` para compensar esse desenho da imagem em x na posição -50 em vez de 0. Outra solução que não exige que o código faça o desenho para saber sobre a mudança de escala, é ajustar o eixo em torno do qual a escala acontece.

Há vários outros métodos além de `scale` que influenciam no sistema de coordenadas para o `canvas`. Você pode girar formas posteriormente desenhados com o método de `rotation` e movê-los com o método de `translate`. É interessante e confuso saber que estas transformações são realizados no estilo de pilha, o que significa que cada uma acontece em relação às transformações anteriores.

Então se nós fizermos um `translate` de 10 pixels na horizontal por duas vezes, tudo será desenhada 20 pixels para a direita. Se primeiro mover o centro do sistema de coordenadas de (50,50) e em seguida girar 20 graus(0.1π em radianos) a rotação vai acontecer em torno do ponto (50,50).



Mas se nós primeiro girarmos 20 graus e em seguida aplicarmos um `translate` de (50,50), o `translate` irá acontecer na rotação do sistema de coordenadas e assim produzir uma orientação diferente. A ordem em que as transformações são aplicadas será assunto nos próximos tópicos.

Para inverter uma imagem em torno da linha vertical em uma determinada posição x podemos fazer o seguinte:

```
function flipHorizontally(context, around) {
```

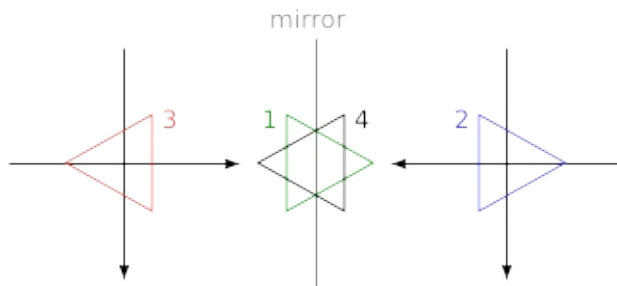


```

context.translate(around, 0);
context.scale(-1, 1);
context.translate(-around, 0);
}

```

Nós deslocamos o eixo-y para onde queremos que o nosso espelho fique e aplicamos, finalmente deslocamos o eixo-y de volta ao seu lugar adequado no universo espelhado. O quadro a seguir explica por que isso funciona:



Isto mostra o sistemas de coordenadas antes e após o espelhamento do outro lado da linha central. Se desenharmos um triângulo em uma posição positiva x, estaria por padrão no lugar onde triângulo 1 esta. Uma chamada para `flipHorizontally` faz primeiro um `translate` para a direita, o que nos leva ao triângulo 2. Em seguida `scale` é lançado e o triângulo volta para a posição 3. Este não é o lugar onde ele deveria estar se fosse espelhada na linha dada. O segundo `translate` para correções da chamadas esta cancelando o `translate` inicial e faz triângulo 4 aparecer exatamente onde deveria.

Agora podemos desenhar um personagem espelhado na posição (100,0) rodando o mundo em torno do centro vertical do personagem.

```

<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var img = document.createElement("img");
  img.src = "img/player.png";
  var spriteW = 24, spriteH = 30;
  img.addEventListener("load", function() {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                  100, 0, spriteW, spriteH);
  });
</script>

```

Armazenar e limpando transformações

Tudo sobre transformações fica por aqui. Qualquer outra coisa que desenhar depois desse personagem espelhado também ficara espelhado. Isso pode ser um problema.

É possível salvar a transformação atual, fazer algum desenho e transformar e em seguida restaurar a velho transformação. Isso geralmente é a coisa certa a fazer para uma função que necessita se transformar temporariamente o sistema de coordenadas. Em primeiro lugar vamos salvar qualquer que seja a transformação do código que chamou a função que estava utilizando. Em seguida a função faz a sua parte(no topo da transformação existente) possivelmente adicionando mais transformações. E finalmente revertermos a transformação que nós fizemos.

Os salvar e o restaurar nos métodos em contexto `canvas` 2D realizam um tipo de gerenciamento na transformação. Eles conceitualmente mantêm uma pilha de estados de transformação. Quando você chama o salvar o estado atual é colocado na pilha, e quando você chama o restaurar, o estado no topo da pilha é retirado e utilizado a transformação atual do contexto.

A função de ramificação no exemplo a seguir ilustra o que você pode fazer com uma função que altera a transformação e em seguida chama outra função que continua a desenhar com a transformação dada no desenho anterior.

Esta função desenha uma forma que lembra um desenho de uma árvore com linhas; movendo o sistema de coordenadas do centro para o fim da linha, e chamando ele novamente. A primeiro `rotate` acontece para a esquerda e depois para a direita. Cada chamada reduz o comprimento do ramo desenhado e a recursividade para quando o comprimento cai abaixo de 8.

```
<canvas width="600" height="300"></canvas>
<script>
  var 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>
```

Se as chamadas para salvar e restaurar não estivessem lá, a segunda chamada recursiva dos galho acabariam com a mesma posição de rotação criado pela primeira chamada. Não estariam ligados ao ramo atual, mas estaria a direita do ramo desenhado pela primeira chamada. A forma resultante também poderia ser interessante mas não é definitivamente uma árvore.

De volta para o jogo

Agora sabemos o suficiente sobre desenho no `canvas` para começarmos a trabalhar em um sistema de visualização baseada em `canvas` para o jogo a partir do capítulo anterior. O novo visual não será apenas mostrando caixas coloridas. Mas vamos usar `drawImage` para desenhar imagens que representam os elementos do jogo.

Vamos definir um tipo de objeto `CanvasDisplay`, suportando a mesma interface que `DOMDisplay` a partir do capítulo 15, ou seja os métodos `drawFrame` e `clear`.

Este objeto mantém um pouco mais de informação do que `DOMDisplay`. Ao invés de usar a posição de rolagem do seu elemento DOM, ele controla o seu próprio visor, que nos diz qual parte do nível atualmente que estamos olhando. Ele também rastreia o tempo e usa isso para decidir qual quadro da animação deve ser usado. E finalmente ele mantém uma propriedade `flipPlayer` de modo que mesmo quando o jogador ainda está de pé ele continua voltada para a direção do último movimento.

```
function CanvasDisplay(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.level = level;
  this.animationTime = 0;
  this.flipPlayer = false;

  this.viewport = {
    left: 0,
```

```

        top: 0,
        width: this.canvas.width / scale,
        height: this.canvas.height / scale
    };

    this.drawFrame(0);
}

CanvasDisplay.prototype.clear = function() {
    this.canvas.parentNode.removeChild(this.canvas);
};

```

O contador `animationTime` é a razão pela qual passou o tamanho do passo para `drawFrame` no Capítulo 15 embora `DOMDisplay` não utilizasse. Nossa nova função `drawFrame` iremos utilizar para controlar o tempo de modo que possa alternar entre quadros de animação com base no tempo atual.

```

CanvasDisplay.prototype.drawFrame = function(step) {
    this.animationTime += step;

    this.updateViewport();
    this.clearDisplay();
    this.drawBackground();
    this.drawActors();
};

```

Diferente do controle de tempo, o método atualiza a janela de exibição para a posição atual do jogador, preenche toda a tela com uma cor de fundo, desenha o fundo e os atores. Note que que é diferente da abordagem no capítulo 15 onde traçamos o plano de fundo toda vez que movemos qualquer elemento do DOM envolvido.

Como as formas em uma tela são apenas pixels, depois que atraído, não há nenhuma maneira de removê-los. A única maneira de atualizar a exibição de tela é limpar e redesenhar a cena.

O método `updateViewport` é semelhante ao método de `scrollPlayerIntoView` no `DOMDisplay`. Ele verifica se o jogador está demasiado perto da borda da tela e move a janela de exibição quando for o caso.

```

CanvasDisplay.prototype.updateViewport = function() {
    var view = this.viewport, margin = view.width / 3;
    var player = this.level.player;
    var 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,
                               this.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,
                               this.level.height - view.height);
};

```

As chamadas para `Math.max` e `Math.min` garantem que a janela de exibição não acabe mostrando espaço fora do nível. `Math.max(x, 0)` tem o efeito de assegurar que o número resultante não seja inferior a zero. `Math.min` da mesma forma, garante que um valor permaneça abaixo de um dado vinculado.

Ao limpar a tela vamos usar uma cor ligeiramente diferente dependendo se o jogo for ganho(mais claro) ou perdido(mais escura).

```

CanvasDisplay.prototype.clearDisplay = function() {
    if (this.level.status == "won")

```

```

        this.cx.fillStyle = "rgb(68, 191, 255)";
    else if (this.level.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);
};

```

Para desenhar o plano de fundo, corremos por entre as telhas que são visíveis na janela de exibição atual, usando o mesmo truque usado em `obstacleAt` no capítulo anterior.

```

var otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function() {
    var view = this.viewport;
    var xStart = Math.floor(view.left);
    var xEnd = Math.ceil(view.left + view.width);
    var yStart = Math.floor(view.top);
    var yEnd = Math.ceil(view.top + view.height);

    for (var y = yStart; y < yEnd; y++) {
        for (var x = xStart; x < xEnd; x++) {
            var tile = this.level.grid[y][x];
            if (tile == null) continue;
            var screenX = (x - view.left) * scale;
            var screenY = (y - view.top) * scale;
            var tileX = tile == "lava" ? scale : 0;
            this.cx.drawImage(otherSprites,
                              tileX, 0, scale, scale,
                              screenX, screenY, scale, scale);
        }
    }
};

```

Azulejos que não estão vazias(null) são desenhados com `drawImage`. A imagem `otherSprites` contém os outros elementos do jogo. Como os azulejos da parede, a telha de lava, e o sprite para uma moeda.



Azulejos de fundo são 20 por 20 pixels, usaremos a mesma escala que usamos no `DOMDisplay`. Assim o deslocamento para telhas de lava é de 20(o valor da variável de escala) e o deslocamento para paredes é 0.

Nós não nos incomodamos em esperar a imagem do sprite carregar. Chamando `drawImage` com uma imagem que não foi carregado e simplesmente ele não ira fazer nada. Assim não chamaremos o jogo corretamente para os primeiros frames enquanto a imagem ainda está sendo carregado, mas isso não é um problema grave, desde que mantenhamos a atualização da tela na cena correta, assim que carregamento terminar.

O carácter para caminhar que foi utilizado, sera usado para representar o jogador. O código que chama ele precisa pegar a posição da sprite com base no movimento atual do jogador. Os primeiros oito sprites contém uma animação curta. Quando o jogador está se movendo ao longo de um chão os ciclos são alternados entre as propriedades de `animationTime` da tela. Este é medido em segundos, e queremos mudar os quadros 12 vezes por segundo, assim que o tempo é multiplicado por 12. Quando o jogador está parado, vamos traçar a nona Sprite. Durante saltos que são reconhecidos pelo fato de que a velocidade vertical não é zero, nós usamos o décimo elemento que esta na sprite mais a direita.

Porque os sprites são ligeiramente mais largo do que o jogador? 24 ao invés de 16 pixels? Isso é para permitir algum espaço para os pés e braços em movimento, o método tem de ajustar a coordenada x e largura por um determinado montante(`playerXoverlap`).

```

var playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
var playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(x, y, width,
    var sprite = 8, player = this.level.player;
    width += playerXOverlap * 2;
    x -= playerXOverlap;
    if (player.speed.x != 0)
        this.flipPlayer = player.speed.x < 0;

    if (player.speed.y != 0)
        sprite = 9;
    else if (player.speed.x != 0)
        sprite = Math.floor(this.animationTime * 12) % 8;

    this.cx.save();
    if (this.flipPlayer)
        flipHorizontally(this.cx, x + width / 2);

    this.cx.drawImage(playerSprites, sprite * width, 0, width, height, x, y, width, height);
    this.cx.restore();
};

```

O método `drawPlayer` é chamado por `drawActors`, que é responsável pela elaboração de todos os atores no jogo.

```

CanvasDisplay.prototype.drawActors = function() {
    this.level.actors.forEach(function(actor) {
        var width = actor.size.x * scale;
        var height = actor.size.y * scale;
        var x = (actor.pos.x - this.viewport.left) * scale;
        var y = (actor.pos.y - this.viewport.top) * scale;
        if (actor.type == "player") {
            this.drawPlayer(x, y, width, height);
        } else {
            var tileX = (actor.type == "coin" ? 2 : 1) * scale;
            this.cx.drawImage(otherSprites,
                                tileX, 0, width, height,
                                x, y, width, height);
        }
    }, this);
};

```

Ao desenhar algo que não é o jogador, verificamos o seu tipo para encontrar o deslocamento correto na sprite. A telha de lava é encontrado no deslocamento 20 o sprite moeda é encontrada em 40 (duas vezes escala).

Nós temos que subtrair a posição da janela de exibição ao computar a posição do ator, (0,0) corresponde ao canto superior esquerdo da janela da exibição do nosso `canvas` na parte superior esquerda do `level`. Nós também poderíamos ter usado o `translate` para isso. De qualquer maneira funcionaria.

O documento minúsculo mostrado a seguir conecta o novo `display` em `runGame`:

```

<body>
  <script>
    runGame(GAME_LEVELS, CanvasDisplay);
  </script>
</body>

```

Escolhendo uma interface gráfica

Sempre que você precisar gerar gráficos no navegador, você pode escolher entre HTML, SVG, e `canvas`. Não há uma abordagem melhor que funciona em todas as situações. Cada opção tem pontos fortes e fracos.

HTML tem a vantagem de ser simples. Ele se integra bem com textos. Ambos SVG e Canvas permitem que você desenhe texto mas eles não ajudam no posicionamento ou envolvimento quando ocupam mais de uma linha. Em uma imagem baseada em HTML é fácil incluir blocos de texto.

SVG pode ser usado para produzir gráficos nítidos que ficam bem em qualquer nível de zoom. É mais difícil de usar do que HTML mas também é muito mais potente.

Ambos SVG e HTML podem construir uma estrutura de dados(DOM) que represente uma imagem. Isto torna possível modificar os elementos depois de serem desenhados. Se você precisa mudar várias vezes uma pequena parte de um grande desenho em resposta ao que o usuário está fazendo ou como parte de uma animação em `canvas` isso pode ser extremamente caro. O DOM também nos permite registrar manipuladores de eventos de mouse sobre cada elemento da imagem(mesmo em formas desenhadas com SVG). E isso não pode ser feito em `canvas`.

Mas a abordagem orientada a pixel da tela pode ser uma vantagem quando o desenho usa uma enorme quantidade de elementos minúsculos. O fato de não se criar uma estrutura de dados, mas de apenas chamá-los repetidamente sobre a mesma superfície de pixel, `canvas` dá um menor custo em performance.

Há também efeitos, como renderizar uma cena de um pixel de cada vez(por exemplo, fazer um desenho de raios) ou pós-processamento de uma imagem com JavaScript(com efeito de embaçado ou distorcida) que só pode ser realisticamente manipulados por uma técnica baseada em pixel.

Em alguns casos, você pode querer combinar várias destas técnicas. Por exemplo, você pode desenhar um gráfico com SVG ou `canvas`, mas mostrar a informação textual posicionando um elemento HTML em cima da imagem.

Para aplicações que não exigem muito, não importa muito por qual interface você ira escolher. A segunda exibição feita para o nosso jogo neste capítulo poderia ter sido implementado com qualquer uma dessas três tecnologias de gráficos, uma vez que não precisamos desenhar texto nem lidar com a interação do mouse ou trabalhar com um número extraordinariamente grande de elementos.

Sumário

Neste capítulo, discutimos as técnicas para desenhar gráficos no navegador, com foco no elemento `<canvas>`.

Um nó `canvas` representa uma área em um documento que o nosso programa pode desenhar. Este desenho é feito através do contexto do objeto de desenho, criado com o método `getContext`.

A interface de desenho em 2D nos permite preencher e traçar várias formas. Propriedade `fillStyle` do contexto determina como as formas são preenchidas. As propriedades `strokeStyle` e `lineWidth` controlam a forma de como as linhas são desenhadas.

Retângulos e pedaços de texto podem ser tiradas com uma única chamada de método. Os métodos `fillRect` e `strokeRect` desenharam retângulos e os métodos `fillText` e `strokeText` desenharam texto. Para criar formas personalizadas é preciso primeiro construir um `path`.

Chamando `beginPath` inicia um novo caminho. Uma série de outros métodos podem adicionar linhas e curvas para o `path` atual. Por exemplo `lineTo` pode adicionar uma linha reta. Quando um caminho é terminado ele pode ser preenchido com o método `fill` ou traçado com o método `stroke`.

Mover os pixels de uma imagem ou de outra tela no nosso `canvas` é realizado com o método `drawImage`. Por padrão esse método desenha a imagem da origem por inteiro, mas passando mais parâmetros você pode copiar uma área específica da imagem. Usamos isso para o nosso jogo onde copiamos poses individuais do personagem do jogo a partir de uma imagem que tinha muitas cenas.

Transformações permitem que você desenhe uma forma de múltiplas orientações. Um contexto de desenho em 2D tem uma transformação em curso que pode ser alterado com os métodos `translate`, `scale` e `rotate`. Estes irão afetar todas as operações dos desenhos subsequentes. Um estado de transformação podem ser salvas com o

método `save` e restaurado com o método `restore`.

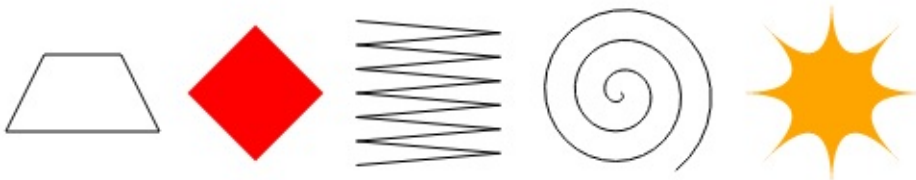
Ao desenhar uma animação sobre uma tela, o método `clearRect` pode ser usado para limpar parte da tela antes de redenhá-la novamente.

Exercícios

Shapes

Escreva um programa que tira as seguintes formas de uma tela:

- Um trapézio(um retângulo que é mais largo de um lado)
- Um diamante vermelho(um retângulo rotacionado em 45 graus ou $\frac{1}{4}\pi$ radianos)
- A linha em ziguezague
- Uma espiral composta de 100 segmentos de linha reta
- Uma estrela amarela



Ao desenhar os dois últimos, você pode querer referir-se a explicação do `Math.cos` e `Math.sin` do capítulo 13 que descreve como obter coordenadas em um círculo usando essas funções.

Eu recomendo a criação de uma função para cada forma. Passar a posição e outras propriedades como algo opcional tais como o tamanho ou o número de pontos. A alternativa é para tirar o hard-code do seu código, tende tornar o código fácil de ler e modificar.

```
<canvas width="600" height="200"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");

  // Your code here.
</script>
```

Dicas

O trapézio(1) é fácil desenhar usando um `path`. Escolha as coordenadas do centro adequado e adicione cada um dos quatro cantos em torno dele.

O diamante(2) pode ser desenhado de forma fácil com um `path`, uma maneira interessante pode ser feito com `transform` e `rotation`. Para usar `rotation` você terá que aplicar um truque semelhante ao que fizemos na função `flipHorizontally`. Você pode girar em torno do centro do seu retângulo e não em torno do ponto (0,0), primeiro você deve utilizar o `translate` em seguida `rotation` e então `translate` para voltar.

Para o ziguezague(3) torna-se impraticável escrever uma novo `path` para cada `lineTo` do segmento de uma linha. Em vez disso você deve usar um loop. Você pode desenhar com dois segmentos de linha(à direita e depois à esquerda). Use a regularidade(2%) do índice de loop para decidir se vai para a esquerda ou direita.

Você também vai precisar de um loop para a espiral(4). Se você desenhar uma série de pontos com cada ponto que se move mais ao longo de um círculo e ao redor do centro do espiral, você começara a fazer um círculo. Se durante o loop você variar o raio do círculo em que você está colocando o ponto atual o resultado sera um espiral.

A estrela(5) representado é construída a partir de linhas `quadraticCurveTo` . Você também pode tirar uma com linhas retas. Divida um círculo em oito pedaços, ou um pedaço para cada ponto que você quer que sua estrela tenha. Desenhar linhas entre estes pontos, tornam as curvas na direção do centro da estrela. Com `quadraticCurveTo` , você pode usar o centro como o ponto de controle.

Gráfico de pizza

No início do capítulo vimos um exemplo de programa que desenhou um gráfico de pizza. Modifique este programa para que o nome de cada categoria seja mostrado e fique ao lado de cada fatia que representa. Tente encontrar uma maneira agradável de mostrar e posicionar automaticamente este texto. Você pode assumir que as categorias não são menores do que 5 por cento.

Você pode precisar de novo do `Math.sin` e `Math.cos` conforme descrito no exercício anterior.

```
<canvas width="600" height="300"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var total = results.reduce(function(sum, choice) {
    return sum + choice.count;
  }, 0);

  var currentAngle = -0.5 * Math.PI;
  var centerX = 300, centerY = 150;
  // Add code to draw the slice labels in this loop.
  results.forEach(function(result) {
    var sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    cx.arc(centerX, centerY, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(centerX, centerY);
    cx.fillStyle = result.color;
    cx.fill();
  });
</script>
```

Dicas

Você vai precisar chamar `fillText` , definir `textAlign` e `textBaseline` para as propriedades do contexto de tal forma que o texto acabe onde quiser.

Uma forma sensata para posicionar os rótulos seria colocar o texto na linha que vai do centro de uma fatia ate o meio. Você não quer colocar o texto diretamente de encontro ao lado da fatia mas sim mover o texto para o lado da fatia por um determinado número de pixels.

O ângulo desta linha é `currentAngle + 0,5 * sliceAngle` . O código a seguir encontra-se em uma posição sobre esta linha de `120 pixels` para centro:

```
var middleAngle = currentAngle + 0.5 * sliceAngle;
var textX = Math.cos(middleAngle) * 120 + centerX;
var textY = Math.sin(middleAngle) * 120 + centerY;
```

Para `textBaseline` o valor `"middle"` é provavelmente uma abordagem a ser utilizada. O que for usado para `textAlign` depende do lado do círculo em que estamos. À esquerda deve ser `"center"` , a direita deve usar `"right"` , e `left` para texto que estiver posicionado longe do pedaço.

Se você não tem certeza de como descobrir qual lado do círculo um determinado ângulo esta, olhe para a explicação de `Math.cos` no exercício anterior. O cosseno de um ângulo nos diz qual coordenada x corresponde, que por sua vez nos diz exatamente que lado do círculo em que estamos.

Quicando a bola

Use a técnica `requestAnimationFrame` que vimos no Capítulo 13 e no Capítulo 15 para desenhar uma caixa com uma bola quicando dentro. A bola se move a uma velocidade constante e rebate nos lados da caixa quando é tocada.

```
<canvas width="400" height="400"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");

  var lastTime = null;
  function frame(time) {
    if (lastTime != null)
      updateAnimation(Math.min(100, time - lastTime) / 1000);
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);

  function updateAnimation(step) {
    // Your code here.
  }
</script>
```

Dicas

A caixa é fácil de desenhar com `strokeRect`. Definir uma variável que contém o seu tamanho e definir duas variáveis da largura e altura da sua caixa. Para criar uma bola redonda, inicie um `path` chamando `arc(x, y, raio, 0, 7)` que cria um arco que vai de zero pra cima para um círculo completo, e depois preencha.

Para modelar a posição da bola e velocidade, você pode usar o tipo vetor a partir do capítulo 15. Dê uma velocidade de partida de preferência um que não é puramente vertical ou horizontal, e a cada quadro multiplique a velocidade com a quantidade de tempo que decorreu. Quando a bola fica muito perto de uma parede vertical inverta o componente x em sua velocidade. Da mesma forma inverta o componente y quando ela atinge uma parede na horizontal.

Depois de encontrar a nova posição e velocidade da bola, use `clearRect` para excluir a cena e redesenhá-lo usando a nova posição.

Espelhamento pre computado

Uma coisa ruim sobre `transformation` é que eles diminuem a qualidade do desenho de bitmaps. Para gráficos vectoriais o efeito é menos grave uma vez que apenas alguns pontos (por exemplo, o centro de um círculo) precisam de ser transformado, após ser desenhado normalmente. Para uma imagem de bitmap a posição de cada pixel tem que ser transformado, embora seja possível que os navegadores vão ficar mais inteligente sobre isso no futuro, atualmente este provoca um aumento considerável no tempo em que leva para desenhar um bitmap.

Em um jogo como o nosso, onde estamos desenhando apenas uma única entidade gráfica e transformando, isto não é um problema. Mas imagine que precisamos desenhar centenas de personagens ou milhares de partículas em rotação de uma explosão.

Pense em uma maneira que nos permite desenhar um personagem invertido sem carregar arquivos de imagem e sem ter que ficar transformando `drawImage` a cada frame que se chama.

Dica

A chave para a solução é o fato de que nós podemos usar um elemento de tela como uma imagem de origem ao usar `drawImage`. É possível criar um elemento extra de `<canvas>` sem adicioná-lo ao documento e colocar nossas sprites invertidas. Ao desenhar um quadro real, nós apenas copiamos as sprites já invertidos para a tela principal.

Alguns cuidados seria necessária porque as imagens não são carregadas instantaneamente. Fazemos o desenho invertido apenas uma vez e se fizermos isso antes do carregamento das imagens ele não vai chamar nada. Um manipulador `"load"` sobre a imagem pode ser usada para desenhar as imagens invertidas para o `canvas` extra. Esta área pode ser usado como uma fonte de desenho imediatamente (ele vai simplesmente ficar em branco até que desenhar o personagem apareça).

HTTP

O sonho por trás da Web é o de um espaço comum de informações no qual podemos nos comunicar compartilhando informações. Sua universalidade é essencial. O fato de que um link pode apontar para qualquer coisa, seja ela pessoal, local ou global, seja ela um rascunho ou algo refinado.

— Tim Berners-Lee, *The World Wide Web: A very short personal history*

O *Hypertext Transfer Protocol*, já mencionado no [capítulo 12](#), é o mecanismo no qual dados são requisitados e entregues na *World Wide Web*. Esse capítulo descreve o protocolo com mais detalhes e explica como o JavaScript executado no navegador tem acesso a ele.

O Protocolo

Se você digitar `eloquentjavascript.net/17_http.html` na barra de endereços do seu navegador, ele irá, primeiramente, procurar o endereço do servidor associado ao domínio `eloquentjavascript.net` e, em seguida, tentar abrir uma conexão TCP com ele na porta 80, a porta padrão para tráfego HTTP. Se o servidor existir e aceitar a conexão, o navegador enviará algo parecido com:

```
GET /17_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

Então, por meio da mesma conexão, o servidor responde.

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT

<!doctype html>
... the rest of the document
```

O navegador participa da resposta após a linha em branco e a mostra como um documento HTML.

A informação enviada pelo cliente é chamada de requisição (*request*) e inicia com essa linha:

```
GET /17_http.html HTTP/1.1
```

A primeira palavra é o *método* da requisição. `GET` significa que queremos acessar o recurso em questão. Outros métodos comuns são `DELETE` para deletar um recurso, `PUT` para substituí-lo e `POST` para enviar uma informação. Note que o servidor não é obrigado a processar todas as requisições que receber. Se você acessar um website aleatório e fizer uma requisição `DELETE` em sua página principal, ele provavelmente irá recusar essa ação.

A parte após o nome do método é o caminho do recurso ao qual a requisição está sendo aplicada. No caso mais simples, um recurso é simplesmente um arquivo no servidor, entretanto, o protocolo não requer que o recurso seja necessariamente um arquivo. Um recurso pode ser qualquer coisa que possa ser transferida *como se fosse* um arquivo. Muitos servidores geram as respostas na medida em que são solicitados. Por exemplo, se você acessar twitter.com/marijnjh, o servidor irá procurar em seu banco de dados por um usuário chamado *marijnjh* e, se encontrá-lo, irá gerar a página de perfil desse usuário.

Após o caminho do recurso, a primeira linha da requisição menciona `HTTP/1.1` para indicar a versão do protocolo HTTP que está sendo usada.

A resposta do servidor irá iniciar também com a versão, seguida pelo *status* da resposta, representado primeiramente por um código de três dígitos e, em seguida, por um texto legível.

```
HTTP/1.1 200 OK
```

Os *status code* (códigos de *status*) que iniciam com o número 2 indicam que a requisição foi bem-sucedida. Códigos que começam com 4, indicam que houve algum problema com a requisição. O código de resposta HTTP provavelmente mais famoso é o 404, que significa que o recurso solicitado não foi encontrado. Códigos que começam com 5 indicam que houve um erro no servidor e que a culpa não é da requisição.

A primeira linha de uma requisição ou resposta pode ser seguida por qualquer quantidade de *headers* (cabeçalhos). Eles são representados por linhas na forma de "nome: valor" que especificam informações extra sobre a requisição ou resposta. Os *headers* abaixo fazem parte do exemplo de resposta usado anteriormente:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT
```

Eles nos informam o tamanho e o tipo do documento da resposta. Nesse caso, é um documento HTML com 65.585 bytes. Além disso, ele nos mostra quando foi a última vez que o documento foi modificado.

Na maioria das vezes, o cliente ou o servidor decidem quais *headers* serão incluídos em uma requisição ou resposta, apesar de alguns serem obrigatórios. Por exemplo, o *header* `Host`, que especifica o *hostname*, deve ser incluído na requisição pois o servidor pode estar servindo múltiplos *hostnames* em um mesmo endereço IP e, sem esse *header*, o servidor não saberá qual *host* o cliente está tentando se comunicar.

Após os *headers*, tanto as requisições quanto as respostas podem incluir uma linha em branco seguida por um *body* (corpo), que contém os dados que estão sendo enviados. As requisições `GET` e `DELETE` não enviam nenhum tipo dado, mas `PUT` e `POST` enviam. De maneira similar, alguns tipos de resposta, como respostas de erro, não precisam de um *body*.

Navegadores e o HTTP

Como vimos no exemplo anterior, o navegador irá fazer uma requisição quando submetermos uma URL na barra de endereços. Quando a página HTML resultante faz referências a outros arquivos como imagens e arquivos JavaScript, eles também são requisitados.

Um website razoavelmente complicado pode facilmente ter algo em torno de dez a duzentos recursos. Para ser capaz de buscá-los rapidamente, ao invés de esperar pelo retorno das respostas de cada requisição feita, os navegadores fazem várias requisições simultaneamente. Tais documentos são sempre requisitados usando requisições `GET`.

Páginas HTML podem incluir *formulários*, que permitem ao usuário preencher e enviar informações para o servidor. Esse é um exemplo de um formulário:

```
<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>
```

Esse código descreve um formulário com dois campos: um campo menor que solicita um nome e um campo maior que solicita que o usuário escreva uma mensagem. Quando você clicar no botão *Send* (enviar), a informação contida nos campos serão convertidas em uma *query string*. Quando o método do atributo do elemento `<form>` for `GET` (ou o método for omitido), a *query string* é associada à URL contida em `action` e o navegador executa a requisição `GET` para essa URL.

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

O início de uma *query string* é indicado por um ponto de interrogação seguido por pares de nomes e valores, correspondendo ao atributo `name` de cada campo do formulário e seus respectivos valores. O caractere `&` é usado para separar os pares.

A mensagem codificada na URL anterior é "Yes?", mesmo que o ponto de interrogação tenha sido substituído por um código estranho. Alguns caracteres nas *query strings* precisam ser escapados. O ponto de interrogação, representado como `%3F`, é um desses casos. Parece haver uma regra não escrita de que cada formato necessita ter sua própria forma de escapar caracteres. Esse formato que está sendo usado é chamado de *URL encoding* e utiliza o sinal de porcentagem seguido por dois dígitos hexadecimais que representam o código daquele caractere. Nesse caso, o 3F significa 63 na notação decimal, que é o código do caractere de interrogação. O JavaScript fornece as funções `encodeURIComponent` e `decodeURIComponent` para codificar e decodificar esse formato.

```
console.log(encodeURIComponent("Hello & goodbye"));  
// → Hello%20%26%20goodbye  
console.log(decodeURIComponent("Hello%20%26%20goodbye"));  
// → Hello & goodbye
```

Se alterarmos o método do atributo do formulário HTML no exemplo anterior para `POST`, a requisição HTTP que será feita para enviar o formulário irá usar o método `POST` e a *query string* será adicionada ao corpo da requisição, ao invés de ser colocada diretamente na URL.

```
POST /example/message.html HTTP/1.1  
Content-length: 24  
Content-type: application/x-www-form-urlencoded  
  
name=Jean&message=Yes%3F
```

Por convenção, o método `GET` é usado para requisições que não produzem efeitos colaterais, tais como fazer uma pesquisa. Requisições que alteram alguma coisa no servidor, como criar uma nova conta ou postar uma nova mensagem, devem ser expressadas usando outros métodos, como `POST`. Aplicações *client-side*, como os navegadores, sabem que não devem fazer requisições `POST` cegamente, mas frequentemente farão requisições `GET` implícitas para, por exemplo, pré-carregar um recurso que ele acredita que o usuário irá precisar no curto prazo.

O [próximo capítulo](#) irá retomar o assunto formulários e explicará como podemos desenvolvê-los usando JavaScript.

XMLHttpRequest

A interface pela qual o JavaScript do navegador pode fazer requisições HTTP é chamada de `XMLHttpRequest` (observe a forma inconsistente de capitalização). Ela foi elaborada pela Microsoft, para o seu navegador Internet Explorer, no final dos anos 90. Naquela época, o formato de arquivo XML era *muito* popular no contexto dos softwares corporativos, um mundo no qual sempre foi a casa da Microsoft. O formato era tão popular que o acrônimo XML foi adicionado ao início do nome de uma interface para o HTTP, a qual não tinha nenhuma relação com o XML.

Mesmo assim, o nome não é completamente sem sentido. A interface permite que você analise os documentos de resposta como XML, caso queira. Combinar dois conceitos distintos (fazer uma requisição e analisar a resposta) em uma única coisa é com certeza um péssimo design.

Quando a interface `XMLHttpRequest` foi adicionada ao Internet Explorer, foi permitido às pessoas fazerem coisas com JavaScript que eram bem difíceis anteriormente. Por exemplo, websites começaram a mostrar listas de sugestões enquanto o usuário digitava algo em um campo de texto. O script mandava o texto para o servidor usando HTTP enquanto o usuário estivesse digitando. O servidor, que tinha um banco de dados com possíveis entradas, comparava as possíveis entradas com a entrada parcial digitada pelo usuário, enviando de volta possíveis combinações de resultados para mostrar ao usuário. Isso era considerado espetacular, pois as pessoas estavam acostumadas a aguardar por uma atualização completa da página para cada interação com o website.

O outro navegador relevante naquela época, chamado Mozilla (mais tarde Firefox), não queria ficar para trás. Para permitir que as pessoas pudessem fazer coisas similares em seu navegador, eles copiaram a interface, incluindo o controverso nome. A próxima geração de navegadores seguiram esse exemplo e, por isso, a interface `XMLHttpRequest` é um padrão atualmente.

Enviando uma requisição

Para fazer uma simples requisição, criamos um objeto de requisição com o construtor `XMLHttpRequest` e chamamos os seus métodos `open` e `send`.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.responseText);
// → This is the content of data.txt
```

O método `open` configura a requisição. Nesse caso, escolhemos fazer uma requisição `GET` para o arquivo `example/data.txt`. As URLs que não começam com um nome de protocolo (como por exemplo `http:`) são relativas, ou seja, são interpretadas em relação ao documento atual. Quando elas iniciam com uma barra (`/`), elas substituem o caminho atual, que é a parte após o nome do servidor. No caso de não iniciarem com uma barra, a parte do caminho em questão até (e incluindo) a última barra é colocada em frente à URL relativa.

Após abrir a requisição, podemos enviá-la usando o método `send`. O argumento a ser enviado é o corpo da requisição. Para requisições `GET`, podemos passar `null`. Se o terceiro argumento passado para `open` for `false`, o método `send` irá retornar apenas depois que a resposta da nossa requisição for recebida. Podemos ler a propriedade `responseText` do objeto da requisição para acessar o corpo da resposta.

As outras informações incluídas na resposta também podem ser extraídas desse objeto. O *status code* (código de status) pode ser acessado por meio da propriedade `status` e a versão legível em texto do *status* pode ser acessada por meio da propriedade `statusText`. Além disso, os cabeçalhos podem ser lidos com `getResponseHeader`.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.status, req.statusText);
// → 200 OK
console.log(req.getResponseHeader("content-type"));
// → text/plain
```

Os nomes dos cabeçalhos são *case-insensitive* (não faz diferença entre letras maiúsculas e minúsculas). Eles são normalmente escritos com letra maiúscula no início de cada palavra, como por exemplo "Content-Type". Entretanto, as respectivas variações "content-type" e "cOnTeNt-TyPe" fazem referência ao mesmo cabeçalho.

O navegador irá automaticamente adicionar alguns cabeçalhos da requisição, tais como "Host" e outros necessários para o servidor descobrir o tamanho do corpo da requisição. Mesmo assim, você pode adicionar os seus próprios cabeçalhos usando o método `setRequestHeader`. Isso é necessário apenas para usos avançados e requer a cooperação do servidor ao qual você está se comunicando (o servidor é livre para ignorar cabeçalhos que ele não sabe lidar).

Requisições Assíncronas

Nos exemplos que vimos, a requisição finaliza quando a chamada ao método `send` retorna. Isso é conveniente pois significa que as propriedades como `responseText` ficam disponíveis imediatamente. Por outro lado, o nosso programa fica aguardando enquanto o navegador e o servidor estão se comunicando. Quando a conexão é ruim, o servidor lento ou o arquivo é muito grande, o processo pode demorar um bom tempo. Ainda pior, devido ao fato de que nenhum manipulador de evento pode ser disparado enquanto nosso programa está aguardando, todo o documento ficará não responsivo.

Se passarmos `true` como terceiro argumento para `open`, a requisição é *assíncrona*. Isso significa que quando chamar o método `send`, a única coisa que irá acontecer imediatamente é o agendamento da requisição que será enviada. Nosso programa pode continuar a execução e o navegador irá ser responsável por enviar e receber os dados em segundo plano.

Entretanto, enquanto a requisição estiver sendo executada, nós não podemos acessar a resposta. É necessário um mecanismo que nos avise quando os dados estiverem disponíveis.

Para isso, precisamos escutar o evento `"load"` no objeto da requisição.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
    console.log("Done:", req.status);
});
req.send(null);
```

Assim como o uso de `requestAnimationFrame` no [Capítulo 15](#), essa situação nos obriga a usar um estilo assíncrono de programação, encapsulando as coisas que precisam ser executadas após a requisição em uma função e preparando-a para que possa ser chamada no momento apropriado. Voltaremos nesse assunto [mais a frente](#).

Recuperando Dados XML

Quando o recurso recebido pelo objeto `XMLHttpRequest` é um documento XML, a propriedade `responseXML` do objeto irá conter uma representação desse documento. Essa representação funciona de forma parecida com o DOM, discutida no [Capítulo 13](#), exceto que ela não contém funcionalidades específicas do HTML, como por exemplo a propriedade `style`. O objeto contido em `responseXML` corresponde ao objeto do documento. Sua propriedade `documentElement` se refere à *tag* mais externa do documento XML. No documento a seguir (*example/fruit.xml*), essa propriedade seria a *tag* `<fruits>`:

```
<fruits>
  <fruit name="banana" color="yellow"/>
  <fruit name="lemon" color="yellow"/>
  <fruit name="cherry" color="red"/>
</fruits>
```

Podemos recuperar esse arquivo da seguinte forma:

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.xml", false);
req.send(null);
console.log(req.responseXML.querySelectorAll("fruit").length);
// → 3
```

Documentos XML podem ser usados para trocar informações estruturadas com o servidor. Sua forma (*tags* dentro de outras *tags*) faz com que seja fácil armazenar a maioria dos tipos de dados, sendo uma alternativa melhor do que usar um simples arquivo de texto. Entretanto, extrair informações da interface DOM é um pouco trabalhoso e os documentos XML tendem a ser verbosos. Normalmente é uma ideia melhor se comunicar usando dados JSON, os quais são mais fáceis de ler e escrever tanto para programas quanto para humanos.

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.json", false);
req.send(null);
console.log(JSON.parse(req.responseText));
// → {banana: "yellow", lemon: "yellow", cherry: "red"}
```

HTTP Sandboxing

Fazer requisições HTTP usando *scripts* em uma página web levanta preocupações em relação à segurança. A pessoa que controla o *script* pode não ter os mesmos interesses que a pessoa do computador o qual o *script* está executando. Mais especificamente, se eu visitar *themafia.org*, eu não quero que seus *scripts* possam fazer requisições para *mybank.com*, usando informações de identificação do meu navegador, com instruções para transferir todo meu dinheiro para alguma conta da máfia.

É possível que websites se protejam desses tipos de ataques, porém, é preciso muito esforço e muitos websites falham ao fazê-lo. Por essa razão, os navegadores nos protegem não permitindo que *scripts* façam requisições HTTP para outros domínios (nomes como *themafia.org* e *mybank.com*).

Isso pode ser um problema irritante quando construímos sistemas que queiram acessar diferentes domínios por razões legítimas. Felizmente, servidores podem incluir um cabeçalho como esse em sua resposta para indicar explicitamente aos navegadores que é permitido requisições que venham de outros domínios:

```
Access-Control-Allow-Origin: *
```

Abstraindo Requisições

No [Capítulo 10](#), em nossa implementação do módulo AMD, usamos uma função hipotética chamada

`backgroundReadFile`. Ela recebia um nome de arquivo e uma função e, após o carregamento do arquivo, chamava a função com o conteúdo recuperado. Aqui está uma implementação simples dessa função:

```
function backgroundReadFile(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
  });
  req.send(null);
}
```


Essa simples abstração torna mais fácil usar `XMLHttpRequest` para fazer simples requisições `GET`. Se você está escrevendo um programa que precisa fazer requisições HTTP, é uma boa ideia usar uma função auxiliar para que você não acabe repetindo o esquisito padrão `XMLHttpRequest` por todo o seu código.

O nome da função argumento (`callback`) é um termo frequentemente usado para descrever funções como essa. Uma função *callback* é fornecida para outro código de forma que possa "ser chamada" mais tarde.

Não é difícil escrever uma função utilitária HTTP adaptada para o que sua aplicação está fazendo. A função anterior apenas faz requisições `GET` e não nos dá controle sobre os cabeçalhos ou sobre o corpo da requisição. Você pode escrever outra variação da função para requisições `POST` ou uma versão genérica que suporte vários tipos de requisições. Muitas bibliotecas JavaScript também fornecem funções *wrappers* para `XMLHttpRequest`.

O principal problema com a função *wrapper* anterior é sua capacidade de lidar com falhas. Quando a requisição retorna um código de *status* que indica um erro (400 para cima), ela não faz nada. Isso pode ser aceitável em algumas circunstâncias, mas imagine que colocamos na página um indicador de "carregando" para dizer que estamos carregando informação. Se a requisição falhar por causa de queda do servidor ou a conexão for interrompida, a página irá apenas manter o seu estado, parecendo que está fazendo alguma coisa. O usuário irá esperar um tempo, ficar impaciente e, por fim, considerar que o site é problemático.

Nós também devemos ter uma opção de ser notificado quando a requisição falhar para que possamos tomar uma ação apropriada. Por exemplo, podemos remover a mensagem "carregando" e informar ao usuário que algo errado aconteceu.

Manipular erros em código assíncrono é ainda mais trabalhoso do que manipulá-los em código síncrono. Frequentemente, adiamos parte do nosso trabalho colocando-o em uma função *callback* e, por isso, o escopo de um bloco `try` acaba não fazendo sentido. No código a seguir, a exceção *não* será capturada pois a chamada à função `backgroundReadFile` retorna imediatamente. O controle de execução então sai do bloco `try` e a função que foi fornecida não será chamada até um momento posterior.

```
try {
  backgroundReadFile("example/data.txt", function(text) {
    if (text !== "expected")
      throw new Error("That was unexpected");
  });
} catch (e) {
  console.log("Hello from the catch block");
}
```

Para lidar com requisições que falharam, precisamos permitir que uma função adicional seja passada para nosso *wrapper* e chamá-la quando ocorrer algo errado. Alternativamente, podemos usar a convenção de que, caso a requisição falhar, um argumento adicional descrevendo o problema é passado para a chamada regular da função *callback*. Segue um exemplo:

```
function getURL(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
    else
      callback(null, new Error("Request failed: " +
                              req.statusText));
  });
  req.addEventListener("error", function() {
    callback(null, new Error("Network error"));
  });
  req.send(null);
}
```

Adicionamos um manipulador para o evento de `"error"` (erro), o qual será sinalizado quando a requisição falhar por completo. Também chamamos a função *callback* com um argumento de erro quando a requisição finaliza com um código de *status* que indica um erro.

O código que utiliza `getUrl` deve, então, verificar se um erro foi fornecido e, caso tenha sido, tratá-lo.

```
getUrl("data/nonsense.txt", function(content, error) {
  if (error != null)
    console.log("Failed to fetch nonsense.txt: " + error);
  else
    console.log("nonsense.txt: " + content);
});
```

Isso não funciona quando se trata de exceções. Ao encadear várias ações assíncronas, uma exceção em qualquer ponto da cadeia ainda irá (a não ser que você envolva cada função em seu próprio bloco `try / catch`) chegar ao topo e abortar a sequência de ações.

Promises

Para projetos complicados, escrever código assíncrono usando o estilo de *callbacks* é difícil de ser feito corretamente. É fácil esquecer de verificar um erro ou permitir que uma exceção inesperada encerre a execução do programa. Além disso, lidar com erros quando os mesmos devem ser passados por um fluxo de múltiplas funções *callback* e blocos *catch* é tedioso.

Já foram feitas várias tentativas para resolver esse problema usando abstrações adicionais. Uma das mais bem-sucedidas é chamada de *promises*. *Promises* encapsulam uma ação assíncrona em um objeto, que pode ser passado e instruído a fazer certas coisas quando a ação finalizar ou falhar. Essa interface está definida para ser parte da próxima versão da linguagem JavaScript, mas já pode ser usada em forma de biblioteca.

A interface para usar *promises* não é muito intuitiva, mas é poderosa. Esse capítulo irá brevemente descrevê-la. Você pode encontrar mais informações em promisejs.org

Para criar um objeto *promise*, chamamos o construtor `Promise` passando uma função que inicia a ação assíncrona. O construtor chama essa função passando dois argumentos, os quais também são funções. A primeira função deve ser chamada quando a ação terminar com sucesso e a segunda quando ela falhar.

Mais uma vez, segue nosso *wrapper* para requisições `GET`, dessa vez retornando uma *promise*. Nós vamos apenas chamá-lo de `get` dessa vez.

```
function get(url) {
  return new Promise(function(succeed, fail) {
    var req = new XMLHttpRequest();
    req.open("GET", url, true);
    req.addEventListener("load", function() {
      if (req.status < 400)
        succeed(req.responseText);
      else
        fail(new Error("Request failed: " + req.statusText));
    });
    req.addEventListener("error", function() {
      fail(new Error("Network error"));
    });
    req.send(null);
  });
}
```

Note que a interface da função em si é bem mais simples. Você passa uma URL e ela retorna uma *promise*. Essa *promise* atua como um manipulador do resultado da requisição. Ela possui um método `then` que pode ser chamado com duas funções: uma para tratar o sucesso e outra para tratar a falha.

```
get("example/data.txt").then(function(text) {  
  console.log("data.txt: " + text);  
}, function(error) {  
  console.log("Failed to fetch data.txt: " + error);  
});
```

Até agora, isso é apenas outra forma de expressar a mesma coisa que já havíamos expressado. É apenas quando você precisa encadear ações que as *promises* fazem uma diferença significativa.

Chamar o método `then` produz uma nova *promise*, a qual o resultado (o valor passado ao manipulador em caso de sucesso) depende do valor de retorno da primeira função fornecida ao `then`. Essa função pode retornar outra *promise* para indicar que mais tarefas assíncronas estão sendo executadas. Nesse caso, a *promise* retornada pelo `then` irá esperar pela *promise* retornada pela função manipuladora, obtendo sucesso ou falhando com o mesmo valor quando for resolvida. Quando a função manipuladora retornar um valor que não seja uma *promise*, a *promise* retornada pelo `then` imediatamente retorna com sucesso usando esse valor como seu resultado.

Isso significa que você pode usar `then` para transformar o resultado de uma *promise*. Por exemplo, o código a seguir retorna uma *promise* a qual o resultado é o conteúdo de uma dada URL representada como JSON:

```
function getJSON(url) {  
  return get(url).then(JSON.parse);  
}
```

A última chamada ao `then` não especificou um manipulador para falhas. Isso é permitido. O erro será passado para a *promise* retornada pelo `then`, que é exatamente o que queremos — `getJSON` não sabe o que fazer quando algo der errado mas, esperançosamente, a função que o chamou sabe.

Como um exemplo que demonstra o uso de *promises*, iremos construir um programa que carrega um número de arquivos JSON do servidor e, enquanto isso é feito, mostra a palavra "carregando". Os arquivos JSON contêm informações sobre pessoas, com links para arquivos que representam outras pessoas em condições como `father`, `mother` ou `spouse` (pai, mãe ou cônjuge).

Nós queremos recuperar o nome da mãe do cônjuge de `example/bert.json` e, se algo der errado, remover o texto "carregando" e mostrar uma mensagem de erro. Segue uma forma de como isso pode ser feito usando *promises*:

```
<script>  
  function showMessage(msg) {  
    var elt = document.createElement("div");  
    elt.textContent = msg;  
    return document.body.appendChild(elt);  
  }  
  
  var loading = showMessage("Loading...");  
  getJSON("example/bert.json").then(function(bert) {  
    return getJSON(bert.spouse);  
  }).then(function(spouse) {  
    return getJSON(spouse.mother);  
  }).then(function(mother) {  
    showMessage("The name is " + mother.name);  
  }).catch(function(error) {  
    showMessage(String(error));  
  }).then(function() {  
    document.body.removeChild(loading);  
  });  
</script>
```

O programa acima é relativamente compacto e legível. O método `catch` é similar ao `then`, exceto que ele espera um manipulador de erro como argumento e passará pelo resultado sem alterá-lo em caso de sucesso. Muito parecido com o `catch` em um bloco `try`, o controle de execução irá continuar normalmente depois que a falha é capturada. Dessa forma, o `then` que executa ao final e é responsável por remover a mensagem de "carregando", é sempre executado, mesmo se algo der errado.

Você pode pensar na interface de *promise* como uma implementação de uma linguagem própria para o controle de fluxo assíncrono. As chamadas adicionais de métodos e expressões de funções fazem com que o código pareça um pouco estranho, mas não tão estranhos quanto se tivéssemos que lidar com todos os erros nós mesmos.

Apreciando o HTTP

Quando estamos construindo um sistema que requer comunicação entre um programa executando JavaScript no navegador (*client-side*) e um programa em um servidor (*server-side*), existem várias maneiras diferentes de modelar essa comunicação.

Um modelo bastante usado é o de *chamadas de procedimentos remotos*. Nesse modelo, a comunicação segue o padrão de chamadas normais de função, exceto pelo fato de que a função está sendo executada em outra máquina. Essa chamada envolve fazer uma requisição ao servidor, incluindo o nome da função e seus argumentos. A resposta para essa requisição contém o valor retornado.

Quando estiver pensando sobre chamadas de procedimentos remotos, o HTTP é apenas um veículo para a comunicação, e você provavelmente escreverá uma camada de abstração para escondê-la.

Outra abordagem é construir sua comunicação em torno do conceito de recursos e métodos HTTP. Ao invés de um procedimento remoto chamado `addUser`, utilizar uma requisição `PUT` para `/users/larry`. Ao invés de passar as propriedades daquele usuário como argumentos da função, você define um formato de documento ou usa um formato existente que represente um usuário. O corpo da requisição `PUT` para criar um novo recurso é simplesmente tal documento. Um recurso pode ser acessado por meio de uma requisição `GET` para a URL do recurso (por exemplo, `/user/larry`), o qual retorna o documento que representa tal recurso.

Essa segunda abordagem torna mais fácil utilizar as funcionalidades que o HTTP fornece, como suporte para *cache* de recursos (mantendo uma cópia no lado do cliente). Além disso, ajuda na coerência de sua interface, visto que os recursos são mais fáceis de serem compreendidos do que um monte de funções.

Segurança e HTTPS

Dados que trafegam pela Internet tendem a seguir uma longa e perigosa estrada. Para chegar ao seu destino, a informação passa por vários ambientes desde redes Wi-Fi em cafeterias até redes controladas por várias empresas e estados. Em qualquer ponto dessa rota, os dados podem ser inspecionados e, até mesmo, modificados.

Se for importante que algo seja secreto, como a senha da sua conta de email, ou que chegue ao destino final sem ser modificado, como o número da conta que você irá transferir dinheiro por meio do site do seu banco, usar simplesmente HTTP não é bom o suficiente.

O protocolo HTTP seguro, o qual as URLs começam com `https://`, encapsula o tráfego HTTP de forma que dificulta a leitura e alteração. Primeiramente, o cliente verifica se o servidor é de fato quem ele diz ser, obrigando-o a provar que possui um certificado criptográfico emitido por uma autoridade certificadora que o navegador reconheça. Por fim, todos os dados que trafegam pela conexão são criptografados de forma que assegure que eles estejam protegidos contra espionagem e violação.

Por isso, quando funciona corretamente, o HTTPs previne ambas situações onde alguém finja ser o website ao qual você estava tentando se comunicar e quando alguém está vigiando sua comunicação. O protocolo não é perfeito e já houveram vários incidentes onde o HTTPs falhou por causa de certificados forjados, roubados e software corrompido. Mesmo assim, é trivial burlar o HTTP simples, enquanto que burlar o HTTPs requer um certo nível de esforço que apenas estados ou organizações criminosas sofisticadas estão dispostas a fazer.

Resumo

Vimos nesse capítulo que o HTTP é um protocolo para acessar recursos usando a Internet. O *cliente* envia uma requisição, a qual contém um método (normalmente `GET`) e um caminho que identifica o recurso. O *servidor* então decide o que fazer com a requisição e responde com um código de *status* e o corpo da resposta. Tanto requisições quanto respostas podem conter *headers* (cabeçalhos) que fornecem informação adicional.

Navegadores fazem requisições `GET` para acessar recursos necessários para mostrar uma página web. Uma página pode também conter formulários, os quais permitem que informações inseridas pelo usuário sejam enviadas juntas com a requisição quando o formulário é submetido. Você aprenderá mais sobre esse assunto no [próximo capítulo](#).

A interface na qual o JavaScript do navegador pode fazer requisições HTTP é chamada `XMLHttpRequest`. Você normalmente pode ignorar o "XML" do nome (mas mesmo assim precisa digitá-lo). Existem duas formas em que a interface pode ser usada. A primeira forma é síncrona, bloqueando toda a execução até que a requisição finalize. A segunda é assíncrona, precisando usar um manipulador de eventos para avisar que a resposta chegou. Em quase todos os casos é preferível usar a forma assíncrona. Fazer uma requisição é algo parecido com o código a seguir:

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
  console.log(req.status);
});
req.send(null);
```

Programação assíncrona é traiçoeira. *Promises* são interfaces que tornam a programação assíncrona um pouco mais fácil, ajudando a rotear condições de erro e exceções para os manipuladores corretos e abstraindo muitos elementos repetitivos e suscetíveis a erro presentes nesse estilo de programação.

Exercícios

Negociação de conteúdo

Uma das coisas que o HTTP pode fazer, mas que não discutimos nesse capítulo, é chamada de *negociação de conteúdo*. O cabeçalho `Accept` de uma requisição pode ser usado para dizer ao servidor qual o tipo de documento que o cliente gostaria de receber. Muitos servidores ignoram esse cabeçalho, mas quando o servidor sabe como converter um recurso de várias formas, ele pode olhar esse cabeçalho e enviar a forma que o cliente prefere.

A URL eloquentjavascript.net/author é configurada para responder tanto com formato simples de texto quanto com HTML ou JSON, dependendo de como o cliente solicita. Esses formatos são identificados pelos *tipos de mídia* `text/plain`, `text/html` e `application/json`.

Envie requisições para recuperar todos os três formatos desse recurso. Use o método `setRequestHeader` do seu objeto `XMLHttpRequest` para definir o valor do cabeçalho chamado `Accept` para um dos tipos de mídia descritos acima. Certifique-se de configurar o cabeçalho *após* chamar o método `open` e antes de chamar o método `send`.

Por fim, tente solicitar pelo tipo de mídia `application/rainbows+unicorns` e veja o que acontece.

```
// Your code here.
```

Dicas:

Veja os vários exemplos que usam `XMLHttpRequest` nesse capítulo, para ter uma ideia de como são as chamadas de métodos que envolvem fazer uma requisição. Você pode usar uma requisição síncrona (informando `false` como o terceiro parâmetro para `open`), se preferir.

Solicitar por um tipo de mídia inexistente, fará com que a resposta seja retornada com o código 406, "*Not acceptable*" (Não aceitável), que é o código que o servidor deve retornar quando ele não pode satisfazer o cabeçalho `Accept`.

Esperando por múltiplas *promises*

O construtor *Promise*, possui um método chamado `all` que, quando fornecido um *array* de *promises*, retorna uma *promise* que aguarda a finalização de todas as *promises* do *array*. O método `all`, então, finaliza com sucesso, gerando um *array* com os valores dos resultados. Se qualquer uma das *promises* do *array* falhar, a *promise* retornada pelo `all` também falha, recebendo o valor de falha da *promise* que falhou inicialmente.

Tente implementar algo parecido com isso usando uma função normal chamada `all`.

Observe que depois que a *promise* é resolvida (obtendo sucesso ou falhado), ela não pode ter sucesso ou falhar novamente, ignorando as chamadas às funções posteriores que tentam resolvê-la. Isso pode facilitar a maneira que você manipula as falhas em sua *promise*.

```
function all(promises) {
  return new Promise(function(success, fail) {
    // Your code here.
  });
}

// Test code.
all([]).then(function(array) {
  console.log("This should be []:", array);
});
function soon(val) {
  return new Promise(function(success) {
    setTimeout(function() { success(val); },
      Math.random() * 500);
  });
}
all([soon(1), soon(2), soon(3)]).then(function(array) {
  console.log("This should be [1, 2, 3]:", array);
});
function fail() {
  return new Promise(function(success, fail) {
    fail(new Error("boom"));
  });
}
all([soon(1), fail(), soon(3)]).then(function(array) {
  console.log("We should not get here");
}, function(error) {
  if (error.message !== "boom")
    console.log("Unexpected failure:", error);
});
```

Dicas:

A função passada ao construtor `Promise` terá que chamar o método `then` para cada uma das *promises* do *array* fornecido. Quando uma delas obtiver sucesso, duas coisas precisam acontecer. O valor resultante precisa ser armazenado na posição correta em um *array* de resultados e devemos, também, verificar se essa foi a última

promise pendente, finalizando nossa própria *promise* caso tenha sido.

O último pode ser feito usando um contador, que é inicializado com o valor do tamanho do *array* fornecido e do qual subtraímos uma unidade cada vez que uma *promise* for bem-sucedida. No momento em que o contador chega ao valor zero, terminamos. Certifique-se de lidar com a situação em que o *array* fornecido é vazio e, consequentemente, nenhuma *promise* será resolvida.

Tratar falhas requer um pouco mais de esforço, mas acaba sendo extremamente simples. Basta passar, para cada *promise* do *array*, a função que lida com a falha da *promise* responsável por encapsular as *promises* do *array*, de forma que uma falha em qualquer uma delas, irá disparar a falha para a *promise* encapsuladora.

Capítulo 18

Formulários e Campos de Formulários

Em inglês:

"I shall this very day, at Doctor's feast, My bounden service duly pay thee. But one thing! — For insurance' sake, I pray thee, Grant me a line or two, at least. Mephistopheles, in Goethe's Faust"

— Mephistopheles, in Goethe's Faust

Formulários e Campos de Formulário

Formulários foram introduzidos brevemente no [capítulo anterior](#) como uma forma de apresentar informações fornecidas pelo usuário através do HTTP. Eles foram projetados para uma web pré-javascript, atribuindo essa interação com o servidor sempre fazendo a navegação para uma nova página.

Mas os seus elementos são parte do DOM como o resto da página, e os elementos DOM representam campos de formulários suportados um número de propriedades e eventos que não são presente em outros elementos. Esses tornam possíveis para inspecionar e controlar os campos *inputs* com programas JavaScript e fazem coisas como adicionar funcionalidades para um formulário tradicional ou usam formulários e campos como blocos de construção em aplicações JavaScript.

Campos

Um formulário web consiste em qualquer número de campos `input` agrupados em uma tag `<form>`. O HTML permite que um número de diferentes estilos de campos, que vão desde simples on/off checkboxes para drop-down menus e campos de texto. Este livro não vai tentar discutir de forma abrangente todos os tipos de campos, mas nós podemos iniciar com a visão geral.

Muitos tipos de campos usam a tag `<input>`. Essa tag é um tipo de atributo usado para estilos do campo. Estes são alguns tipos de `<input>` comumente usados:

text	Um campo de texto em uma única linha
password	O mesmo que o campo de texto, mas esconde o texto que é digitado
checkbox	Um modificador on/off
radio	Campo de múltipla escolha
file	Permite que o usuário escolha um arquivo de seu computador

Os campos de formulários não aparecem necessariamente em uma tag `<form>`. Você pode colocá-los em qualquer lugar. Esses campos não podem ser apresentados (somente em um formulário como um todo), mas ao retornar para o *input* com JavaScript, muitas vezes não querem submeter campos de qualquer maneira.

```
<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="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" checked> (file)</p>
```

A interface JavaScript de tais elementos se diferem com o tipo dos elementos. Nós vamos passar por cima de cada um deles no final do capítulo.

Campos de texto de várias linhas têm a sua própria tag `<textarea>`, principalmente porque quando é usando um atributo para especificar um valor de várias linhas poderia ser inicialmente estranho. A tag `<textarea>` requer um texto que é usado entre as duas tags `</textarea>`, ao invés de utilizar o texto no atributo `value`.

```
<textarea>
  um
  dois
  três
</textarea>
```

Sempre que o valor de um campo de formulário é modificado, ele dispara um evento "change".

Focus

Diferentemente da maioria dos elementos em um documento HTML, campos de formulário podem obter o foco do teclado. Quando clicado, ou ativado de alguma outra forma, eles se tornam o elemento ativo no momento, o principal destinatário de entrada do teclado.

Se o documento tem um campo de texto, o campo é focado quando texto é digitado. Outros campos respondem diferente ao evento de teclado. Por exemplo, um menu `<select>` vai para uma opção que contém o texto que o usuário digitou e responde às teclas de seta, movendo sua seleção para cima e para baixo.

Podemos controlar focus do JavaScript com os métodos *focus* e *blur*. O primeiro modifica o foco do elemento que é chamado no DOM, e do segundo remove o focus. O valor no `document.activeElement` corresponde atualmente ao elemento focado.

```
<input type="text">
```

```
document.querySelector("input").focus();
console.log(document.activeElement.tagName);
// → INPUT
document.querySelector("input").blur();
console.log(document.activeElement.tagName);
// → BODY
```

Para algumas páginas, espera-se que o usuário interaja com um campo de formulário imediatamente. JavaScript pode ser usado para dar um focus nesse campo quando o documento é carregado, mas o HTML também fornece o atributo `autofocus`, que faz o mesmo efeito, mas permite que o navegador saiba o que estamos tentando realizar. Isso faz com que seja possível o navegador desativar o comportamento quando não é o caso, por exemplo, quando o usuário tem focado em outra coisa.

```
<input type="text" autofocus>
```

Navegadores tradicionais também permitem que o usuário mova o foco através do documento pressionando a tecla [Tab]. Nós podemos influenciar a ordem na qual os elementos recebem o focus com o atributo `tabindex`. Seguindo o exemplo do documento vai pular o foco do input text para o botão OK em vez de passar em primeiro pelo link de help.

```
<input type="text" tabindex=1> <a href=".">(help)</a>  
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

Por padrão, a maioria dos tipos de elementos HTML não podem ser focado. Mas você pode adicionar um atributo `tabindex` a qualquer elemento, o que tornará focalizável.

Campos desativados

Todos o campos dos formulários podem ser desabilitados por meio do seu atributo `disabled`, que também existe como uma propriedade no elemento do objeto DOM.

```
<button>I'm all right</button>  
<button disabled>I'm out</button>
```

Campos desabilitados não podem ser focalizados ou alterados, e ao contrário de campos ativos, eles ficam cinza e desbotado.

Quando um programa está sendo processado quando uma ação causada por algum botão ou outro controle, que poderia requerer comunicação com o servidor e assim levar um tempo, pode ser uma boa ideia para desabilitar o controle até que a ação termine. Dessa forma, quando o usuário fica impaciente e clica novamente, eles não vão acidentalmente repetir a sua ação.

Formulários Como um Todo

Quando o campo é contido em um elemento `<form>`, elemento DOM deve estar em uma propriedade `<form>` que faz a ligação por trás do formulário com o elemento DOM. O elemento `<form>`, tem uma propriedade chamada *elements* que contém uma coleção de array-like dos campos internos dentro dele.

O atributo `name` de um campo de formulário determina como seu valor será identificado quando o formulário é enviado. Ele também pode ser usado como um nome de propriedade quando acessar *elements* de propriedades do formulário, que atua tanto como um objeto array-like (acessível pelo número) e um map (acessível pelo nome).

```
<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>
```

```
var 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
```

Um botão com um atributo `type` do submit, quando pressionado, faz com que o formulário seja enviado. Pressionando Enter quando um campo de formulário é focado tem alguns efeitos.

O envio de um formulário normalmente significa que o navegador navega para a página indicada pelo atributo `action`, utilizando uma requisição *GET* ou *POST*. Mas Antes que isso aconteça, um evento "submit" é disparado. Esse evento pode ser manipulado pelo JavaScript, e o manipulador pode impedir o comportamento padrão chamando `preventDefault` no objeto evento.

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
```

```
var form = document.querySelector("form");
form.addEventListener("submit", function(event) {
  console.log("Saving value", form.elements.value.value);
  event.preventDefault();
});
```

Interceptar eventos *submit* em JavaScript tem vários usos. Podemos escrever código para verificar se o valores que o usuário digitou faz sentido imediatamente mostrar uma mensagem de erro, em vez de enviar o formulário. Ou nós podemos desabilitar o modo regular de enviar o formulário por completo, como no exemplo anterior, temos o nosso programa que manipula o `input`, possivelmente usando *XMLHttpRequest* para enviá-lo para um servidor sem recarregar a página.

Campos de Texto

Campos criados pela tag `<input>` com um tipo de `text` ou `password`, bem como uma tag `textarea`, compartilha uma interface comum. Seus elementos DOM tem uma propriedade de valor que mantém o seu conteúdo atual como um valor de string. A definição dessa propriedade para outra sequência altera o conteúdo dos campos.

As propriedades `selectionEnd` e `selectionStart` de campos de texto nos dão informações sobre o curso e seleção do texto. Quando não temos nada selecionado, estas duas propriedades tem o mesmo número o que indica a posição do cursor. Por exemplo, 0 indica o início do texto, e 10 indica o curso está após o décimo caractere. Quando uma parte do campo é selecionada as duas propriedades serão diferentes, nos dando o final e início do texto selecionado. Essas propriedades também podem ser gravadas como valores.

Como exemplo, imagine que você está escrevendo um artigo sobre Khasekhemwy, mas tem alguns problemas para soletrar o seu nome. As seguintes linhas de código até a tag `<textarea>` com um manipulador de eventos que, quando você pressionar F2, a string "Khasekhemwy" é inserida para você.

```
< textarea > < / textarea >
```

```
var textarea = document.querySelector("textarea");
textarea.addEventListener("keydown", function(event) {
  // The key code for F2 happens to be 113
  if (event.keyCode == 113) {
    replaceSelection(textarea, "Khasekhemwy");
    event.preventDefault();
  }
});
function replaceSelection(field, word) {
  var from = field.selectionStart, to = field.selectionEnd;
  field.value = field.value.slice(0, from) + word +
    field.value.slice(to);
  // Put the cursor after the word
  field.selectionStart = field.selectionEnd =
    from + word.length;
}
```

A função `replaceSelection` substitui a parte selecionada de um campo de texto com a palavra dada e em seguida, move o cursor depois que a palavra de modo que o usuário pode continuar a escrever.

O evento altera um campo de texto e não dispara cada vez que algo é digitado. Em vez disso, ele é acionado quando o campo perde o foco após o seu conteúdo foi alterado. Para responder imediatamente a mudanças em um campo de texto você deve registrar um manipulador para o evento "input" em vez disso, que é acionado para cada vez que o usuário digitar um caractere, exclui do texto, ou de outra forma manipula o conteúdo do campo.

O exemplo a seguir mostra um campo de texto e um contador que mostra o comprimento atual do texto inserido:

```
< input type="text" > length: < span id="length" >0< / span >
```

```
var text = document.querySelector("input");
var output = document.querySelector("#length");
text.addEventListener("input", function() {
    output.textContent = text.value.length;
});
```

Checkboxes e radio buttons

Um Checkbox é uma alternância binária simples. Seu valor pode ser extraído ou alterado por meio de sua propriedade checked, que tem um valor booleano.

```
<input type="checkbox" id="purple"> <label for="purple">Make this page purple</label>
```

```
var checkbox = document.querySelector("#purple");
checkbox.addEventListener("change", function() {
    document.body.style.background =
checkbox.checked ? "mediumpurple" : "";
});
```

A tag `<label>` é usada para associar uma parte de texto com um campo de entrada. Seu atributo deverá acessar o id do campo. Clicando no label irá ativar o campo, que se concentra nele e altera o seu valor quando é um checkbox ou button radio.

Um radio button é semelhante a um checkbox, mas está implicitamente ligado a outros radio buttons com o mesmo atributo de nome, de modo que apenas um deles pode estar ativo a qualquer momento.

Color:

```
<input type="radio" name="color" value="mediumpurple"> Purple
<input type="radio" name="color" value="lightgreen"> Green
<input type="radio" name="color" value="lightblue"> Blue
```

```
var buttons = document.getElementsByName("color");
function setColor(event) {
    document.body.style.background = event.target.value;
}
for (var i = 0; i < buttons.length; i++)
    buttons[i].addEventListener("change", setColor);
```

O método `document.getElementsByName` nos dá todos os elementos com um determinado atributo name. O exemplo faz um loop sobre aqueles (com um loop regular for , não forEach, porque a coleção retornada não é uma matriz real) e registra um manipulador de eventos para cada elemento. Lembre-se que os eventos de objetos tem uma propriedade target referindo-se ao elemento que disparou o evento. Isso é muitas vezes útil para manipuladores de eventos como este, que será chamado em diferentes elementos e precisa de alguma forma de acessar o target atual.

Campos Select

Os campos select são conceitualmente similares aos radio buttons, eles também permitem que o usuário escolha a partir de um conjunto de opções. Mas onde um botão de opção coloca a disposição das opções sob o nosso controle, a aparência de uma tag `<select>` é determinada pelo browser.

Campos select também têm uma variante que é mais parecido com uma lista de checkboxes, em vez de radio boxes. Quando dado o atributo múltiplo, um `<select>` tag vai permitir que o usuário selecione qualquer número de opções, em vez de apenas uma única opção.

```
<select multiple>
  <option>Pancakes</option>
  <option>Pudding</option>
  <option>Ice cream</option>
</select>
```

Isto, na maioria dos navegadores, mostra-se diferente do que um campo select não-múltiplo, que é comumente desenhado como um controle *drop-down* que mostra as opções somente quando você abrir.

O atributo `size` da tag `<select>` é usada para definir o número de opções que são visíveis ao mesmo tempo, o que lhe dá o controle sobre a aparência do *drop-down*. Por exemplo, definir o atributo `size` para "3" fará com que o campo mostre três linhas, se ele tem a opção de `multiple` habilitado ou não.

Cada tag `<option>` tem um valor. Este valor pode ser definido com um atributo de `value`, mas quando isso não for dado, o texto dentro do `option` irá contar como o valor do `option`. O valor da propriedade de um elemento `<select>` reflete a opção selecionada no momento. Para um campo `multiple`, porém, esta propriedade não significa muito, uma vez que vai possuir o valor apenas uma das opções escolhidas no momento.

As tags `<option>` de um campo `<select>` pode ser acessada como um objeto de array-like através de opções propriedade do campo. Cada opção tem uma propriedade chamada `selected`, o que indica se essa opção for selecionada. A propriedade também pode ser escrita para marcar ou desmarcar uma opção.

O exemplo a seguir extrai os valores selecionados a partir de um campo de seleção múltiplo e as utiliza para compor um número binário de bits individuais. Segure Ctrl (ou Comand no Mac) para selecionar várias opções.

```
<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>
```

```
var select = document.querySelector("select");
var output = document.querySelector("#output");
select.addEventListener("change", function() {
  var number = 0;
  for (var i = 0; i < select.options.length; i++) {
    var option = select.options[i];
    if (option.selected)
      number += Number(option.value);
  }
  output.textContent = number;
});
```

Campo Arquivo (Campo File)

Os campos de arquivo - (file), foram originalmente concebidos como uma maneira de fazer upload de arquivos de uma máquina do navegador através de um formulário. Em navegadores modernos, eles também fornecem uma maneira de ler esses arquivos a partir de programas de JavaScript. O campo atua como uma forma de porteiro). O script não pode simplesmente começar a ler arquivos privados do computador do usuário, mas se o usuário seleciona um arquivo em tal campo, o navegador interpreta que a ação no sentido de que o script pode ler o arquivo.

Um campo *file* geralmente parece um botão rotulado com algo como "escolha o arquivo" ou "procurar", com informações sobre o arquivo escolhido ao lado dele.

```
<input type="file">
```

```
var input = document.querySelector("input");
input.addEventListener("change", function() {
  if (input.files.length > 0) {
    var file = input.files[0];
    console.log("You chose", file.name);
    if (file.type)
      console.log("It has type", file.type);
  }
});
```

A propriedade `files` de um elemento campo file é um objeto de array-like (novamente, não um array autêntico) que contém os arquivos escolhidos no campo. É inicialmente vazio. A razão não é simplesmente uma propriedade de arquivo é que os campos file também suportam um atributo múltiplo, o que torna possível selecionar vários arquivos ao mesmo tempo.

Objetos na propriedade files têm propriedades como name (o nome do arquivo), size (o tamanho do arquivo em bytes), e type (o tipo de mídia do arquivo, como text/plain ou image/jpeg).

O que ele não tem é uma propriedade que contém o conteúdo do arquivo. Como é um pouco mais complicado. Desde a leitura de um arquivo do disco pode levar tempo, a interface terá de ser assíncrona para evitar o congelamento do documento. Você pode pensar o construtor *FileReader* como sendo semelhante a *XMLHttpRequest*, mas para arquivos.

```
<input type="file" multiple>
```

```
var input = document.querySelector("input");
input.addEventListener("change", function() {
  Array.prototype.forEach.call(input.files, function(file) {
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      console.log("File", file.name, "starts with",
        reader.result.slice(0, 20));
    });
    reader.readAsText(file);
  });
});
```

A leitura de um arquivo é feita através da criação de um objeto *FileReader*, registrando um manipulador de eventos "load" para ele, e chamando seu método *readAsText*, dando-lhe o arquivo para leitura. Uma vez finalizado o carregamento, a propriedade de leitura *result* tem o conteúdo do arquivo. O exemplo usa `Array.prototype.forEach` para iterar o array, uma vez em um loop (laço) normal, seria estranho obter os objetos *file* e *read* a partir de um manipulador de eventos. As variáveis poderiam compartilhar todas as iterações do loop.

FileReaders também aciona um evento "error" ao ver o arquivo falhar por algum motivo. O próprio objeto de erro vai acabar na propriedade de "error" de leitura. Se você não quer lembrar dos detalhes de mais uma interface assíncrona inconsistente, você pode envolvê-lo em uma *Promise* (ver [Capítulo 17](#)) como este:

```
function readFile(file) {
  return new Promise(function(succeed, fail) {
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      succeed(reader.result);
    });
    reader.addEventListener("error", function() {
      fail(reader.error);
    });
    reader.readAsText(file);
  });
}
```

É possível ler apenas parte de um arquivo chamando *slice* sobre ele e passando o resultado (uma chamada de um objeto *blob*) para o leitor de arquivos.

Armazenamento de dados Cliente-side

Páginas simples em HTML com um pouco de JavaScript pode ser um ótimo meio para "mini aplicativos" programas auxiliares -Pequenos ajudantes que automatizam coisas cotidianas. Ao ligar alguns campos de formulário com os manipuladores de eventos, você pode fazer qualquer coisa, desde a conversão entre graus Celsius e Fahrenheit às senhas de computação de uma senha mestra e um nome de site.

Quando tal aplicativo precisa lembrar-se de algo entre as sessões, você não pode usar variáveis JavaScript uma vez que aqueles são jogados fora a cada vez que uma página é fechada. Você pode configurar um servidor, conectar-se à Internet, e ter o seu aplicativo de armazenamento de alguma coisa lá. Vamos ver como fazer isso no capítulo 20. Mas isso adiciona um monte de trabalho extra e complexidade. Às vezes é suficiente apenas para manter os dados no navegador. Mas como?

Você pode armazenar dados de string data de uma forma que ainda continue ao recarregar a página, colocando-o no objeto *localStorage*. Este objeto permite-lhe apresentar valores de strings sob nomes (também strings), como neste exemplo:

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

Um valor em *localStorage* continua na página até que seja substituído, ele é removido com *removeItem*, ou o usuário apaga seus dados locais.

Sites de domínios diferentes obtém diferentes espaços de armazenamento. Isso significa que os dados armazenados em *localStorage* por um determinado site pode, a princípio, ser lido (e sobrescritos) por scripts desse mesmo site.

Os navegadores também impor um limite para o tamanho dos dados de um site pode armazenar em *localStorage*, tipicamente na ordem de poucos megabytes. Esta restrição, juntamente com o fato de encher os discos rígidos das pessoas com lixo não é realmente viável, impede esse recurso de ocupar muito espaço.

O código a seguir implementa uma simples aplicação de anotações. Ele mantém notas do usuário como um objeto, associando títulos de notas com strings de conteúdo. Este objeto é codificado como JSON e armazenados em *localStorage*. O usuário pode selecionar uma nota de um campo `<select>` e mudar o texto da nota em um

`<textarea>` . A nota pode ser adicionado clicando em um botão.

Notes:

```
<select id="list"></select>
<button onclick="addNote()">new</button><br>
<textarea id="currentnote" style="width: 100%; height: 10em">
</textarea>
```

```
var list = document.querySelector("#list");
function addToList(name) {
    var option = document.createElement("option");
    option.textContent = name;
    list.appendChild(option);
}

// Initialize the list from localStorage
var notes = JSON.parse(localStorage.getItem("notes")) ||
    {"shopping list": ""};
for (var name in notes)
    if (notes.hasOwnProperty(name))
        addToList(name);

function saveToStorage() {
    localStorage.setItem("notes", JSON.stringify(notes));
}

var current = document.querySelector("#currentnote");
current.value = notes[list.value];

list.addEventListener("change", function() {
    current.value = notes[list.value];
});
current.addEventListener("change", function() {
    notes[list.value] = current.value;
    saveToStorage();
});

function addNote() {
    var name = prompt("Note name", "");
    if (!name) return;
    if (!notes.hasOwnProperty(name)) {
        notes[name] = "";
        addToList(name);
        saveToStorage();
    }
    list.value = name;
    current.value = notes[name];
}
```

O script inicializa a variável `notes` para o valor armazenado em `localStorage` ou um valor que não existe, para objeto simples `notes` "shopping list" vazio . A leitura de um campo que não existe de `localStorage` será nulo. Passando nulo para `JSON.parse` irá analisar uma string "null" e retornar *null*. Assim, o operador `||` pode ser utilizada para fornecer um valor *default* de uma situação como esta.

Sempre que as alterações de dados de notas (quando uma nova nota é adicionado ou uma nota existente é modificada), a função `saveToStorage` é chamado para atualizar o campo de armazenamento. Se esta aplicação foi destinado a lidar com milhares de notas, em vez de muitos, isso seria muito caro, e nós teríamos que chegar a uma maneira mais complicada para armazená-los, como dar cada nota de seu próprio campo de armazenamento.

Quando o usuário adiciona uma nova nota, o código deve atualizar o campo de texto explicitamente, mesmo que o campo `<select>` tenha um manipulador de "change" que faz a mesma coisa. Isso é necessário porque o eventos "change" disparam apenas quando o usuário altera o valor do campo, e não quando um script executa.

Há um outro objeto semelhante para `LocalStorage` chamado `sessionStorage`. A diferença entre as duas é que o conteúdo de `sessionStorage` é esquecido no fim de cada sessão, o que para a maioria dos navegadores significa quando o navegador é fechado.

Sumário

HTML pode expressar vários tipos de campos de formulário, tais como text fields, checkboxes, campos multiple-choice, e file pickers.

Esses campos podem ser inspecionados e manipulados com JavaScript. Eles acionam o evento "change" quando alterado, o evento "input" quando o texto é digitado, e vários eventos de teclado. Estes eventos permitem-nos a perceber quando o usuário está interagindo com os campos. Propriedades como `value` (para texto e seleção campos) ou `checked` (para checkboxes e radio buttons) são usados para ler ou definir o conteúdo do campo.

Quando um formulário é enviado, o evento "submit" dispara. Um manipulador de JavaScript pode chamar `preventDefault` para impedir que que dispare o evento submit. Elementos de campo de formulário não precisam ser envolvidos em tags `<form>`.

Quando o usuário tenha selecionado um campo de seu sistema de arquivos local em um campo picker field, a interface `FileReader` pode ser usado para acessar o conteúdo deste arquivo a partir de um programa de JavaScript.

Os objetos `LocalStorage` e `sessionStorage` pode ser usado para guardar informações de uma forma que continue mesmo recarregando a página. O primeiro salva os dados para sempre (ou até que o usuário decida limpá-la), e o segundo salvá-lo até que o navegador é fechado.

Exercícios

A JavaScript workbench

Construa uma interface que permite que as pessoas a digitem e executem partes do código JavaScript.

Coloque um botão ao lado de um campo `<textarea>`, ao ser pressionado, usa o construtor `Function` vimos no Capítulo 10 para dividir o texto em uma função e chamá-lo. Converter o valor de retorno da função, ou qualquer erro que é elevado, em uma string e exibi-lo depois de o campo de texto.

```
<textarea id="code">return "hi";</textarea>
<button id="button">Run</button>
<pre id="output"></pre>
```

```
<script>
  // Your code here.
</script>
```

Use `document.querySelector` ou `document.getElementById` para ter acesso aos elementos definidos em seu HTML. Um manipulador de eventos para o "click" ou eventos no botão "mousedown" pode ter a propriedade `value` do campo de texto e chamada `new Function` nele.

Certifique-se de envolver tanto a chamada para a `new function` e a chamada para o seu resultado em um bloco `try` para que você possa capturar exceções que ela produz. Neste caso, nós realmente não sabemos que tipo de exceção que estamos procurando, então pegar tudo.

A propriedade `textContent` do elemento de saída pode ser usada para preenchê-lo com uma mensagem de string. Ou, se você quiser manter o conteúdo antigo ao redor, criar um novo nó de texto usando `document.createTextNode` e anexá-lo ao elemento. Lembre-se de adicionar um caractere de nova linha até o fim, de modo que nem todas as saídas apareçam em uma única linha.

Autocompletion

Estender um campo de texto para quando o usuário digitar uma lista de valores sugeridos é mostrado abaixo do campo. Você tem um conjunto de possíveis valores disponíveis e deve mostrar aqueles que começam com o texto que foi digitado. Quando uma sugestão é clicada, substitua o valor atual do campo de texto com ele.

```
<input type="text" id="field">
<div id="suggestions" style="cursor: pointer"></div>
```

```
// Builds up an array with global variable names, like
// 'alert', 'document', and 'scrollTo'
var terms = [];
for (var name in window)
    terms.push(name);

// Your code here.
```

O melhor evento para a atualização da lista de sugestões é " `input` ", uma vez que será acionado imediatamente quando o conteúdo do campo é alterado.

Em seguida, um loop por meio do array de termos e ver se eles começam com a string dada. Por exemplo, você poderia chamar `indexOf` e ver se o resultado é zero. Para cada sequência correspondente, adicionar um elemento para as sugestões `<div>` . Você deve, provavelmente, cada vez que você inicia começar vazio e atualizar as sugestões, por exemplo, definindo sua `textContent` para a string vazia.

Você poderia adicionar um manipulador de evento " `click` " [para cada elemento ou adicionar um único para fora `<div>` que prendê-los e olhar para a propriedade `target` do evento para descobrir qual sugestão foi clicada.]

Para obter o texto sugestão de um nó DOM, você pode olhar para a sua `textContent` ou definir um atributo para armazenar explicitamente o texto quando você cria o elemento.

Conway's Game of Life

Jogo da Vida de Conway é uma simulação simples que cria a "vida" artificial em uma grade, cada célula, que é ao vivo ou não. Cada geração (virar), as seguintes regras são aplicadas:

Qualquer célula viva com menos de dois ou mais de três vizinhos vivos morre.

Qualquer célula viva com dois ou três vizinhos vivos vive para a próxima geração.

Qualquer célula morta com exatamente três vizinhos vivos se torna um ce ao vivo

Um vizinho é definido como qualquer célula adjacente, inclusive na diagonal adjacentes.

Nota-se que estas regras são aplicadas a toda a rede de uma só vez, e não um quadrado de cada vez. Isso significa que a contagem de vizinhos baseia-se na situação no início da produção, e mudanças acontecendo com as células vizinhas durante esta geração não deve influenciar o novo estado de uma dada célula.

Implementar este jogo usando qualquer estrutura de dados que você ache mais apropriado. Use `Math.random` para preencher a grade com um padrão aleatório inicialmente. Exibi-lo como uma grade de campos de checkboxes, com um botão ao lado dele para avançar para a próxima geração. Quando os controles de utilizador ou desmarca as checkboxes, as alterações devem ser incluídos no cálculo a próxima geração.

```
<div id="grid"></div>
<button id="next">Next generation</button>
```

```
<script>
  // Your code here.
</script>
```

Para resolver o problema de ter conceitualmente as alterações ocorram ao mesmo tempo, tente ver o cálculo de uma geração como uma função pura, que tem uma grelha e produz uma nova grade que representa a curva seguinte.

Representando a grade pode ser feito em qualquer das formas mostradas nos capítulos 7 e 15. Contando vizinhos vivos podem ser feitas com dois loops aninhados, percorrer coordenadas adjacentes. Tome cuidado para não contar as células fora do campo e ignorar o celular no centro, cujos vizinhos estamos contando.

Fazer alterações em check-boxes em vigor na próxima geração pode ser feito de duas maneiras. Um manipulador de eventos pode perceber essas alterações e atualizar a grade atual para refleti-los, ou você poderia gerar uma nova grade a partir dos valores nas caixas de seleção antes de calcular o próximo turno.

Se você optar utilizar manipuladores de eventos, você pode querer anexar atributos que identificam a posição que cada caixa corresponde ao modo que é fácil descobrir qual célula de mudar.

Para desenhar a rede de caixas de seleção, você ou pode usar um elemento `<table>` (olhe o [Capítulo 13](#)) ou simplesmente colocá-los todos no mesmo elemento e colocar `
` (quebra de linha) elementos entre as linhas.

Um programa de pintura

"Eu olho para muitas cores antes de mim. Eu olho para minha tela em branco. Então, eu tento aplicar cores como palavras que moldam poemas, como as notas que formam a música." Joan Miro

O material do capítulo anterior dá para você todo os elementos que você precisa para construir uma aplicação web simples. Nesse capítulo, vamos fazer exatamente isso.

Nosso aplicativo será um programa de desenho baseado na web, aos moldes do Microsoft Paint. Você poderá usá-lo para abrir arquivos de imagem, rabiscar sobre ela com o mouse e salvá-la. Isso é como vai se parecer:



Pintar pelo computador é ótimo. Você não precisa se preocupar com materiais, habilidades ou talento. Você apenas precisa começar a manchar.

Implementação

A interface para o programa de pintura mostra um grande elemento `<canvas>` na parte superior, com os campos do formulário abaixo dele. O usuário desenha na imagem ao selecionar uma ferramenta de um campo `<select>` e em seguida clicando ou arrastando em toda a tela. Existem ferramentas para desenhar linhas, apagar partes da imagem, adicionar texto e assim por diante.

Clicando na tela, será delegado o evento `mousedown` para a ferramenta selecionada no momento, que poderá manipulá-lo em qualquer maneira que escolher. A ferramenta de desenhar linha, por exemplo, vai ouvir os eventos de `mousemove` até que o botão do mouse seja liberado e desenhará linhas através do caminho do mouse usando a cor atual e o tamanho do pincel.

Cor e o tamanho do pincel são selecionados com campos adicionais no formulário. Esses são ativados para atualizar a tela desenhando o conteúdo `fillStyle`, `strokeStyle` e `linewidth` toda hora que eles forem alterados.

Você pode carregar uma imagem no programa de duas formas. A primeira usa um campo de arquivo, onde o usuário pode selecionar um arquivo no seu computador. A segunda pede uma URL e vai pegar a imagem na internet.

Imagens são salvas em um lugar atípico. O link de salvar está no canto direito da imagem ao lado do tamanho do pincel. Ele pode ser seguido, compartilhado ou salvo. Eu vou explicar como isso é possível em um momento.

Construindo o DOM

A interface do nosso programa é criado a partir de mais de 30 elementos DOM. Nós precisamos construir esses de alguma maneira.

HTML é o formato mais óbvio para definir estrutura complexas do DOM. Mas, separando o programa em pedaços de HTML e um script é dificultada pelo fato de muitos elementos do DOM precisar de manipuladores de eventos ou ser tocado por outro script de alguma outra forma. Assim, nosso script precisa fazer muitas chamadas de `querySelector` (ou similar), afim de encontrar algum elemento DOM que ele precise para agir.

Seria bom se a estrutura DOM para cada parte da nossa interface fosse definida perto do código JavaScript que vai interagir com ela. Assim, eu escolhi por fazer toda a criação dos nós do DOM no JavaScript. Como nós vimos no [Capítulo 13](#), a interface integrada para a criação da estrutura DOM é terrivelmente verbosa. Se vamos fazer um monte de construções DOM, precisamos de uma função auxiliar.

Essa função auxiliar é uma função estendida da função `elt` a partir do [Capítulo 13](#). Ela cria um elemento com o nome e os atributos dado e acrescenta todos os argumentos que ela recebe como nós filho, automaticamente convertendo strings em nós de texto.

```
function elt(name, attributes) {
  var node = document.createElement(name);
  if (attributes) {
    for (var attr in attributes)
      if (attributes.hasOwnProperty(attr))
        node.setAttribute(attr, attributes[attr]);
  }
  for (var i = 2; i < arguments.length; i++) {
    var child = arguments[i];
    if (typeof child == "string")
      child = document.createTextNode(child);
    node.appendChild(child);
  }
  return node;
}
```

Isso nos permite criar elementos facilmente, sem fazer nosso código fonte tão longo e maçante quanto um contrato de usuário final corporativo.

A fundação

O núcleo do nosso programa é a função `createPaint`, que acrescenta a interface de pintura em um elemento do DOM que é dado como argumento. Porque nós queremos construir nosso programa pedaço por pedaço, definimos um objeto chamado `controls`, que realizará funções para inicializar vários controles abaixo da imagem.

```
var controls = Object.create(null);

function createPaint(parent) {
  var canvas = elt("canvas", {width: 500, height: 300});
  var cx = canvas.getContext("2d");
  var toolbar = elt("div", {class: "toolbar"});
  for (var name in controls)
    toolbar.appendChild(controls[name](cx));

  var panel = elt("div", {class: "picturepanel"}, canvas);
  parent.appendChild(elt("div", null, panel, toolbar));
}
```

Cada controle tem acesso ao contexto da tela de desenho e, através da propriedade `tela` desse contexto, para o elemento `<canvas>`. A maior parte do estado do programa vive nesta tela - que contém a imagem atual bem como a cor selecionada (em sua propriedade `fillStyle`) e o tamanho do pincel (em sua propriedade `lineWidth`).

Nós envolvemos a tela e os controles em elementos `<div>` com classes que seja possível adicionar um pouco de estilo, como uma borda cinza envolta da imagem.

Ferramenta de seleção

O primeiro controle que nós adicionamos é o elemento `<select>` que permite o usuário a selecionar uma ferramenta de desenho. Tal como com os controles, vamos usar um objeto para coletar as várias ferramentas de modo que não temos que codificá-las em um só lugar e nós podemos adicionar ferramentas novas mais tarde. Esse objeto associa os nomes das ferramentas com a função que deverá ser chamada quando ela for selecionada e quando for clicado na tela.

```

var tools = Object.create(null);

controls.tool = function(cx) {
  var select = elt("select");
  for (var name in tools)
    select.appendChild(elt("option", null, name));

  cx.canvas.addEventListener("mousedown", function(event) {
    if (event.which == 1) {
      tools[select.value](event, cx);
      event.preventDefault();
    }
  });

  return elt("span", null, "Tool: ", select);
};

```

O campo de ferramenta é preenchida com elementos `<option>` para todas as ferramentas que foram definidas e um manipulador `mousedown` no elemento `canvas` cuida de chamar a função para a ferramenta atual, passando tanto o objeto do evento quanto o contexto do desenho como argumentos. E também chama `preventDefault` para que segurando o botão do mouse e arrastando não cause uma seleção do navegador em qualquer parte da página.

A ferramenta mais básica é a ferramenta de linha, o qual permite o usuário a desenhar linhas com o mouse. Para colocar o final da linha no lugar certo, temos que ser capazes de encontrar as coordenadas relativas do `canvas` que um determinado evento do mouse corresponde. O método `getBoundingClientRect`, brevemente mencionado no [Capítulo 13](#), pode nos ajudar aqui. Nos diz que um elemento é exibido, relativo ao canto `top-left` da tela. As propriedades `clientX` e `clientY` dos eventos do mouse também são relativas a esse canto, então podemos subtrair o canto `top-left` da tela a partir deles para obter uma posição em relação a esse canto.

```

function relativePos(event, element) {
  var rect = element.getBoundingClientRect();
  return {x: Math.floor(event.clientX - rect.left),
    y: Math.floor(event.clientY - rect.top)};
}

```

Várias das ferramentas de desenho precisam ouvir os eventos `mousemove` até que o botão do mouse mantiver pressionado. A função `trackDrag` cuida de registrar e cancelar o registro de eventos para essas situações.

```

function trackDrag(onMove, onEnd) {
  function end(event) {
    removeEventListener("mousemove", onMove);
    removeEventListener("mouseup", end);
    if (onEnd)
      onEnd(event);
  }
  addEventListener("mousemove", onMove);
  addEventListener("mouseup", end);
}

```

Essa função recebe dois argumentos. Um é a função para chamar a cada evento `mousemove` e o outro é uma função para chamar quando o botão do mouse deixa de ser pressionado. Qualquer argumento pode ser omitido quando não for necessário.

A ferramenta de linha usa esses dois métodos auxiliares para fazer o desenho real.

```

tools.Line = function(event, cx, onEnd) {
  cx.lineCap = "round";

  var pos = relativePos(event, cx.canvas);
  trackDrag(function(event) {

```

```

    cx.beginPath();
    cx.moveTo(pos.x, pos.y);
    pos = relativePos(event, cx.canvas);
    cx.lineTo(pos.x, pos.y);
    cx.stroke();
  }, onEnd);
};

```

A função inicia por definir a propriedade do contexto de desenho `lineCap` para `round`, que faz com que ambas as extremidades de um caminho traçado, fique arredondada e não na forma padrão quadrada. Esse é o truque para se certificar de que várias linhas separadas, desenhadas em resposta a eventos separados, pareçam a mesma, coerente. Com larguras maiores de linhas, você vai ver as lacunas nos cantos se você usar as linhas planas padrão.

Então, para cada evento `mousemove` que ocorre enquanto o botão do mouse está pressionado, um simples segmento de linha entre o botão do mouse apertado e a nova posição, usando qualquer `strokeStyle` e `lineWidth` começa a ser desenhado no momento.

O argumento `onEnd` para a ferramenta de linha é simplesmente passado através do `trackDrag`. O caminho normal para executar as ferramentas não vai passar o terceiro argumento, portando, quando usar a ferramenta linha, esse argumento mantém `undefined` e nada acontece no final do movimento do mouse. O argumento está lá para nos permitir implementar a ferramenta de apagar na parte superior da ferramenta da linha com muito pouco código adicional.

```

tools.Erase = function(event, cx) {
  cx.globalCompositeOperation = "destination-out";
  tools.Line(event, cx, function() {
    cx.globalCompositeOperation = "source-over";
  });
};

```

A propriedade `globalCompositeOperation` influencia o modo como as operações de desenhar em uma tela de desenho altera a cor dos pixels que tocam. Por padrão, o valor da propriedade é `source-over`, o que significa que a cor do desenho é sobreposta sobre a cor já existente naquele ponto. Se a cor é opaca, vai apenas substituir a cor antiga, mas, se é parcialmente transparente, as duas serão misturadas.

A ferramenta apagar define `globalCompositeOperation` para `destination-out`, que tem o efeito de apagar os pixels que tocamos, tornando-os transparentes novamente.

Isso nos dá duas ferramentas para nosso programa de pintura. Nós podemos desenhar linhas pretas em um único pixel de largura (o padrão `strokeStyle` e `lineWidth` para uma tela) e apagá-los novamente. É um trabalho, mesmo que ainda bastante limitado, programa de pintura.

Cor e tamanho do pincel

Partindo do princípio que os usuários vão querer desenhar em cores diferentes do preto e usar tamanhos diferentes de pincéis, vamos adicionar controles para essas duas definições.

No [Capítulo 8](#), eu discuti um número de diferentes campos de formulário. Campo de cor não estava entre aqueles. Tradicionalmente, os navegadores não tem suporte embutido para seletores de cores, mas nos últimos anos, uma série de tipos de campos foram padronizados. Uma delas é `<input type='color'>`. Outros incluem `date`, `email`, `url` e `number`. Nem todos os navegadores suportam eles ainda - no momento da escrita, nenhuma versão do Internet Explorer suporta campos de cor. O tipo padrão de uma tag `<input>` é `text`, e quando um tipo não suportado é usado, os navegadores irão tratá-lo como um campo de texto. Isso significa que usuários do Internet Explorer executando o nosso programa de pintura vão ter que digitar o nome da cor que quiser, ao invés de selecioná-la a partir de um componente conveniente.

```
controls.color = function(cx) {
  var input = elt("input", {type: "color"});
  input.addEventListener("change", function() {
    cx.fillStyle = input.value;
    cx.strokeStyle = input.value;
  });
  return elt("span", null, "Color: ", input);
};
```

Sempre que o valor do campo cor muda, o `fillStyle` e `strokeStyle` do contexto são atualizados para manter o novo.

O campo para configurar o tamanho do pincel funciona de forma semelhante.

```
controls.brushSize = function(cx) {
  var select = elt("select");
  var sizes = [1, 2, 3, 5, 8, 12, 25, 35, 50, 75, 100];
  sizes.forEach(function(size) {
    select.appendChild(elt("option", {value: size},
      size + " pixels"));
  });
  select.addEventListener("change", function() {
    cx.lineWidth = select.value;
  });
  return elt("span", null, "Brush size: ", select);
};
```

O código gera opções a partir de uma `array` de tamanhos de pincel e, novamente, garante que o `lineWidth` seja atualizado quando um tamanho de pincel é escolhido.

Salvando

Para explicar a implementação do link salvar, eu preciso falar sobre `data URLs`. Um `data URL` é uma URL com dados: como seu protocolo. Ao contrário de `http` : normal e `https` : URLs, URLs de dados não apontam para algum recurso mas sim, contém todo o recurso em si. Esta é uma URL de dados contendo um simples documento HTML.

```
data:text/html,<h1 style="color:red">Hello!</h1>
```

Essas URLs são úteis para várias tarefas, tais como a inclusão de pequenas imagens diretamente em um arquivo de folha de estilo. Eles também nos permitem linkar para arquivos que nós criamos no lado do cliente, no navegador, sem antes mover para algum servidor.

Elementos `canvas` tem um método conveniente, chamado `toDataURL`, que irá retornar a URL de dados que contém a imagem no `canvas` como um arquivo de imagem. Nós não queremos para atualizar nosso link de salvar toda vez que a imagem for alterada. Para imagens grandes, que envolve a transferência de um monte de dados em um link, seria visivelmente lento. Em vez disso, nós atualizaremos o atributo `href` do link sempre que o foco do teclado estiver sobre ele ou o mouse é movido sobre ele.

```
controls.save = function(cx) {
  var link = elt("a", {href: "/"}, "Save");
  function update() {
    try {
      link.href = cx.canvas.toDataURL();
    } catch (e) {
      if (e instanceof SecurityError)
        link.href = "javascript:alert(" +
          JSON.stringify("Can't save: " + e.toString()) + ")";
      else
        throw e;
    }
  }
}
```



```

    }
  }
  link.addEventListener("mouseover", update);
  link.addEventListener("focus", update);
  return link;
};

```

Assim, o link fica calmamente ali sentado, apontando para a coisa errada, mas quando o usuário se aproxima, ele magicamente se atualiza para apontar para a imagem atual.

Se você carregar uma imagem grande, alguns navegadores vão ter problemas com as URLs de dados gigantes que essa produz. Para imagens pequenas, essa abordagem funciona sem problemas.

Mas aqui estamos, mais uma vez para correr para as sutilezas do navegador sandboxing. Quando uma imagem é carregada a partir de uma URL para outro domínio, se a resposta do servidor não incluir o header que diz ao navegador que o recurso pode ser usado para outro domínio (ver o [Capítulo 17](#)), a tela irá conter as informações que o usuário pode olhar, mas que o script não pode.

Podemos ter solicitado uma imagem que contenha informações privadas (por exemplo, um gráfico que mostre o saldo da conta bancária do usuário) usando a sessão do usuário. Se os scripts puderem obter as informações dessa imagem, eles podem espionar o usuário de formas indesejáveis.

Para evitar esse tipo vazamento de informações, os navegadores irá deixar a tela tão manchada quando se trata de uma imagem que o script não pode ver. Dados de pixel, incluindo URLs de dados, não podem ser extraídos de uma tela manchada. Você pode escrever para ele, mas você não pode mais ler.

É por isso que precisamos das instruções `try/catch` na função `update` para o link salvar. Quando o `canvas` ficou corrompido, chamando `toDataURL` irá lançar uma exceção que é uma instância do `SecurityError`. Quando isso acontece, nós definimos o link para apontar para outro tipo de URL, usando o `javascript:` protocolo. Esses links simplesmente executam o script dado após os dois pontos para que o link irá mostrar uma janela de `alert` informando o usuário do problema quando é clicado.

Carregando arquivos de imagem

Os dois controles finais são usados para carregar arquivos de imagens locais e a partir de URLs. Vamos precisar da seguinte função auxiliar, que tenta carregar um arquivo de imagem a partir de uma URL e substitui o conteúdo do `canvas` por ela.

```

function loadImageURL(cx, url) {
  var image = document.createElement("img");
  image.addEventListener("load", function() {
    var color = cx.fillStyle, size = cx.lineWidth;
    cx.canvas.width = image.width;
    cx.canvas.height = image.height;
    cx.drawImage(image, 0, 0);
    cx.fillStyle = color;
    cx.strokeStyle = color;
    cx.lineWidth = size;
  });
  image.src = url;
}

```

Nós queremos alterar o tamanho do `canvas` para ajustar precisamente a imagem. Por alguma razão, alterar o tamanho do `canvas` vai causar perda das configurações do contexto do desenho como `fillStyle` e `lineWidth`, então a função salva elas e restaura depois de ter atualizado o tamanho do `canvas`.

O controle para o carregamento de um arquivo local utiliza a técnica `FileReader` a partir do [Capítulo 18](#). Além do método `readAsText` que nós usamos lá, tais objetos de leitura também tem um método chamado `readAsDataURL`, que é exatamente o que precisamos aqui. Nós carregamos o arquivo que o usuário escolheu como URL de dados e passaremos para `loadImageURL` para colocá-lo no `canvas`.

```
controls.openFile = function(cx) {
  var input = elt("input", {type: "file"});
  input.addEventListener("change", function() {
    if (input.files.length == 0) return;
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      loadImageURL(cx, reader.result);
    });
    reader.readAsDataURL(input.files[0]);
  });
  return elt("div", null, "Open file: ", input);
};
```

Carregando um arquivo através de uma URL é ainda mais simples. Mas, com o campo de texto, é menos limpo quando o usuário termina de escrever a URL, por isso nós não podemos simplesmente ouvir pelos eventos `change`. Ao invés disso, vamos envolver o campo de um formulário e responder quando o formulário for enviado, seja porque o usuário pressionou `Enter` ou porque clicou no botão de carregar.

```
controls.openURL = function(cx) {
  var input = elt("input", {type: "text"});
  var form = elt("form", null,
    "Open URL: ", input,
    elt("button", {type: "submit"}, "load"));
  form.addEventListener("submit", function(event) {
    event.preventDefault();
    loadImageURL(cx, form.querySelector("input").value);
  });
  return form;
};
```

Nós temos agora definidos todos os controles que o nosso simples programa de pintura precisa, mas isso ainda poderia usar mais algumas ferramentas.

Finalizando

Nós podemos facilmente adicionar uma ferramenta de texto que peça para o usuário qual é o texto que deve desenhar.

```
tools.Text = function(event, cx) {
  var text = prompt("Text:", "");
  if (text) {
    var pos = relativePos(event, cx.canvas);
    cx.font = Math.max(7, cx.lineWidth) + "px sans-serif";
    cx.fillText(text, pos.x, pos.y);
  }
};
```

Você pode adicionar campos extras para o tamanho da fonte, mas para simplificar, sempre use uma fonte `sans-serif` e baseie o tamanho da fonte com o tamanho atual do pincel. O tamanho mínimo é de 7 pixels porque texto menor do que isso, é ilegível.

Outra ferramenta indispensável para a desenhos de computação amadores é a ferramenta tinta spray. Este desenha pontos em locais aleatórios sobre o pincel, desde que o mouse é pressionado, criando mais denso ou salpicado menos densa com base em quão rápido ou lento os movimentos do mouse são.

```
tools.Spray = function(event, cx) {
    var radius = cx.linewidth / 2;
    var area = radius * radius * Math.PI;
    var dotsPerTick = Math.ceil(area / 30);

    var currentPos = relativePos(event, cx.canvas);
    var spray = setInterval(function() {
        for (var i = 0; i < dotsPerTick; i++) {
            var offset = randomPointInRadius(radius);
            cx.fillRect(currentPos.x + offset.x,
                        currentPos.y + offset.y, 1, 1);
        }
    }, 25);
    trackDrag(function(event) {
        currentPos = relativePos(event, cx.canvas);
    }, function() {
        clearInterval(spray);
    });
};
```

A ferramenta spray usa `setInterval` para cuspir pontos coloridos a cada 25 milissegundos enquanto o botão do mouse é pressionado. A função `trackDrag` é usada para manter o `currentPos` apontando para a posição atual do mouse e para desligar o intervalo quando o mouse é liberado.

Para determinar quantos pontos serão desenhados a cada intervalo de cliques, a função calcula a área do pincel e divide por 30. Para encontrar uma posição aleatória sob o pincel, é usada a função `randomPointInRadius`.

```
function randomPointInRadius(radius) {
    for (;;) {
        var x = Math.random() * 2 - 1;
        var y = Math.random() * 2 - 1;
        if (x * x + y * y <= 1)
            return {x: x * radius, y: y * radius};
    }
}
```

A função gera pontos no quadrado entre (-1,-1) e (1,1). Usando o teorema de Pitágoras, ele testa se o ponto gerado encontra-se dentro de um círculo de raio 1. Logo que a função encontrar esse ponto, ele retorna o ponto multiplicado pelo argumento de raio.

O círculo é necessário para a distribuição uniforme dos pontos. A maneira simples de gerar um ponto aleatório dentro de um círculo seria a utilização de um ângulo e distância aleatória e chamar `Math.sin` e `Math.cos` para criar o ponto correspondente. Mas com esse método, os pontos são mais prováveis de aparecerem perto do centro do círculo. Existem outras maneiras de contornar isso, mas elas são mais complicadas do que o ciclo anterior.

Nós agora temos um programa de pintura funcionando. Execute o código abaixo para experimentá-lo.

```
<link rel="stylesheet" href="css/paint.css">

<body>
  <script>createPaint(document.body);</script>
</body>
```

Exercícios

Ainda há muito espaço para melhorias nesse programa. Vamos adicionar mais algumas funcionalidades como exercício.

Retângulos

Definir uma ferramenta chamada `Retângulo` que preenche um retângulo (veja o método `fillRect` a partir do [Capítulo 16](#)) com a cor atual. O retângulo deve espalhar a partir do ponto onde o usuário pressionar o botão do mouse para o ponto onde ele é liberado. Note-se que este último pode estar acima ou a esquerda do primeiro.

Uma vez que ele funcionar, você vai perceber que é um pouco chocante não ver o retângulo como você está arrastando o mouse para definir o seu tamanho. Você pode chegar a uma maneira de mostrar algum tipo de retângulo durante o movimento do mouse, sem realmente ir desenhando no `canvas` até que o botão do mouse seja liberado?

Se nada lhe vem a mente, lembre `position: absolute` discutido no [Capítulo 13](#), que pode ser usado para sobrepor um nó no resto do documento. As propriedades `pageX` e `pageY` de um evento de mouse pode ser usada para a posicionar um elemento precisamente sob o mouse, definindo os estilos `left`, `top`, `width` e `height` para os valores corretos de pixel.

```
<script>
  tools.Rectangle = function(event, cx) {
    // Your code here.
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Dicas

Você pode utilizar `relativePos` para encontrar o canto correspondente ao início do arrasto do mouse. Descobrir aonde as extremidades de arrasto termina pode ser com `trackDrag` ou registrando seu próprio manipulador de eventos.

Quando você tem dois cantos do retângulo, você deve de alguma forma traduzi-los em argumentos que o `fillRect` espera: O canto `top-left`, `width` e `height` do retângulo. `Math.min` pode ser usado para encontrar a coordenada mais a esquerda X e a coordenada superior Y. Para obter o `width` e `height`, você pode chamar `Math.abs` (o valor absoluto) sobre a diferença entre os dois lados.

Mostrando o retângulo durante o arrastar do mouse requer um conjunto semelhante de números, mas no contexto de toda a página em vez de em relação ao `canvas`. Considere escrever um `findRect`, que converte dois pontos em um objeto com propriedades `top`, `left`, `width` e `height`, de modo que você não tenha que escrever a mesma lógica duas vezes.

Você pode, então, criar uma `<div>` e definir seu `style.position` como `absolute`. Ao definir os estilos de posicionamento, não se esqueça de acrescentar `px` para os números. O nó deve ser adicionado ao documento (você pode acrescentá-la à `document.body`) e também remover novamente quando o arrasto do mouse terminar e o retângulo real for desenhado na tela.

Seletor de cores

Outra ferramenta que é comumente encontrada em programas gráficos é um seletor de cores, o que permite que o usuário clique na imagem e seleciona a cor sob o ponteiro do mouse. Construa este.

Para esta ferramenta, precisamos de uma maneira para acessar o conteúdo do `canvas`. O método `toDataURL` mais ou menos faz isso, mas recebendo informações de pixel para fora de tal URL de dados é difícil. Em vez disso, vamos usar o método `getImageData` no contexto do desenho, que retorna um pedaço retangular da imagem como um objeto com propriedades `width`, `height` e `data`. A propriedade `data` contém um array de números de 0 a 255, com quatro números para representar, componentes de cada pixel vermelho, verde, azul e opacidade.

Este exemplo recupera os números para um único pixel de uma tela de uma vez, quando o `canvas` está em branco (todos os pixels são preto transparente) e uma vez quando o pixel foi colorido de vermelho.

```
function pixelAt(cx, x, y) {
  var data = cx.getImageData(x, y, 1, 1);
  console.log(data.data);
}

var canvas = document.createElement("canvas");
var cx = canvas.getContext("2d");
pixelAt(cx, 10, 10);
// → [0, 0, 0, 0]

cx.fillStyle = "red";
cx.fillRect(10, 10, 1, 1);
pixelAt(cx, 10, 10);
// → [255, 0, 0, 255]
```

Os argumentos para `getImageData` indica o início das coordenadas x e y do retângulo que queremos recuperar, seguido por `width` e `height`.

Ignore transparência durante este exercício e se importe apenas com os três primeiros valores para um determinado pixel. Além disso, não se preocupe com a atualização do campo de cor quando o usuário escolher uma cor. Apenas certifique-se de que `fillStyle` do contexto do desenho e `strokeStyle` foram definidos com a cor sob o cursor do mouse.

Lembre-se que essas propriedades aceita qualquer cor que o CSS entende, que inclui o `rgb (R, G, B)` estilo que você viu no [Capítulo 15](#).

O método `getImageData` está sujeito as mesmas restrições como `toDataURL` irá gerar um erro quando a tela conter pixels que se originam a partir de outro domínio. Use um `try/catch` para relatar tais erros com um diálogo de alerta.

```
<script>
  tools["Pick color"] = function(event, cx) {
    // Your code here.
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Exibir dicas

Você de novo vai precisar usar `relativePos` para descobrir qual pixel foi clicado. A função `pixelAt` no exemplo demonstra como obter os valores para um determinado pixel. Colocar eles em uma `string rgb` exige apenas algumas concatenações.

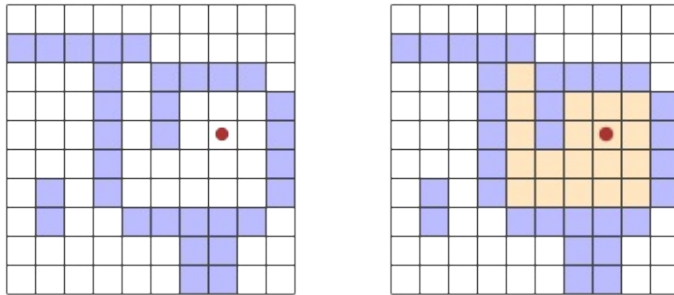
Certifique-se de verificar se a exceção que você pegar é uma instância de `SecurityError` de modo que você não trate acidentalmente o tipo errado de exceção.

Preenchimento

Este é um exercício mais avançado do que os dois anteriores, e isso vai exigir que você projete uma solução não trivial para um problema complicado. Certifique-se de que você tem bastante tempo e paciência antes de começar a trabalhar neste exercício e não desanime por falhas iniciais.

A ferramenta preenchimento de cores do pixel sob o mouse e todo o grupo de pixels em torno dele que têm a mesma cor. Para efeitos deste exercício, vamos considerar esse grupo para incluir todos os pixels que podem ser alcançadas a partir de nosso pixel inicial movendo em um único pixel de medidas horizontais e verticais (não diagonal), sem nunca tocar um pixel que tenha uma cor diferente a partir do pixel de partida.

A imagem a seguir ilustra o conjunto de pixels coloridos quando a ferramenta de preenchimento é usada no pixel marcado:



O preenchimento não vaza através de aberturas diagonais e não toca pixels que não são acessíveis, mesmo que tenham a mesma cor que o pixel alvo.

Você vai precisar mais uma vez do `getImageData` para descobrir a cor para cada pixel. É provavelmente uma boa ideia para buscar a imagem inteira de uma só vez e, em seguida, selecionar os dados de pixel do array resultante. Os pixels são organizados nesse array de uma forma semelhante aos elementos de rede, no [Capítulo 7](#), uma linha de cada vez, exceto que cada pixel é representado por quatro valores. O primeiro valor para o pixel em (x, y) é a posição $(x + y \times \text{width}) \times 4$.

Não incluam o quarto valor (alpha) desta vez, já que queremos ser capazes de dizer a diferença entre pixels pretos e vazios.

Encontrar todos os pixels adjacentes com a mesma cor exige que você "ande" sobre a superfície do pixel, um pixel para cima, baixo, esquerda ou direita, até que os novos pixels da mesma cor possam ser encontrados. Mas você não vai encontrar todos os pixels em um grupo na primeira caminhada. Em vez disso, você tem que fazer algo semelhante para o retrocesso feito pela expressão de correspondência regular, descrito no [Capítulo 9](#). Sempre que for possível proceder por mais de uma direção for possível, você deve armazenar todas as instruções que você não tomar imediatamente e voltar para elas mais tarde, quando terminar a sua caminhada atual.

Em uma imagem de tamanho normal, há um grande número de pixels. Assim, você deve ter o cuidado de fazer a quantidade mínima de trabalho necessário ou o seu programa vai levar muito tempo para ser executado. Por exemplo, todos os passos devem ignorar pixels vistos por caminhadas anteriores, de modo que ele não refaça o trabalho que já foi feito.

Eu recomendo chamando `fillRect` para pixels individuais quando um pixel que deve ser colorido é encontrado e manter alguma estrutura de dados que informe sobre todos os pixels que já foram analisados.

```
<script>
  tools["Flood fill"] = function(event, cx) {
    // Your code here.
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
```

```
</body>
```

Exibir dicas

Dado um par de coordenadas de partida e os dados da imagem para todo o canvas, esta abordagem deve funcionar:

- Criar uma array para armazenar informações sobre coordenadas já coloridas.
- Criar uma array de lista de trabalho para segurar coordenadas que devem ser analisadas. Coloque a posição inicial na mesma.
- Quando a lista de trabalho estiver vazia, estamos prontos.
- Remova um par de coordenadas a partir da lista de trabalho.
- Se essas coordenadas já estão em nosso array de pixels coloridos, volte para o passo 3.
- Colorir o pixel nas coordenadas atuais e adicionar as coordenadas para o array de pixels coloridos.
- Adicionar as coordenadas de cada pixel adjacente cuja cor é a mesma que a cor original do pixel inicial para a lista de trabalho.
- Retorne ao passo 3.

A lista de trabalho pode ser simplesmente um array de objetos vetoriais. A estrutura de dados que rastreia pixels coloridos serão consultados com muita frequência. Buscar por toda a coisa toda vez que um novo pixel é visitado vai custar muito tempo. Você poderia, ao invés de criar um array que tenha um valor nele para cada pixel, usando novamente $x + y \times$ esquema de largura para a associação de posições com pixels. Ao verificar se um pixel já foi colorido, você pode acessar diretamente o campo correspondente ao pixel atual.

Você pode comparar cores, executando sobre a parte relevante do array de dados, comparando um campo de cada vez. Ou você pode "condensar" uma cor a um único número ou o texto e comparar aqueles. Ao fazer isso, certifique-se de que cada cor produza um valor único. Por exemplo, a simples adição de componentes da cor não é segura, pois várias cores terá a mesma soma.

Ao enumerar os vizinhos de um determinado ponto, tenha o cuidado de excluir os vizinhos que não estão dentro da tela ou o seu programa poderá correr em uma direção para sempre.

Node.js

"Um estudante perguntou 'Os programadores de antigamente usavam somente máquinas simples e nenhuma linguagem de programação, mas mesmo assim eles construíram lindos programas. Por que nós usamos máquinas complicadas e linguagens de programação?'. Fu-Tzu respondeu 'Os construtores de antigamente usaram somente varas e barro, mas mesmo assim eles construíram lindas cabanas.'" Mestre Yuan-Ma, *The Book of Programming*

Até agora você vem aprendendo e usando a linguagem JavaScript num único ambiente: o navegador. Esse capítulo e o próximo vão introduzir brevemente você ao Node.js, um programa que permite que você aplique suas habilidades de JavaScript fora do navegador. Com isso, você pode construir desde uma ferramenta de linha de comando até servidores HTTP dinâmicos.

Esses capítulos visam te ensinar conceitos importantes nos quais o Node.js foi construído, e também te dar informação suficiente para escrever alguns programas úteis. Esses capítulos não detalham completamente o funcionamento do Node.

Você vem executando o código dos capítulos anteriores diretamente nessas páginas, pois eram pura e simplesmente JavaScript ou foram escritos para o navegador, porém os exemplos de códigos nesse capítulo são escritos para o Node e não vão rodar no navegador.

Se você quer seguir em frente e rodar os códigos desse capítulo, comece indo em <http://nodejs.org> e seguindo as instruções de instalação para o seu sistema operacional. Guarde também esse site como referência para uma documentação mais profunda sobre Node e seus módulos integrados.

Por Trás dos Panos

Um dos problemas mais difíceis em escrever sistemas que se comunicam através de uma rede é administrar a entrada e saída — ou seja, ler/escrever dados na rede, num disco rígido, e outros dispositivos. Mover os dados desta forma consome tempo, e planejar isso de forma inteligente pode fazer uma enorme diferença na velocidade em que um sistema responde ao usuário ou às requisições da rede.

A maneira tradicional de tratar a entrada e saída é ter uma função, como `readfile`, que começa a ler um arquivo e só retorna quando o arquivo foi totalmente lido. Isso é chamado *I/O síncrono* (I/O quer dizer input/output ou entrada/saída).

Node foi inicialmente concebido para o propósito de tornar a assincronicidade I/O mais fácil e conveniente. Nós já vimos interfaces síncronas antes, como o objeto `XMLHttpRequest` do navegador, discutido no Capítulo 17. Uma interface assíncrona permite que o script continue executando enquanto ela faz seu trabalho e chama uma função de *callback* quando está finalizada. Isso é como Node faz todo seu I/O.

JavaScript é ideal para um sistema como Node. É uma das poucas linguagens de programação que não tem uma maneira embutida de fazer I/O. Dessa forma, JavaScript poderia encaixar-se bastante na abordagem excêntrica do Node para o I/O sem acabar ficando com duas interfaces inconsistentes. Em 2009, quando Node foi desenhado, as pessoas já estavam fazendo I/O baseado em funções de *callback* no navegador, então a comunidade em volta da linguagem estava acostumada com um estilo de programação assíncrono.

Assincronia

Eu vou tentar ilustrar I/O síncrono contra I/O assíncrono com um pequeno exemplo, onde um programa precisa buscar recursos da Internet e então fazer algum processamento simples com o resultado dessa busca.

Em um ambiente síncrono, a maneira óbvia de realizar essa tarefa é fazer uma requisição após outra. Esse método tem a desvantagem de que a segunda requisição só será realizada após a primeira ter finalizado. O tempo total de execução será no mínimo a soma da duração das duas requisições. Isso não é um uso eficaz da máquina, que vai estar inativa por boa parte do tempo enquanto os dados são transmitidos através da rede.

A solução para esse problema, num sistema síncrono, é iniciar *threads* de controle. (Dê uma olhada no Capítulo 14 para uma discussão sobre *threads*.) Uma segunda *thread* poderia iniciar a segunda requisição, e então ambas as *threads* vão esperar os resultados voltarem, e após a resincronização elas vão combinar seus resultados.

No seguinte diagrama, as linhas grossa representam o tempo que o programa gastou em seu processo normal, e as linhas finas representam o tempo gasto esperando pelo I/O. Em um modelo síncrono, o tempo gasto pelo I/O faz parte da linha do tempo de uma determinada *thread* de controle. Em um modelo assíncrono, iniciar uma ação de I/O causa uma divisão na linha do tempo, conceitualmente falando. A *thread* que iniciou o I/O continua rodando, e o I/O é finalizado juntamente à ela, chamando uma função de *callback* quando é finalizada.

synchronous, single thread of control



synchronous, two threads of control



asynchronous



Uma outra maneira de mostrar essa diferença é que essa espera para que o I/O finalize é implícita no modelo síncrono, enquanto que é explícita no assíncrono. Mas assincronia é uma faca de dois gumes. Ela faz com que expressivos programas que seguem uma linha reta se tornem mais estranhos.

No capítulo 17, eu já mencionei o fato de que todos esses *callbacks* adicionam um pouco de ruído e rodeios para um programa. Se esse estilo de assincronia é uma boa ideia ou não, em geral isso pode ser discutido. De qualquer modo, levará algum tempo para se acostumar.

Mas para um sistema baseado em JavaScript, eu poderia afirmar que esse estilo de assincronia com *callback* é uma escolha sensata. Uma das forças do JavaScript é sua simplicidade, e tentar adicionar múltiplas *threads* de controle poderia causar uma grande complexidade. Embora os *callbacks* não tendem a ser códigos simples, como conceito, eles são agradavelmente simples e ainda assim poderosos o suficiente para escrever servidores web de alta performance.

O Comando Node

Quando Node.js está instalado em um sistema, ele disponibiliza um programa chamado `node`, que é usado para executar arquivos JavaScript. Digamos que você tenha um arquivo chamado `ola.js`, contendo o seguinte código:

```
var mensagem = "Olá mundo";
console.log(mensagem);
```

Você pode então rodar `node` a partir da linha de comando para executar o programa:

```
$ node ola.js
Olá mundo
```

O método `console.log` no Node tem um funcionamento bem parecido ao do navegador. Ele imprime um pedaço de texto. Mas no Node, o texto será impresso pelo processo padrão de saída, e não no console JavaScript do navegador.

Se você rodar `node` sem especificar nenhum arquivo, ele te fornecerá um *prompt* no qual você poderá escrever códigos JavaScript e ver o resultado imediatamente.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

A variável `process`, assim como a variável `console`, está disponível globalmente no Node. Ela fornece várias maneiras de inspecionar e manipular o programa atual. O método `exit` finaliza o processo e pode receber um código de saída, que diz ao programa que iniciou `node` (nesse caso, a linha de comando) se o programa foi completado com sucesso (código zero) ou se encontrou algum erro (qualquer outro código).

Para encontrar os argumentos de linha de comando recebidos pelo seu script, você pode ler `process.argv`, que é um *array* de *strings*. Note que também estarão inclusos o nome dos comandos `node` e o nome do seu script, fazendo com que os argumentos comecem na posição 2. Se `showargv.js` contém somente o *statement* `console.log(process.argv)`, você pode rodá-lo dessa forma:

```
$ node showargv.js one --and two
["node", "/home/braziljs/showargv.js", "one", "--and", "two"]
```

Todas as variáveis JavaScript globais, como `Array`, `Math` and `JSON`, estão presentes também no ambiente do Node. Funcionalidades relacionadas ao navegador, como `document` e `alert` estão ausentes.

O objeto global do escopo, que é chamado `window` no navegador, passa a ser `global` no Node, que faz muito mais sentido.

Módulos

Além de algumas variáveis que mencionei, como `console` e `process`, Node também colocou pequenas funcionalidades no escopo global. Se você quiser acessar outras funcionalidades embutidas, você precisa pedir esse módulo ao sistema.

O sistema de módulo CommonJS, baseado na função `require`, estão descritos no Capítulo 10. Esse sistema é construído em Node e é usado para carregar desde módulos integrados até bibliotecas transferidas, ou até mesmo, arquivos que fazem parte do seu próprio programa.

Quando `require` é chamado, Node tem que transformar a string recebida em um arquivo real a ser carregado. Nomes de caminhos que começam com `"/"`, `"/."`, ou `"../"` são resolvidos relativamente ao atual caminho do módulo, aonde `"/."` significa o diretório corrente, `"../"` para um diretório acima, e `"/"` para a raiz do sistema de arquivos. Então se você solicitar por `"/world/world"` do arquivo `/home/braziljs/elifa/run.js`, Node vai tentar carregar o arquivo `/home/braziljs/elifa/world/world.js`. A extensão `.js` pode ser omitida.

Quando uma *string* recebida pelo `require` não parece ter um caminho relativo ou absoluto, fica implícito que ela se refere a um módulo integrado ou que está instalado no diretório `node_modules`. Por exemplo, `require(fs)` disponibilizará o módulo de sistema de arquivos integrado ao Node, `require("elifa")` vai tentar carregar a biblioteca encontrada em `node_modules/elifa`. A maneira mais comum de instalar bibliotecas como essas é usando NPM, que em breve nós vamos discutir.

Para ilustrar o uso do `require`, vamos configurar um projeto simples que consiste de dois arquivos. O primeiro é chamado `main.js`, que define um script que pode ser chamado da linha de comando para alterar uma *string*.

```
var garble = require("./garble");

// O índice 2 possui o valor do primeiro parâmetro da linha de comando
var parametro = process.argv[2];

console.log(garble(parametro));
```

O arquivo `garble.js` define uma biblioteca para alterar string, que pode ser usada tanto da linha de comando quanto por outros scripts que precisam ter acesso direto a função de alterar.

```
module.exports = function(string) {
  return string.split("").map(function(ch) {
    return String.fromCharCode(ch.charCodeAt(0) + 5);
  }).join("");
}
```

Lembre-se que substituir `module.exports`, ao invés de adicionar propriedades à ele, nos permite exportar um valor específico do módulo. Nesse caso, nós fizemos com que o resultado ao requerer nosso arquivo `garble` seja a própria função de alterar.

A função separa a *string* recebida em dois caracteres únicos separando a *string* vazia e então substituindo cada caractere cujo código é cinco pontos maior. Finalmente, o resultado é reagrupado novamente numa *string*.

Agora nós podemos chamar nossa ferramenta dessa forma:

```
$ node main.js JavaScript
Of{fXhwnuy
```

Instalando com NPM

NPM, que foi brevemente discutido no Capítulo 10, é um repositório online de módulos JavaScript, muitos deles escritos para Node. Quando você instala o Node no seu computador, você também instala um programa chamado `npm`, que fornece uma interface conveniente para esse repositório.

Por exemplo, um módulo que você vai encontrar na NPM é `figlet`, que pode converter texto em *ASCII art*—desenhos feitos de caracteres de texto. O trecho a seguir mostra como instalar e usar esse módulo:

```
$ npm install figlet
npm GET https://registry.npmjs.org/figlet
npm 200 https://registry.npmjs.org/figlet
npm GET https://registry.npmjs.org/figlet/-/figlet-1.0.9.tgz
npm 200 https://registry.npmjs.org/figlet/-/figlet-1.0.9.tgz
figlet@1.0.9 node_modules/figlet
$ node
> var figlet = require("figlet");
> figlet.text("Hello world!", function(error, data) {
  if (error)
    console.error(error);
  else
    console.log(data);
});
```

Depois de rodar `npm install`, NPM já vai ter criado um diretório chamado `node_modules`. Dentro desse diretório haverá um outro diretório chamado `figlet`, que vai conter a biblioteca. Quando rodamos `node` e chamamos `require("figlet")`, essa biblioteca é carregada, e nós podemos chamar seu método `text` para desenhar algumas letras grandes.

Talvez de forma inesperada, ao invés de retornar a string que faz crescer as letras, `figlet.text` têm uma função de *callback* que passa o resultado para ela. Ele também passa outro parâmetro no *callback*, `error`, que vai possuir um objeto de erro quando alguma coisa sair errada ou nulo se tudo ocorrer bem.

Isso é um padrão comum em Node. Renderizar alguma coisa com `figlet` requer a biblioteca para ler o arquivo que contém as formas das letras. Lendo esse arquivo do disco é uma operação assíncrona no Node, então `figlet.text` não pode retornar o resultado imediatamente. Assincronia é, de certa forma, infecciosa—qualquer função que chamar uma função assincronamente precisa se tornar assíncrona também.

Existem muito mais coisas no NPM além de `npm install`. Ele pode ler arquivos `package.json`, que contém informações codificadas em JSON sobre o programa ou biblioteca, como por exemplo outras bibliotecas que depende. Rodar `npm install` em um diretório que contém um arquivo como esse vai instalar automaticamente todas as dependências, assim como as dependências das dependências. A ferramenta `npm` também é usada para publicar bibliotecas para o repositório NPM online de pacotes para que as pessoas possam encontrar, transferir e usá-los.

Esse livro não vai abordar detalhes da utilização do NPM. Dê uma olhada em npmjs.org para uma documentação mais detalhada e para uma maneira simples de procurar por bibliotecas.

O módulo de arquivos de sistema

Um dos módulos integrados mais comuns que vêm com o Node é o módulo `"fs"`, que significa *file system*. Esse módulo fornece funções para o trabalho com arquivos de diretórios.

Por exemplo, existe uma função chamada `readFile`, que lê um arquivo e então chama um *callback* com o conteúdo desse arquivo.

```
var fs = require("fs");
fs.readFile("file.txt", "utf8", function(error, text) {
  if (error)
    throw error;
  console.log("The file contained:", text);
});
```

O segundo argumento passado para `readFile` indica a codificação de caracteres usada para decodificar o arquivo numa *string*. Existem muitas maneiras de codificar texto em informação binária, mas a maioria dos sistemas modernos usam UTF-8 para codificar texto, então a menos que você tenha razões para acreditar que outra forma de codificação deve ser usada, passar "utf8" ao ler um arquivo de texto é uma aposta segura. Se você não passar uma codificação, o Node vai assumir que você está interessado na informação binária e vai te dar um objeto `Buffer` ao invés de uma *string*. O que por sua vez, é um objeto *array-like* que contém números representando os *bytes* nos arquivos.

```
var fs = require("fs");
fs.readFile("file.txt", function(error, buffer) {
  if (error)
    throw error;
  console.log("The file contained", buffer.length, "bytes.",
    "The first byte is:", buffer[0]);
});
```

Uma função similar, `writeFile`, é usada para escrever um arquivo no disco.

```
var fs = require("fs");
fs.writeFile("graffiti.txt", "Node was here", function(err) {
  if (err)
    console.log("Failed to write file:", err);
  else
    console.log("File written.");
});
```

Aqui, não foi necessário especificar a codificação de caracteres, pois a função `writeFile` assume que recebeu uma *string* e não um objeto `Buffer`, e então deve escrever essa *string* como texto usando a codificação de caracteres padrão, que é UTF-8.

O módulo `"fs"` contém muitas outras funções úteis: `readdir` que vai retornar os arquivos em um diretório como um *array* de *strings*, `stat` vai buscar informação sobre um arquivo, `rename` vai renomear um arquivo, `unlink` vai remover um arquivo, e assim por diante. Veja a documentação em nodejs.org para especificidades.

Muitas das funções em `"fs"` vêm com variantes síncronas e assíncronas. Por exemplo, existe uma versão síncrona de `readFile` chamada `readFileSync`.

```
var fs = require("fs");
console.log(fs.readFileSync("file.txt", "utf8"));
```

Funções síncronas requerem menos formalismo na sua utilização e podem ser úteis em alguns scripts, onde a extra velocidade oferecida pela assincronia *I/O* é irrelevante. Mas note que enquanto tal operação síncrona é executada, seu programa fica totalmente parado. Se nesse período ele deveria responder ao usuário ou a outras máquinas na rede, ficar preso com um *I/O* síncrono pode acabar produzindo atrasos inconvenientes.

O Módulo HTTP

Outro principal é o `"http"`. Ele fornece funcionalidade para rodar servidores HTTP e realizar requisições HTTP.

Isso é tudo que você precisa para rodar um simples servidor HTTP:

```
var http = require("http");
var server = http.createServer(function(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);
```

Se você rodar esse script na sua máquina, você pode apontar seu navegador para o endereço <http://localhost:8000/hello> para fazer uma requisição no seu servidor. Ele irá responder com uma pequena página HTML.

A função passada como um argumento para `createServer` é chamada toda vez que um cliente tenta se conectar ao servidor. As variáveis `request` e `response` são os objetos que representam a informação que chega e sai. A primeira contém informações sobre a requisição, como por exemplo a propriedade `url`, que nos diz em qual URL essa requisição foi feita.

Para enviar alguma coisa de volta, você chama métodos do objeto `response`. O primeiro, `writeHead`, vai escrever os cabeçalhos de resposta (veja o Capítulo 17). Você define o código de status (200 para "OK" nesse caso) e um objeto que contém valores de cabeçalho. Aqui nós dizemos ao cliente que estaremos enviando um documento HTML

de volta.

Em seguida, o corpo da resposta (o próprio documento) é enviado com `response.write`. Você pode chamar esse método quantas vezes você quiser para enviar a resposta peça por peça, possibilitando que a informação seja transmitida para o cliente assim que ela esteja disponível. Finalmente, `response.end` assina o fim da resposta.

A chamada de `server.listen` faz com que o servidor comece a esperar por conexões na porta 8000. Por isso você precisa se conectar a `localhost:8000`, ao invés de somente `localhost` (que deveria usar a porta 80, por padrão), para se comunicar com o servidor.

Para parar de rodar um script Node como esse, que não finaliza automaticamente pois está aguardando por eventos futuros (nesse caso, conexões de rede), aperte Ctrl+C.

Um servidor real normalmente faz mais do que o que nós vimos no exemplo anterior—ele olha o método da requisição (a propriedade `method`) para ver que ação o cliente está tentando realizar e olha também a URL da requisição para descobrir que recurso essa ação está executando. Você verá um servidor mais avançado daqui a pouco neste capítulo.

Para agir como um *cliente HTTP*, nós podemos usar a função `request` no módulo `"http"`.

```
var http = require("http");
var request = http.request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, function(response) {
  console.log("Server responded with status code",
    response.statusCode);
});
request.end();
```

O primeiro parâmetro passado para `request` configura a requisição, dizendo pro Node qual o servidor que ele deve se comunicar, que caminho solicitar daquele servidor, que método usar, e assim por diante. O segundo parâmetro é a função que deverá ser chamada quando uma resposta chegar. É informado um objeto que nos permite inspecionar a resposta, para descobrir o seu código de status, por exemplo.

Assim como o objeto `response` que vimos no servidor, o objeto `request` nos permite transmitir informação na requisição com o método `write` e finalizar a requisição com o método `end`. O exemplo não usa `write` porque requisições `GET` não devem conter informação no corpo da requisição.

Para fazer requisições para URLs HTTP seguras (HTTPS), o Node fornece um pacote chamado `https`, que contém sua própria função `request`, parecida a `http.request`.

Streams

Nós já vimos dois exemplos de *streams* em HTTP—são, consecutivamente, o objeto de resposta no qual o servidor pode escrever e o objeto de requisição que foi retornado do `http.request`.

Streams de gravação são um conceito amplamente usado nas interfaces Node. Todos os *streams* de gravação possuem um método `write`, que pode receber uma *string* ou um objeto `Buffer`. Seus métodos `end` fecham a transmissão e, se passado um parâmetro, também vai escrever alguma informação antes de fechar. Ambos métodos podem receber um *callback* como um parâmetro adicional, que eles vão chamar ao fim do escrever ou fechar a transmissão.

É possível criar *streams* de gravação que apontam para um arquivo com a função `fs.createWriteStream`. Então você pode usar o método `write` no objeto resultante para escrever o arquivo peça por peça, ao invés de escrever tudo de uma só vez com o `fs.writeFile`.

Streams de leitura são um pouco mais fechados. Em ambos a variável `request` que foi passada para a função de *callback* do servidor HTTP e a variável `response` para o cliente HTTP são *streams* de leitura. (Um servidor lê os pedidos e então escreve as respostas, enquanto que um cliente primeiro escreve um pedido e então lê a resposta.) Para ler de um *stream* usamos manipuladores de eventos, e não métodos.

Objetos que emitem eventos no Node têm um método chamado `on` que é similar ao método `addEventListener` no navegador. Você dá um nome de evento e então uma função, e isso irá registrar uma função para ser chamada toda vez que um dado evento ocorrer.

Streams de leitura possuem os eventos `"data"` e `"end"`. O primeiro é acionado sempre que existe alguma informação chegando, e o segundo é chamado sempre que a *stream* chega ao fim. Esse modelo é mais adequado para um *streaming* de dados, que pode ser imediatamente processado, mesmo quando todo documento ainda não está disponível. Um arquivo pode ser lido como uma *stream* de leitura usando a função `fs.createReadStream`.

O seguinte código cria um servidor que lê o corpo da requisição e o devolve em caixa alta para o cliente via *stream*:

```
var http = require("http");
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", function(chunk) {
    response.write(chunk.toString().toUpperCase());
  });
  request.on("end", function() {
    response.end();
  });
}).listen(8000);
```

A variável `chunk` enviada para o manipulador de dados será um `Buffer` binário, que nós podemos converter para uma *string* chamando `toString` nele, que vai decodificá-lo usando a codificação padrão (UTF-8).

O seguinte trecho de código, se rodado enquanto o servidor que transforma letras em caixa alta estiver rodando, vai enviar uma requisição para esse servidor e retornar a resposta que obtiver:

```
var http = require("http");
var request = http.request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, function(response) {
  response.on("data", function(chunk) {
    process.stdout.write(chunk.toString());
  });
});
request.end("Hello server");
```

O exemplo escreve no `process.stdout` (a saída padrão de processos, como uma *stream* de escrita) ao invés de usar `console.log`. Nós não podemos usar `console.log` porque isso adicionaria uma linha extra depois de cada pedaço de texto escrito, o que é adequado no nosso exemplo.

Um servidor de arquivos simples

Vamos combinar nossas novas descobertas sobre servidores HTTP e conversas sobre sistema de arquivos e criar uma ponte entre eles: um servidor HTTP que permite acesso remoto ao sistema de arquivos. Um servidor desse tipo possui diversos usuários. Ele permite que aplicações web guardem e compartilhem dados ou dá direito para um determinado grupo de pessoas compartilhar muitos arquivos.

Quando lidamos com arquivos de recursos HTTP, os métodos HTTP `GET`, `PUT` e `DELETE` podem ser usados, respectivamente, para ler, escrever e apagar esses arquivos. Nós vamos interpretar o caminho na requisição como o caminho do arquivo referido por aquela requisição.

Provavelmente nós não queremos compartilhar todo nosso sistema de arquivos, então nós vamos interpretar esses caminhos como se começassem no diretório de trabalho do servidor, que é o diretório no qual ele começou. Se eu rodar o servidor de `/home/braziljs/public/` (ou `C:\Users\braziljs\public\` no Windows), então a requisição por `/file.txt` deve ser referir a `/home/braziljs/public/file.txt` (ou `C:\Users\braziljs\public\file.txt`).

Nós vamos construir um programa peça por peça, usando um objeto chamado `methods` para guardar as funções que tratam os vários métodos HTTP.

```
var http = require("http"), fs = require("fs");

var methods = Object.create(null);

http.createServer(function(request, response) {
  function respond(code, body, type) {
    if (!type) type = "text/plain";
    response.writeHead(code, {"Content-Type": type});
    if (body && body.pipe)
      body.pipe(response);
    else
      response.end(body);
  }
  if (request.method in methods)
    methods[request.method](urlToPath(request.url),
                           respond, request);
  else
    respond(405, "Method " + request.method +
            " not allowed.");
}).listen(8000);
```

Isso vai começar um servidor que apenas retorna erro 405 nas respostas, que é o código usado para indicar que dado método não está sendo tratado pelo servidor.

A função `respond` é passada para as funções que tratam os vários métodos e agem como *callback* para finalizar a requisição. Isso carrega um código de status do HTTP, um corpo e opcionalmente um tipo conteúdo como argumentos. Se o valor passado para o corpo é um *stream* de leitura, ele terá um método `pipe`, que será usado para encaminhar uma *stream* de leitura para uma *stream* de escrita. Caso contrário, assumimos que o corpo será `null` (não há corpo) ou uma *string* é passada diretamente para o método `end` da resposta.

Para obter um caminho de uma URL em uma requisição, a função `urlToPath` usa o módulo `"url"` embutido no Node para parsear a URL. Ela pega o nome do caminho, que será algo parecido a `/file.txt`, o decodifica para tirar os códigos de escape (como `%20` e etc), e coloca um único ponto para produzir um caminho relativo ao diretório atual.

```
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
}
```

É provável que você esteja preocupado com a segurança da função `urlToPath`, e você está certo, deve se preocupar mesmo. Nós vamos retornar a ela nos exercícios.

Nós vamos fazer com que o método `GET` retorne uma lista de arquivos quando lermos um diretório e retornar o conteúdo do arquivo quando lermos um arquivo regular.

Uma questão delicada é que tipo de cabeçalho `Content-Type` nós devemos adicionar quando retornar um conteúdo de um arquivo. Tendo em vista que esses arquivos podem ser qualquer coisa, nosso servidor não pode simplesmente retornar o mesmo tipo para todos eles. Mas o NPM pode ajudar com isso. O pacote `mime` (indicadores de tipo de conteúdo como `text/plain` também são chamados *MIME types*) sabe o tipo adequado de um grande número de extensões de arquivos.

Se você rodar o seguinte comando `npm` no diretório aonde o script do servidor está, você estará apto a usar `require("mime")` para acessar essa biblioteca:

```
$ npm install mime
npm http GET https://registry.npmjs.org/mime
npm http 304 https://registry.npmjs.org/mime
mime@1.2.11 node_modules/mime
```

Quando um arquivo requisitado não existe, o código de erro HTTP adequado a ser retornado é 404. Nós vamos usar `fs.stat`, que obtém informações sobre um arquivo, para saber se o arquivo existe e/ou se é um diretório.

```
methods.GET = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(404, "File not found");
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.readdir(path, function(error, files) {
        if (error)
          respond(500, error.toString());
        else
          respond(200, files.join("\n"));
      });
    else
      respond(200, fs.createReadStream(path),
        require("mime").lookup(path));
  });
};
```

Como ele pode levar um bom tempo para encontrar o arquivo no disco, `fs.stat` é assíncrono. Quando o arquivo não existe, `fs.stat` vai passar um objeto de erro com `"ENOENT"` em uma propriedade chamada `code` para o seu *callback*. Isso seria muito bom se o Node definisse diferentes subtipos de `Error` para diferentes tipos de erros, mas ele não o faz. Ao invés disso, Node coloca um código obscuro, inspirado no sistema Unix lá.

Nós vamos reportar qualquer erro que não esperamos com o código de status 500, que indica que o problema está no servidor, ao contrário dos códigos que começam com 4 (como o 404), que se referem a requisições ruins. Existem algumas situações nas quais isso não totalmente preciso, mas para um programa pequeno de exemplo como esse, deverá ser bom o suficiente.

O objeto `stats` retornado pelo `fs.stat` nos diz uma porção de coisas sobre um arquivo, tais como tamanho (propriedade `size`) e sua data de modificação (propriedade `mtime`). Nosso interesse aqui é saber se isso é um diretório ou um arquivo regular, e quem nos diz isso é o método `isDirectory`.

Nós usamos `fs.readdir` para ler a lista de arquivos em um diretório e, ainda em outro *callback*, retornar o resultado para o usuário. Para arquivos comuns, nós criamos uma *stream* de leitura com o `fs.createReadStream` e passamos ela ao `respond`, junto com o tipo de conteúdo que o módulo `"mime"` nos deu para esse nome de arquivo.

O código que trata as requisições de `DELETE` é um pouco mais simples.

```

methods.DELETE = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(204);
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.rmdir(path, respondErrorOrNothing(respond));
    else
      fs.unlink(path, respondErrorOrNothing(respond));
  });
};

```

Você deve estar se perguntando porque tentar deletar um arquivo inexistente retornar um status 204, e não um erro. Quando o arquivo que será deletado não existe, você pode dizer que o objetivo da requisição já foi cumprido. O padrão HTTP recomenda que as pessoas façam requisições *idempotentes*, o que significa que independente da quantidade de requisições, elas não devem produzir um resultado diferente.

```

function respondErrorOrNothing(respond) {
  return function(error) {
    if (error)
      respond(500, error.toString());
    else
      respond(204);
  };
}

```

Quando uma resposta HTTP não contém nenhum dado, o status 204 ("no content") pode ser usado para indicar isso. Tendo em vista que a gente precisa construir *callbacks* que reportam um erro ou retornam uma resposta 204 em diferentes situações, eu escrevi uma função chamada `respondErrorOrNothing` que cria esse *callback*.

Aqui está a função que trata as requisições `PUT` :

```

methods.PUT = function(path, respond, request) {
  var outputStream = fs.createWriteStream(path);
  outputStream.on("error", function(error) {
    respond(500, error.toString());
  });
  outputStream.on("finish", function() {
    respond(204);
  });
  request.pipe(outputStream);
};

```

Aqui, nós não precisamos checar se o arquivo existe - se ele existe, nós simplesmente sobrescrevemos ele. Novamente nós usamos `pipe` para mover a informação de um *stream* de leitura para um de escrita, nesse caso de uma requisição para um arquivo. Se a criação do *stream* falhar, um evento `"error"` é disparado e reportado na nossa resposta. Quando a informação for transferida com sucesso, `pipe` vai fechar ambos *streams*, o que vai disparar o evento `"finish"` no *stream* de escrita. Quando isso acontecer, nós podemos reportar sucesso na nossa resposta para o cliente com um status 204.

O script completo para o servidor está disponível em eloquentjavascript.net/code/file_server.js. Você pode fazer o download e rodá-lo com Node pra começar seu próprio servidor de arquivos. E é claro, você pode modificá-lo e estendê-lo para resolver os exercícios desse capítulo ou para experimentar.

A ferramenta de linha de comando `curl`, amplamente disponível em sistemas Unix, pode ser usada para fazer requisições HTTP. A sessão a seguir é um rápido teste do nosso servidor. Note que `-x` é usado para escolher o método da requisição e `-d` é usado para incluir o corpo da requisição.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

A primeira requisição feita para o arquivo `file.txt` falha pois o arquivo ainda não existe. A requisição `PUT` cria o arquivo, para que então a próxima requisição consiga encontrá-lo com sucesso. Depois de deletar o arquivo com uma requisição `DELETE`, o arquivo passa a não ser encontrado novamente.

Tratamento de erros

No código para o servidor de arquivos, existem seis lugares aonde nós estamos explicitando exceções de rota que nós não sabemos como tratá-los como respostas de erro. Como exceções são passadas como argumentos e, portanto, não são automaticamente propagadas para os *callbacks*, elas precisam ser tratadas a todo momento de forma explícita. Isso acaba completamente com a vantagem de tratamento de exceções, isto é, a habilidade de centralizar o tratamento das condições de falha.

O que acontece quando alguma coisa *joga* uma exceção em seu sistema? Como não estamos usando nenhum bloco `try`, a exceção vai propagar para o topo da pilha de chamada. No Node, isso aborta o programa e escreve informações sobre a exceção (incluindo um rastro da pilha) no programa padrão de *stream* de erros.

Isso significa que nosso servidor vai colidir sempre que um problema for encontrado no código do próprio servidor, ao contrário dos problemas assíncronos, que são passados como argumentos para os *callbacks*. Se nós quisermos tratar todas as exceções *levantadas* durante o tratamento de uma requisição, para ter certeza que enviamos uma resposta, precisamos adicionar blocos de `try/catch` para todos os *callbacks*.

Isso é impraticável. Muitos programas em Node são escritos para fazer o menor uso possível de exceções, assumindo que se uma exceção for *levantada*, aconteceu algo que o programa não conseguiu resolver, e colidir é a resposta certa.

Outra abordagem é usar promessas, que foram introduzidas no Capítulo 17. Promessas capturam as exceções *levantadas* por funções de *callback* e propagam elas como falhas. É possível carregar uma biblioteca de promessa no Node e usá-la para administrar seu controle assíncrono. Algumas bibliotecas Node fazem integração com as promessas, mas as vezes é trivial envolvê-las. O excelente módulo `"promise"` do NPM contém uma função chamada `denodeify`, que converte uma função assíncrona como a `fs.readFile` para uma função de retorno de promessa.

```
var Promise = require("promise");
var fs = require("fs");

var readFile = Promise.denodeify(fs.readFile);
readFile("file.txt", "utf8").then(function(content) {
  console.log("The file contained: " + content);
}, function(error) {
  console.log("Failed to read file: " + error);
});
```

A título de comparação, eu escrevi uma outra versão do servidor de arquivos baseado em promessas, que você pode encontrar em eloquentjavascript.net/code/file_server_promises.js. Essa versão é um pouco mais clara pois as funções podem retornar seus resultados, ao invés de ter que chamar *callbacks*, e a rota de exceções está implícito, ao invés de explícito.

Eu vou mostrar algumas linhas do servidor de arquivos baseado em promessas para ilustrar a diferença no estilo de programação.

O objeto `fsp` que é usado por esse código contém estilos de promessas variáveis para determinado número de funções `fs`, envolvidas por `Promise.denodeify`. O objeto retornado, com propriedades `code` e `body`, vai se tornar o resultado final de uma cadeia de promessas, e vai ser usado para determinar que tipo de resposta vamos mandar pro cliente.

```
methods.GET = function(path) {
  return inspectPath(path).then(function(stats) {
    if (!stats) // Does not exist
      return {code: 404, body: "File not found"};
    else if (stats.isDirectory())
      return fsp.readdir(path).then(function(files) {
        return {code: 200, body: files.join("\n")};
      });
    else
      return {code: 200,
        type: require("mime").lookup(path),
        body: fs.createReadStream(path)};
  });
};

function inspectPath(path) {
  return fsp.stat(path).then(null, function(error) {
    if (error.code == "ENOENT") return null;
    else throw error;
  });
}
```

A função `inspectPath` simplesmente envolve o `fs.stat`, que trata o caso de arquivo não encontrado. Nesse caso, nós vamos substituir a falha por um sucesso que representa `null`. Todos os outros erros são permitidos a propagar. Quando a promessa retornada desses manipuladores falha, o servidor HTTP responde com um status 500.

Resumo

Node é um sistema bem íntegro e legal que permite rodar JavaScript em um contexto fora do navegador. Ele foi originalmente concebido para tarefas de rede para desempenhar o papel de um *nó* na rede. Mas ele se permite a realizar todas as tarefas de script, e se escrever JavaScript é algo que você gosta, automatizar tarefas de rede com Node funciona de forma maravilhosa.

O NPM disponibiliza bibliotecas para tudo que você possa imaginar (e algumas outras coisas que você provavelmente nunca pensou), e permite que você atualize e instale essas bibliotecas rodando um simples comando. Node também vêm com um bom número de módulos embutidos, incluindo o módulo `"fs"`, para trabalhar com sistema de arquivos e o `"http"`, para rodar servidores HTTP e fazer requisições HTTP.

Toda entrada e saída no Node é feita de forma assíncrona, a menos que você explicitamente use uma variante síncrona da função, como a `fs.readFileSync`. Você fornece as funções de *callback* e o Node vai chamá-las no tempo certo, quando o I/O que você solicitou tenha terminado.

Exercícios

Negociação de Conteúdo, novamente

No Capítulo 17, o primeiro exercício era fazer várias requisições para eloquentjavascript.net/author, pedindo por tipos diferentes de conteúdo passando cabeçalhos `Accept` diferentes.

Faça isso novamente usando a função `http.request` do Node. Solicite pelo menos os tipos de mídia `text/plain`, `text/html` e `application/json`. Lembre-se que os cabeçalhos para uma requisição podem ser passados como objetos, na propriedade `headers` do primeiro argumento da `http.request`.

Escreva o conteúdo das respostas para cada requisição.

Dica: Não se esqueça de chamar o método `end` no objeto retornado pela `http.request` para de fato disparar a requisição.

O objeto de resposta passado ao *callback* da `http.request` é um *stream* de leitura. Isso significa que ele não é muito trivial pegar todo o corpo da resposta dele. A função a seguir lê todo o *stream* e chama uma função de *callback* com o resultado, usando o padrão comum de passar qualquer erro encontrado como o primeiro argumento do *callback*:

```
function readStreamAsString(stream, callback) {
  var data = "";
  stream.on("data", function(chunk) {
    data += chunk.toString();
  });
  stream.on("end", function() {
    callback(null, data);
  });
  stream.on("error", function(error) {
    callback(error);
  });
}
```

Corrigindo uma falha

Para um fácil acesso remoto aos arquivos, eu poderia adquirir o hábito de ter o servidor de arquivos definido nesse capítulo na minha máquina, no diretório `/home/braziljs/public/`. E então, um dia, eu encontro alguém que tenha conseguido acesso a todas as senhas que eu gravei no navegador.

O que aconteceu?

Se ainda não está claro para você, pense novamente na função `urlToPath` definida dessa forma:

```
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
}
```

Agora considere o fato de que os caminhos para as funções `fs` podem ser relativos-eles podem conter `../` para voltar a um diretório acima. O que acontece quando um cliente envia uma requisição para uma dessas URLs abaixo?

```
http://myhostname:8000/../../config/config/google-chrome/Default/Web%20Data
http://myhostname:8000/../../ssh/id_dsa
http://myhostname:8000/../../etc/passwd
```

Mudar o `urlToPath` corrige esse problema. Levando em conta o fato de que o Node no Windows permite tanto barras quanto contrabarras para separar diretórios.

Além disso, pense no fato de que assim que você expor algum sistema *meia boca* na internet, os *bugs* nesse sistema podem ser usado para fazer coisas ruins para sua máquina.

Dicas Basta remover todas as recorrências de dois pontos que tenham uma barra, uma contrabarra ou as extremidades da *string*. Usando o método `replace` com uma expressão regular é a maneira mais fácil de fazer isso. Não se esqueça da *flag* `g` na expressão, ou o `replace` vai substituir somente uma única instância e as pessoas

ainda poderiam incluir pontos duplos no caminho da URL a partir dessa medida de segurança! Também tenha certeza de substituir *depois* de decodificar a *string*, ou seria possível despistar o seu controle que codifica pontos e barras.

Outro caso de preocupação potencial é quando os caminhos começam com barra, que são interpretados como caminhos absolutos. Mas por conta do `urlToPath` colocar um ponto na frente do caminho, é impossível criar requisições que resultam em tal caminho. Múltiplas barras numa linha, dentro do caminho, são estranhas mas serão tratadas como uma única barra pelo sistema de arquivos.

Criando diretórios

Embora o método `DELETE` esteja envolvido em apagar diretórios (usando `fs.rmdir`), o servidor de arquivos não disponibiliza atualmente nenhuma maneira de *criar* diretórios.

Adicione suporte para o método `MKCOL`, que deve criar um diretório chamando `fs.mkdir`. `MKCOL` não é um método básico do HTTP, mas ele existe nas normas da *WebDAV*, que especifica um conjunto de extensões para o HTTP, tornando-o adequado para escrever recursos, além de os ler.

Dicas Você pode usar a função que implementa o método `DELETE` como uma planta baixa para o método `MKCOL`. Quando nenhum arquivo é encontrado, tente criar um diretório com `fs.mkdir`. Quando um diretório existe naquele caminho, você pode retornar uma resposta 204, então as requisições de criação de diretório serão *idempotentes*. Se nenhum diretório de arquivo existe, retorne um código de erro. O código 400 ("*bad request*") seria o mais adequado nessa situação.

Um espaço público na rede

Uma vez que o servidor de arquivos serve qualquer tipo de arquivo e ainda inclui o cabeçalho `Content-Type`, você pode usá-lo para servir um website. Mas uma vez que seu servidor de arquivos permita que qualquer um delete e sobrescreva arquivos, seria um tipo interessante de website: que pode ser modificado, vandalizado e destruído por qualquer um que gaste um tempo para criar a requisição HTTP correta. Mas ainda assim, seria um website.

Escreva uma página HTML básica que inclui um simples arquivo JavaScript. Coloque os arquivos num diretório servido pelo servidor de arquivos e abra isso no seu navegador.

Em seguida, como um exercício avançado ou como um projeto de fim de semana, combine todo o conhecimento que você adquiriu desse livro para construir uma interface mais amigável pra modificar o website de dentro do website.

Use um formulário HTML (Capítulo 18) para editar os conteúdos dos arquivos que fazem parte do website, permitindo que o usuário atualize eles no servidor fazendo requisições HTTP como vimos no Capítulo 17.

Comece fazendo somente um único arquivo editável. Então faça de uma maneira que o usuário escolha o arquivo que quer editar. Use o fato de que nosso servidor de arquivos retorna uma lista de arquivos durante a leitura de um diretório.

Não trabalhe diretamente no código do servidor de arquivos, tendo em vista que se você cometer um engano você vai afetar diretamente os arquivos que estão lá. Ao invés disso, mantenha seu trabalho em um diretório sem acessibilidade pública e copie ele pra lá enquanto testa.

Se seu computador está diretamente ligado a internet, sem um *firewall*, roteador, ou outro dispositivo interferindo, você pode ser capaz de convidar um amigo para usar seu website. Para checar, vá até whatismyip.com, copie e cole o endereço de IP que ele te deu na barra de endereço do seu navegador, e adicione `:8000` depois dele para selecionar a porta correta. Se isso te levar ao seu website, está online para qualquer um que quiser ver.

Dicas Você pode criar um elemento `<textarea>` para conter o conteúdo do arquivo que está sendo editado. Uma requisição `GET`, usando `XMLHttpRequest`, pode ser usada para pegar o atual conteúdo do arquivo. Você pode usar URLs relativas como `index.html`, ao invés de `http://localhost:8000/index.html`, para referir-se aos arquivos do mesmo servidor que está rodando o script.

Então, quando o usuário clicar num botão (você pode usar um elemento `<form>` e um evento `"submit"` ou um simples manipulador `"click"`), faça uma requisição `PUT` para a mesma URL, com o conteúdo do `<textarea>` no corpo da requisição para salvar o arquivo.

Você pode então adicionar um elemento `<select>` que contenha todos os arquivos na raiz do servidor adicionando elementos `<option>` contendo as linhas retornadas pela requisição `GET` para a URL `/`. Quando um usuário seleciona outro arquivo (um evento `"change"` nesse campo), o script deve buscar e mostrar o arquivo. Também tenha certeza que quando salvar um arquivo, você esteja usando o nome do arquivo selecionado.

Infelizmente, o servidor é muito simplista para ser capaz de ler arquivos de subdiretórios de forma confiável, uma vez que ele não nos diz se a coisa que está sendo buscado com uma requisição `GET` é um arquivo ou um diretório. Você consegue pensar em uma maneira de estender o servidor para solucionar isso?

Projeto - Website de compartilhamento de habilidades

Uma reunião de compartilhamento de habilidades é um evento onde as pessoas com um interesse em comum se juntam e dão pequenas apresentações informais sobre coisas que eles sabem. Em uma reunião de compartilhamento de habilidade de jardinagem alguém pode explicar como cultivar um Aipo. Ou em um grupo de compartilhamento de habilidades orientadas para a programação você poderia aparecer e dizer a todos sobre Node.js.

Tais reuniões muitas vezes também são chamados de grupos de usuários quando eles estão falando sobre computadores. Isso é uma ótima maneira de aprofundar o seu conhecimento e aprender sobre novos desenvolvimentos ou simplesmente reunir pessoas com interesses semelhantes. Muitas cidades têm grandes grupos de JavaScript. Eles são tipicamente livre para assistir ou visitar.

Neste último capítulo do projeto o nosso objetivo é a criação de um site para gerenciar estas palestras dadas em um encontro de compartilhamento de habilidade. Imagine um pequeno grupo de pessoas que se encontra regularmente no escritório de um dos membros para falar sobre Monociclo. O problema é que quando um organizador de alguma reunião anterior muda de cidade ninguém se apresentará para assumir esta tarefa. Queremos um sistema que permite que os participantes proponha e discuta as palestras entre si sem um organizador central.



Assim como no capítulo anterior, o código neste capítulo é escrito em Node.js, e executá-lo diretamente em uma página HTML é improvável que funcione. O código completo para o projeto pode ser baixado [aqui](#).

Projeto

Há uma parte do servidor para este projeto escrito em Node.js e a outra parte do cliente escrito para o browser. O servidor armazena os dados do sistema e fornece para o cliente. Ela também serve os arquivos HTML e JavaScript que implementam o sistema do lado do cliente.

O servidor mantém uma lista de palestras propostas para a próxima reunião e o cliente mostra esta lista. Cada palestra tem um nome do apresentador, um título, um resumo e uma lista de comentários dos participantes. O cliente permite que os usuários proponha novas palestras (adicionando a lista), exclua as palestras e comente sobre as palestras existentes. Sempre que o usuário faz tal mudança o cliente faz uma solicitação `HTTP` para informar para o servidor o que fazer.

Your name:

Unituning
by Carlos

Modifying your cycle for extra style

Alice: Will you talk about raising a cycle?
Carlos: Definitely!
Alice: I'll be there

Submit a talk

Title:

Summary:

O aplicativo será configurado para mostrar uma exibição em tempo real das atuais palestras propostas e seus comentários. Sempre que alguém apresentar uma nova palestra ou adicionar um comentário, todas as pessoas que têm a página aberta no navegador devem visualizar a mudança imediatamente. Isto coloca um pouco de um desafio, pois não há `path` para um servidor web abrir uma conexão com um cliente nem há uma boa maneira de saber o que os clientes estão olhando atualmente no site.

Uma solução comum para este problema é chamado de `long polling` que passa a ser uma das motivações para o projeto ser em Node.

Long polling

Para ser capaz de se comunicar imediatamente com um cliente que algo mudou precisamos de uma conexão com o cliente. Os navegadores não tradicionais, bloqueiam de qualquer maneira tais conexões que deveriam ser aceitas pelo cliente; toda essa ligação deve ser feita via servidor o que não é muito prático.

Nós podemos mandar o cliente abrir a conexão e mantê-la de modo que o servidor possa usá-la para enviar informações quando for preciso.

Uma solicitação `HTTP` permite apenas um fluxo simples de informações, onde o cliente envia a solicitação e o servidor devolve uma única resposta. Há uma tecnologia que chamamos de soquetes web, suportado pelos navegadores modernos, isso torna possível abrir as ligações para a troca de dados arbitrária. É um pouco difícil usá-las corretamente.

Neste capítulo vamos utilizar uma técnica relativamente simples, `long polling`, onde os clientes continuamente pedem ao servidor para obter novas informações usando solicitações `HTTP` e o servidor simplesmente barrará sua resposta quando ele não houver nada de novo para relatar.

Enquanto o cliente torna-se constantemente um `long polling` aberto, ele irá receber informações do servidor imediatamente. Por exemplo, se Alice tem o nosso aplicativo de compartilhamento de habilidade aberto em seu navegador, ele terá feito um pedido de atualizações e estará a espera de uma resposta a esse pedido. Quando Bob submeter uma palestra sobre a `extrema Downhill Monociclo` o servidor vai notificar que Alice está esperando por atualizações e enviar essas informações sobre a nova palestra como uma resposta ao seu pedido pendente. O navegador de Alice receberá os dados e atualizará a tela para mostrar a nova palestra.

Para evitar que as conexões excedam o tempo limite (sendo anulado por causa de uma falta de atividade) podemos definir uma técnica que define um tempo máximo para cada pedido do `long polling`; após esse tempo o servidor irá responder de qualquer maneira mesmo que ele não tenha nada a relatar, daí então o cliente inicia um novo pedido.

Reiniciar o pedido periodicamente torna a técnica mais robusta a qual permite aos clientes se recuperarem de falhas de conexão temporárias ou de problemas no servidor.

Um servidor que esta ocupado usando `long polling` pode ter milhares de pedidos em espera com conexões `TCP` em aberto. Node torna fácil de gerenciar muitas conexões sem criar uma thread separada com controle para cada uma, sendo assim, Node é uma boa opção para esse sistema.

Interface HTTP

Antes de começarmos a comunicar servidor e cliente vamos pensar sobre o ponto em que é feita a comunicação: a interface `HTTP`.

Vamos basear nossa interface em `JSON` e como vimos no servidor de arquivos a partir do capítulo 20 vamos tentar fazer um bom uso dos métodos `HTTP`. A interface é centrado em torno de um path `/talks`. `Paths` que não começam com `/talks` serão usado para servir arquivos estáticos como: código HTML e JavaScript que serão implementados no sistema do lado do cliente.

A solicitação do tipo `GET` para `/talks` devolve um documento `JSON` como este:

```
{"serverTime": 1405438911833,
 "talks": [{"title": "Unituning",
             "presenter": "Carlos",
             "summary": "Modifying your cycle for extra style",
             "comment": []}]}
```

O campo `serverTime` vai ser usado para fazer a sondagem de `long polling`. Voltarei a explicar isso mais adiante.

Para criar um novo talk é preciso uma solicitação do tipo `PUT` para a URL `/talks/unituning/`, onde após a segunda barra é o título da palestra. O corpo da solicitação `PUT` deve conter um objeto `JSON` que tem o apresentador e o sumário como propriedade do corpo da solicitação.

O títulos da palestra pode conter espaços e outros caracteres que podem não aparecerem normalmente em um URL, a `string` do título deve ser codificado com a função `encodeURIComponent` ao construir a URL.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

O pedido para criação de uma palestra é parecido com isto:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Dana",
 "summary": "Standing still on a unicycle"}
```

Estas URLs também suportam requisições `GET` para recuperar a representação do `JSON` de uma palestra ou `DELETE` para exclusão de uma palestra.

Para adicionar um comentário a uma palestra é necessário uma solicitação `POST` para uma URL `/talks/Unituning/comments` com um objeto `JSON` contendo o autor e a mensagem como propriedades do corpo da solicitação.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72
```

```
{"author": "Alice",  
  "message": "Will you talk about raising a cycle?"}
```

Para termos apoio do `long polling` precisamos de pedidos `GET` para `/talks`. Podemos incluir um parâmetro de consulta chamado `changesSince` ele será usado para indicar que o cliente está interessado em atualizações que aconteceram desde de um determinado tempo. Quando existem tais mudanças eles são imediatamente devolvidos. Quando não há a resposta é adiada até que algo aconteça em um determinado período de tempo (vamos determinar 90 segundos).

O tempo deve ser indicado em números por milissegundos decorridos desde do início de 1970, o mesmo tipo de número que é retornado por `Date.now()`. Para garantir que ele recebeu todas as atualizações sem receber a mesma atualização repetida; o cliente deve passar o tempo da última informação recebida ao servidor. O relógio do servidor pode não ser exatamente sincronizado com o relógio do cliente e mesmo se fosse seria impossível para o cliente saber a hora exata em que o servidor enviou uma resposta porque a transferência de dados através de rede pode ter um pouco de atraso.

Esta é a razão da existência da propriedade `serverTime` em respostas enviadas a pedidos `GET` para `/talks`. Essa propriedade diz ao cliente o tempo preciso do servidor em que os dados foram recebidos ou criados. O cliente pode então simplesmente armazenar esse tempo e passá-los no seu próximo pedido de `polling` para certificar de que ele vai receber exatamente as atualizações que não tenha visto antes.

```
GET /talks?changesSince=1405438911833 HTTP/1.1
```

```
(time passes)
```

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Content-Length: 95
```

```
{"serverTime": 1405438913401,  
  "talks": [{"title": "Unituning",  
             "deleted": true}]}
```

Quando a palestra é alterada, criada ou tem um comentário adicionado; a representação completa da palestra estará incluída na próxima resposta de solicitação que o cliente busca. Quando a palestra é excluída, somente o seu título e a propriedade que será excluída da onde ela estará incluída. O cliente pode então adicionar os títulos das palestras que não estão sendo exibidas na página, atualizar as que já estão sendo exibidas ou remover aquelas que foram excluídas.

O protocolo descrito neste capítulo não irá fazer qualquer controle de acesso. Todos podem comentar, modificar a palestras e até mesmo excluir. Uma vez que a Internet está cheia de arruaceiros ao colocarmos um sistema on-line sem proteção adicional é provável que acabe em um desastre.

Uma solução simples seria colocar um sistema de `proxy` reverso por trás, sendo ele um servidor `HTTP` que aceita conexões de fora do sistema e os encaminha para servidores `HTTP` que estão sendo executados localmente. O `proxy` pode ser configurado para exigir um nome de usuário e senha, você pode ter certeza de que somente os participantes do grupo de compartilhamento de habilidade tenham essa senha.

O serviço

Vamos começar a escrever código do lado do servidor. O código desta seção é executado em `Node.js`.

Roteamento

O nosso servidor irá utilizar `http.createServer` para iniciar um servidor HTTP. Na função que lida com um novo pedido, iremos distinguir entre os vários tipos de solicitações (conforme determinado pelo método e o `path`) que suportamos. Isso pode ser feito com uma longa cadeia de `if` mas há uma maneira mais agradável.

As rotas é um componente que ajuda a enviar uma solicitação através de uma função. Você pode dizer para as rotas que os pedidos combine com um `path` usando expressão regular `/^\/talks\/([^\/]*)$/` (que corresponde a `/talks/` seguido pelo título) para tratar por uma determinada função. Isso pode ajudar a extrair as partes significativas de um `path`, neste caso o título da palestra, que estará envolto entre os parênteses na expressão regular, após disto é passado para o manipulador de função.

Há uma série de bons pacotes de roteamento na `NPM` mas vamos escrever um nós mesmos para ilustrar o princípio.

Este é `router.js` que exigirá mais tarde do nosso módulo de servidor:

```
var Router = module.exports = function() {
  this.routes = [];
};

Router.prototype.add = function(method, url, handler) {
  this.routes.push({method: method,
    url: url,
    handler: handler});
};

Router.prototype.resolve = function(request, response) {
  var path = require("url").parse(request.url).pathname;

  return this.routes.some(function(route) {
    var match = route.url.exec(path);
    if (!match || route.method !== request.method)
      return false;

    var urlParts = match.slice(1).map(decodeURIComponent);
    route.handler.apply(null, [request, response]
      .concat(urlParts));

    return true;
  });
};
```

O módulo exporta o construtor de `Router`. Um objeto de `Router` permite que novos manipuladores sejam registrados com o método `add` e resolver os pedidos com o método `resolve`.

Este último irá retornar um `booleano` que indica se um manipulador foi encontrado. Há um método no conjunto de rotas que tenta uma rota de cada vez (na ordem em que elas foram definidos) e retorna a verdadeira quando alguma for correspondida.

As funções de manipulação são chamadas com os objetos de solicitação e resposta. Quando algum grupo da expressão regular corresponder a uma URL, as `string` que correspondem são passadas para o manipulador como argumentos extras. Essas sequências tem que ser uma URL decodificada tendo a URL codificada assim `%20-style code`.

Servindo arquivos

Quando um pedido não corresponde a nenhum dos tipos de solicitação que esta sendo definidos em nosso `router` o servidor deve interpretar como sendo um pedido de um arquivo que esta no diretório público. Seria possível usar o servidor de arquivos feito no Capítulo 20 para servir esses arquivos; nenhuma destas solicitações sera do tipo `PUT` e `DELETE`, nós gostaríamos de ter recursos avançados como suporte para armazenamento em cache. Então vamos usar um servidor de arquivo estático a partir de uma NPM.

Optei por `ecstatic`. Este não é o único tipo de servidor NPM, mas funciona bem e se encaixa para nossos propósitos. O módulo de `ecstatic` exporta uma função que pode ser chamada com um objeto de configuração para produzir uma função de manipulação de solicitação. Nós usamos a opção `root` para informar ao servidor onde ele deveria procurar pelos arquivos. A função do manipulador aceita solicitação e resposta através de parâmetros que pode ser passado diretamente para `createServer` onde é criado um servidor que serve apenas arquivos. Primeiro verificamos se na solicitações não ha nada de especial, por isso envolvemos em uma outra função.

```
var http = require("http");
var Router = require("./router");
var ecstatic = require("ecstatic");

var fileServer = ecstatic({root: "./public"});
var router = new Router();

http.createServer(function(request, response) {
  if (!router.resolve(request, response))
    fileServer(request, response);
}).listen(8000);
```

`response` e `respondJSON` serão funções auxiliares utilizadas em todo o código do servidor para ser capaz de enviar as respostas com uma única chamada de função.

```
function respond(response, status, data, type) {
  response.writeHead(status, {
    "Content-Type": type || "text/plain"
  });
  response.end(data);
}

function respondJSON(response, status, data) {
  respond(response, status, JSON.stringify(data),
    "application/json");
}
```

Recursos das palestras

O servidor mantém as palestras que têm sido propostas em um objeto chamado `talks`, cujos os títulos são propriedades de nomes de uma palestra. Estes serão expostos como recursos `HTTP` sob `/talks/[title]` e por isso precisamos adicionar manipuladores ao nosso roteador que implementaram vários métodos que podem serem utilizados pelo o cliente.

O manipulador de solicitações serve uma única resposta, quer seja alguns dados do tipo `JSON` da palestra, uma resposta de 404 ou um erro.

```
var talks = Object.create(null);

router.add("GET", /^\/talks\/([^\/]*)$/,
  function(request, response, title) {
    if (title in talks)
      respondJSON(response, 200, talks[title]);
    else
      respond(response, 404, "No talk '" + title + "' found");
  });
```

A exclusão de um `talk` é feito para remove-la do objeto de `talks`.

```
router.add("DELETE", /^\/talks\/([^\/]*)$/,
  function(request, response, title) {
    if (title in talks) {
```

```

    delete talks[title];
    registerChange(title);
  }
  respond(response, 204, null);
});

```

A função `registerChange` que iremos definir; notifica alterações enviando uma solicitação de `long polling` ou simplesmente espera.

Para ser capaz de obter facilmente o conteúdo do `body` de uma solicitação de `JSON`, teremos que definir uma função chamada `readStreamAsJSON` que lê todo o conteúdo de um `stream`, analisa o `JSON` e em seguida chama uma função de retorno.

```

function readStreamAsJSON(stream, callback) {
  var data = "";
  stream.on("data", function(chunk) {
    data += chunk;
  });
  stream.on("end", function() {
    var result, error;
    try { result = JSON.parse(data); }
    catch (e) { error = e; }
    callback(error, result);
  });
  stream.on("error", function(error) {
    callback(error);
  });
}

```

Um manipulador que precisa ler respostas JSON é o manipulador `PUT` que é usado para criar novas palestras. Nesta `request` devemos verificar se os dados enviados tem um apresentador e o título como propriedades ambos do tipo `String`. Quaisquer dados que vêm de fora do sistema pode conter erros e nós não queremos corromper o nosso modelo de dados interno ou mesmo travar quando os pedidos ruins entrarem.

Se os dados se parecem válidos o manipulador armazena-os em um novo objeto que representa uma nova palestra no objeto `talks`, possivelmente substituindo uma palestra que já exista com este título e mais uma vez chama `registerChange`.

```

router.add("PUT", /^\/talks\/([^\/]*)$/,
  function(request, response, title) {
    readStreamAsJSON(request, function(error, talk) {
      if (error) {
        respond(response, 400, error.toString());
      } else if (!talk ||
        typeof talk.presenter !== "string" ||
        typeof talk.summary !== "string") {
        respond(response, 400, "Bad talk data");
      } else {
        talks[title] = {title: title,
          presenter: talk.presenter,
          summary: talk.summary,
          comments: []};
        registerChange(title);
        respond(response, 204, null);
      }
    });
  });
});

```

Para adicionar um comentário a uma palestra, funciona de forma semelhante. Usamos `readStreamAsJSON` para obter o conteúdo do pedido, validar os dados resultantes e armazená-los como um comentário quando for válido.

```

router.add("POST", /^\/talks\/([^\s]+)\/comments$/,
  function(request, response, title) {
    readStreamAsJSON(request, function(error, comment) {
      if (error) {
        respond(response, 400, error.toString());
      } else if (!comment ||
        typeof comment.author !== "string" ||
        typeof comment.message !== "string") {
        respond(response, 400, "Bad comment data");
      } else if (title in talks) {
        talks[title].comments.push(comment);
        registerChange(title);
        respond(response, 204, null);
      } else {
        respond(response, 404, "No talk '" + title + "' found");
      }
    });
  });
});

```

Ao tentar adicionar um comentário a uma palestra inexistente é claro que devemos retornar um erro 404.

Apoio a long polling

O aspecto mais interessante do servidor é a parte que trata de `long polling`. Quando uma requisição `GET` chega para `/talks` pode ser um simples pedido de todas as palestras ou um pedido de atualização com um parâmetro `changesSince`.

Haverá várias situações em que teremos que enviar uma lista de palestra para o cliente de modo que primeiro devemos definir uma pequena função auxiliar que atribuirá um campo `servertime` para tais respostas.

```

function sendTalks(talks, response) {
  respondJSON(response, 200, {
    serverTime: Date.now(),
    talks: talks
  });
}

```

O manipulador precisa olhar para os parâmetros de consulta da `URL` do pedido para ver se o parâmetro `changesSince` foi enviado. Se você entregar a `url` para o módulo da função `parse` teremos um segundo argumento que será `true`; também teremos que analisar parte por parte de uma URL. Se o objeto que ele retornou tem uma propriedade `query` removemos o outro objeto que mapeia os parâmetros de nomes para os valores.

```

router.add("GET", /^\/talks$/, function(request, response) {
  var query = require("url").parse(request.url, true).query;
  if (query.changesSince == null) {
    var list = [];
    for (var title in talks)
      list.push(talks[title]);
    sendTalks(list, response);
  } else {
    var since = Number(query.changesSince);
    if (isNaN(since)) {
      respond(response, 400, "Invalid parameter");
    } else {
      var changed = getChangedTalks(since);
      if (changed.length > 0)
        sendTalks(changed, response);
      else
        waitForChanges(since, response);
    }
  }
});

```

Quando o parâmetro `changesSince` não é enviado, o manipulador simplesmente acumula uma lista de todas as palestras e retorna.

Caso contrário o parâmetro `changeSince` tem que ser verificado primeiro para certificar-se de que é um número válido. A função `getChangedTalks` a ser definido em breve retorna um `array` de palestras que mudaram desde um determinado tempo. Se retornar um `array` vazio significa que o servidor ainda não tem nada para armazenar no objeto de resposta e retorna de volta para o cliente (usando `waitForChanges`), o que pode também ser respondida em um momento posterior.

```
var waiting = [];

function waitForChanges(since, response) {
  var waiter = {since: since, response: response};
  waiting.push(waiter);
  setTimeout(function() {
    var found = waiting.indexOf(waiter);
    if (found > -1) {
      waiting.splice(found, 1);
      sendTalks([], response);
    }
  }, 90 * 1000);
}
```

O método `splice` é utilizado para cortar um pedaço de um `array`. Você dá um índice e uma série de elementos para transforma é um `array` removendo o restante dos elementos após o índice dado. Neste caso nós removeremos um único elemento do objeto que controla a resposta de espera cujo índice encontramos pelo `indexOf`. Se você passar argumentos adicionais para `splice` seus valores serão inseridas no `array` na posição determinada substituindo os elementos removidos.

Quando um objeto de resposta é armazenado no `array` de espera o tempo é ajustado imediatamente. Determinamos 90 segundos para ser o tempo limite do pedido, caso ele ainda estiver a espera ele envia uma resposta de `array` vazio e remove a espera.

Para ser capaz de encontrar exatamente essas palestras que foram alterados desde um determinado tempo precisamos acompanhar o histórico de alterações. Registrando a mudança com `registerChange`, podemos escutar as mudança juntamente com o tempo atual do `array` chamado de `waiting`. Quando ocorre uma alteração isso significa que há novos dados, então todos os pedidos em espera podem serem respondidos imediatamente.

```
var changes = [];

function registerChange(title) {
  changes.push({title: title, time: Date.now()});
  waiting.forEach(function(waiter) {
    sendTalks(getChangedTalks(waiter.since), waiter.response);
  });
  waiting = [];
}
```

Finalmente `getChangedTalks` poderá usar o `array` de mudanças para construir uma série de palestras alteradas, incluindo no objetos uma propriedade de `deleted` para as palestras que não existem mais. Ao construir esse `array`, `getChangedTalks` tem de garantir que ele não incluiu a mesma palestra duas vezes; isso pode acontecer se houver várias alterações em uma palestra desde o tempo dado.

```
function getChangedTalks(since) {
  var found = [];
  function alreadySeen(title) {
    return found.some(function(f) {return f.title == title;});
  }
```



```

    }
    for (var i = changes.length - 1; i >= 0; i--) {
      var change = changes[i];
      if (change.time <= since)
        break;
      else if (alreadySeen(`change.title`))
        continue;
      else if (change.title in talks)
        found.push(talks[change.title]);
      else
        found.push({title: change.title, deleted: true});
    }
    return found;
  }
}

```

Aqui concluímos o código do servidor. Executando o programa definido até agora você vai ter um servidor rodando na porta `8000` que serve arquivos do subdiretório `public` ao lado de uma interface de gerenciamento de palestras sob a URL `/talks`.

O cliente

A parte do cliente é onde vamos gerenciar as palestras, basicamente isso consiste em três arquivos: uma página HTML, uma folha de estilo e um arquivo JavaScript.

HTML

Servir arquivos com o nome de `index.html` é uma convenção amplamente utilizado para servidores web quando uma solicitação é feita diretamente de um `path`, onde corresponde a um diretório. O módulo de servidor de arquivos que usamos foi o `ecstatic`, ele suporta esta convenção. Quando um pedido é feito para o `path /` o servidor procura pelo arquivo em `./public/index.html` (`./public` é a raiz que especificamos) e retorna esse arquivo se for encontrado.

Se quisermos uma página para mostrar quando um navegador estiver apontado para o nosso servidor devemos colocá-la em `public/index.html`. Esta é a maneira que o nosso arquivo `index` irá começar:

```

<!doctype html>

<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill sharing</h1>

<p>Your name: <input type="text" id="name"></p>

<div id="talks"></div>

```

Ele define o título do documento e inclui uma folha de estilo que define alguns estilos, adicionei uma borda em torno das palestras. Em seguida ele adiciona um `input` de nome; onde é esperado que o usuário coloque seu nome para que ele possa ser redirecionado para a observação das palestras.

O elemento `<div>` com o `id "talks"` conterá a lista atual de todas as palestras. O `script` preenche a lista quando recebe as palestras do servidor.

Segue o formulário que é usado para criar uma nova palestra:

```

<form id="newtalk">
  <h3>Submit a talk</h3>
  Title: <input type="text" style="width: 40em" name="title">
  <br>

```

```
Summary: <input type="text" style="width: 40em" name="summary">
<button type="submit">Send</button>
</form>
```

Um `script` irá adicionar um manipulador de evento `"submit"` para este formulário, a partir do qual ele fará a solicitação `HTTP` que informará ao servidor sobre a palestra.

Em seguida, vem um bloco bastante misterioso, que tem seu estilo de exibição definido como `none`, impedindo que ele apareça na página. Você consegue adivinhar para o que serve?

```
<div id="template" style="display: none">
  <div class="talk">
    <h2>{{title}}</h2>
    <div>by <span class="name">{{presenter}}</span></div>
    <p>{{summary}}</p>
    <div class="comments"></div>
    <form>
      <input type="text" name="comment">
      <button type="submit">Add comment</button>
      <button type="button" class="del">Delete talk</button>
    </form>
  </div>
  <div class="comment">
    <span class="name">{{author}}</span>: {{message}}
  </div>
</div>
```

Criar estruturas de DOM com JavaScript é complicado e produz um código feio. Você pode tornar o código um pouco melhor através da introdução de funções auxiliares como a função `elt` do capítulo 13, mas o resultado vai ficar ainda pior do que no HTML que foi pensado como uma linguagem de domínio específico para expressar estruturas do DOM.

Para criar uma estrutura DOM para as palestras, o nosso programa vai definir um sistema de `templates` simples que utiliza estruturas incluídas no DOM que estejam escondidas no documento para instanciar novas estruturas, substituindo os espaços reservados entre chaves duplas para os valores de uma palestra em específico.

Por fim, o documento `HTML` inclui um arquivo de `script` que contém o código do lado do cliente.

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

O início

A primeira coisa que o cliente tem que fazer quando a página é carregada é pedir ao servidor um conjunto atual de palestras. Uma vez que estamos indo fazer um monte de solicitações `HTTP`, vamos novamente definir um pequeno invólucro em torno `XMLHttpRequest` que aceita um objeto para configurar o pedido, bem como um `callback` para chamar quando o pedido for concluído.

```
function request(options, callback) {
  var req = new XMLHttpRequest();
  req.open(options.method || "GET", options.pathname, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(null, req.responseText);
    else
      callback(new Error("Request failed: " + req.statusText));
  });
  req.addEventListener("error", function() {
    callback(new Error("Network error"));
  });
  req.send(options.body || null);
}
```

```
}
```

O pedido inicial mostra as palestras que recebeu na tela e inicia o processo de `long polling` chamando o método `waitForChanges`.

```
var lastServerTime = 0;

request({pathname: "talks"}, function(error, response) {
  if (error) {
    reportError(error);
  } else {
    response = JSON.parse(response);
    displayTalks(response.talks);
    lastServerTime = response.serverTime;
    waitForChanges();
  }
});
```

A variável `lastServerTime` é usado para controlar o tempo da última atualização que foi recebido do servidor. Após o pedido inicial as palestras exibidas pelo cliente correspondem ao tempo da resposta das palestras que foram devolvidas pelo servidor. Assim a propriedade `serverTime` que foi incluída na resposta fornece um valor inicial apropriado para `lastServerTime`.

Quando a solicitação falhar nós não queremos que a nossa página não faça nada sem explicação. Assim definimos uma função simples chamada de `reportError` que pelo menos irá mostra ao usuário uma caixa de diálogo que diz que algo deu errado.

```
function reportError(error) {
  if (error)
    alert(error.toString());
}
```

A função verifica se existe um erro real, alertando somente quando houver. Dessa forma podemos passar diretamente esta função para solicitar pedidos onde podemos ignorar a resposta. Isso garante que se a solicitação falhar o erro será relatado ao usuário.

Resultados das palestras

Para ser capaz de atualizar a visualização das palestras quando as mudanças acontecem, o cliente deve se manter atualizado das palestras que estão sendo mostradas. Dessa forma quando uma nova versão de uma palestra que já está na tela sofre atualizações ela deve ser substituído pela atual. Da mesma forma quando a informação que vem de uma palestra deve ser eliminada o elemento do DOM pode ser removido direto do documento.

A função `displayTalks` é usada tanto para construir a tela inicial tanto para atualizá-la quando algo muda. Ele vai usar o objeto `shownTalks` que associa os títulos da palestras com os nós do DOM para lembrar das palestras que se tem atualmente na tela.

```
var talkDiv = document.querySelector("#talks");
var shownTalks = Object.create(null);

function displayTalks(talks) {
  talks.forEach(function(talk) {
    var shown = shownTalks[talk.title];
    if (talk.deleted) {
      if (shown) {
        talkDiv.removeChild(shown);
        delete shownTalks[talk.title];
      }
    }
  });
}
```

```

    } else {
      var node = drawTalk(talk);
      if (shown)
        talkDiv.replaceChild(node, shown);
      else
        talkDiv.appendChild(node);
      shownTalks[talk.title] = node;
    }
  });
}

```

Para construir a estrutura DOM para as palestras usaremos os `templates` que foram incluídas no documento HTML. Primeiro temos que definir o método `instantiateTemplate` que verifica e preenche com um `template`.

O parâmetro `name` é o nome do `template`. Para buscar o elemento de `templates` buscamos um elemento cujo nome da classe corresponda ao nome do `template` que é o filho do elemento com id do `template`. O método `querySelector` facilita essa busca. Temos `templates` nomeados como `talk` e `comment` na página HTML.

```

function instantiateTemplate(name, values) {
  function instantiateText(text) {
    return text.replace(/\{\{(\w+)\}\}/g, function(_, name) {
      return values[name];
    });
  }
  function instantiate(node) {
    if (node.nodeType == document.ELEMENT_NODE) {
      var copy = node.cloneNode();
      for (var i = 0; i < node.childNodes.length; i++)
        copy.appendChild(instantiate(node.childNodes[i]));
      return copy;
    } else if (node.nodeType == document.TEXT_NODE) {
      return document.createTextNode(
        instantiateText(node.nodeValue));
    }
  }

  var template = document.querySelector("#template ." + name);
  return instantiate(template);
}

```

O método `cloneNode` cria uma cópia de um nó. Ele não copia os filhos do nó a menos que `true` seja enviado como primeiro argumento. A função instancia recursivamente uma cópia do `template` preenchendo onde o `template` deve aparecer.

O segundo argumento para `instantiateTemplate` deve ser um objecto cujas propriedades são `strings` com os mesmos atributos que estão presente no `teplate`. Um espaço reservado como `{{title}}` será substituído com o valor da propriedade do atributo `title`.

Esta é uma abordagem básica para a implementação de um `template` mas suficiente para implementar o `drawTalk`.

```

function drawTalk(talk) {
  var node = instantiateTemplate("talk", talk);
  var comments = node.querySelector(".comments");
  talk.comments.forEach(function(comment) {
    comments.appendChild(
      instantiateTemplate("comment", comment));
  });

  node.querySelector("button.del").addEventListener(
    "click", deleteTalk.bind(null, talk.title));

  var form = node.querySelector("form");
  form.addEventListener("submit", function(event) {

```

```

    event.preventDefault();
    addComment(talk.title, form.elements.comment.value);
    form.reset();
  });
  return node;
}

```

Depois de instanciar o `template` de `talk` há várias coisas que precisamos fazermos. Em primeiro lugar os comentários têm que ser preenchido pelo `template comments` e anexar os resultados no nó da classe `commnets`. Em seguida os manipuladores de eventos tem que anexar um botão que apaga a palestra e um formulário que adiciona um novo comentário.

Atualizando o servidor

Os manipuladores de eventos registrados pela `drawTalk` chamam a função `deleteTalk` e `addComment` para executar as ações necessárias para excluir uma palestra ou adicionar um comentário. Estes terão de construir as URLs que se referem as palestras com um determinado título para o qual se define a função auxiliar de `talkURL`.

```

function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}

```

A função `deleteTalk` dispara uma requisição `DELETE` e informa o erro quando isso falhar.

```

function deleteTalk(title) {
  request({pathname: talkURL(title), method: "DELETE"},
    reportError);
}

```

Adicionar um comentário requer a construção de uma representação JSON dos comentários e delegar que ele seja parte de um pedido `POST`.

```

function addComment(title, comment) {
  var comment = {author: nameField.value, message: comment};
  request({pathname: talkURL(title) + "/comments",
    body: JSON.stringify(comment),
    method: "POST"},
    reportError);
}

```

A variável `nameField` é usado para definir a propriedade autor de um comentário com referência no campo `<input>` na parte superior da página que permite que o usuário especifique o seu nome. Nós também inserimos o nome no `localStorage` para que ele não tem que ser preenchido novamente a cada vez que a página é recarregada.

```

var nameField = document.querySelector("#name");

nameField.value = localStorage.getItem("name") || "";

nameField.addEventListener("change", function() {
  localStorage.setItem("name", nameField.value);
});

```

O formulário na parte inferior da página propoe uma nova palestra, ele recebe um manipulador de evento `"submit"`. Este manipulador impede o efeito padrão do evento (o que causaria um recarregamento da página) passando a ter o comportamento de disparar uma solicitação `PUT` para criar uma palestra e limpar o formulário.

```
var talkForm = document.querySelector("#newtalk");

talkForm.addEventListener("submit", function(event) {
  event.preventDefault();
  request({pathname: talkURL(talkForm.elements.title.value),
    method: "PUT",
    body: JSON.stringify({
      presenter: nameField.value,
      summary: talkForm.elements.summary.value
    }), reportError);
  talkForm.reset();
});
```

Notificando mudanças

Gostaria de salientar que as várias funções que alteram o estado do pedido de criação, exclusão da palestras ou a adição de um comentário não fazem absolutamente nada para garantir que as mudanças que eles fazem sejam visíveis na tela. Eles simplesmente dizem ao servidor que contam com o mecanismo de `long polling` para acionar as atualizações apropriadas para a página.

Dado o mecanismo que implementamos em nosso servidor e da maneira que definimos `displayTalks` para lidar com atualizações das palestras que já estão na página, o `long polling` é surpreendentemente simples.

```
function waitForChanges() {
  request({pathname: "talks?changesSince=" + lastServerTime},
    function(error, response) {
      if (error) {
        setTimeout(waitForChanges, 2500);
        console.error(error.stack);
      } else {
        response = JSON.parse(response);
        displayTalks(response.talks);
        lastServerTime = response.serverTime;
        waitForChanges();
      }
    });
}
```

Esta função é chamada uma vez quando o programa inicia e em seguida continua a chamar assegurando que um pedido de `polling` esteja sempre ativo. Quando a solicitação falhar não podemos chamar o método `reportError` pois se o servidor cair a cada chamada uma popup irá aparecer para o usuário deixando nosso programa bem chato de se usar. Em vez disso o `output` do erro deverá aparecer no console (para facilitar a depuração) e uma outra tentativa é feita em dois segundos e meio depois.

Quando o pedido for bem-sucedido os novos dados é colocado na tela e `lastServerTime` é atualizado para refletir o fato de que recebemos dados correspondentes nesse novo momento. O pedido é imediatamente reiniciado para esperar pela próxima atualização.

Se você executar o servidor e abrir duas janelas do navegador em `localhost:8000/` um ao lado do outro você vai observar que as ações que você executa em uma janela são imediatamente visíveis no outro.

Exercícios

Os exercícios a seguir vai envolver uma modificação definida neste capítulo. Para trabalhar com elas certifique-se de baixar o [código primeiro](#) e ter instalado [Node](#).

Persistência no disco

O servidor de compartilhamento de habilidade mantém seus dados puramente na memória. Isto significa que quando o servidor travar ou reiniciar por qualquer motivo todas as palestras e comentários serão perdidos.

Estenda o servidor e faça ele armazenar os dados da palestra em disco. Ache uma forma automática de recarrega os dados quando o servidor for reiniciado. Não se preocupe com performance faça o mais simples possível e funcional.

Dica:

A solução mais simples que posso dar para você é transformar todas as palestras em objeto `JSON` e coloca-las em um arquivo usando `fs.writeFile`. Já existe uma função (`registerChange`) que é chamada toda vez que temos alterações no servidor. Ela pode ser estendida para escrever os novos dados no disco.

Escolha um nome para o arquivo, por exemplo `./talks.json`. Quando o servidor é iniciado ele pode tentar ler esse arquivo com `fs.readFile` e se isso for bem sucedido o servidor pode usar o conteúdo de arquivo como seus dados iniciais.

Porém cuidado, as palestras começam como um protótipo menos como um objeto para que possa ser operado normalmente. `JSON.parse` retorna objetos regulares com `Object.prototype` como sendo do seu protótipo. Se você usar `JSON` como formato de arquivo você terá que copiar as propriedades do objeto retornados por `JSON.parse` em um novo objeto.

Melhorias nos templates

A maioria dos sistemas de templates fazem mais do que apenas preencher algumas strings. No mínimo permitem a inclusão de condicional em alguma parte do template como `if` ou uma repetição de um pedaço de template semelhante a um `while`.

Se formos capazes de repetir um pedaço de template para cada elemento em um `array` não precisaremos de um segundo template (`comment`). Em vez disso poderíamos especificar que o template `talk` verifica os conjuntos das propriedade de uma palestra e dos comentários realizando uma interação para cada comentário que esteja no `array`.

Ele poderia ser assim:

```
<div class="comments">
  <div class="comment" template-repeat="comments">
    <span class="name">{{author}}</span>: {{message}}
  </div>
</div>
```

A idéia é que sempre que um nó com um atributo `template-repeat` é encontrado durante a instânciação do template, o código faz um loop sobre o `array` na propriedade chamada por esse atributo. Para cada elemento do `array` ele adiciona um exemplo de nó. O contexto do template (a variável com valores em `instantiateTemplate`) durante este ciclo aponta para o elemento atual do `array` para que `{{author}}` seja o objeto de um comentário e não o contexto original do objeto `talk`.

Reescreva `instantiateTemplate` para implementar isso e em seguida altere os templates para usar este recurso e remover a prestação explícita dos comentários na função `drawTalk`.

Como você gostaria de acrescentar a instânciação condicional de nós tornando-se possível omitir partes do template quando um determinado valor é falso ou verdadeiro?

Dica:

Você poderia mudar `instantiateTemplate` de modo que sua função interna não tenha apenas um nó mas também um contexto atual como um argumento. Você pode verificar se no loop sobre os nós filhos de um nó exista um atributo filho em `template-repeat`. Se isso acontecer não instancie, em vez de um loop sobre o `array` indicado pelo

valor do atributo, faça uma instancia para cada elemento do `array` passando o elemento atual como contexto.

Condicionais pode ser implementado de uma forma semelhante aos atributos de chamadas, por exemplo `template-when` e `template-unless` quando inserido no template irá instanciar ou não um nó dependendo de uma determinada propriedade que pode ser verdadeiro ou falso.

Os unscriptables

Quando alguém visita o nosso site com um navegador que tenha JavaScript desabilitado ou o navegador que não suporta a execução de JavaScript eles vão conseguir ver uma página inoperável e completamente quebrada. Isso não é bom.

Alguns tipos de aplicações web realmente não pode ser feito sem JavaScript. Para outros você simplesmente não tem o orçamento ou paciência para se preocupar com os clientes que não podem executar scripts. Mas para páginas com um grande público é uma forma educada apoiar os usuários que não tenha suporte a script.

Tente pensar de uma maneira com que o site de compartilhamento de habilidade poderia ser configurado para preservar a funcionalidade básica quando executado sem JavaScript. As atualizações automáticas não seria mais suportada e as pessoas teram que atualizar sua página da maneira antiga. Seria bom sermos capazes de possibilitar esses usuários de ver as palestras existentes, criar novas e apresentar comentários.

Não se sinta obrigado a implementar isso. Esboçar uma solução é o suficiente. Será que a abordagem vista é mais ou menos elegante do que fizemos inicialmente?

Dica:

Dois aspectos centrais da abordagem feita neste capítulo como a interface `HTTP` e um `template` do lado do cliente de renderização não funcionam sem JavaScript. Formulários HTML normais podem enviar solicitações `GET` e `POST`, mas não solicitações `PUT` ou `DELETE` e os dados podem serem enviados apenas por uma `URL` fixa.

Assim o servidor teria que ser revisto para aceitar comentários, novas palestras e remover palestras através de solicitações `POST`, cujos corpos não podem serem JSONs, então devemos converter para o formato codificado em uma `URL` para conseguirmos usar os formulários HTML (ver Capítulo 17). Estes pedidos teriam que retornar a nova página inteira para que os usuários vejam os novos estados depois que é feito alguma mudança. Isso não seria muito difícil de fazer e poderia ser aplicado conjuntamente com uma interface `HTTP` mais "clean".

O código para mostrar as palestras teria que ser duplicado no servidor. O arquivo `index.html` ao invés de ser um arquivo estático, teria de ser gerado dinamicamente adicionando um manipulador para que o roteador incluía as palestras e comentários atuais quando for servido.