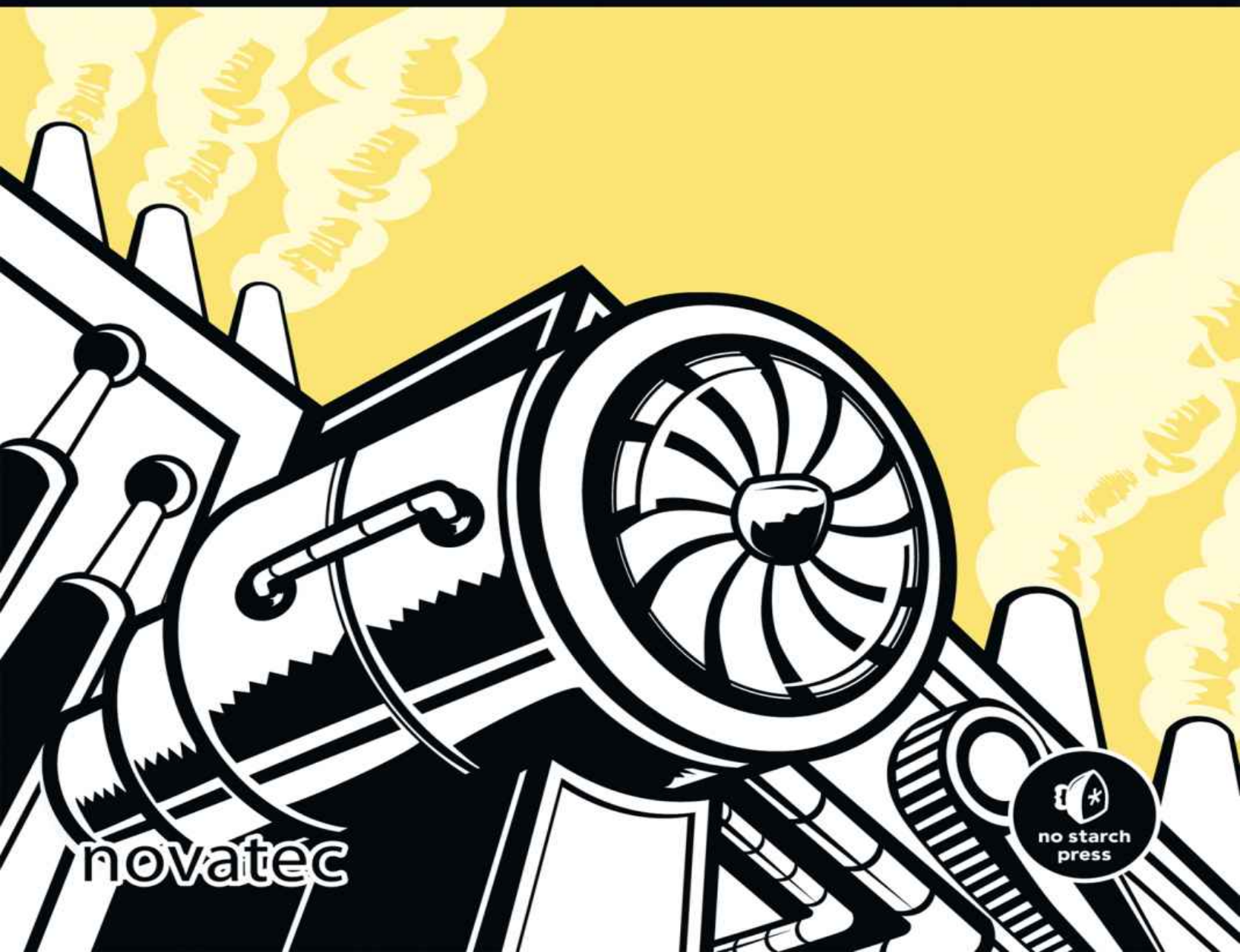
PRINCÍPIOS DE ORIENTAÇÃO A OBJETOS EM JAVASCRIPT

NICHOLAS C. ZAKAS



novatec

no starch press

Nicholas C. Zakas

Novatec

Editor: Rubens Prates
Tradução: Pietro Pereira/Lúcia A. Kinoshita
Revisão gramatical: Marta Almeida de Sá
Assistente editorial: Priscila Ayumi Yoshimatsu
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-589-9

Histórico de edições impressas:

Outubro/2014 Primeira edição

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

Sumário

Sobre o autor

Nicholas C. Zakas é engenheiro de software na Box e é conhecido por escrever e falar sobre o que há de mais moderno nas melhores práticas de JavaScript. Adquiriu experiência durante os cinco anos em que trabalhou no Yahoo! como principal engenheiro responsável pelo front-end da página inicial dessa empresa. É autor de diversos livros, incluindo *Maintainable JavaScript* (O'Reilly Media, 2012) e *Professional JavaScript for Web Developers* (Wrox, 2012).

Sobre o revisor técnico

Originalmente do Reino Unido, Angus Croll atualmente faz parte da equipe de framework web do Twitter em São Francisco e é coautor e principal mantenedor do framework open source Flight do Twitter. Ele é obcecado tanto por JavaScript quanto por literatura e é defensor apaixonado de um maior envolvimento entre artistas e pensadores criativos no desenvolvimento de software. Angus frequentemente faz palestras em conferências pelo mundo todo e, atualmente, está trabalhando em dois livros para a editora No Starch Press. Ele pode ser contatado no Twitter pelo nome de usuário *@angustweets*.

Apresentação

Por Cody Lindley

O nome Nicholas Zakas é sinônimo do próprio desenvolvimento em JavaScript. Eu poderia divagar por várias páginas fazendo elogios ao seu trabalho, mas não farei isso. Nicholas é bem conhecido como desenvolvedor JavaScript de alto nível, além de ser autor, e dispensa apresentação. Entretanto gostaria de oferecer minha opinião antes de elogiar o conteúdo deste livro.

Meu relacionamento com Nicholas vem de anos estudando seus livros, lendo seus posts em blogs, assistindo a suas palestras e monitorando as suas atualizações no Twitter, como um aprendiz de JavaScript. Nós nos conhecemos pessoalmente quando lhe pedi que fizesse uma palestra em uma conferência de jQuery há alguns anos. Ele proferiu uma palestra de alto nível à comunidade jQuery e, desde então, temos nos falado de forma tanto pública quanto privada pela Internet. Naquela época, comecei a admirá-lo como mais que apenas um líder e um desenvolvedor na comunidade JavaScript. Suas palavras são sempre agradáveis e profundas, e seu comportamento é sempre gentil.

Seu objetivo como desenvolvedor, palestrante e autor é sempre ajudar, educar e aperfeiçoar. Quando ele fala, você deve ouvi-lo, não só porque ele é um expert em JavaScript, mas porque seu caráter se destaca acima de seu status profissional.

O título e a introdução deste livro deixam claras as intenções de Nicholas: ele o escreveu para ajudar programadores que têm um background em orientação a objetos baseado em classes (como programadores C++ ou Java) a migrar para uma linguagem que não tem classes. No livro, ele explica como podemos implementar o encapsulamento, a agregação, a herança e o polimorfismo quando programamos em JavaScript. Esse é o conteúdo perfeito para trazer um programador experiente ao

desenvolvimento JavaScript orientado a objetos. Se você está lendo este livro e for um desenvolvedor com experiência em outra linguagem de programação, está prestes a ter a satisfação de ler um livro JavaScript conciso e habilmente escrito.

Contudo este livro também é ideal para programadores que já desenvolvam em JavaScript. Muitos desenvolvedores JavaScript têm somente um entendimento de objetos compatível com ECMAScript 3 (ES3), e eles precisam de uma introdução adequada aos recursos de objetos do ECMAScript 5 (ES5). Este livro pode servir como essa introdução, preenchendo a lacuna de conhecimento que separa os objetos em ES3 e em ES5.

Porém você pode estar pensando: "Grande coisa. Muitos livros já incluem capítulos ou notas com as características adicionais do JavaScript, que se encontram na ES5." Bem, isso é verdade. Contudo acredito que este seja o único livro, até agora, que foca na natureza dos objetos, dando um tratamento de primeira classe aos objetos ES5 em toda a narrativa. Este livro apresenta uma introdução coesa não somente aos objetos ES5, mas também às partes do ES3 que você precisa conhecer enquanto aprende muitos aspectos novos introduzidos em ES5.

Como autora, realmente acredito que este seja exatamente o livro que precisava ser escrito, considerando o seu foco nos princípios de orientação a objetos e nas atualizações referentes aos objetos em ES5, enquanto esperamos pelas atualizações de ES6 para os ambientes de scripting.

Cody Lindley (*www.codylindley.com*)

Autora de *Javascript Enlightenment*, *DOM Enlightenment* e *jQuery Enlightenment.*

Boise, Idaho

16 de dezembro de 2013

Agradecimentos

Gostaria de agradecer a Kate Matsudaira por me convencer de que publicar um e-book era a melhor maneira de divulgar essas informações. Sem o seu conselho, provavelmente eu ainda estaria pensando no que fazer com o conteúdo deste livro.

Obrigado a Rob Friesel por, novamente, prover um excelente feedback em uma versão preliminar deste livro, e a Cody Lindley por suas sugestões. Agradecimentos adicionais a Angus Croll por sua revisão técnica da versão final – sua meticulosidade tornou este livro muito melhor.

Obrigado também a Bill Pollock, que conheci em uma conferência e que foi o ponto de partida na empreitada para a publicação deste livro.

Introdução

Muitos desenvolvedores associam a programação orientada a objetos com linguagens que normalmente são ensinadas nas escolas, como C++ e Java, que têm as classes como base para esse tipo de programação. Antes de poder fazer algo nessas linguagens, é preciso criar uma classe, mesmo que você esteja apenas escrevendo um simples programa de linha de comando.

Os padrões de design comuns da indústria da programação também reforçam o conceito baseado em classes. No entanto o JavaScript não utiliza classes, e isso é parte do motivo pelo qual as pessoas ficam confusas quando tentam aprender JavaScript depois de C++ ou de Java.

As linguagens orientadas a objetos apresentam algumas características:

- **Encapsulamento** – os dados podem ser agrupados com funcionalidades que operam sobre eles. Essa é, basicamente, a definição de um objeto.
- **Agregação** – um objeto pode referenciar outro objeto.
- **Herança** – um objeto recém-criado tem as mesmas características que outro objeto sem duplicar as mesmas funcionalidades explicitamente.
- **Polimorfismo** – uma interface pode ser implementada por múltiplos objetos.

O JavaScript apresenta todas essas características, mas, como a linguagem não abrange o conceito de classes, algumas delas não são implementadas exatamente da maneira que você imaginaria. À primeira vista, um programa JavaScript pode até se parecer com um programa procedural escrito em C. Se puder escrever uma função e passar algumas variáveis a ela, você terá um script funcional que aparentemente não tem objetos. Um olhar mais profundo na linguagem, no entanto, revela a existência de objetos por meio do uso da notação de ponto (.).

Muitas linguagens orientadas a objetos usam a notação de ponto para

acessar propriedades e métodos de objetos, e no JavaScript, sintaticamente, ocorre o mesmo. Mas em JavaScript você jamais irá precisar criar uma definição de classe, importar um pacote ou incluir um arquivo de cabeçalho. Basta começar a codificar com os tipos de dados que você quiser, e eles poderão ser agrupados de várias maneiras. Certamente, é possível escrever um programa JavaScript de modo procedural, mas o verdadeiro poder da linguagem surge quando você tira vantagem de sua natureza orientada a objetos. Esse é o assunto deste livro.

Não se engane: muitos conceitos que você pode ter aprendido em linguagens de programação orientadas a objetos mais tradicionais não se aplicam necessariamente ao JavaScript. Embora muitos desses conceitos, com frequência, confundam os iniciantes, durante a leitura, você irá descobrir rapidamente que a natureza de tipagem fraca do JavaScript permite escrever menos código para realizar as mesmas tarefas implementadas em outras linguagens. Podemos simplesmente começar a escrever código sem planejar as classes de que precisaremos com antecedência. Você precisa de um objeto com campos específicos? Simplesmente crie um objeto *ad hoc* no local desejado. Você se esqueceu de adicionar um método a esse objeto? Sem problemas – basta adicionar depois.

Neste livro, você conhecerá a forma única com que o JavaScript lida com a programação orientada a objetos. Deixe para trás as noções de classes e de herança baseadas em classes e conheça a herança baseada em protótipos e as funções construtoras que se comportam de modo semelhante. Você aprenderá a criar objetos, definir seus próprios tipos, usar herança e manipular objetos para tirar o máximo de proveito deles. Resumindo, você aprenderá tudo o que você precisa saber para entender e escrever JavaScript de forma profissional. Divirta-se!

Público-alvo deste livro

O propósito deste livro é ser um guia para aqueles que já entendem de programação orientada a objetos, mas desejam saber exatamente como o

conceito funciona em JavaScript. Familiaridade com Java, C# ou programação orientada a objetos em outras linguagens é um forte indicador de que este livro é ideal para você. Em particular, este livro está voltado para três grupos de leitores:

- desenvolvedores que tenham familiaridade com conceitos de programação orientada a objetos e desejam aplicá-los ao JavaScript;
- desenvolvedores de aplicações web que usam Node.js e que desejam estruturar seu código mais eficientemente;
- desenvolvedores iniciantes em JavaScript que desejam ter um conhecimento mais profundo dessa linguagem.

Este não é um livro para iniciantes que nunca escreveram uma linha de JavaScript. Você precisará de um bom conhecimento sobre como escrever e executar códigos JavaScript para acompanhar a leitura.

Organização

O *Capítulo 1: Tipos primitivos e de referência* apresenta os dois tipos diferentes de valores em JavaScript: primitivos e de referência. Você saberá como eles se diferenciam e por que entender suas diferenças é importante para ter um conhecimento geral de JavaScript.

O *Capítulo 2: Funções* explica os prós e os contras das funções em JavaScript. Funções de primeira classe são o que deixam o JavaScript interessante.

O *Capítulo 3: Entendendo os objetos* detalha a criação de objetos em JavaScript. Os objetos em JavaScript se comportam de maneira diferente dos objetos em outras linguagens, de modo que um conhecimento profundo sobre o funcionamento dos objetos é fundamental para dominar a linguagem.

O *Capítulo 4: Construtores e protótipos* expande a discussão anterior sobre funções ao observar mais especificamente os construtores. Todos os construtores são funções, mas eles são usados de forma um pouco diferente. Este capítulo explora essas diferenças ao mesmo tempo que discute a criação de seus próprios tipos.

O *Capítulo 5: Herança* explica como a herança é implementada em JavaScript.

Embora o JavaScript não tenha classes, isso não significa que a herança seja impossível. Nesse capítulo, você conhecerá a herança prototípica e saberá como ela difere da herança baseada em classes.

O *Capítulo 6: Padrões de objetos* discorre sobre padrões comuns de objetos. Há muitas maneiras de criar e compor objetos em JavaScript, e esse capítulo apresenta os padrões mais populares para fazer isso.

Ajuda e suporte

O fórum oficial de ajuda (em inglês) pode ser encontrado neste link: *http://groups.google.com/group/zakasbooks*.

CAPÍTULO 1

Tipos primitivos e de referência

Muitos desenvolvedores aprendem programação orientada a objetos ao trabalharem com linguagens baseadas em classes, como Java ou C#. Quando começam a aprender JavaScript, esses desenvolvedores ficam desorientados porque o JavaScript não tem um suporte formal para classes. Em vez de definir as classes desde o início, com o JavaScript, você pode simplesmente escrever código e criar as estruturas de dados à medida que precisar delas. Como não há classes em JavaScript, também não há agrupamento de classes, comumente chamados de "pacotes" (packages). Enquanto em linguagens como Java os pacotes e os nomes das classes definem tanto os tipos de objeto usados quanto o layout dos arquivos e das pastas em seu projeto, programar em JavaScript é como começar com uma folha em branco: você pode organizar tudo do jeito que quiser. Alguns desenvolvedores preferem copiar estruturas de outras linguagens, enquanto outros aproveitam a flexibilidade do JavaScript para estruturar de um jeito completamente novo. Para os iniciantes, essa liberdade de escolha pode ser confusa, mas depois que você se habituar a ela, verá que o JavaScript é uma linguagem altamente flexível, que se adapta as suas preferências facilmente.

Para facilitar a transição a partir de linguagens orientadas a objetos tradicionais, em JavaScript, os objetos constituem a parte central da linguagem. Quase todos os dados em JavaScript são um objeto ou são acessados por meio deles. De fato, até as funções (para as quais, em linguagens tradicionais, é preciso fazer alguns malabarismos para obter referências) são representadas como objetos em JavaScript, o que as tornam *funções de primeira-classe*.

Trabalhar com objetos e entendê-los são aspectos fundamentais para

compreender o JavaScript como um todo. Você pode criar objetos a qualquer momento e adicionar ou remover propriedades desses objetos sempre que quiser. Além do mais, os objetos JavaScript são extremamente flexíveis e têm funcionalidades que geram padrões únicos e interessantes, simplesmente impossíveis de serem criados em outras linguagens.

Este capítulo foca em identificar e trabalhar com os dois tipos de dados principais do JavaScript: tipos primitivos e tipos de referência. Embora ambos sejam acessados por meio de objetos, eles se comportam de maneira diferente, e é importante entender isso.

O que são tipos?

Embora o JavaScript não tenha nenhum conceito de classe, ele utiliza dois *tipos*: primitivos e de referência. Os *tipos primitivos* são armazenados como tipos de dados simples. *Os tipos de referência* são armazenados como objetos e, na realidade, são apenas referências a posições de memória.

O truque está no fato de que o JavaScript permite tratar tipos primitivos como tipos de referência para fazer com que a linguagem seja mais consistente para o desenvolvedor.

Enquanto outras linguagens de programação distinguem tipos primitivos de tipos de referência ao armazenar os tipos primitivos na pilha e os tipos de referência no heap, o JavaScript não utiliza, de modo algum, esse conceito: ele controla as variáveis de um escopo em particular usando um *objeto variável*. Valores primitivos são armazenados diretamente no objeto variável, enquanto os valores de referência são criados como ponteiros no objeto variável. No entanto, como você verá mais adiante neste capítulo, valores primitivos e valores de referência se comportam de modo bem diferente, embora possam parecer iguais à primeira vista.

É claro que há outras diferenças entre tipos primitivos e de referência.

Tipos primitivos

Os tipos primitivos representam simples porções de dados armazenados da forma como são, por exemplo, como `true` ou `25`. Há cinco tipos

primitivos em JavaScript:

Boolean	`true` ou `false`
Number	Qualquer valor numérico inteiro ou de ponto flutuante
String	Um caractere ou uma sequência de caracteres delimitados tanto por aspas simples como por aspas duplas
Null	Um tipo primitivo que tem apenas um valor: `null`
Undefined	Um tipo primitivo que tem apenas um valor: `undefined` (`undefined` é o valor atribuído a uma variável não inicializada)

Os três primeiros tipos (Boolean, number e string) se comportam de forma semelhante, enquanto os dois últimos (null e undefined) são um pouco diferentes, conforme discutiremos ao longo deste capítulo. Todos os tipos primitivos têm representações literais de seus valores. *Os literais* representam valores que não são armazenados em uma variável, por exemplo, um nome ou um preço fixo no código. Aqui estão alguns exemplos de cada tipo com suas formas literais:

```
// strings
var name = "Nicholas";
var selection = "a";

// number
var count = 25;
var cost = 1.51;

// boolean
var found = true;

// null
var object = null;

// undefined
var flag = undefined;
var ref; // undefined é atribuído automaticamente
```

Em JavaScript, assim como em muitas outras linguagens, as variáveis que armazenam um primitivo contêm diretamente o valor primitivo (em vez de conter um ponteiro para um objeto). Quando um valor primitivo é atribuído a uma variável, o valor é copiado para essa variável. Isso significa que se você definir que uma variável é igual a outra, cada

variável terá a sua própria cópia do dado. Por exemplo:

```
var color1 = "red";
var color2 = color1;
```

Nesse código, `color1` é definida como `"red"` (vermelho). A variável `color2` então recebe o valor de `color1`, o que faz com que `"red"` seja armazenado em `color2`. Embora `color1` e `color2` contenham o mesmo valor, eles são totalmente diferentes um do outro, e você pode mudar o valor de `color1` sem afetar `color2` e vice-versa. Isso ocorre porque há dois lugares diferentes de armazenamento, um para cada variável. A figura 1.1 mostra o objeto variável (Variable Object) para esse trecho de código.

Figura 1.1 – O objeto variável.

Como cada variável que contém um valor primitivo usa seu próprio espaço de armazenamento, as alterações em uma variável não afetam a outra. Por exemplo:

```
var color1 = "red";
var color2 = color1;

console.log(color1); // "red"
console.log(color2); // "red"

color1 = "blue";

console.log(color1); // "blue"
console.log(color2); // "red"
```

Nesse código, `color1` é alterada para `"blue"` (azul) e `color2` preserva o seu valor original, `"red"` (vermelho).

Identificando tipos primitivos

A melhor maneira de identificar tipos primitivos é por meio do operador `typeof`, que funciona com qualquer variável e retorna uma string indicando o tipo de dado. O operador `typeof` funciona bem com strings, numbers, Booleans e `undefined`. O código a seguir mostra a saída quando usamos o operador `typeof` em diferentes valores primitivos:

```
console.log(typeof "Nicholas"); // "string"
console.log(typeof 10); // "number"
console.log(typeof 5.1); // "number"
```

```
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
```

Como esperado, `typeof` retorna `"string"` quando o valor é uma string; `number` quando um valor é um número (independentemente de o valor ser um inteiro ou um float); `boolean` quando o valor é um Boolean e `undefined` quando o valor for indefinido.

A parte confusa é aquela que envolve `null`.

Você não seria o primeiro desenvolvedor a se confundir com o resultado desta linha de código:

```
console.log(typeof null); // "object"
```

Ao executar `typeof null`, o resultado é `"object"`. Mas por que é um objeto se o tipo é `null`? (De fato, isso foi reconhecido como um erro pelo TC39, que é o comitê que faz o design e mantém o JavaScript. Você poderia argumentar que `null` é uma referência a um objeto vazio, fazendo com que `object` seja um valor de retorno correto, porém ainda assim é confuso).

A melhor maneira de determinar se um valor é `null` é compará-lo diretamente com `null`, desta maneira:

```
console.log(value === null); // true ou false
```

COMPARANDO SEM CONVERSÃO

Note que esse código usa o operador de igualdade triplo (`===`) em vez de usar o operador de igualdade duplo. Isso ocorre porque o operador triplo faz a comparação sem converter a variável para outro tipo. Para entender o motivo pelo qual isso é importante, veja o código a seguir:

```
console.log("5" == 5); // true
console.log("5" === 5); // false

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

Quando a igualdade dupla é utilizada, a string "5" e o número 5 são considerados iguais porque a igualdade dupla converte a string em um número antes de fazer a comparação. O operador de igualdade tripla não considera esses valores iguais porque eles são de tipos diferentes. Da mesma forma, quando comparamos `undefined` e `null`, a igualdade dupla mostra que eles são equivalentes, enquanto a igualdade tripla diz que não. Quando estiver tentando identificar um `null`, use a igualdade tripla para que você possa identificar o tipo corretamente.

Métodos primitivos

Apesar de serem tipos primitivos, strings, numbers e Booleans contêm métodos. (Os tipos `null` e `undefined` não contêm métodos). As Strings, em particular, têm vários métodos para ajudar você a trabalhar com elas. Por exemplo:

```
var name = "Nicholas";
var lowercaseName = name.toLowerCase(); // Converte para minúsculo
var firstLetter = name.charAt(0); // Obtém o primeiro caractere
var middleOfName = name.substring(2, 5); // Obtém os caracteres de 2 a 4

var count = 10;
var fixedCount = count.toFixed(2); // Converte para "10.00"
var hexCount = count.toString(16); // Converte para "a"

var flag = true;
var stringFlag = flag.toString(); // Converte para "true"
```

NOTA Apesar de terem métodos, os valores primitivos não são objetos. O JavaScript faz com que eles pareçam ser objetos para oferecer uma experiência consistente na linguagem, como você verá mais adiante neste capítulo.

Tipos de referência

Os tipos de referência representam objetos em JavaScript e são o recurso encontrado na linguagem que mais se assemelha às classes. Valores de referência são *instâncias* de tipos de referência e são sinônimos de objetos (o restante deste capítulo chama os valores de referência simplesmente de *objetos*). Um objeto é uma lista não ordenada de *propriedades* constituídas de um nome (sempre uma string) e um valor. Quando o valor de uma propriedade for uma função, ela será chamada de *método*. As funções propriamente ditas, na verdade, são valores de referência em JavaScript, de modo que há pouca diferença entre uma propriedade que contém um array e uma que contém uma função, exceto pelo fato de que a função pode ser executada.

É claro que é necessário criar os objetos antes de começar a trabalhar com eles.

Criando objetos

Às vezes, pensar em objetos JavaScript como nada mais que tabelas hash, como mostrado na figura 1.2, pode ajudar:

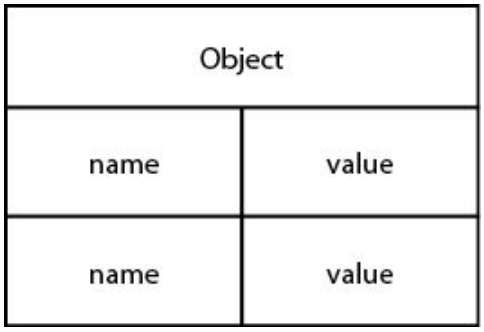


Figura 1.2 – Estrutura de um objeto.

Existem algumas maneiras de criar, ou seja, de *instanciar* um objeto. A primeira é por meio do operador `new` e um *construtor*. (Um construtor é simplesmente uma função que usa `new` para criar um objeto – qualquer função pode ser um construtor.) Por convenção, os nomes dos construtores em JavaScript iniciam com uma letra maiúscula para distingui-los de funções que não são construtoras. Por exemplo, este código instancia um objeto genérico e armazena uma referência a ele em `object`:

```
var object = new Object();
```

Os tipos de referência não armazenam o objeto diretamente na variável à qual ele foi atribuído, portanto a variável `object` nesse exemplo não contém a instância do objeto. Em vez disso, ela armazena um ponteiro (ou uma referência) para a posição de memória em que esse objeto está. Essa é a principal diferença entre objetos e valores primitivos, pois os valores primitivos são armazenados diretamente na variável em que são definidos.

Quando um objeto for atribuído a uma variável, você estará, na verdade, atribuindo-lhe um ponteiro. Isso significa que se você atribuir uma variável a outra, cada variável terá uma cópia dessa referência, e ambos continuarão referenciando o mesmo objeto na memória. Por exemplo:

```
var object1 = new Object();
var object2 = object1;
```

Esse código inicialmente cria um objeto (com `new`) e então armazena uma referência em `object1`. Em seguida, `object2` recebe o valor de `object1`. Continua havendo somente uma única instância do objeto criado na primeira linha, mas ambas as variáveis agora apontam para esse objeto, como mostrado na figura 1.3:

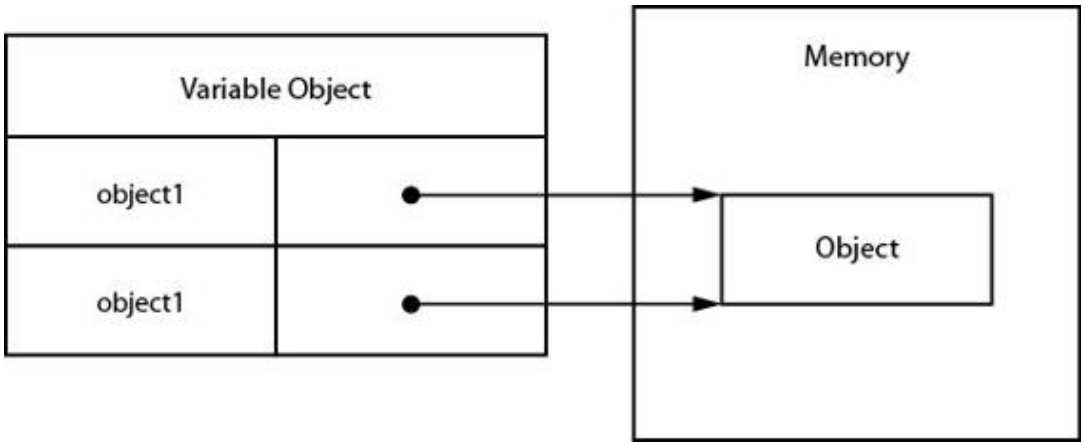


Figura 1.3 – Duas variáveis que apontam para o mesmo objeto.

Removendo a referência a objetos

O JavaScript é uma linguagem que tem garbage-collection (coletor de lixo), portanto não é realmente necessário se preocupar com alocações de memória ao usar tipos de referência. Entretanto é melhor remover a referência aos objetos que não sejam mais necessários para que o coletor de lixo possa liberar essa memória. A melhor maneira de fazer isso é atribuir `null` à variável do objeto.

```
var object1 = new Object();
// faça algo
object1 = null; // remove a referência
```

Nesse caso, `object1` é criado e usado antes de finalmente ser definido como `null`. Quando não houver mais referências a um objeto na memória, o coletor de lixo poderá usar essa área de memória para algo diferente. (Remover a referência aos objetos é especialmente importante em aplicações grandes, que utilizam milhares de objetos.)

Adicionando ou removendo propriedades

Outro aspecto interessante sobre os objetos em JavaScript é que você pode adicionar ou remover propriedades a qualquer momento. Por exemplo:

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Awesome!";
console.log(object2.myCustomProperty); // "Awesome!"
```

Nesse caso, `myCustomProperty` é adicionada a `object1` com valor igual a `"Awesome!"`. Essa propriedade também é acessível em `object2` porque tanto `object1` quanto `object2` apontam para o mesmo objeto.

NOTA Esse exemplo demonstra uma particularidade única do JavaScript: você pode modificar objetos quando quiser, mesmo que eles não tenham sido definidos antes. Também há maneiras de evitar essas modificações, como você verá mais adiante neste livro.

Além dos tipos de referência genéricos para objetos, o JavaScript tem outros tipos próprios da linguagem à sua disposição.

Instanciando tipos próprios

Você viu como criar e interagir com objetos genéricos criados por meio de `new Object()`. O tipo `Object` é somente um entre uma grande variedade de tipos de referência que o JavaScript oferece. Os outros tipos próprios são mais especializados no que diz respeito ao uso pretendido e podem ser instanciados a qualquer momento. Os tipos próprios são:

- `Array`: uma lista ordenada de valores indexados numericamente
- `Date`: uma data e uma hora
- `Error`: um erro de execução (há vários subtipos mais específicos de erros)
- `Function`: uma função
- `Object`: um objeto genérico
- `RegExp`: uma expressão regular

Você pode instanciar cada tipo próprio usando `new`, como mostrado aqui:

```
var items = new Array();
var now = new Date();
```

```
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

Formas literais

Vários tipos próprios apresentam formas literais. Uma forma *literal* é uma sintaxe que permite definir um valor de referência sem criar um objeto explicitamente, usando o operador `new` e o construtor do objeto. (Anteriormente, neste capítulo, você viu exemplos de literais primitivos incluindo strings literais, literais numéricos, literais booleanos, o literal `null` e o literal `undefined`).

Literais de objetos e de arrays

Para criar um objeto com a sintaxe de *objeto literal*, você pode definir as propriedades de um novo objeto entre chaves. As propriedades são compostas de um identificador ou de uma string, dois pontos e um valor, com várias propriedades separadas por vírgula. Por exemplo:

```
var book = {
    name: "Princípios de orientação a objetos em JavaScript",
    year: 2014
};
```

Você também pode usar strings literais para nomes de propriedade, o que é muito útil se você quiser definir o nome de uma propriedade usando espaços ou outros caracteres especiais:

```
var book = {
    "name": "Princípios de orientação a objetos em JavaScript",
    "year" : 2014
};
```

Esse exemplo é equivalente ao anterior, apesar das diferenças sintáticas. Ambos os exemplos também são logicamente equivalentes a este:

```
var book = new Object();
book.name = "Princípios de orientação a objetos em JavaScript";
book.year = 2014;
```

O resultado final de cada um dos três exemplos anteriores é o mesmo: um objeto com duas propriedades. A escolha do padrão a ser usado cabe a você, pois as funcionalidades, em última instância, são as mesmas.

NOTA Usar um literal de objeto, na verdade, não faz `new Object()` ser chamado. Em vez disso, a engine do JavaScript segue os mesmos passos usados em `new Object()` sem chamar o construtor. Isso vale para todos os literais de referência.

Você pode definir um *literal de array* de forma semelhante, inserindo qualquer quantidade de valores separados por vírgula dentro de colchetes. Por exemplo:

```
var colors = [ "red", "blue", "green" ];
console.log(colors[0]); // "red"
```

Esse código é equivalente a:

```
var colors = new Array("red", "blue", "green");
console.log(colors[0]); // "red"
```

Literais de função

Quase sempre, as funções são definidas por meio de sua forma literal. Na verdade, usar o construtor `Function` normalmente é desaconselhável por ser mais difícil de manter, ler e depurar uma string de código do que o código em si, de modo que, raramente, você o verá.

Criar funções é muito mais simples e menos sujeito a erros quando usamos a forma literal. Por exemplo:

```
function reflect(value) {
    return value;
}

// é o mesmo que:
var reflect = new Function("value", "return value;");
```

Esse código define a função `reflect()`, que retorna qualquer valor passado a ela. Mesmo no caso dessa função simples, a forma literal é mais fácil de escrever e de ser entendida se comparada ao código que utiliza o construtor. Além disso, não há maneira eficiente de depurar funções criadas com o método do construtor: essas funções não são reconhecidas pelos debuggers de JavaScript e, desse modo, atuam como caixas pretas

em sua aplicação.

Literais de expressões regulares

O JavaScript também tem *literais para expressões regulares*, que permitem definir essas expressões sem usar o construtor `RegExp`. Literais de expressões regulares se parecem muito com as expressões regulares em Perl: a expressão é definida entre duas barras, e qualquer opção adicional corresponde a uma única letra após a segunda barra. Por exemplo:

```
var numbers = /\d+/g;
// é o mesmo que:
var numbers = new RegExp("\\d+", "g");
```

É mais fácil lidar com a forma literal das expressões regulares em JavaScript do que usar o método com o construtor porque não é necessário se preocupar com caracteres de escape nas strings. Quando o construtor `RegExp` é usado, a expressão é passada na forma de uma string, de modo que é preciso usar o escape em qualquer barra invertida (esse é o motivo pelo qual `\d` é usado na forma literal e `\\d` é usado no construtor). Literais de expressões regulares são preferíveis ao método do construtor em JavaScript, exceto quando a expressão regular for construída dinamicamente a partir de uma ou mais strings. Apesar do que foi dito, com exceção de `Function`, não há realmente uma maneira certa ou errada de instanciar tipos próprios. Muitos desenvolvedores preferem usar literais, enquanto outros preferem os construtores. Escolha o método que você se sentir mais confortável em usar.

Acesso a propriedades

Propriedades são pares de nomes/valores armazenados em um objeto. A notação de ponto é a forma mais comum de acessar propriedades em JavaScript (e em muitas linguagens orientadas a objetos), mas você também pode acessar propriedades em objetos JavaScript usando a notação de colchetes com uma string. Por exemplo, você pode escrever este código, que usa a notação de ponto:

```
var array = [];
```

```
array.push(12345);
```

Com a notação de colchetes, o nome do método passa a ser incluído em uma string dentro de colchetes, como neste exemplo:

```
var array = [];
array["push"](12345);
```

Essa sintaxe é muito útil quando você precisa decidir dinamicamente qual propriedade deverá ser acessada. Por exemplo, no caso a seguir, a notação de colchetes permite usar uma variável em vez de usar uma string literal para especificar a propriedade a ser acessada:

```
var array = [];
var method = "push";
array[method](12345);
```

Nesse exemplo, a variável `method` tem valor igual a `"push"`, portanto `push()` é chamada no array. Essa funcionalidade é muito útil, como você verá ao longo deste livro. A questão a ser lembrada é que, além da sintaxe, a única diferença – considerando o desempenho e outros aspectos – entre a notação de ponto e a notação de colchetes é que a notação de colchetes permite usar caracteres especiais em nomes de propriedades. Os desenvolvedores, normalmente, acham a notação de ponto mais fácil de ler, de modo que você a verá sendo mais usada que a notação de colchetes.

Identificando tipos de referência

Uma função é o tipo de referência mais fácil de ser identificado porque, ao usar o operador `typeof` em uma função, ele deverá retornar `"function"`:

```
function reflect(value) {
    return value;
}

console.log(typeof reflect); // "function"
```

Outros tipos de referência podem ser mais difíceis de identificar porque, para todos os tipos de referência que não sejam funções, o operador `typeof` retornará `"object"`. Isso não ajuda muito quando você estiver lidando com

vários tipos diferentes. Para identificar tipos de referência mais facilmente, o operador `instanceof` do JavaScript pode ser utilizado.

O operador `instanceof` recebe um objeto e um construtor como parâmetros. Quando o valor for uma instância do tipo especificado pelo construtor, o operador `instanceof` retornará `true`. Caso contrário, ele retornará `false`, como você pode ver aqui:

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}
console.log(items instanceof Array); // true
console.log(object instanceof Object); // true
console.log(reflect instanceof Function); // true
```

Nesse exemplo, diversos valores foram testados usando `instanceof` e um construtor. Cada tipo de referência é identificado corretamente por meio do operador `instanceof` e do construtor que representa seu tipo verdadeiro (mesmo que o construtor não tenha sido usado na criação da variável).

O operador `instanceof` pode identificar tipos herdados. Isso significa que todo objeto na verdade é uma instância de `Object` porque todo tipo de referência herda de `Object`.

Para demonstrar, o próximo exemplo analisa as três referências anteriormente criadas com `instanceof`:

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}

console.log(items     instanceof Array); // true
console.log(items     instanceof Object); // true
console.log(objects   instanceof Object); // true
console.log(object    instanceof Array); // false
console.log(reflect    instanceof Function); // true
console.log(reflect    instanceof Object); // true
```

Cada tipo de referência é identificado corretamente como uma instância de `Object`, da qual todos os tipos de referência herdam.

Identificando arrays

Embora `instanceof` possa identificar arrays, há uma exceção que afeta os desenvolvedores web: valores em JavaScript podem ser passados entre frames na mesma página web. Isso se torna um problema somente quando você tenta identificar o tipo de um valor de referência, pois cada página web tem o seu próprio contexto global – sua própria versão de `Object`, `Array` e de todos os demais tipos próprios. Como resultado, ao passar um array de um frame para outro, `instanceof` não funciona porque o array é uma instância de `Array` de um frame diferente.

Para corrigir esse problema, o ECMAScript 5 introduziu `Array.isArray()`, que identifica definitivamente o valor como uma instância de `Array`, sem se importar com a origem do valor. Esse método deve retornar `true` quando receber um valor que seja um array nativo de qualquer contexto. Se o seu ambiente de desenvolvimento suporta ECMAScript 5, `Array.isArray()` será a melhor maneira de identificar arrays:

```
var items = [];

console.log(Array.isArray(items)); // true
```

O método `Array.isArray()` é suportado na maioria dos ambientes de desenvolvimento, tanto em browsers quanto em Node.js. Esse método não é suportado no Internet Explorer 8 e em versões anteriores.

Tipos wrapper primitivos

Talvez uma das partes mais confusas do JavaScript seja o conceito de *tipos wrapper primitivos*. Há três tipos wrapper primitivos: `String`, `Number` e `Boolean`. Esses tipos de referência especiais existem para fazer com que trabalhar com valores primitivos seja tão simples quanto trabalhar com objetos (seria muito confuso se você tivesse que usar uma sintaxe diferente ou mudar para o estilo de programação procedural somente para obter uma substring de um texto).

Os tipos wrapper primitivos são tipos de referência criados automaticamente por baixo dos panos sempre que strings, numbers ou booleans são lidos. Repare que, na primeira linha do próximo exemplo, um valor primitivo de string é atribuído a `name`. A segunda linha trata `name` como um objeto e chama `charAt(0)` usando a notação de ponto:

```
var name = "Nicholas";
var firstChar = name.charAt(0);
console.log(firstChar); // "N"
```

Isto é o que acontece internamente:

```
// O que a engine do JavaScript faz
var name = "Nicholas";
var temp = new String(name);
var firstChar = temp.charAt(0);
temp = null;
console.log(firstChar); // "N"
```

Como a segunda linha (do primeiro exemplo) usa uma string (que é um primitivo) como um objeto, a engine do JavaScript cria uma instância de `String` para que `charAt(0)` funcione. O objeto `String` existe somente em uma instrução antes de ser destruído (um processo chamado de *autoboxing*). Para testar isso, tente adicionar uma propriedade a uma string como se ela fosse um objeto comum:

```
var name = "Nicholas";
name.last = "Zakas";

console.log(name.last); // undefined
```

Esse código tenta adicionar a propriedade `last` à string `name`. O código funciona normalmente, exceto pelo fato de que a propriedade desaparece. O que aconteceu? Quando trabalhamos com objetos normais, você pode adicionar propriedades a qualquer momento e elas estarão presentes até serem manualmente removidas. Com tipos wrapper primitivos, as propriedades parecem desaparecer porque o objeto no qual a propriedade foi definida é destruído imediatamente na sequência.

Aqui está o que realmente aconteceu na engine do JavaScript:

```
// O que a engine do JavaScript faz
var name = "Nicholas":
```

```
var temp = new String(name);
temp.last = "Zakas";
temp = null; // objeto temporário é destruído

var temp = new String(name);
console.log(temp.last); // undefined
temp = null;
```

Em vez de atribuir uma nova propriedade à string, o código cria uma nova propriedade em um objeto temporário que então é destruído. Ao tentar acessar essa propriedade posteriormente, um objeto temporário diferente será criado e a nova propriedade não estará presente ali. Embora valores de referência sejam criados automaticamente para valores primitivos, quando o tipo desses valores é verificado por meio de `instanceof`, o resultado é `false`:

```
var name = "Nicholas";
var count = 10;
var found = false;
console.log(name instanceof String); // false
console.log(count instanceof Number); // false
console.log(found instanceof Boolean); // false
```

O operador `instanceof` retorna `false` porque um objeto temporário é criado somente quando um valor é lido. E como `instanceof` na verdade não lê nada, nenhum objeto temporário é criado, e ele nos diz que esses valores não são instâncias de tipos wrapper primitivos. Você pode criar tipos wrapper primitivos manualmente, mas há alguns efeitos colaterais:

```
var name = new String("Nicholas");
var count = new Number(10);
var found = new Boolean(false);
console.log(typeof name); // "object"
console.log(typeof count); // "object"
console.log(typeof found); // "object"
```

Como você pode ver, criar uma instância de um tipo wrapper primitivo apenas dá origem a outro objeto, o que significa que `typeof` não pode identificar o tipo de dado que você pretende armazenar.

Além do mais, você não pode usar objetos `String`, `Number` ou `Boolean` como faria com valores primitivos. Por exemplo, o próximo código utiliza um

objeto `Boolean`. O objeto `Boolean` é `false`, embora `console.log("Found")` continue sendo executado porque um objeto será sempre considerado `true` em instruções condicionais. Não importa se o objeto representa `false`; ele é um objeto, portanto será avaliado como `true`.

```
var found = new Boolean(false);
if (found) {
    console.log("Found"); // isso é executado
}
```

Instanciar manualmente tipos wrapper primitivos também pode ser confuso em outros aspectos, portanto, a menos que você se depare com um caso especial em que faça sentido fazer isso, evite-os. Na maioria dos casos, usar objetos wrappers primitivos em vez de usar somente primitivos resultará em erros.

Sumário

Embora não tenha classes, o JavaScript tem tipos. Cada variável ou porção de dado é associado a um tipo primitivo ou de referência específico. Os cinco tipos primitivos (strings, numbers, Booleans, `null` e `undefined`) representam valores simples armazenados diretamente no objeto variável em um determinado contexto. Você pode usar `typeof` para identificar tipos primitivos com exceção de `null`, que deve ser comparado diretamente com o valor especial `null`.

Tipos de referência são o recurso que mais se assemelha a classes em JavaScript, e os objetos são instâncias de tipos de referência. Você pode criar novos objetos usando o operador `new` ou uma forma literal. As propriedades e os métodos podem ser acessados por meio da notação de ponto ou de colchetes. Funções são objetos em JavaScript, e você pode identificá-las usando o operador `typeof`. Use o operador `instanceof` com um construtor para identificar objetos de outros tipos de referência.

Para fazer com que os primitivos se assemelhem mais às referências, o JavaScript tem três tipos wrapper primitivos: `String`, `Number` e `Boolean`. O JavaScript cria esses objetos internamente para que você possa tratar os primitivos como se fossem objetos normais, porém os objetos

temporários são destruídos assim que a instrução que os utiliza for executada. Embora você possa criar suas próprias instâncias de wrappers primitivos, é melhor não fazer isso porque poderá ser confuso.

CAPÍTULO 2

Funções

Como discutimos no capítulo 1, as funções são objetos em JavaScript. A característica determinante de uma função – o que a distingue de outros objetos – é a presença de uma *propriedade interna* chamada `[[Call]]`. As propriedades internas não são acessíveis por meio do código, porém definem o seu comportamento à medida que ele é executado. O ECMAScript define várias propriedades internas de objetos em JavaScript, e essas propriedades são indicadas pela notação de colchetes duplos.

A propriedade `[[Call]]` é exclusiva das funções e indica que o objeto pode ser executado. Como somente as funções têm essa propriedade, o operador `typeof` é definido pela ECMAScript para retornar `"function"` para qualquer objeto que tenha a propriedade `[[Call]]`. Isso resultou em um pouco de confusão no passado porque alguns browsers também incluíam uma propriedade `[[Call]]` para expressões regulares, que eram, desse modo, erroneamente identificadas como funções. Todos os browsers atualmente se comportam da mesma maneira e o operador `typeof` não identifica mais as expressões regulares como funções.

Este capítulo discute as diferentes maneiras pelas quais as funções são definidas e executadas em JavaScript. Como as funções são objetos, elas se comportam de modo diferente das funções em outras linguagens, e entender esse comportamento é fundamental para ter uma boa compreensão de JavaScript.

Declarações versus expressões

Há duas formas literais para as funções. A primeira é a *declaração de função*, que começa com a palavra-chave `function` e inclui o nome da função logo

em seguida. O conteúdo da função é definido entre chaves, como mostrado nesta declaração:

```
function add(num1, num2) {
    return num1 + num2;
}
```

A segunda forma é a *expressão de função*, que não exige um nome após a palavra-chave `function`. Essas funções são consideradas anônimas porque o objeto função propriamente dito não tem um nome. Em vez disso, as expressões de funções normalmente são referenciadas por meio de uma variável ou de uma propriedade, como nesta expressão:

```
var add = function(num1, num2) {
    return num1 + num2;
};
```

Esse código atribui um valor de função à variável `add`. As duas formas são quase idênticas, exceto pela ausência do nome da função e pelo ponto e vírgula necessário no final da expressão de função. Expressões de atribuição normalmente terminam com ponto e vírgula, exatamente como ocorre na atribuição de qualquer outro valor.

Apesar de essas duas formas serem muito parecidas, elas diferem fundamentalmente em relação a um aspecto. As declarações de função são "*içadas*" (hoisted) para o topo do contexto (seja na função em que a declaração é feita ou no escopo global) quando o código é executado. Isso significa que você pode definir uma função depois de ela ter sido utilizada no código, sem que um erro seja gerado. Por exemplo:

```
var result = add(5, 5);

function add(num1, num2) {
    return num1 + num2;
}
```

Esse código pode passar a impressão de que causará um erro, porém ele funciona perfeitamente. Isso ocorre porque a engine do JavaScript efetua o hoisting da declaração de função para o topo e executa o código como se ele estivesse escrito desta maneira:

```
// Como a engine do JavaScript interpreta o código
```

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 5);
```

O hoisting de funções ocorre somente em declarações de funções porque o nome da função é previamente conhecido. Expressões de função, por outro lado, não podem sofrer hoisting porque as funções podem ser referenciadas somente por meio de uma variável. Desse modo, este código irá causar um erro:

```
// Erro!
var result = add(5, 5);

var add = function(num1, num2) {
    return num1 + num2;
};
```

Desde que as funções sejam sempre definidas antes de serem utilizadas, tanto as declarações quanto as expressões de função poderão ser usadas.

Funções como valores

Como o JavaScript tem funções de primeira classe, você pode usá-las assim como faria com qualquer outro objeto. Você pode atribuí-las a variáveis, adicioná-las a objetos, passá-las para outras funções como argumentos e retorná-las a partir de outras funções. Basicamente, uma função pode ser usada em qualquer local em que outro valor de referência pode ser utilizado. Isso faz com que as funções JavaScript sejam incrivelmente eficazes. Considere o exemplo a seguir:

```
❶ function sayHi() {
    console.log("Hi!");
}
sayHi(); // exibe "Hi!"

❷ var sayHi2 = sayHi;

sayHi2(); // exibe "Hi!";
```

Nesse código, há uma declaração de função para `sayHi` ❶. Uma variável chamada `sayHi2` é criada e recebe o valor de `sayHi` ❷. Tanto `sayHi` quanto

`sayHi2` apontam para a mesma função, e isso significa que qualquer uma delas pode ser executada gerando o mesmo resultado. Para entender por que isso acontece, vamos observar o mesmo código reescrito de modo a usar o construtor `Function`:

```
var sayHi = new Function("console.log('Hi');");
sayHi(); // exibe "Hi!"
var sayHi2 = sayHi;
sayHi2(); // exibe "Hi!"
```

O construtor `Function` deixa mais explícito o fato de que `sayHi` possa ser atribuído como qualquer outro objeto. Quando você tem em mente que funções são objetos, muitos comportamentos começam a fazer sentido.

Por exemplo, você pode passar uma função para outra função como argumento. O método `sort()` de arrays em JavaScript aceita uma função de comparação como parâmetro opcional. A função de comparação é chamada sempre que dois valores do array tiverem de ser comparados. Se o primeiro valor for menor que o segundo, a função de comparação deverá retornar um número negativo. Se o primeiro valor for maior que o segundo, a função retornará um número positivo. Se os dois valores forem iguais, a função retornará zero.

Por padrão, `sort()` converte todos os itens de um array em uma string e, em seguida, efetua a comparação. Isso significa que você não poderá ordenar um array de números de forma precisa sem especificar uma função de comparação. Por exemplo, é necessário incluir uma função de comparação para ordenar um array numérico de modo preciso, como em:

```
var numbers = [ 1, 5, 8, 4, 7, 10, 2, 6];
❶ numbers.sort(function(first, second) {
    return first - second;
});
console.log(numbers); // "[1, 2, 4, 5, 6, 7, 8, 10]"
❷ numeros.sort();
console.log(numbers); // "[1, 10, 2, 4, 5, 6, 7, 8]"
```

Nesse exemplo, a função de comparação ❶ passada para `sort()`, na realidade, é uma expressão de função. Note que não há nome para a

função; ela existe somente como uma referência que é passada para outra função (tornando-a uma *função anônima*). Subtrair os dois valores faz o resultado correto ser retornado pela função de comparação.

Compare isso com a segunda chamada a `sort()` ❷, que não utiliza uma função de comparação. A ordem do array é diferente do esperado, pois 10 vem depois de 1. Isso ocorre porque a comparação-padrão converte todos os valores para string antes de compará-los.

Parâmetros

Outro aspecto único das funções JavaScript é que você pode passar qualquer número de parâmetros para qualquer função sem causar erros. Isso ocorre porque os parâmetros de funções são armazenados em uma estrutura semelhante a arrays chamada `arguments`. Assim como um array comum em JavaScript, `arguments` pode ser estendido de modo a receber qualquer quantidade de valores. Os valores são referenciados por meio de índices numéricos, e há uma propriedade `length` para determinar quantos valores estão presentes.

O objeto `arguments` está automaticamente disponível em qualquer função. Isso significa que parâmetros nomeados em uma função existem apenas por conveniência e não limitam o número de argumentos que uma função pode aceitar.

NOTA O objeto `arguments` não é uma instância de `Array` e, portanto, não tem os mesmos métodos de um array. `Array.isArray(arguments)` sempre retornará `false`.

Por outro lado, o JavaScript também não ignora os parâmetros nomeados de uma função. O número de argumentos que uma função espera é armazenado na propriedade `length` da função. Lembre-se de que uma função é apenas um objeto, portanto ela pode ter propriedades. A propriedade `length` indica a *aridade* (arity) da função, ou seja, o número de parâmetros que ela espera. Conhecer a aridade das funções é importante em JavaScript porque as funções não gerarão um erro se você lhes passar parâmetros a mais ou a menos.

A seguir, temos um exemplo simples de uso de `arguments` e da aridade da

função; observe que o número de argumentos passados para a função não tem efeito sobre a aridade exibida:

```
function reflect(value) {
    return value;
}

console.log(reflect("Hi!")); // "Hi!"
console.log(reflect("Hi!", 25)); // "Hi!"
console.log(reflect.length); // 1

reflect = function() {
    return arguments[0]
};

console.log(reflect("Hi!")); // "Hi!"
console.log(reflect("Hi!", 25)); // "Hi!"
console.log(reflect.length); // 0
```

Nesse exemplo, inicialmente a função `reflect()` é definida com apenas um parâmetro nomeado, mas não há erro quando um segundo parâmetro é passado para a função. A propriedade `length` é igual a `1` porque há somente um parâmetro nomeado. A função `reflect()` então é redefinida sem parâmetros nomeados; ela retorna `arguments[0]`, que é o primeiro argumento passado para a função. Essa função redefinida funciona exatamente como a versão anterior, porém seu `length` é `0`.

A primeira versão da função `reflect()` é muito mais fácil de entender porque ela utiliza um argumento nomeado (como você faria em outras linguagens). A versão com o objeto `arguments` pode ser confusa porque não há argumentos nomeados, e você deve ler o corpo da função para determinar se há argumentos. Esse é o motivo pelo qual muitos desenvolvedores evitam usar `arguments`, a menos que seja necessário.

Em algumas ocasiões, porém, usar `arguments` é mais eficiente do que utilizar parâmetros nomeados. Por exemplo, suponha que você queira criar uma função que aceite qualquer número de parâmetros e que retorne a soma deles. Não é possível usar parâmetros nomeados porque não sabemos quantos serão necessários, portanto, nesse caso, usar `arguments` é a melhor opção:

```
function sum() {
```

```
    var result = 0,
        i = 0,
        len = arguments.length;

    while (i < len) {
        result += arguments[i];
        i++;
    }

    return result;
}

console.log(sum(1, 2)); // 3
console.log(sum(3, 4, 5, 6)); // 18
console.log(sum(50)); // 50
console.log(sum()); // 0
```

A função `sum()` aceita qualquer quantidade de parâmetros e efetua a soma deles por meio de uma iteração pelos valores em `arguments` utilizando um loop `while`. Isso é exatamente o mesmo que somar todos os valores de um array de números. A função funciona até mesmo quando nenhum parâmetro é passado porque a variável `result` é inicializada com um valor igual a `0`.

Sobrecarga

A maioria das linguagens orientadas a objetos suporta *sobrecarga de função*, que é a possibilidade de uma função ter diversas *assinaturas*. A assinatura de uma função é composta do nome da função e da quantidade e dos tipos de parâmetros esperados pela função. Assim, uma única função pode ter uma assinatura que aceite apenas uma string como argumento e outra que aceite dois números. A linguagem determina qual versão da função será chamada de acordo com os argumentos passados.

Como mencionado anteriormente, as funções em JavaScript aceitam qualquer quantidade de parâmetros, e os tipos de parâmetro que uma função aceita não são especificados. Isso significa que em JavaScript as funções não têm assinatura. A ausência de assinatura nas funções também significa a ausência de sobrecarga de função. Observe o que acontece quando você tenta declarar duas funções com o mesmo nome:

```
function sayMessage(message) {
    console.log(message);
}
function sayMessage() {
    console.log("Default message");
}
sayMessage("Hello!"); // exibe "Default message"
```

Se fosse em outra linguagem, a saída de `sayMessage("Hello!")` provavelmente seria `"Hello!"`. Em JavaScript, no entanto, quando várias funções são definidas com o mesmo nome, a função que aparecer por último em seu código será a vencedora. As funções declaradas anteriormente serão totalmente removidas e a última será usada. Mais uma vez, usar objetos ajuda a entender essa situação:

```
var sayMessage = new Function("message","console.log(message);");
sayMessage = new Function("console.log(\"Default message\");");
sayMessage("Hello!"); // exibe "Default message"
```

Ao observar o código dessa maneira, fica claro por que o código anterior não funcionou. Um objeto de função é atribuído duas vezes seguidas a `sayMessage`, portanto faz sentido que o primeiro objeto seja perdido.

O fato de as funções não terem assinatura em JavaScript não quer dizer que você não possa imitar o comportamento da sobrecarga de função. O número de parâmetros passados pode ser obtido por meio do objeto `arguments`, e essa informação pode ser usada para decidir o que deve ser feito. Por exemplo:

```
function sayMessage(message) {
    if (arguments.length === 0) {
        message = "Default message";
    }
    console.log(message);
}
sayMessage("Hello!"); // exibe "Hello!"
```

Nesse exemplo, a função `sayMessage()` se comporta de modo diferente de acordo com o número de argumentos passados. Se nenhum argumento

for passado (`arguments.length === 0`), uma mensagem-padrão será usada. Caso contrário, o primeiro parâmetro será usado como a mensagem. Isso é um pouco mais complicado que a sobrecarga de função em outras linguagens, mas o resultado final será o mesmo. Se você realmente deseja verificar se há tipos de dados diferentes, os operadores `typeof` ou `instanceof` poderão ser utilizados.

NOTA Na prática, comparar o parâmetro nomeado com `undefined` é mais comum do que basear-se em `arguments.length === 0`.

Métodos de objetos

Como mencionado no capítulo 1, você pode adicionar e remover propriedades dos objetos a qualquer momento. Quando um valor de propriedade é uma função, esse valor é considerado um método. Você pode adicionar um método a um objeto da mesma maneira que uma propriedade é adicionada. Por exemplo, no código a seguir, a variável `person` recebe um objeto literal com uma propriedade chamada `name` e um método chamado `sayName`:

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(person.name);
    }
};

person.sayName(); // exibe "Nicholas"
```

Note que a sintaxe para o valor de uma propriedade de qualquer tipo e um método é a mesma: um identificador seguido de dois-pontos e o valor. No caso de `sayName`, o valor é uma função. Você pode então chamar o método diretamente a partir do objeto, como em `person.sayName("Nicholas")`.

Objeto this

Você pode ter notado algo estranho no exemplo anterior. O método `sayName()` referencia `person.name` diretamente, o que gera um alto nível de acoplamento entre o método e o objeto. Isso representa um problema por

vários motivos. Em primeiro lugar, se o nome da variável for alterado, você terá de se lembrar de alterar a referência a esse nome no método. Em segundo lugar, esse tipo de alto acoplamento faz com que seja difícil usar a mesma função em diferentes objetos. Felizmente, em JavaScript, há uma maneira de resolver esse problema.

Todo escopo em JavaScript tem um objeto `this` que representa o objeto que chama a função. No escopo global, `this` representa o objeto global (`window` em web browsers). Quando uma função associada a um objeto é chamada, por padrão, o valor de `this` é igual a esse objeto. Portanto, em vez de referenciar diretamente um objeto em um método, `this` pode ser referenciado em seu lugar. Por exemplo, o código do exemplo anterior pode ser reescrito de modo a usar `this`:

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

person.sayName(); // exibe "Nicholas"
```

Esse código funciona do mesmo modo que a versão anterior, mas, dessa vez, `sayName()` referencia `this` em vez de `person`. Isso significa que você pode facilmente alterar o nome da variável ou até mesmo reutilizar a função em objetos diferentes.

```
function sayNameForAll() {
    console.log(this.name);
}

var person1 = {
    name: "Nicholas",
    sayName: sayNameForAll
};

var person2 = {
    name: "Greg",
    sayName: sayNameForAll
};

var name = "Michael";
```

```
person1.sayName(); // exibe "Nicholas"
person2.sayName(); // exibe "Greg"

sayNameForAll(); // exibe "Michael"
```

Nesse exemplo, uma função chamada `sayNameForAll()` inicialmente é criada. Em seguida, dois objetos literais são criados, em que `sayName` é definido para que seja igual à função `sayNameForAll`. As funções são apenas valores de referência, portanto elas podem ser definidas como valores de propriedade em qualquer objeto. Quando `sayName()` é chamada em `person1`, `"Nicholas"` é exibido; quando é chamada em `person2`, `"Greg"` é exibido. Isso ocorre porque `this` é definido quando a função é chamada, portanto `this.name` estará correto.

A última parte desse exemplo define uma variável global chamada `name`. Quando `sayNameForAll()` é chamada diretamente, ela exibe `"Michael"` porque a variável global é considerada uma propriedade do objeto global.

Mudando this

A capacidade de usar e de manipular o valor de `this` das funções é fundamental para um bom entendimento de orientação a objetos em JavaScript. As funções podem ser usadas em muitos contextos diferentes e elas devem funcionar em todas as situações. Embora `this` normalmente seja definido automaticamente, seu valor poderá ser alterado para obter resultados diferentes. Há três métodos que permitem mudar o valor de `this`. (Lembre-se de que as funções são objetos e os objetos podem ter métodos, portanto as funções também podem.)

Método call()

O primeiro método para manipular `this` é `call()`, que executa a função com um determinado valor de `this` e com parâmetros específicos. O primeiro parâmetro de `call()` é o valor que `this` deve ter quando a função for executada. Todos os parâmetros seguintes correspondem aos parâmetros que devem ser passados para a função. Por exemplo, suponha que você atualize `sayNameForAll()` para que essa função receba um parâmetro:

```
function sayNameForAll(label) {
    console.log(label + ":" + this.name);
}

var person1 = {
    name: "Nicholas"
};

var person2 = {
    name: "Greg"
};
var name = "Michael";

sayNameForAll.call(this, "global"); // exibe "global: Michael"
sayNameForAll.call(person1,"person1"); // exibe "person1:Nicholas"
sayNameForAll.call(person2,"person2"); // exibe "person2:Greg"
```

Nesse exemplo, `sayNameForAll()` aceita um parâmetro que é usado como label para exibir o valor de saída. Em seguida, a função é chamada três vezes. Note que não há parênteses depois do nome da função porque ela é acessada como um objeto em vez de ser acessada como um código a ser executado. A primeira chamada de função utiliza o `this` global e passa o parâmetro `"global"` para ser exibido em `"global: Michael"`. A mesma função é chamada mais duas vezes, uma para `person1` e outra para `person2`. Como o método `call()` está sendo usado, não é preciso adicionar a função diretamente a cada objeto – você especifica explicitamente o valor de `this` em vez de deixar a engine do JavaScript fazer isso automaticamente.

Método apply()

O segundo método de função que você pode usar para manipular `this` é `apply()`. O método `apply()` funciona exatamente como `call()`, exceto pelo fato de ele aceitar somente dois parâmetros: o valor de `this` e um array ou um objeto semelhante a um array contendo os parâmetros a serem passados para a função (isso significa que você pode usar um objeto `arguments` como segundo parâmetro). Desse modo, em vez de nomear individualmente cada parâmetro usando `call()`, você pode facilmente passar arrays para `apply()` como segundo argumento. Exceto por isso, `call()` e `apply()` se comportam de modo idêntico. Esse exemplo mostra o método `apply()` em ação:

```
function sayNameForAll(label) {
    console.log(label + ":" + this.name);
}

var person1 = {
    name: "Nicholas"
};

var person2 = {
    name: "Greg"
};

var name = "Michael";

sayNameForAll.apply(this,["global"]); // exibe "global:Michael"
sayNameForAll.apply(person1, ["person1"]); // exibe "person1:Nicholas"
sayNameForAll.apply(person2, ["person2"]); // exibe "person2:Greg"
```

Esse código parte do exemplo anterior e substitui `call()` por `apply()`; o resultado é exatamente o mesmo. O método usado normalmente dependerá do tipo de dado que você tiver. Se você já tiver um array de dados, use `apply()`; se tiver apenas variáveis individuais, use `call()`.

Método bind()

O terceiro método para mudar o valor de `this` é `bind()`. Esse método foi adicionado no ECMAScript 5 e se comporta de modo bem diferente dos outros dois. O primeiro argumento de `bind()` corresponde ao valor de `this` para a nova função. Todos os demais argumentos representam parâmetros nomeados que devem ser definidos permanentemente na nova função. Você ainda pode passar qualquer parâmetro que não seja definido de modo permanente mais tarde.

O código a seguir mostra dois exemplos que utilizam `bind()`. A função `sayNameForPerson1()` é criada ao efetuar a ligação (binding) do valor de `this` com `person1`, enquanto `sayNameForPerson2()` faz a ligação de `this` com `person2` e do primeiro parâmetro com `"person2"`.

```
function sayNameForAll(label) {
    console.log(label + ":" + this.name);
}

var person1 = {
    name: "Nicholas"
```

```
};

var person2 = {
    name: "Greg"
};
// cria uma função somente para person1
❶ var sayNameForPerson1 = sayNameForAll.bind(person1);
sayNameForPerson1("person1"); // exibe "person1:Nicholas"

// cria uma função somente para person2
❷ var sayNameForPerson2 = sayNameForAll.bind(person2, "person2");
sayNameForPerson2(); // exibe "person2:Greg"

// associar um método a um objeto não altera 'this'
❸ person2.sayName = sayNameForPerson1;
person2.sayName("person2"); // exibe "person2:Nicholas"
```

Nenhum parâmetro foi associado a `sayNameForPerson1()` ❶ , portanto continua sendo necessário passar `"label"` para gerar a saída. A função `sayNameForPerson2()` não só faz o binding de `this` com `person2` como também associa o primeiro parâmetro a `"person2"` ❷. Isso significa que você pode chamar `sayNameForPerson2()` sem passar nenhum argumento adicional. Essa parte do exemplo adiciona `sayNameForPerson1()` a `person2` com o nome `sayName` ❸. A função está associada a `person1`, portanto o valor de `this` não se altera mesmo que `sayNameForPerson1` agora seja uma função de `person2`. O método continua mostrando o valor de `person1.name`.

Sumário

As funções em JavaScript são únicas porque elas também são objetos, o que significa que podem ser acessadas, copiadas, sobrescritas e normalmente tratadas como qualquer outro objeto. A principal diferença entre uma função JavaScript e outros objetos está na propriedade interna especial `[[Call]]`, que contém as instruções de execução da função. O operador `typeof` procura essa propriedade interna em um objeto e, se ela for encontrada, `"function"` será retornado.

Há duas formas literais de função: as declarações e as expressões As declarações de função são compostas do nome da função à direita da palavra-chave `function` e são "içadas" (hoisted) para o topo do contexto em

que são definidas. As expressões de função são usadas nos locais em que outros valores também podem ser usados, por exemplo, em expressões de atribuição, em parâmetros de função ou no valor de retorno de outra função.

Como as funções são objetos, o construtor `Function` está presente. Você pode criar novas funções com o construtor `Function`, mas isso, normalmente, não é recomendável porque pode tornar o seu código mais difícil de entender e o debugging será muito mais complicado. Apesar do que foi dito, é provável que você vá ver o uso do construtor de vez em quando, em situações em que a verdadeira forma de uma função não é conhecida até a execução do programa.

É necessário ter um bom domínio sobre as funções para entender como a programação orientada a objetos funciona em JavaScript. Como em JavaScript não há o conceito de classes, tudo o que você tem para trabalhar a fim de implementar a agregação e a herança são as funções e os demais objetos.

CAPÍTULO 3

Entendendo os objetos

Mesmo que haja vários tipos prontos de referência em JavaScript, é provável que você vá criar seus próprios objetos com bastante frequência. Quando fizer isso, tenha em mente que os objetos em JavaScript são dinâmicos, o que significa que eles podem mudar a qualquer momento durante a execução do código. Enquanto as linguagens baseadas em classes restringem os objetos de acordo com uma definição de classe, os objetos em JavaScript não têm essa limitação.

A maior parte da programação em JavaScript é representada pela manipulação desses objetos, motivo pelo qual entender como eles funcionam é fundamental para compreender o JavaScript como um todo. Isso será discutido com detalhes mais adiante neste capítulo.

Definindo propriedades

De acordo com o capítulo 1, há duas formas básicas de criar seus próprios objetos: usando o construtor `Object` ou usando um objeto literal. Por exemplo:

```
var person1 = {
    name: "Nicholas"
};

var person2 = new Object();
person2.name = "Nicholas";

❶ person1.age = "Redacted";
person2.age = "Redacted";

❷ person1.name = "Greg";
person2.name = "Michael";
```

Tanto `person1` quanto `person2` são objetos que têm uma propriedade `name`.

Mais adiante no exemplo, ambos os objetos recebem uma propriedade chamada `age` ❶. Você poderia fazer isso imediatamente após a definição do objeto ou bem depois. Os objetos que você criar estarão sempre abertos a modificações, a menos que você especifique o contrário (mais sobre isso na seção "Evitando modificações em objetos"). A última parte do exemplo muda o valor de `name` em cada objeto ❷; os valores das propriedades podem ser alterados a qualquer momento também.

Quando uma propriedade é adicionada pela primeira vez a um objeto, o JavaScript utiliza um método interno chamado `[[Put]]` no objeto. O método `[[Put]]` cria um espaço no objeto para armazenar a propriedade. Isso pode ser comparado à adição de uma chave em uma tabela hash pela primeira vez. Essa operação especifica não somente o valor inicial como também alguns atributos da propriedade. Desse modo, no exemplo anterior, quando as propriedades `name` e `age` são inicialmente definidas em cada objeto, o método `[[Put]]` é chamado para cada uma delas.

O resultado da chamada a `[[Put]]` é a criação de uma *propriedade própria* no objeto. Uma propriedade própria simplesmente indica que a instância específica do objeto tem aquela propriedade. A propriedade é armazenada diretamente na instância e todas as operações sobre a propriedade devem ser executadas por meio desse objeto.

NOTA Propriedades próprias são diferentes de *propriedades de protótipos*, que serão discutidas no capítulo 4.

Quando um novo valor é atribuído a uma propriedade, uma operação diferente chamada `[[Set]]` é executada. Essa operação substitui o valor atual da propriedade pelo novo valor. No exemplo anterior, atribuir um novo valor a `name` resulta em uma chamada a `[[Set]]`. Veja a figura 3.1 para ter uma visão passo a passo do que aconteceu com `person1` internamente quando suas propriedades `name` e `age` foram alteradas.

Na primeira parte do diagrama, um objeto literal é usado para criar o objeto `person1`. Isso gera uma chamada implícita a `[[Put]]` para a propriedade `name`. Atribuir um valor a `person1.age` faz com que `[[Put]]` seja executado para a propriedade `age`. Entretanto atribuir um novo valor a `person1.name` faz com que uma operação `[[Set]]` seja executada na

propriedade `name`, sobrescrevendo o valor atual da propriedade.

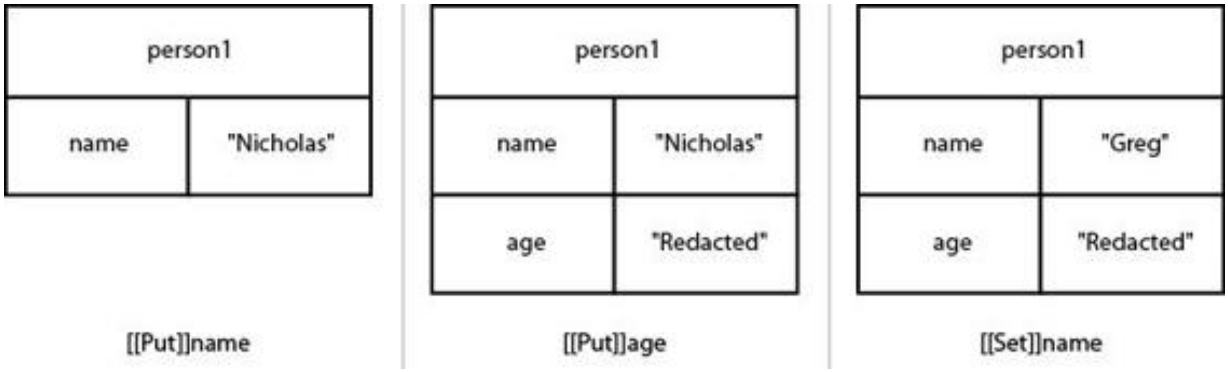


Figura 3.1 – Adicionando e alterando propriedades de um objeto.

Verificando a existência de propriedades

Como as propriedades podem ser adicionadas a qualquer momento, às vezes é necessário verificar se uma propriedade já existe em um objeto. Desenvolvedores JavaScript iniciantes muitas vezes usam padrões incorretos, como mostrado no próximo exemplo, para verificar se uma propriedade existe:

```
// não é confiável
if (person1.age) {
    // faz algo com a idade (age)
}
```

O problema com esse padrão está no modo como as conversões de tipo do JavaScript afetam o resultado. A condição `if` é avaliada como `true` se o valor for *truthy* (um objeto, uma string não vazia, um número diferente de zero ou `true`) e é avaliada como `false` se o valor for *falsy* (`null`, `undefined`, `0`, `NaN` ou uma string vazia). Como uma propriedade de objeto pode conter qualquer um desses valores falsy, o exemplo anterior pode resultar em um falso negativo. Por exemplo, se `person1.age` for `0`, a condição `if` não será verdadeira, mesmo que a propriedade exista. Uma maneira mais confiável de testar a existência de uma propriedade é por meio do operador `in`.

O operador `in` procura uma propriedade com um determinado nome em um objeto específico e retorna `true` se ela for encontrada. Com efeito, o operador `in` verifica se a chave especificada existe na tabela hash. Por

exemplo, aqui está o que acontece quando `in` é usado para verificar se algumas propriedades existem no objeto `person1`:

```
console.log("name" in person1); // true
console.log("age" in person1); // true
console.log("title" in person1); // false
```

Tenha em mente que os métodos são apenas propriedades que referenciam funções, portanto você pode verificar a existência de um método da mesma forma. O exemplo a seguir adiciona uma nova função `sayName()` a `person1` e utiliza `in` para confirmar a sua existência:

```
var person1 = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

console.log("sayName" in person1); // true
```

Na maioria dos casos, o operador `in` é a melhor forma de determinar se uma propriedade existe em um objeto. Essa solução tem também a vantagem adicional de não avaliar o valor da propriedade, o que pode ser importante se uma avaliação como essa puder causar um problema de desempenho ou um erro.

Em outros casos, no entanto, você pode querer verificar a existência de uma propriedade somente se ela for uma propriedade própria. O operador `in` verifica tanto as propriedades próprias quanto as propriedades de protótipos, de modo que você deverá adotar uma abordagem diferente. Use o método `hasOwnProperty()`, que está presente em todos os objetos e retorna `true` somente se a propriedade dada existir e for uma propriedade própria. Por exemplo, o código a seguir compara o resultado do uso de `in` em relação a `hasOwnProperty()` em diferentes propriedades de `person1`:

```
var person1 = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
```

```
    }
};

console.log("name" in person1); // true
console.log(person1.hasOwnProperty("name")); // true

console.log("toString" in person1); // true
❶ console.log(person1.hasOwnProperty("toString")); // false
```

Nesse exemplo, `name` é uma propriedade própria de `person1`, portanto tanto o operador `in` quanto o método `hasOwnProperty()` retornam `true`. O método `toString()`, entretanto, é uma propriedade do protótipo, presente em todos os objetos. O operador `in` retorna `true` para `toString()`, mas `hasOwnProperty` retorna `false` ❶ . Essa importante distinção será discutida com mais detalhes no capítulo 4.

Removendo propriedades

Assim como as propriedades podem ser adicionadas aos objetos a qualquer momento, elas também podem ser removidas. Simplesmente definir uma propriedade como `null` não remove totalmente a propriedade do objeto. Essa operação chama `[[Set]]` com o valor `null`, o que, como você viu antes neste capítulo, somente substitui o valor da propriedade. Você deve usar o operador `delete` para remover totalmente uma propriedade de um objeto.

O operador `delete` atua em uma única propriedade do objeto e chama uma operação interna de nome `[[Delete]]`. Você pode pensar nessa operação como a remoção de um par chave/valor de uma tabela hash. Se o operador `delete` conseguir remover a propriedade, ele retornará `true`. (Algumas propriedades não podem ser removidas, e isso será discutido em mais detalhes neste capítulo). O código a seguir mostra como o operador `delete` funciona.

```
var person1 = {
    name: "Nicholas"
};

console.log("name" in person1); // true

delete person1.name; // true (não exibe nada)
console.log("name" in person1); // false
```

```
❶ console.log(person1.name); // undefined
```

Nesse exemplo, a propriedade `name` é removida de `person1`. O operador `in` retorna `false` depois que a operação é concluída. Note também que tentar acessar uma propriedade que não existe retornará `undefined` ❶. A figura 3.2 mostra como `delete` afeta um objeto:

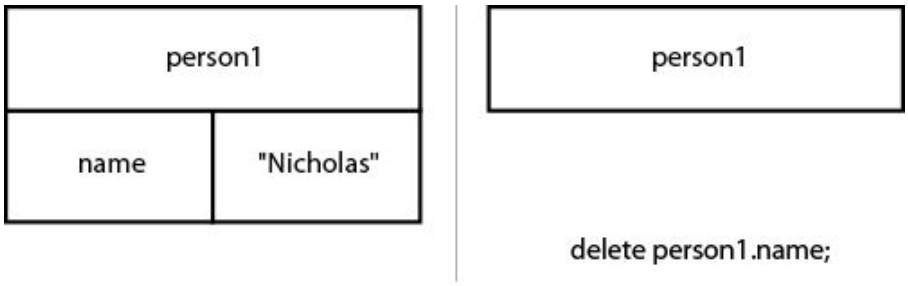


Figura 3.2 – Quando a propriedade `name` *é removida, ela desaparece completamente de person1.*

Enumeração

Por padrão, todas as propriedades que você adicionar a um objeto são *enumeráveis*, o que significa que você pode iterar por elas usando um loop `for-in`. Propriedades enumeráveis têm seu atributo interno `[[Enumerable]]` definido como `true`. O loop `for-in` enumera todas as propriedades enumeráveis de um objeto, atribuindo o nome da propriedade a uma variável. Por exemplo, o loop no código a seguir exibe os nomes das propriedades de um objeto e os seus valores:

```
var property;
for (property in object) {
    console.log("Name :" + property);
    console.log("Value:" + object[property]);
}
```

A cada iteração do loop `for-in`, a variável `property` é preenchida com a próxima propriedade enumerável do objeto até que todas as propriedades desse tipo tenham sido usadas. Nesse ponto, o loop termina e a execução do código continua normalmente. A notação de colchetes foi usada nesse exemplo para obter o valor da propriedade do objeto e exibi-lo no console, que é um dos casos principais de uso da notação de colchetes em JavaScript.

Se você precisar apenas de uma lista das propriedades de um objeto para usar posteriormente em seu programa, o ECMAScript5 introduziu o método `Object.keys()` para obter um array de nomes de propriedades enumeráveis, como mostrado a seguir:

```
❶ var properties = Object.keys(object);

// Se quiser imitar o comportamento do loop for-in
var i, len;

for (i=0, len=properties.length; i < len; i++) {
    console.log("Name :" + properties[i]);
    console.log("Value:" + object[properties[i]]);
}
```

Esse exemplo usa o método `Object.keys()` para obter as propriedades enumeráveis de um objeto ❶. Um loop `for` é então usado para iterar pelas propriedades e exibir o nome e o valor. Normalmente, `Object.keys()` será usado em situações em que você deseja operar sobre um array de nomes de propriedades e `for-in` quando não precisar de um array.

NOTA Há uma diferença entre as propriedades enumeráveis retornadas em um loop `for-in` e aquelas retornadas por `Object.keys()`. O loop `for-in` também enumera propriedades do protótipo, enquanto `Object.keys()` retorna somente as propriedades próprias (da instância). A diferença entre propriedades do protótipo e propriedades próprias será discutida no capítulo 4.

Tenha em mente que nem todas as propriedades são enumeráveis. De fato, a maioria dos métodos nativos dos objetos tem o seu atributo `[[Enumerable]]` definido como `false`. Você pode verificar se uma propriedade é enumerável usando o método `propertyIsEnumerable()`, que está presente em todos os objetos:

```
var person1 = {
    name: "Nicholas"
};

console.log("name" in person1); // true
❶ console.log(person1.propertyIsEnumerable("name")); // true

var properties = Object.keys(person1);

console.log("length" in properties); // true
❷ console.log(properties.propertyIsEnumerable("length")); // false
```

Nesse caso, a propriedade `name` é enumerável, pois é uma propriedade própria definida em `person1` ❶. A propriedade `length` do array `properties`, por outro lado, não é enumerável ❷ porque é uma propriedade incluída em `Array.prototype`. Você irá descobrir que muitas propriedades nativas não são enumeráveis por padrão.

Tipos de propriedade

Há dois tipos diferentes de propriedade: propriedades de dados e propriedades de acesso. As *propriedades de dados* contêm um valor, como a propriedade `name` dos exemplos anteriores deste capítulo. O comportamento-padrão do método `[[Put]]` consiste em criar uma propriedade de dado, e todos os exemplos até este ponto do livro têm usado propriedades de dados. As *propriedades de acesso* não contêm valores; em vez disso, elas definem uma função a ser chamada quando a propriedade é lida (chamada *getter*) e uma função a ser chamada quando a propriedade é atualizada (chamada *setter*). As propriedades de acesso exigem somente um getter ou um setter, embora ambas possam existir.

Há uma sintaxe especial para definir as propriedades de acesso usando um objeto literal:

```
var person1 = {
❶    _name: "Nicholas",
❷    get name() {
       console.log("Reading name");
       return this._name;
    },
❸    set name(value) {
       console.log("Setting name to %s", value);
       this._name = value;
    }
};

console.log(person1.name); // "Reading name" e em seguida "Nicholas"

person1.name = "Greg";
console.log(person1.name); // "Setting name to Greg" e em seguida "Greg"
```

Esse exemplo define uma propriedade de acesso chamada `name`. Há uma

propriedade de dado chamada `_name` que contém o valor propriamente dito da propriedade ❶. (O underline no início é uma convenção comum para indicar que a propriedade é considerada privada, embora, na realidade, ela continue sendo pública). A sintaxe usada para definir o getter ❷ e o setter ❸ para `name` se parece bastante com uma função, mas sem a palavra-chave `function`. As palavras-chave especiais `get` e `set` são usadas antes do nome da propriedade de acesso, seguidas por parênteses e do corpo da função. Espera-se que os getters retornem um valor, enquanto os setters recebem o valor sendo atribuído à propriedade como argumento.

Embora esse exemplo use `_name` para armazenar o valor da propriedade, você poderia facilmente armazenar o valor em uma variável ou até mesmo em outro objeto. Esse exemplo simplesmente adiciona um log ao comportamento da propriedade; em geral, não há razão para usar propriedades de acesso se você estiver apenas armazenando o valor em outra propriedade – basta usar a própria propriedade. As propriedades de acesso serão mais úteis se você quiser que a atribuição de um valor acione algum tipo de comportamento ou quando a leitura de um valor exigir o cálculo do valor de retorno desejado.

NOTA Não é necessário definir tanto um setter quanto um getter; você pode escolher um deles ou ambos. Se apenas um getter for definido, a propriedade será somente de leitura; uma tentativa de escrever nessa propriedade irá falhar silenciosamente em modo não restrito e irá causar um erro em modo restrito. Se somente um setter for definido, a propriedade será somente de escrita, e uma tentativa de ler o valor irá falhar silenciosamente tanto em modo restrito quanto em modo não restrito.

Atributos de propriedades

Antes do ECMAScript 5, não havia maneira de especificar se uma propriedade deveria ser enumerável. Na verdade, não havia nenhum modo de acessar os atributos internos de uma propriedade. O ECMAScript 5 mudou isso ao introduzir várias maneiras de interagir diretamente com esses atributos, assim como introduziu novos atributos para suportar funcionalidades adicionais. Atualmente é possível criar propriedades que se comportam da mesma forma que as propriedades prontas do JavaScript. Esta seção discute em detalhes os atributos das

propriedades tanto de dados quanto de acesso, começando pelos atributos que elas têm em comum.

Atributos comuns

Há dois atributos internos que são compartilhados entre as propriedades de dados e as propriedades de acesso. Um deles é o `[[Enumerable]]`, que determina se você pode iterar pela propriedade. O outro é `[[Configurable]]`, que determina se uma propriedade pode ser alterada. Uma propriedade configurável pode ser removida usando `delete` e pode ter seus atributos alterados a qualquer momento (isso também significa que as propriedades configuráveis podem ser alteradas, passando de propriedades de dados para propriedades de acesso e vice-versa). Por padrão, todas as propriedades declaradas em um objeto são enumeráveis e configuráveis.

Se quiser mudar os atributos das propriedades, você poderá usar o método `Object.defineProperty()`. Esse método aceita três argumentos: o objeto que tem a propriedade, o nome da propriedade e um objeto *descritor da propriedade* que contém os atributos a serem definidos. O descritor tem propriedades com o mesmo nome que os atributos internos, mas sem os colchetes duplos. Portanto você pode usar `enumerable` para definir `[[Enumerable]]` e `configurable` para definir `[[Configurable]]`. Por exemplo, suponha que você queira que uma propriedade do objeto seja não enumerável e não configurável:

```
var person1 = {
❶    name: "Nicholas"
};

Object.defineProperty(person1, "name", {
❷    enumerable: false
});

console.log("name" in person1); // true
❸ console.log(person1.propertyIsEnumerable("name")); // false

var properties = Object.keys(person1);
console.log(properties.length); // 0

Object.defineProperty(person1, "name", {
```

```
❹    configurable: false
});

// tenta apagar a propriedade
delete person1.name;
❺ console.log("name" in person1); // true
console.log(person1.name); // "Nicholas"

❻ Object.defineProperty(person1, "name", { // Erro!
    configurable: true
});
```

A propriedade `name` é definida normalmente ❶, porém é modificada em seguida para que seu atributo `[[Enumerable]]` seja definido com `false` ❷. O método `propertyIsEnumerable()` agora retorna `false` ❸ porque ele faz referência ao novo valor de `[[Enumerable]]`. Depois disso, `name` é alterado para que seja não configurável ❹. A partir daí, tentativas de apagar `name` falham porque a propriedade não pode ser alterada, de modo que `name` continuará presente em `person1` ❺. Chamar `Object.defineProperty()` em `name` mais uma vez também não resultará em mudanças na propriedade. Com efeito, `name` torna-se uma propriedade definitiva de `person1`.

A última parte do código tenta redefinir a propriedade `name` para que ela seja configurável novamente ❻. No entanto isso gera um erro, pois uma propriedade não configurável não pode se tornar uma propriedade configurável novamente. Tentar transformar uma propriedade de dado em uma propriedade de acesso ou vice-versa também deverá lançar um erro, nesse caso.

NOTA Quando o JavaScript estiver sendo executado em modo restrito, tentar apagar uma propriedade não configurável resultará em erro. Em modo não restrito, essa operação falhará silenciosamente.

Atributos de propriedades de dados

As propriedades de dados têm dois atributos adicionais que as propriedades de acesso não têm. O primeiro é `[[Value]]`, que contém o valor da propriedade. Esse atributo é preenchido automaticamente quando uma propriedade é criada em um objeto. Todos os valores de propriedade são armazenados em `[[Value]]`, mesmo que o valor seja uma

função.

O segundo atributo é `[[Writable]]`, que é um valor booleano para indicar se o valor da propriedade pode ser reescrito. Por padrão, todas as propriedades podem ser reescritas, a menos que você especifique o contrário.

Com esses dois atributos adicionais, você pode definir completamente uma propriedade de dado usando `Object.defineProperty()`, mesmo que a propriedade ainda não exista. Considere este código:

```
var person1 = {
    name: "Nicholas"
};
```

Você já viu essa porção de código neste capítulo; ela adiciona a propriedade `name` a `person1` e define o seu valor. O mesmo resultado pode ser obtido com o código a seguir (mais extenso):

```
var person1 = {};

Object.defineProperty(person1, "name", {
    value: "Nicholas",
    enumerable: true,
    configurable: true,
    writable: true
});
```

Quando `Object.defineProperty()` é chamado, inicialmente ele verifica se a propriedade existe. Se não existir, uma nova propriedade será adicionada, com os atributos especificados no descritor. No exemplo anterior, `name` ainda não é uma propriedade de `person1`, portanto ela será criada.

Ao definir uma nova propriedade usando `Object.defineProperty()`, é importante especificar todos os atributos porque, do contrário, os atributos booleanos serão definidos automaticamente como `false` por padrão. Por exemplo, o código a seguir cria uma propriedade `name` que é não enumerável, não configurável e que não pode ser atualizada porque ela não define explicitamente nenhum dos atributos como `true` na chamada a `Object.defineProperty()`.

```
var person1 = {};
```

```
Object.defineProperty(person1, "name", {
    value: "Nicholas"
});
console.log("name" in person1); // true
console.log(person1.propertyIsEnumerable("name")); //false

delete person1.name;
console.log("name" in person1); // true

person1.name = "Greg";
console.log(person1.name); // "Nicholas"
```

Nesse código, você não pode fazer nada com a propriedade `name`, exceto ler o seu valor; qualquer outra operação estará bloqueada. Se você estiver alterando uma propriedade que já existe, tenha em mente que somente os atributos que você especificar serão alterados.

NOTA Propriedades que não podem ser atualizadas lançam um erro em modo restrito se você tenta mudar o seu valor. Em modo não restrito, a operação falhará silenciosamente.

Atributos de propriedades de acesso

As propriedades de acesso também têm dois atributos adicionais. Como não há nenhum valor armazenado nessas propriedades, os atributos `[[Value]]` e `[[Writable]]` não são necessários. Em seu lugar, as propriedades de acesso têm `[[Get]]` e `[[Set]]`, que contêm as funções getter e setter, respectivamente. Como na forma literal de getters e setters, é preciso definir apenas um desses atributos para criar a propriedade.

NOTA Se você tentar criar uma propriedade com atributos tanto de dados quanto de acesso, um erro será gerado.

A vantagem de usar atributos de propriedades de acesso em vez de utilizar a notação literal de objeto para definir as propriedades de acesso é que você também pode definir essas propriedades em objetos que já existem. Se quiser usar a notação literal de objeto, você deve definir as propriedades de acesso quando o objeto for criado.

Como ocorre com as propriedades de dados, você também poderá especificar se as propriedades de acesso são configuráveis ou enumeráveis. Considere o seguinte código de um exemplo anterior:

```
var person1 = {
```

```
    _name: "Nicholas",
    get name() {
        console.log("Reading name");
        return this._name;
    },
    set name(value) {
        console.log("Setting name to %s", value);
        this._name = value;
    }
};
```

Esse código também pode ser escrito da seguinte maneira:

```
var person1 = {
    _name: "Nicholas"
};

Object.defineProperty(person1, "name", {
    get: function() {
        console.log("Reading name");
        return this._name;
    },
    set: function(value) {
        console.log("Setting name to %s", value);
        this._name = value;
    },
    enumerable: true,
    configurable: true
});
```

Note que os nomes das propriedades `get` e `set` do objeto passado para `Object.defineProperty()` são propriedades de dados que contêm uma função. O formato de objeto literal para a propriedade de acesso não pode ser usado nesse caso.

Definir os outros atributos (`[[Enumerable]]` e `[[Configurable]]`) permite alterar o modo como a propriedade de acesso funciona. Por exemplo, você pode criar uma propriedade não configurável, não enumerável e que não pode ser atualizada desta maneira:

```
var person1 = {
    _name: "Nicholas"
};
```

```
Object.defineProperty(person1, "name", {
    get: function() {
        console.log("Reading name");
❶         return this._name;
    }
});
console.log("name" in person1); // true
console.log(person1.propertyIsEnumerable("name")); // false
delete person1.name;
console.log("name" in person1); // true
person1.name = "Greg";
console.log(person1.name); // "Nicholas"
```

Nesse código, `name` é uma propriedade de acesso que tem somente um getter ❶. Não há setter nem outros atributos explicitamente definidos como `true`, portanto o valor poderá ser lido, mas não poderá ser alterado.

NOTA Assim como ocorre com as propriedades de acesso definidas por meio de notação literal de objeto, uma propriedade de acesso sem um setter lançará um erro em modo restrito se você tentar mudar o seu valor. Em modo não restrito, a operação falhará silenciosamente. Tentativas de ler uma propriedade de acesso que tenha somente um setter definido sempre retornarão `undefined`.

Definindo várias propriedades

Também é possível definir várias propriedades em um objeto simultaneamente se `Object.defineProperties()` for usado no lugar de `Object.defineProperty()`. Esse método aceita dois argumentos: o objeto que será usado e um objeto contendo todas as informações das propriedades. As chaves do segundo argumento correspondem aos nomes das propriedades de dados e os valores são objetos descritores que definem os atributos dessas propriedades. Por exemplo, o código a seguir define duas propriedades:

```
var person1 = {};

Object.defineProperties(person1, {

❶     // propriedade de dados para armazenar informações
    _name: {
        value: "Nicholas",
        enumerable: true,
        configurable: true,
```

```
        writable: true
    },
❷   // propriedade de acesso
    name: {
        get: function() {
            console.log("Reading name");
            return this._name;
        },
        set: function(value) {
            console.log("Setting name to %s", value);
            this._name = value;
        },
        enumerable: true,
        configurable: true
    }
});
```

Esse exemplo define `_name` como uma propriedade de dado que contém informações ❶ e `name` como uma propriedade de acesso ❷. É possível definir qualquer quantidade de propriedades usando `Object.defineProperties()`; podemos até mesmo alterar as propriedades já existentes e criar novas propriedades ao mesmo tempo. O efeito será o mesmo que chamar `Object.defineProperty()` várias vezes.

Obtendo atributos de propriedades

Se houver necessidade de acessar os atributos das propriedades, isso poderá ser feito em JavaScript usando o método `Object.getOwnPropertyDescriptor()`. Como o nome sugere, esse método somente funciona com propriedades próprias. Ele aceita dois argumentos: o objeto a ser manipulado e o nome da propriedade a ser acessada. Se a propriedade existir, você deverá receber um objeto descritor com quatro propriedades: `configurable`, `enumerable` e os outros dois valores adequados ao tipo da propriedade. Mesmo que um atributo não tenha sido especificamente definido, você receberá um objeto que contém o valor apropriado para esse atributo. Por exemplo, este código cria uma propriedade e verifica os seus atributos:

```
var person1 = {
```

```
    name: "Nicholas"
};
var descriptor = Object.getOwnPropertyDescriptor(person1, "name");

console.log(descriptor.enumerable); // true
console.log(descriptor.configurable); // true
console.log(descriptor.writable); // true
console.log(descriptor.value); // "Nicholas"
```

Nesse caso, uma propriedade chamada `name` é definida como parte de um objeto literal. A chamada a `Object.getOwnPropertyDescriptor()` retorna um objeto com `enumerable`, `configurable`, `writable` e `value`, apesar de esses atributos não terem sido explicitamente definidos por meio de `Object.defineProperty()`.

Evitando modificações em objetos

Os objetos, assim como as propriedades, têm atributos internos que definem seu comportamento. Um desses atributos é `[[Extensible]]` – um valor booleano que indica se o objeto pode ser modificado. Todos os objetos que você criar serão *extensíveis* por padrão, o que significa que novas propriedades podem ser adicionadas ao objeto a qualquer momento. Isso foi visto várias vezes neste capítulo. Ao definir `[[Extensible]]` com `false`, você pode evitar que novas propriedades sejam adicionadas a um objeto. Há três maneiras de conseguir esse resultado.

Evitando extensões

Uma maneira de criar objetos não extensíveis é com o método `Object.preventExtensions()`. Esse método aceita apenas um argumento, que é o objeto que você deseja tornar não extensível. Assim que esse método for usado em um objeto, você jamais poderá adicionar novas propriedades a ele novamente. O valor de `[[Extensible]]` pode ser verificado com o método `Object.isExtensible()`. O próximo exemplo mostra os dois métodos em ação:

```
var person1 = {
    name: "Nicholas"
};
```

```
❶ console.log(Object.isExtensible(person1)); // true

❷ Object.preventExtensions(person1);
console.log(Object.isExtensible(person1)); // false

❸ person1.sayName = function() {
    console.log(this.name);
};
console.log("sayName" in person1); // false
```

Depois de criar `person1`, o atributo `[[Extensible]]` do objeto é verificado no exemplo ❶ antes de torná-lo não extensível ❷. Depois que `person1` não for mais extensível, o método `sayName()` ❸ não poderá ser adicionado a esse objeto.

NOTA Tentar adicionar uma propriedade a um objeto não extensível irá lançar um erro em modo restrito. Em modo não restrito, a operação falhará silenciosamente. Sempre use o modo restrito com objetos não extensíveis para que você saiba quando um objeto não extensível está sendo usado incorretamente.

Selando objetos

A segunda maneira de criar um objeto não extensível é *selar* o objeto. Um objeto selado é não extensível, e todas as suas propriedades não são configuráveis. Isso significa que você não pode adicionar, remover nem mudar o tipo (de dados para acesso ou vice-versa) das propriedades. Se um objeto estiver selado, você poderá somente ler e atualizar as suas propriedades.

O método `Object.seal()` pode ser usado em um objeto para selá-lo. Quando isso acontecer, o atributo `[[Extensible]]` será definido como `false` e todas as propriedades terão seu atributo `[[Configurable]]` definido como `false`. Você pode verificar se um objeto está selado usando `Object.isSealed()`, como neste exemplo:

```
var person1 = {
    name: "Nicholas"
};
console.log(Object.isExtensible(person1)); // true
console.log(Object.isSealed(person1)); // false

❶ Object.seal(person1);
❷ console.log(Object.isExtensible(person1)); // false
```

```
console.log(Object.isSealed(person1)); // true

❸ person1.sayName = function() {
    console.log(this.name);
};
console.log("sayName" in person1); // false

❹ person1.name = "Greg";
console.log(person1.name); // "Greg"

❺ delete person1.name;
console.log("name" in person1); // true
console.log(person1.name); // "Greg"

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable); // false
```

Esse código sela `person1` ❶, de modo que não podemos adicionar nem remover propriedades do objeto. Como todos os objetos selados não são extensíveis, `Object.isExtensible()` retorna `false` ❷ quando usado em `person1`, e uma tentativa de adicionar um método chamado `sayName()` ❸ falha silenciosamente. Além disso, embora `person1.name` possa ser alterado com sucesso para um novo valor ❹, uma tentativa de apagá-lo irá falhar ❺.

Se você tiver familiaridade com Java ou com C++, também deverá estar familiarizado com objetos selados. Quando uma nova instância de um objeto baseado em uma classe é criada em uma dessas linguagens, novas propriedades não podem ser adicionadas a esse objeto. Entretanto, se uma propriedade contiver um objeto, você poderá modificá-lo. De fato, objetos selados em JavaScript representam a maneira de permitir que você tenha o mesmo grau de controle sem usar classes.

NOTA Não se esqueça de usar o modo restrito com objetos selados para que um erro seja gerado quando alguém tentar usar o objeto incorretamente.

Congelando objetos

A última maneira de criar objetos não extensíveis é *congelá-los*. Se um objeto estiver congelado, não será possível adicionar nem remover propriedades, mudar seus tipos nem atualizar qualquer propriedade de dados. Essencialmente, um objeto congelado é um objeto selado em que as propriedades de dados também são apenas para leitura. Objetos

congelados não podem ser "descongelados", portanto eles permanecem no estado em que estavam quando foram congelados. Um objeto pode ser congelado por meio de `Object.freeze()`, e podemos determinar se um objeto está congelado usando `Object.isFrozen()`. Por exemplo:

```
var person1 = {
    name: "Nicholas"
};

console.log(Object.isExtensible(person1)); // true
console.log(Object.isSealed(person1)); // false
console.log(Object.isFrozen(person1)); // false

❶ Object.freeze(person1);
❷ console.log(Object.isExtensible(person1)); // false
❸ console.log(Object.isSealed(person1)); // true
console.log(Object.isFrozen(person1)); // true

person1.sayName = function() {
    console.log(this.name);
};

console.log("sayName" in person1); // false

❹ person1.name = "Greg";
console.log(person1.name); // "Nicholas"

delete person1.name;
console.log("name" in person1); // true
console.log(person1.name); // "Nicholas"

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable); // false
console.log(descriptor.writable); // false
```

Nesse exemplo, o objeto `person1` está congelado ❶. Objetos congelados também são considerados não extensíveis e selados, de modo que `Object.isExtensible()` retorna `false` ❷ e `Object.isSealed()` retorna `true` ❸. A propriedade `name` não pode ser alterada, portanto, mesmo atribuindo-lhe `"Greg"`, a operação falha ❹ e verificações subsequentes de `name` continuam retornando `"Nicholas"`.

NOTA Objetos congelados são como "fotografias" de objetos em um determinado momento. Eles são muito limitados e raramente devem ser usados. Como ocorre com todos os objetos não extensíveis, você deve usar o modo restrito com objetos congelados.

Sumário

Pensar em objetos JavaScript como uma tabela hash em que as propriedades são apenas pares de chaves e valores pode ajudar. As propriedades dos objetos são acessadas tanto por meio da notação de ponto quanto pela notação de colchetes com um identificador em forma de string. Uma propriedade pode ser adicionada a qualquer momento atribuindo-lhe um valor e pode ser igualmente removida por meio do operador `delete`. É sempre possível verificar se uma propriedade existe usando o operador `in` com o nome de uma propriedade e um objeto. Se a propriedade em questão for uma propriedade própria, o método `hasOwnProperty()`, existente em todos os objetos, também poderá ser utilizado. Todas as propriedades que você adicionar a um objeto serão enumeráveis por padrão, o que significa que elas poderão ser usadas em um loop `for-in` ou ser acessadas com `Object.keys()`.

Há dois tipos de propriedade: propriedade de dados e propriedade de acesso. As propriedades de dados armazenam valores, e você pode lê-las ou sobrescrevê-las. Quando uma propriedade de dado contiver uma função, ela será considerada um método do objeto. De modo diferente das propriedades de dados, as propriedades de acesso não armazenam valores próprios: elas usam uma combinação de getters e setters para executar ações específicas. Você pode criar tanto propriedades de dados quanto propriedades de acesso usando a notação de objeto literal diretamente.

Todas as propriedades têm vários atributos associados. Esses atributos definem o comportamento da propriedade. Tanto as propriedades de dado quanto as propriedades de acesso têm os atributos `[[Enumerable]]` e `[[Configurable]]`. As propriedades de dados também têm os atributos `[[Writable]]` e `[[Value]]`, enquanto as propriedades de acesso têm os atributos `[[Get]]` e `[[Set]]`. Por padrão, `[[Enumerable]]` e `[[Configurable]]` são definidas como `true` para todas as propriedades, e `[[Writable]]` também é definida como `true` para todas as propriedades de dados. Esses atributos podem ser alterados por meio de `Object.defineProperty()` ou `Object.defineProperties()`. Também é possível acessar esses atributos usando

`Object.getOwnPropertyDescriptor()`.

Se você quiser bloquear as propriedades de um objeto, há três maneiras de fazer isso. Se `Object.preventExtensions()` for usado, os objetos não mais permitirão que novas propriedades sejam adicionadas. Você também pode selar um objeto com o método `Object.seal()`, que torna o objeto não extensível e faz com que suas propriedades não sejam configuráveis. O método `Object.freeze()` cria um objeto congelado, que é um objeto selado com propriedades de dados que não podem ser atualizadas. Tenha cuidado com objetos não extensíveis e sempre use o modo restrito para que tentativas de acessar os objetos incorretamente lancem um erro.

CAPÍTULO 4

Construtores e protótipos

Você pode ir bem longe em JavaScript sem entender o que são construtores e protótipos, mas não irá realmente apreciar a linguagem sem ter um bom conhecimento deles. Como o JavaScript não tem classes, cabe aos construtores e aos protótipos proporcionar um comportamento semelhante aos objetos. Entretanto só porque alguns padrões se assemelham a classes não significa que eles se comportem da mesma maneira. Neste capítulo, você irá explorar os construtores e os protótipos em detalhes e verá como o JavaScript os utiliza para criar objetos.

Construtores

Um *construtor* é simplesmente uma função usada com o operador `new` para criar um objeto. Até este ponto, você viu diversos construtores prontos do JavaScript, como `Object`, `Array` e `Function`. A vantagem dos construtores é que os objetos criados com o mesmo construtor têm as mesmas propriedades e os mesmos métodos. Se quiser criar vários objetos semelhantes, você poderá implementar seus próprios construtores e, portanto, seus próprios tipos de referência.

Como um construtor é somente uma função, ele deve ser definido da mesma maneira. A única diferença é que os nomes dos construtores devem começar com uma letra maiúscula para diferenciá-lo de outras funções. Por exemplo, observe a função `Person` vazia a seguir:

```
function Person() {
    // intencionalmente vazia
}
```

Essa função é construtora, mas, sintaticamente, não há nenhuma

diferença entre ela e qualquer outra função. A pista que indica que `Person` é um construtor está no nome – a primeira letra é maiúscula.

Após o construtor ter sido definido, você pode começar a criar instâncias, como o que foi feito para os dois objetos `Person` a seguir:

```
var person1 = new Person();
var person2 = new Person();
```

Se não houver parâmetros para passar para o construtor, você poderá até mesmo omitir os parênteses:

```
var person1 = new Person;
var person2 = new Person;
```

Embora o construtor `Person` não retorne nada explicitamente, tanto `person1` quanto `person2` são considerados instâncias do novo tipo `Person`. O operador `new` cria automaticamente um objeto do tipo especificado e o retorna. Isso significa também que o operador `instanceof` pode ser utilizado para deduzir o tipo do objeto. O código a seguir mostra `instanceof` em ação com os objetos recém-criados:

```
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Person); // true
```

Como `person1` e `person2` foram criados com o construtor `Person`, `instanceof` retorna `true` quando verifica se esses objetos são instâncias do tipo `Person`.

Você também pode verificar o tipo de uma instância usando a propriedade `constructor`. Toda instância de objeto é automaticamente criada com uma propriedade chamada `constructor` que contém uma referência à função construtora que a criou. Para objetos *genéricos* (aqueles criados por meio de uma notação literal de objeto ou com o construtor `Object`), a propriedade `constructor` é definida como `Object`, enquanto para objetos criados com um construtor personalizado, `constructor` apontará para esse construtor. Por exemplo, `Person` corresponde à propriedade `constructor` de `person1` e de `person2`:

```
console.log(person1.constructor === Person); // true
console.log(person2.constructor === Person); // true
```

A função `console.log` exibe `true` em ambos os casos, pois ambos os objetos

foram criados com o construtor `Person`.

Mesmo que esse relacionamento exista entre uma instância e o seu construtor, continua sendo aconselhável usar `instanceof` para verificar o tipo de uma instância. Isso se deve ao fato de a propriedade `constructor` poder ser sobrescrita, e, sendo assim, poderá não ser totalmente precisa.

É claro que um construtor vazio não é muito útil. O propósito de um construtor é fazer com que seja fácil criar mais objetos com as mesmas propriedades e os mesmos métodos. Para fazer isso, simplesmente adicione qualquer propriedade que você quiser a `this` no construtor, como no exemplo a seguir:

```
function Person(name) {
❶    this.name = name;
❷    this.sayName = function() {
       console.log(this.name);
   };
}
```

Essa versão do construtor `Person` aceita um único parâmetro nomeado chamado `name` e o atribui à propriedade `name` do objeto `this` ❶. O construtor também adiciona o método `sayName()` ao objeto ❷. O objeto `this` é automaticamente criado pelo operador `new` quando o construtor é chamado e corresponde a uma instância do tipo do construtor. (Nesse caso, `this` é uma instância de `Person`.) Não há necessidade de retornar um valor na função porque o operador `new` gera o valor de retorno.

Agora você pode usar o construtor `Person` para criar objetos com uma propriedade `name` inicializada:

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name); // "Nicholas"
console.log(person2.name); // "Greg"

person1.sayName(); // exibe "Nicholas"
person2.sayName(); // exibe "Greg"
```

Cada objeto tem sua própria propriedade `name`, portanto `sayName()` deve retornar um valor diferente de acordo com o objeto em que o método for usado.

NOTA Você também pode explicitamente chamar `return` dentro de um construtor. Se o valor retornado for um objeto, ele será retornado no lugar da nova instância do objeto recém-criado. Se o valor retornado for um valor primitivo, o objeto recém-criado será usado e o valor retornado será ignorado.

Os construtores permitem inicializar uma instância de um tipo de maneira consistente, efetuando todas as configurações de propriedades necessárias antes que o objeto possa ser usado. Por exemplo, `Object.defineProperty()` pode ser usado em um construtor para ajudar a inicializar a instância:

```
function Person(name) {
    Object.defineProperty(this, "name", {
        get: function() {
            return name;
        },
        set: function(newName) {
            name = newName;
        },
        enumerable: true,
        configurable: true
    });

    this.sayName = function() {
        console.log(this.name);
    };
}
```

Nessa versão do construtor `Person`, a propriedade `name` é uma propriedade de acesso que usa o parâmetro `name` para armazenar o nome propriamente dito. Isso é possível porque os parâmetros nomeados agem como variáveis locais.

Não se esqueça de sempre chamar os construtores com `new`; caso contrário, você correrá o risco de mudar o objeto global em vez de alterar o objeto recém-criado. Considere o que acontece no código a seguir:

```
var person1 = Person("Nicholas"); // nota: "new" está ausente

console.log(person1 instanceof Person); // false
console.log(typeof person1); // "undefined"
console.log(name); // "Nicholas"
```

Quando `Person` é chamado como uma função, sem o operador `new`, o valor

de `this` no construtor é igual ao objeto `this` global. A variável `person1` não contém um valor porque o construtor `Person` depende de `new` para fornecer um valor de retorno. Sem `new`, `Person` é somente uma função sem uma instrução de `return`. A atribuição a `this.name` cria uma variável global chamada `name`, em que o nome passado para `Person` é armazenado. O capítulo 6 descreve uma solução tanto para esse problema quanto para padrões mais complexos de composição de objetos.

NOTA Um erro será lançado se você chamar o construtor `Person` em modo restrito sem usar `new`. Isso ocorre porque, em modo restrito, `this` não é atribuído ao objeto global. Em vez disso, `this` permanece como `undefined`; e um erro ocorrerá sempre que você tentar criar uma propriedade em `undefined`.

Os construtores permitem configurar instâncias de objetos com as mesmas propriedades, mas os construtores por si só não eliminam as redundâncias de código. Nos exemplos de código até agora, cada instância tem seu próprio método `sayName()`, embora `sayName()` não seja diferente. Isso significa que, se você tiver cem instâncias de um objeto, haverá cem cópias de uma função que faz exatamente o mesmo, porém com dados diferentes.

Seria muito mais eficiente se todas as instâncias compartilhassem um método e esse método pudesse usar `this.name` para acessar o dado apropriado. É nesse caso que os protótipos entram em cena.

Protótipos

Você pode pensar em um *protótipo* como uma receita para um objeto. Quase todas as funções (com exceção de algumas funções prontas) têm uma propriedade `prototype`, que é usada durante a criação de novas instâncias. Esse protótipo é compartilhado entre todas as instâncias do objeto, e essas instâncias podem acessar as propriedades do protótipo. Por exemplo, o método `hasOwnProperty()` é definido no protótipo `Object` genérico, mas pode ser acessado a partir de qualquer objeto como se fosse uma propriedade própria, como mostrado neste exemplo:

```
var book = {
    title = "Princípios de orientação a objetos em JavaScript"
```

```
};
console.log("title" in book); // true
console.log(book.hasOwnProperty("title")); // true
console.log("hasOwnProperty" in book); // true
console.log(book.hasOwnProperty("hasOwnProperty")); // false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty")); // true
```

Embora não haja nenhuma definição de `hasOwnProperty()` em `book`, esse método pode ser acessado como `book.hasOwnProperty()` porque sua definição existe em `Object.prototype`. Lembre-se de que o operador `in` retorna `true` tanto para propriedades de protótipos *quanto* para propriedades próprias.

IDENTIFICANDO UMA PROPRIEDADE DE PROTÓTIPO

É possível determinar se uma propriedade está em um protótipo ao usar uma função como esta:

```
function hasPrototypeProperty(object, name) {
    return name in object && !object.hasOwnProperty(name);
}
console.log(hasPrototypeProperty(book, "title")); // false
console.log(hasPrototypeProperty(book, "hasOwnProperty")); // true"
```

Se a propriedade estiver em um objeto, mas `hasOwnProperty()` retornar `false`, então ela estará no protótipo.

A propriedade [[Prototype]]

Uma instância mantém o controle de seu protótipo por meio de uma propriedade interna chamada `[[Prototype]]`. Essa propriedade é um ponteiro para o objeto referente ao protótipo que a instância está usando. Quando um novo objeto é criado usando `new`, a propriedade `prototype` do construtor é atribuída à propriedade `[[Prototype]]` desse novo objeto. A figura 4.1 mostra como a propriedade `[[Prototype]]` permite que várias instâncias de um tipo de objeto se refiram ao mesmo protótipo, o que pode reduzir a duplicação de código:

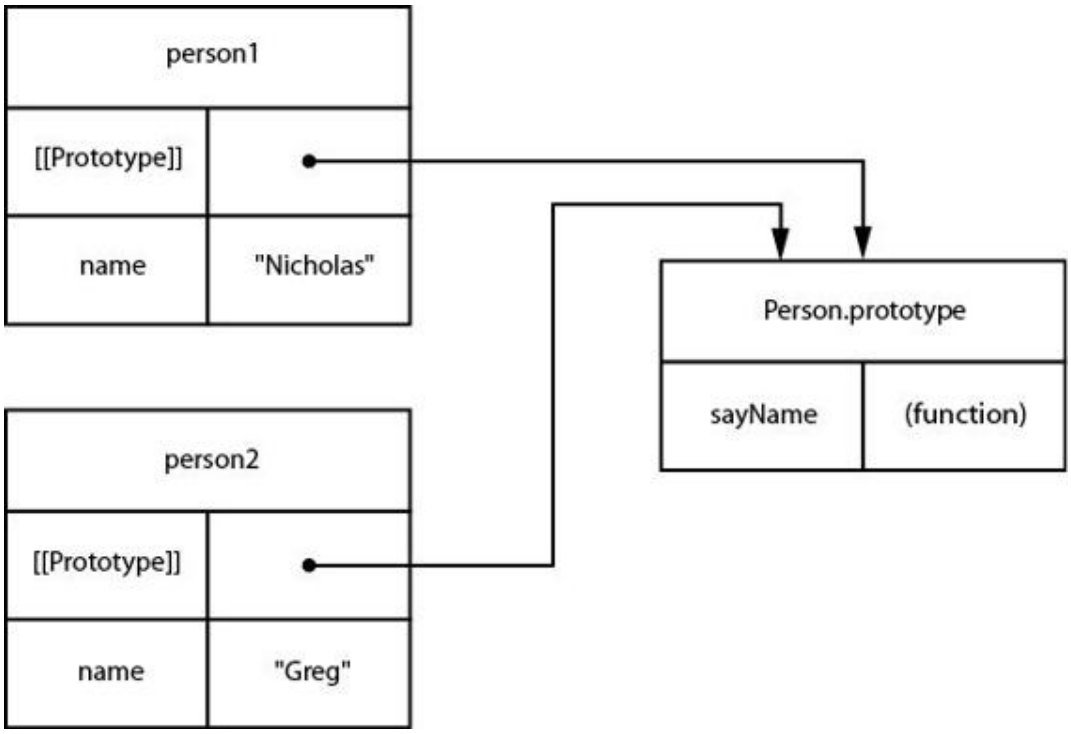


Figura 4.1 – A propriedade [[Prototype]] de person1 e de person2 aponta para o mesmo protótipo.

O valor da propriedade `[[Prototype]]` pode ser lido por meio do método `Object.getPrototypeOf()` de um objeto. Por exemplo, o código a seguir verifica o `[[Prototype]]` de um objeto genérico e vazio:

```
❶ var object = {};
var prototype = Object.getPrototypeOf(object);

console.log(prototype === Object.prototype); // true
```

Para qualquer objeto genérico como esse ❶, `[[Prototype]]` será sempre uma referência a `Object.prototype`.

NOTA Algumas engines de JavaScript também suportam uma propriedade chamada `__proto__` em todos os objetos. Essa propriedade permite tanto ler quanto escrever na propriedade `[[Prototype]]`. O Firefox, o Safari, o Chrome e o Node.js suportam essa propriedade, e `__proto__` está prestes a ser padronizada no ECMAScript 6.

Também é possível verificar se um objeto é protótipo de outro usando o método `isPrototypeOf()`, que está incluído em todos os objetos:

```
var object = {}

console.log(Object.prototype.isPrototypeOf(object)); // true
```

Como `object` é somente um objeto genérico, seu protótipo deve ser

`Object.prototype`, o que significa que `isPrototypeOf()` deve retornar `true`.

Quando uma propriedade é lida em um objeto, a engine do JavaScript inicialmente procura uma propriedade própria com esse nome. Se a engine encontrar devidamente uma propriedade própria com esse nome, ela retornará esse valor. Se não houver nenhuma propriedade própria com esse nome no objeto-alvo, o JavaScript procurará no objeto `[[Prototype]]`. Se houver uma propriedade de protótipo com esse nome, o valor dessa propriedade será retornado. Se a busca terminar sem que uma propriedade com o nome correto tenha sido encontrada, `undefined` será retornado.

Considere o código a seguir, em que um objeto é inicialmente criado sem nenhuma propriedade própria:

```
var object = {};

❶ console.log(object.toString()); // "[object Object]"

object.toString = function() {
    return "[object Custom]";
};

❷ console.log(object.toString()); // "[object Custom]"
// apaga a propriedade própria
delete object.toString;

❸ console.log(object.toString()); // "[object Object]"

// sem efeito, pois delete só funciona em propriedades próprias
delete object.toString;
console.log(object.toString()); // "[object Object]"
```

Nesse exemplo, o método `toString()` é disponibilizado pelo protótipo e retorna `"[object Object]"` ❶ por padrão. Se uma propriedade própria chamada `toString()` for definida, essa propriedade será usada sempre que `toString()` for chamada no objeto novamente ❷. A propriedade própria *encobre* a propriedade do protótipo, de modo que essa propriedade de mesmo nome não será mais usada. A propriedade do protótipo será usada novamente somente se a propriedade própria for apagada do objeto ❸ (tenha em mente que você não pode apagar uma propriedade do protótipo a partir de uma instância porque o operador `delete` atua somente em propriedades próprias). A figura 4.2 mostra o que aconteceu

no exemplo anterior.

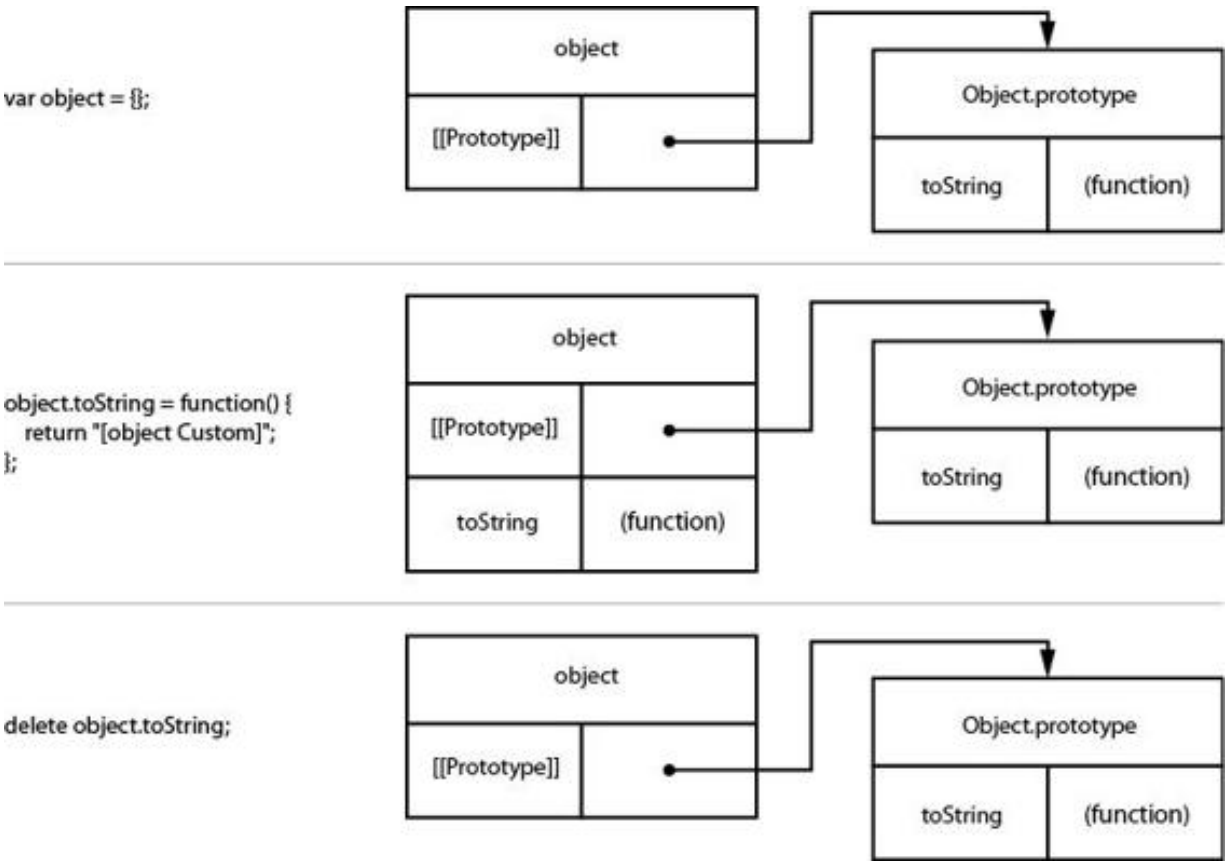


Figura 4.2 – Um objeto sem propriedades próprias (parte superior) tem somente os métodos de seu protótipo. Adicionar uma propriedade `toString()` *ao objeto (parte central) faz com que a propriedade do protótipo seja substituída até que a propriedade própria seja apagada (parte inferior).*

Esse exemplo também enfatiza um conceito importante: não se pode atribuir um valor a uma propriedade do protótipo a partir de uma instância. Como você pode ver na seção central da figura 4.2, atribuir um valor a `toString` faz com que uma nova propriedade própria seja criada na instância, deixando a propriedade do protótipo inalterada.

Usando protótipos com construtores

A natureza compartilhada dos protótipos faz com que eles sejam ideais para definir métodos somente uma vez para todos os objetos de um dado tipo. Como os métodos tendem a fazer o mesmo para todas as instâncias, não há razão para que cada instância deva ter seu próprio conjunto de métodos.

É muito mais eficiente colocar os métodos no protótipo e usar `this` para acessar a instância atual. Por exemplo, considere o construtor `Person` a seguir:

```
function Person(name) {
    this.name = name;
}
❶ Person.prototype.sayName = function() {
    console.log(this.name);
};
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name); // "Nicholas"
console.log(person2.name); // "Greg"

person1.sayName(); // exibe "Nicholas"
person2.sayName(); // exibe "Greg"
```

Nessa versão do construtor `Person`, `sayName()` está definido no protótipo ❶, e não no construtor. As instâncias do objeto funcionam exatamente da mesma maneira que no exemplo mostrado anteriormente neste capítulo, mesmo que `sayName()` seja agora uma propriedade do protótipo em vez de ser uma propriedade própria. Como `person1` e `person2` são referências de base para as chamadas a `sayName()`, o valor `this` é atribuído a `person1` e a `person2`, respectivamente.

Outros tipos de dados também podem ser armazenados no protótipo, mas tenha cuidado ao usar valores de referência. Como esses valores são compartilhados pelas instâncias, não espere que uma instância possa alterar os valores que outra instância irá acessar. O exemplo a seguir mostra o que pode acontecer se você não tomar cuidado com o local para onde seus valores de referência apontam:

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayName = function() {
    console.log(this.name);
}

❶ Person.prototype.favorites = [];
```

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

person1.favorites.push("pizza");
person2.favorites.push("quinoa");

console.log(person1.favorites); // "pizza, quinoa"
console.log(person2.favorites); // "pizza, quinoa"
```

A propriedade `favorites` ❶ é definida no protótipo, o que significa que `person1.favorites` e `person2.favorites` apontam para *o mesmo array*. Qualquer valor adicionado à propriedade `favorites` de qualquer pessoa será um elemento do array que está no protótipo. Esse pode não ser o comportamento que você deseja, portanto é importante ter cuidado com o que for definido no protótipo.

Embora você possa adicionar propriedades ao protótipo, uma a uma, muitos desenvolvedores usam um padrão mais sucinto que envolve substituir o protótipo por um objeto literal:

```
function Person(name) {
    this.name = name;
}

Person.prototype = {
❶    sayName: function() {
        console.log(this.name);
    },
❷    toString: function() {
        return "[Person " + this.name + "]";
    }
};
```

Esse código define dois métodos no protótipo: `sayName()` ❶ e `toString()` ❷. Esse padrão se tornou bem popular porque elimina a necessidade de digitar `Person.prototype` diversas vezes. Porém há um efeito colateral do qual você deve estar ciente:

```
var person1 = new Person("Nicholas");
console.log(person1 instanceof Person); // true
console.log(person1.constructor === Person); // false
❶ console.log(person1.constructor === Object); // true
```

Usar a notação de objeto literal para sobrescrever o protótipo alterou a

propriedade `constructor`, de modo que agora ela aponta para `Object` ❶ em vez de apontar para `Person`. Isso aconteceu porque a propriedade `constructor` está no protótipo, e não na instância do objeto. Quando uma função é criada, sua propriedade `prototype` é criada com uma propriedade `constructor` igual à função. Esse padrão sobrescreve completamente o objeto referente ao protótipo, o que significa que `constructor` será proveniente do novo objeto (genérico) criado, atribuído a `Person.prototype`. Para evitar isso, restaure a propriedade `constructor` para um valor adequado ao sobrescrever o protótipo:

```
function Person(name) {
    this.name = name;
}
Person.prototype = {
❶    constructor: Person,

    sayName: function() {
        console.log(this.name);
    },

    toString: function() {
        return "[Person " + this.name + "]";
    }
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1 instanceof Person); // true
console.log(person1.constructor === Person); // true
console.log(person1.constructor === Object); // false

console.log(person2 instanceof Person); // true
console.log(person2.constructor === Person); // true
console.log(person2.constructor === Object); // false
```

Nesse exemplo, a propriedade `constructor` é especificamente atribuída no protótipo ❶. Fazer com que essa seja a primeira propriedade no protótipo para que você não se esqueça de incluí-la constitui uma boa prática.

Talvez o aspecto mais interessante das relações entre construtores, protótipos e instâncias esteja no fato de não haver uma ligação direta entre a instância e o construtor. Entretanto há uma ligação direta entre a

instância e o protótipo e entre o protótipo e o construtor. A figura 4.3 ilustra essa relação.

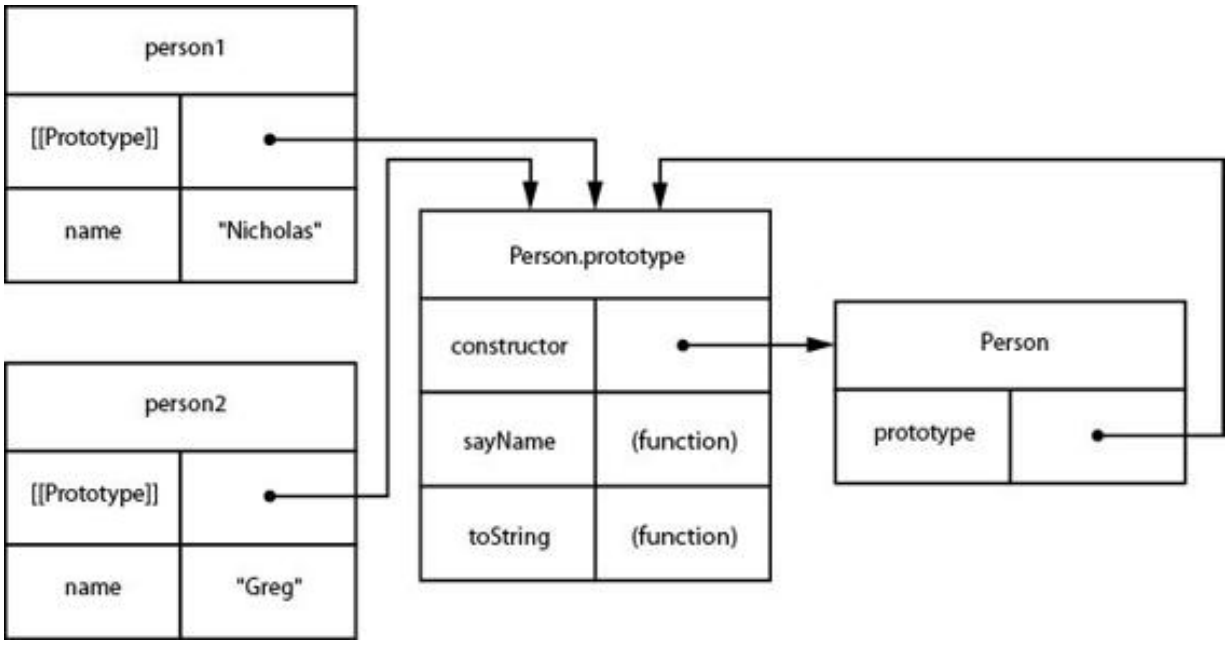


Figura 4.3 – Uma instância e o seu construtor estão ligados pelo protótipo.

Por causa da natureza dessa relação, qualquer rompimento entre a instância e o protótipo também causará um rompimento entre a instância e o construtor.

Alterando os protótipos

Como todas as instâncias de um tipo particular referenciam um protótipo compartilhado, você pode estender todos esses objetos em conjunto a qualquer momento. Lembre-se de que a propriedade `[[Prototype]]` contém somente um ponteiro para o protótipo, e qualquer alteração no protótipo estará imediatamente disponível a qualquer instância que o referenciar. Isso significa que você pode literalmente adicionar novos membros a um protótipo a qualquer momento, e essas mudanças serão refletidas nas instâncias atuais, como neste exemplo:

```
function Person(name) {
    this.name = name;
}
Person.prototype = {
    constructor: Person,
```

```
❶    sayName: function() {
       console.log(this.name;)
   },
❷    toString: function() {
       return "[Person " + this.name + "]";
   }
};
❸ var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log("sayHi" in person1); // false
console.log("sayHi" in person2); // false

// adiciona um novo método
❹ Person.prototype.sayHi = function() {
    console.log("Oi");
};

❺ person1.sayHi(); // exibe "Hi"
person2.sayHi(); // exibe "Hi"
```

Nesse código, o tipo `Person` começa com apenas dois métodos: `sayName()` ❶ e `toString()` ❷. Duas instâncias de `Person` são criadas ❸ e, em seguida, o método `sayHi()` ❹ é adicionado ao protótipo. A partir daí, ambas as instâncias podem acessar `sayHi()` ❺. A procura por uma propriedade nomeada ocorre sempre que a propriedade for acessada, o que proporciona fluidez ao processo.

A capacidade de modificar o protótipo a qualquer momento tem repercussões interessantes para objetos selados e congelados. Quando `Object.seal()` e `Object.freeze()` forem utilizados em um objeto, você estará agindo *somente* na instância do objeto e em suas propriedades próprias. Não é possível adicionar novas propriedades próprias nem mudar as que já existem em objetos congelados, mas, certamente, você poderá continuar adicionando propriedades ao protótipo e poderá estender esses objetos, como mostrado a seguir:

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

❶ Object.freeze(person1);

❷ Person.prototype.sayHi = function() {
```

```
    console.log("Hi");
}

person1.sayHi(); // exibe "Hi"
person2.sayHi(); // exibe "Hi"
```

Nesse exemplo, há duas instâncias de `Person`. A primeira (`person1`) está congelada ❶, enquanto a segunda é um objeto normal. Ao adicionar `sayHi()` ao protótipo ❷, tanto `person1` quanto `person2` ganham um novo método, aparentemente, contradizendo o status de congelado de `person1`. A propriedade `[[Prototype]]` é considerada uma propriedade própria da instância e, embora a propriedade em si esteja congelada, o valor (um objeto) não está.

NOTA Na prática, provavelmente você não usará protótipos dessa maneira com muita frequência quando estiver desenvolvendo em JavaScript. Entretanto é importante entender a relação que existe entre os objetos e seus protótipos, e exemplos incomuns como esse ajudam a esclarecer os conceitos.

Protótipos de objetos prontos

Nesse ponto, você deve estar se perguntando se os protótipos também permitem modificar os objetos prontos que são padrões na engine do JavaScript. A resposta é sim. Todos os objetos prontos têm construtores e, sendo assim, têm protótipos que podem ser alterados. Por exemplo, adicionar um novo método a ser usado em todos os arrays é simples: basta modificar `Array.prototype`:

```
Array.prototype.sum = function() {
    return this.reduce(function(previous, current){
        return previous + current;
    });
};

var numbers = [1, 2, 3, 4, 5, 6];
var result = numbers.sum();

console.log(result); // 21
```

Nesse exemplo, um método chamado `sum()`, que simplesmente soma todos os itens do array e retorna o resultado, é criado em `Array.prototype`. O array `numbers` tem acesso automaticamente a esse método por meio do protótipo.

Em `sum()`, `this` se refere a `numbers`, que é uma instância de `Array`, de modo que o método é livre para usar outros métodos de array, por exemplo, `reduce()`.

Você deve estar lembrado de que strings, numbers e booleans têm tipos wrapper primitivos prontos, que são usados para acessar valores primitivos, como se fossem objetos. Se o protótipo de um tipo wrapper primitivo for modificado, como no exemplo a seguir, você poderá adicionar mais funcionalidades a esses valores primitivos:

```
String.prototype.capitalize = function() {
    return this.charAt(0).toUpperCase() + this.substring(1);
};

var message = "hello world!";
console.log(message.capitalize()); // "Hello world!"
```

Esse código cria um novo método `capitalize()` para strings. O tipo `String` é o wrapper primitivo para strings, e modificar o seu protótipo significa que todas as strings automaticamente terão essas mudanças à disposição.

NOTA Embora possa parecer divertido e interessante modificar objetos prontos para testar novas funcionalidades, não é uma boa ideia fazer isso em ambiente de produção. Os desenvolvedores esperam que os objetos prontos se comportem de uma determinada maneira e que tenham determinados métodos. Alterar objetos prontos deliberadamente viola essas expectativas e faz com que outros desenvolvedores não tenham certeza de como os objetos devem funcionar.

Sumário

Os construtores são apenas funções normais chamadas com o operador `new`. Você pode definir seus próprios construtores sempre que quiser ter vários objetos com as mesmas propriedades. Os objetos criados por meio de construtores podem ser identificados pelo operador `instanceof` ou pelo acesso direto à sua propriedade `constructor`.

Toda função tem uma propriedade `prototype` que define qualquer propriedade compartilhada pelos objetos criados com um determinado construtor. Métodos e propriedades com valores primitivos compartilhados normalmente são definidos nos protótipos, enquanto todas as demais propriedades são definidas no construtor. A propriedade

`constructor` está definida no protótipo porque ela é compartilhada pelas instâncias do objeto.

O protótipo de um objeto é armazenado internamente na propriedade `[[Prototype]]`. Essa propriedade é uma referência, e não uma cópia. Se o protótipo for alterado em algum momento, essas mudanças ocorrerão em todas as instâncias por causa da maneira como o JavaScript procura as propriedades. Ao tentar acessar uma propriedade de um objeto, uma pesquisa é feita no objeto, em busca de qualquer propriedade própria com o nome especificado. Se uma propriedade própria não for encontrada, a procura será feita no protótipo. Esse sistema de pesquisa implica que o protótipo pode continuar a sofrer mudanças e que as instâncias dos objetos que referenciam esse protótipo irão refletir essas mudanças imediatamente.

Objetos prontos também têm protótipos que podem ser modificados. Embora não seja recomendável fazer isso em ambiente de produção, pode ser útil para testes e provas de conceito relacionados a novas funcionalidades.

CAPÍTULO 5

Herança

Aprender a criar objetos é o primeiro passo para entender a programação orientada a objetos. O segundo passo é entender a herança. Nas linguagens orientadas a objetos tradicionais, as classes herdam propriedades de outras classes. Em JavaScript, entretanto, a herança pode ocorrer entre objetos que não tenham uma estrutura semelhante a classes que defina a sua relação. Você já está familiarizado com o sistema usado nessa herança: ela é feita por meio dos protótipos.

Cadeia de protótipos e Object.prototype

A abordagem que o JavaScript tem para lidar com a herança chama-se *cadeia de protótipos* ou *herança prototípica*. Como vimos no capítulo 4, as propriedades dos protótipos estão automaticamente disponíveis nas instâncias dos objetos, o que é uma forma de herança. As instâncias dos objetos herdam as propriedades do protótipo. Como o protótipo também é um objeto, ele tem seu próprio protótipo e herda suas propriedades. Essa é a *cadeia de protótipos*: um objeto herda de seu protótipo, enquanto esse protótipo, por sua vez, herda de seu protótipo, e assim por diante.

Todos os objetos, incluindo aqueles que você mesmo define, automaticamente herdam de `Object`, a menos que você especifique o contrário (isso será discutido posteriormente neste capítulo). Mais especificamente, todos os objetos herdam de `Object.prototype`. Qualquer objeto definido por meio de notação literal tem seu `[[Prototype]]` definido com `Object.prototype`, o que significa que ele herda as propriedades de `Object.prototype`, como a variável `book` neste exemplo:

```
var book = {
    title: "Princípios de orientação a objetos em JavaScript"
```

```
};
var prototype = Object.getPrototypeOf(book);
console.log(prototype === Object.prototype); // true
```

Nesse caso, `book` tem um protótipo igual a `Object.prototype`. Nenhum código adicional é necessário para que isso aconteça, pois esse é o comportamento-padrão quando novos objetos são criados. Essa relação significa que `book` automaticamente recebe os métodos de `Object.prototype`.

Métodos herdados de Object.prototype

Muitos métodos usados em capítulos anteriores deste livro estão, na verdade, definidos em `Object.prototype` e, portanto, são herdados por todos os demais objetos. Esses métodos são:

- `hasOwnProperty()` – Determina se uma propriedade própria com o nome especificado existe.
- `propertyIsEnumerable()` – Determina se uma propriedade própria é enumerável.
- `isPrototypeOf()` – Determina se o objeto é protótipo de outro.
- `valueOf()` – Retorna a representação do valor do objeto.
- `toString()` – Retorna uma representação do objeto em forma de string.

Esses cinco métodos estão presentes em todos os objetos por meio de herança. Os dois últimos são importantes quando você quiser que os objetos funcionem de forma consistente em JavaScript e, às vezes, você irá querer defini-los por conta própria.

valueOf()

O método `valueOf()` é chamado sempre que um operador é usado em um objeto. Por padrão, `valueOf()` simplesmente retorna a instância do objeto. Os tipos wrapper primitivos sobrescrevem `valueOf()` de modo que uma string é retornada para `String`, um booleano é retornado para `Boolean` e um número é retornado para `Number`. Da mesma maneira, o método `valueOf()` do objeto `Date` retorna o instante no tempo em milissegundos (assim como é feito por `Date.prototype.getTime()`). É isso que permite que você escreva um

código que compare datas desta maneira:

```
var now = new Date();
var earlier = new Date(2010, 1, 1);
❶ console.log(now > earlier); // true
```

Nesse exemplo, `now` é um `Date` que representa o instante atual, e `earlier` é uma data fixa no passado. Quando o operador maior que (>) é usado ❶, o método `valueOf()` é chamado em ambos os objetos antes de a comparação ser executada. Você pode até mesmo subtrair uma data de outra e obter a diferença de tempo por causa de `valueOf()`.

Será sempre possível definir o seu próprio método `valueOf()` se seus objetos tiverem o propósito de ser usados com operadores. Se um método `valueOf()` for definido, tenha em mente que você não irá alterar o modo como os operadores funcionam, mas apenas o valor a ser usado com o comportamento-padrão do operador.

toString()

O método `toString()` é chamado como fallback sempre que `valueOf()` retorna um valor de referência no lugar de um valor primitivo. Ele também será chamado implicitamente em valores primitivos sempre que o JavaScript estiver esperando uma string. Por exemplo, quando uma string for usada como um operando da operação de adição, o outro operando será automaticamente convertido em uma string. Se o outro operando for um valor primitivo, ele será convertido para a sua representação em forma de string (por exemplo, `true` se torna `"true"`), mas se ele for um valor de referência, `valueOf()` será chamado. Se `valueOf()` retornar um valor de referência, `toString()` será chamado e o valor retornado será usado. Por exemplo:

```
var book = {
    title: "Princípios de orientação a objetos em JavaScript"
};
var message = "Book = " + book;
console.log(message); // "Book = [object Object]"
```

Esse código constrói a string por meio da combinação de `"Book ="` com `book`. Como `book` é um objeto, seu método `toString()` é chamado. Esse

método é herdado de `Object.prototype` e retorna o valor-padrão `"[object Object]"` na maioria das engines de JavaScript. Se você estiver satisfeito com esse valor, não haverá necessidade de alterar o método `toString()` de seu objeto. Às vezes, porém, é útil definir seu próprio método `toString()` para que as conversões de string retornem um valor que ofereça mais informações. Suponha, por exemplo, que você queira que o código anterior exiba o título do livro:

```
var book = {
    title: "Princípios de orientação a objetos em JavaScript",
    toString: function() {
        return "[Book " + this.title + "]"
    }
};
var message = "Book = " + book;

// "Book = [Book Princípios de orientação a objetos em JavaScript]"
❶ console.log(message);
```

Esse código define um método `toString()` personalizado para `book`, que retorna um valor mais útil ❶ que a versão herdada. Normalmente, não é preciso se preocupar com a definição de um método `toString()` personalizado, mas é bom saber que isso é possível se for necessário.

Modificando Object.prototype

Todos os objetos herdam de `Object.prototype` por padrão, portanto mudanças em `Object.prototype` afetam todos os objetos. Essa é uma situação muito perigosa. No capítulo 4, você foi aconselhado a não modificar os protótipos de objetos prontos, e esse conselho é reforçado para `Object.prototype`. Observe o que pode acontecer:

```
Object.prototype.add = function(value) {
    return this + value;
};
var book = {
    title: "Princípios de orientação a objetos em JavaScript"
};

console.log(book.add(5)); // "[object Object]5"
console.log("title".add("end")); // "titleend"
```

```
// em um web browser
console.log(document.add(true)); // "[object HTMLDocument]true"
console.log(window.add(5)); // "[object Window]5"
```

Adicionar `Object.prototype.add()` faz com que todos os objetos tenham um método `add()`, não importa se isso faça sentido ou não. Essa questão tem sido um problema não só para os desenvolvedores, mas também para o comitê que trabalha na linguagem JavaScript: foi necessário inserir novos métodos em locais diferentes porque adicionar métodos em `Object.prototype` pode provocar consequências inesperadas.

Outro aspecto desse problema envolve a adição de propriedades enumeráveis em `Object.prototype`. No exemplo anterior, `Object.prototype.add()` é uma propriedade enumerável, o que significa que ela irá aparecer em um loop `for-in`, como em:

```
var empty = {}
for (var property in empty) {
    console.log(property);
}
```

Nesse caso, um objeto vazio irá exibir `"add"` como propriedade porque ela existe no protótipo e é enumerável. Considerando a frequência com que a estrutura `for-in` é usada em JavaScript, modificar `Object.prototype` com propriedades enumeráveis tem o potencial de afetar muito código. Por essa razão, Douglas Crockford recomenda sempre usar `hasOwnProperty()` em loops `for-in`[1], como em:

```
var empty = {};

for (var property in empty) {
    if (empty.hasOwnProperty(property)) {
        console.log(property);
    }
}
```

Se, por um lado, essa abordagem é eficiente contra possíveis propriedades indesejadas dos protótipos, por outro, ela também limita o uso do loop `for-in` apenas às propriedades próprias, que pode ou não ser o que você quer. Sua melhor aposta para ter o máximo de flexibilidade é não modificar `Object.prototype`.

Herança entre objetos

O tipo mais simples de herança é a herança entre objetos. Tudo o que você deve fazer é especificar que objeto deve ser o `[[Prototype]]` do novo objeto. Objetos literais têm seu `[[Prototype]]` definido como `Object.prototype` implicitamente, mas `[[Prototype]]` também pode ser explicitamente especificado no método `Object.create()`.

O método `Object.create()` aceita dois argumentos. O primeiro argumento é o objeto que corresponderá ao `[[Prototype]]` do novo objeto. O segundo argumento opcional é um objeto contendo descritores de propriedade no mesmo formato usado por `Object.defineProperties()` (veja o capítulo 3). Considere o código a seguir:

```
var book = {
    title: "Princípios de orientação a objetos em JavaScript"
};

// é o mesmo que:

var book = Object.create(Object.prototype, {
              title: {
                configurable: true,
                enumerable: true,
                value: "Princípios de orientação a objetos em JavaScript",
                writable: true
                }
            });
```

As duas declarações nesse código são exatamente iguais. A primeira declaração usa um objeto literal para definir um objeto com uma única propriedade chamada `title`. Esse objeto herda automaticamente de `Object.prototype`, e a propriedade é definida como configurável, enumerável e pode ser escrita por padrão. A segunda declaração executa os mesmos passos, mas faz isso explicitamente usando `Object.create()`. O objeto `book` resultante de cada declaração se comporta exatamente da mesma maneira. Mas é provável que você jamais vá escrever código que herde de `Object.prototype` diretamente, pois isso já é padrão. Herdar de outros objetos é muito mais interessante:

```
var person1 = {
```

```
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

var person2 = Object.create(person1, {
    name: {
        configurable: true,
        enumerable: true,
        value: "Greg",
        writable: true
    }
});

person1.sayName(); // exibe "Nicholas"
person2.sayName(); // exibe "Greg"

console.log(person1.hasOwnProperty("sayName")); // true
console.log(person1.isPrototypeOf(person2)); // true
console.log(person2.hasOwnProperty("sayName")); // false
```

Esse código cria um objeto `person1` com uma propriedade `name` e um método `sayName()`. O objeto `person2` herda de `person1`, portanto ele herda tanto `name` quanto `sayName()`. Entretanto `person2` é definido por meio de `Object.create()`, que também define uma propriedade própria chamada `name` para `person2`. Essa propriedade própria encobre a propriedade de mesmo nome do protótipo e é usada em seu lugar. Então `person1.sayName()` exibe `"Nicholas"`, enquanto `person2.sayName()` exibe `"Greg"`. Tenha em mente que `sayName()` existe somente em `person1` e está sendo herdado por `person2`.

A cadeia de herança nesse exemplo é maior para `person2` do que para `person1`. O objeto `person2` herda do objeto `person1`, e o objeto `person1` herda de `Object.prototype`. Veja a figura 5.1.

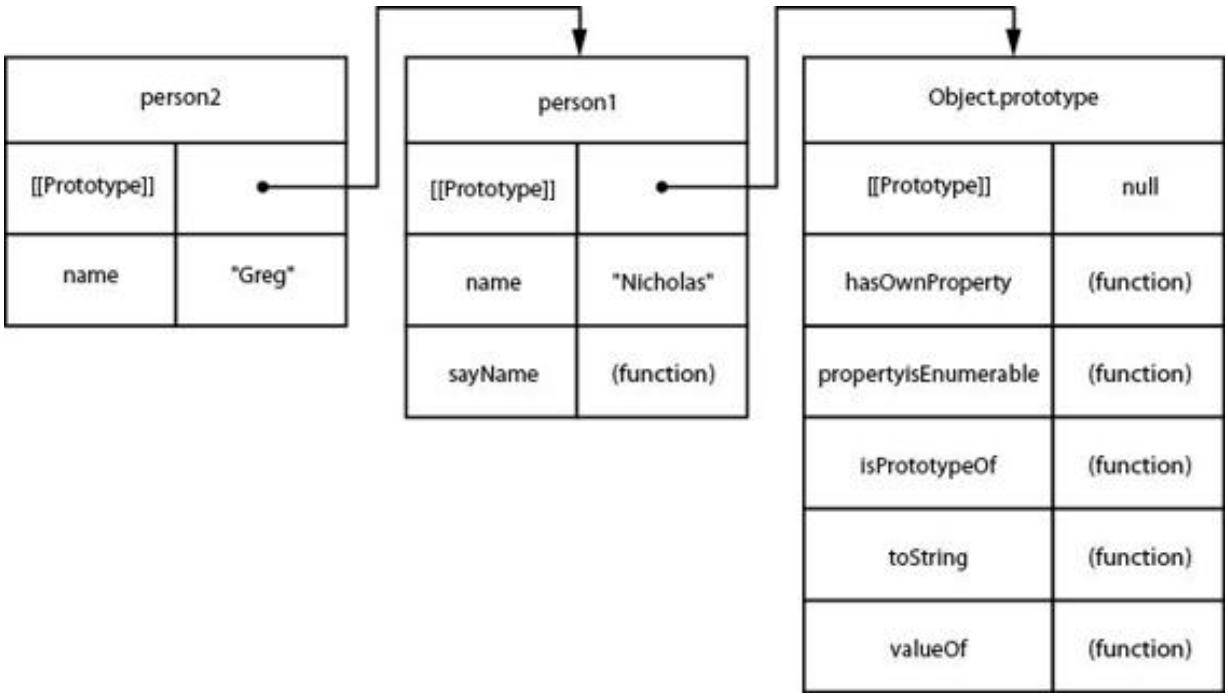


Figura 5.1 – A cadeia de protótipos de person2 inclui person1 e Object.prototype.

Quando uma propriedade é acessada em um objeto, a engine do JavaScript executa um processo de pesquisa. Se a propriedade for encontrada na instância (ou seja, se for uma propriedade própria), o valor dessa propriedade será usado. Se a propriedade não for encontrada na instância, a procura continuará no `[[Prototype]]` desse objeto, e assim por diante até o fim da cadeia ser alcançado. Essa cadeia normalmente termina com `Object.prototype`, cujo `[[Prototype]]` é definido com `null`.

Você também pode criar objetos com um `[[Prototype]]` igual a `null` por meio de `Object.create()`, como mostrado a seguir:

```
var nakedObject = Object.create(null);

console.log("toString" in nakedObject); // false
console.log("valueOf" in nakedObject); // false
```

O `nakedObject` nesse exemplo é um objeto sem cadeia de protótipos. Isso significa que métodos prontos como `toString()` e `valueOf()` não estão presentes no objeto. De fato, esse objeto é um quadro totalmente em branco, sem propriedades predefinidas, o que o torna perfeito para a criação de uma tabela de pesquisa hash sem que você tenha de se preocupar com colisões de nomes com as propriedades herdadas.

Não há muitos usos diferentes para um objeto como esse, e você não pode usá-lo como se ele herdasse de `Object.prototype`. Por exemplo, sempre que um operador for usado em `nakedObject`, um erro do tipo "Cannot convert object to primitive value" (Não é possível converter o objeto em um valor primitivo) será gerado. Apesar disso, a possibilidade de criar objetos sem protótipos é um aspecto interessante da linguagem JavaScript.

Herança de construtores

A herança de objetos em JavaScript também é a base da herança de construtores. Lembre-se do capítulo 4, em que vimos que quase toda função tem uma propriedade `prototype` que pode ser modificada ou substituída. Essa propriedade `prototype` é automaticamente definida para conter um novo objeto genérico que herda de `Object.prototype` e que tem uma única propriedade própria chamada `constructor`. Na verdade, a engine do JavaScript faz o seguinte:

```
// Você escreve isto:
function YourConstrutor() {
    // inicialização
}

// A engine do JavaScript faz isto internamente
YourConstrutor.prototype = Object.create(Object.prototype, {
                            constructor: {
                                configurable: true,
                                enumerable: true,
                                value: YourConstrutor,
                                writable: true
                            }
                        });
```

Sem fazer nada adicional, esse código define a propriedade `prototype` do construtor com um objeto que herda de `Object.prototype`, o que significa que qualquer instância de `YourConstrutor` também herdará de `Object.prototype`. `YourConstrutor` é um *subtipo* de `Object`, e `Object` é um *supertipo* de `YourConstrutor`.

Como a propriedade `prototype` pode ser atualizada, a cadeia de protótipos

pode ser alterada ao ser sobrescrita. Considere o exemplo a seguir:

```
❶ function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}
Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

Rectangle.prototype.toString = function() {
    return "[Rectangle " + this.length + "x" + this.width + "]";
};

// herda de Rectangle
❷ function Square(size) {
    this.length = length;
    this.width = size;
}
Square.prototype = new Rectangle();
Square.prototype.constructor = Square;

Square.prototype.toString = function() {
    return "[Square " + this.length + "x" + this.width + "]";
};
var rect = new Rectangle(5, 10);
var square = new Square(6);

console.log(rect.getArea()); // 50
console.log(square.getArea()); // 36

console.log(rect.toString()); // "[Rectangle 5x10]"
console.log(square.toString()); // "[Square 6x6]"

console.log(rect instanceof Rectangle); // true
console.log(rect instanceof Object); // true

console.log(square instanceof Square); // true
console.log(square instanceof Rectangle); // true
console.log(square instanceof Object); // true
```

Nesse código, há dois construtores: `Rectangle` ❶ e `Square` ❷. O construtor `Square` tem sua propriedade `prototype` sobrescrita com uma instância de `Rectangle`. Nenhum argumento é passado para `Rectangle` nesse ponto porque eles não precisam ser usados, e se fossem, todas as instâncias de `Square` compartilhariam as mesmas dimensões. Para mudar a cadeia de

protótipos dessa maneira, você deve sempre garantir que o construtor não irá lançar um erro se os argumentos não forem fornecidos (muitos construtores contêm uma lógica de inicialização que pode exigir os argumentos) e que o construtor não alterará nenhum tipo de estado global, por exemplo, controlando quantas instâncias já foram criadas. A propriedade `constructor` é restaurada em `Square.prototype` após o valor original ter sido sobrescrito.

Depois disso, `rect` é criado como uma instância de `Rectangle`, e `square` é criado como uma instância de `Square`. Ambos os objetos têm o método `getArea()` porque ele é herdado de `Rectangle.prototype`. A variável `square` é considerada uma instância de `Square` bem como de `Rectangle` e de `Object` porque `instanceof` usa a cadeia de protótipos para determinar o tipo do objeto. Observe a figura 5.2:

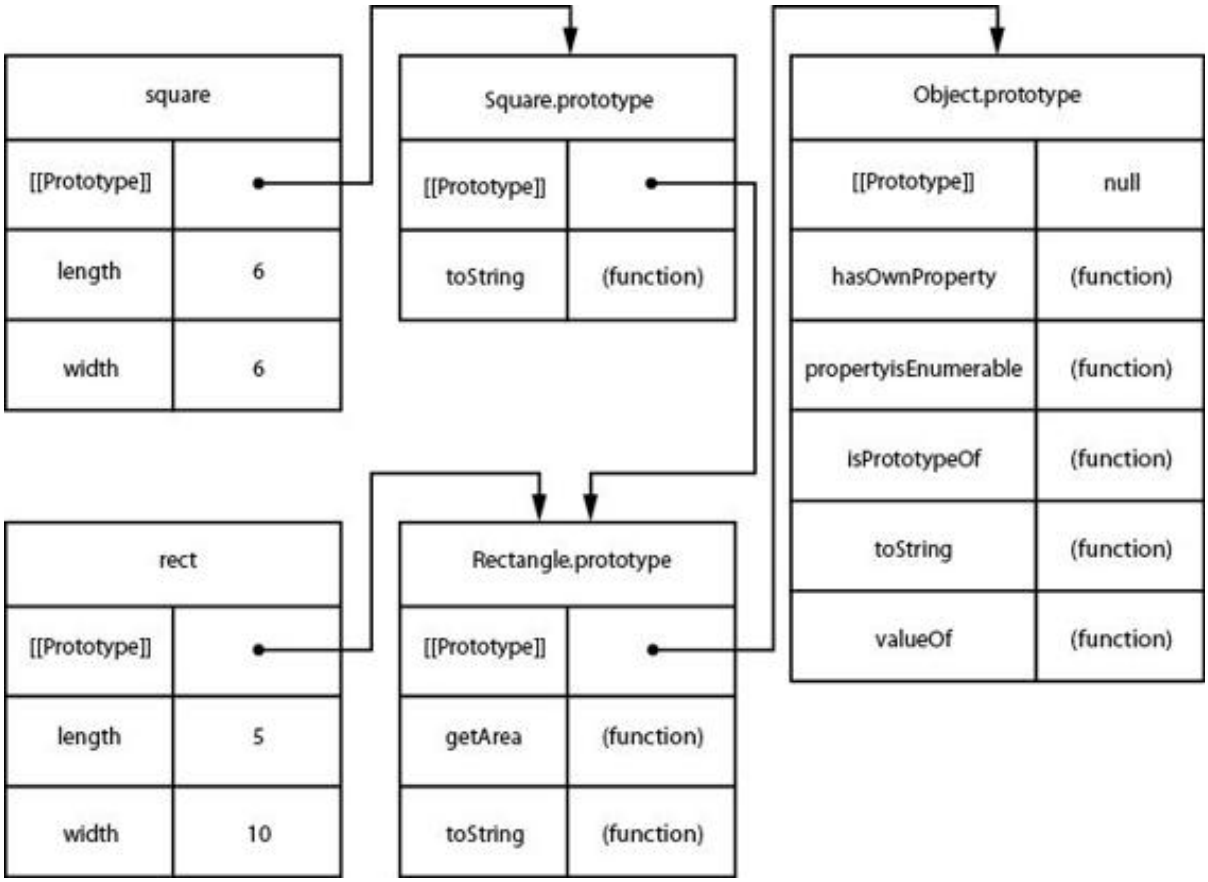


Figura 5.2 – A cadeia de protótipos para quadrado e rect mostra que ambos herdam de Rectangle.prototype e de Object.prototype, mas somente quadrado herda de Square.prototype.

`Square.prototype`, porém, não precisa ser sobrescrito por um objeto `Rectangle`;

o construtor `Rectangle` não está fazendo nada que seja necessário a `Square`. De fato, a única parte relevante é que `Square.prototype`, de alguma maneira, deve se ligar a `Rectangle.prototype` para que a herança aconteça. Isso significa que você pode simplificar esse exemplo usando `Object.create()` novamente:

```
// herda de Rectangle
function Square(size) {
    this.length = size;
    this.width = size;
}

Square.prototype = Object.create(Rectangle.prototype, {
                        constructor: {
                            configurable: true,
                            enumerable: true,
                            value: Square,
                            writable: true
                        }
                    });

Square.prototype.toString = function() {
    return "[Square " + this.length + "x" + this.width + "]";
};
```

Nessa versão do código, `Square.prototype` é sobrescrito por um novo objeto que herda de `Rectangle.prototype`, e o construtor `Rectangle` jamais é chamado. Isso significa que você não precisa mais se preocupar em causar um erro ao chamar o construtor sem argumentos. Esse código se comporta exatamente da mesma maneira que o código anterior. A cadeia de protótipos continua intacta, de modo que todas as instâncias de `Square` herdam de `Rectangle.prototype` e a propriedade `constructor` é restaurada no mesmo passo.

NOTA Sempre garanta que você irá sobrescrever `prototype` *antes* de adicionar propriedades a ele; do contrário, você perderá todos os métodos adicionados quando a sobrescrita ocorrer.

Furto de construtor

Como a herança é implementada por meio de cadeias de protótipos em

JavaScript, não é preciso chamar o construtor do supertipo de um objeto. Se quiser chamar o construtor do supertipo a partir do construtor do subtipo, você deverá tirar vantagem do modo como as funções em JavaScript funcionam.

No capítulo 2, você conheceu os métodos `call()` e `apply()`, que permitem que as funções sejam chamadas com um valor `this` diferente. É exatamente dessa maneira que o *furto de construtor* (constructor stealing) funciona. Basta chamar o construtor do supertipo a partir do construtor do subtipo usando `call()` ou `apply()` para passar o objeto recém-criado. De fato, você estará roubando o construtor do supertipo para o seu próprio objeto, como neste exemplo:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

Rectangle.prototype.toString = function() {
    return "[Rectangle " + this.length + "x" + this.width + "]";
};

// herda de Rectangle
❶ function Square(size) {
    Rectangle.call(this, size, size);
    // opcional: adiciona novas propriedades ou sobrescreve as
    // propriedades existentes aqui
}

Square.prototype = Object.create(Rectangle.prototype, {
                        constructor: {
                            configurable: true,
                            enumerable: true,
                            value: Square,
                            writable: true
                        }
                    });

Square.prototype.toString = function() {
    return "[Square " + this.length + "x" + this.width + "]";
```

```
};

var square = new Square(6);

console.log(square.length); // 6
console.log(square.width); // 6
console.log(square.getArea()); // 36
```

O construtor `Square` ❶ chama o construtor `Rectangle`, passa `this` e também `size` duas vezes (uma para `length`, outra para `width`). Fazer isso cria as propriedades `length` e `width` no novo objeto e faz cada uma delas ser igual a `size`. Essa é a maneira de evitar a redefinição de propriedades de um construtor do qual queremos herdar. Você pode adicionar novas propriedades ou sobrescrever as propriedades existentes após aplicar o construtor do supertipo.

Esse processo de duas etapas é útil quando queremos implementar a herança entre tipos personalizados. Você sempre deverá modificar o protótipo de um construtor, e poderá precisar chamar o construtor do supertipo a partir do construtor do subtipo. Geralmente, você irá modificar o protótipo para a herança de métodos e usará o furto de construtor para as propriedades. Essa abordagem normalmente é conhecida como *herança pseudoclássica* porque ela imita a herança clássica das linguagens baseadas em classe.

Acessando os métodos do supertipo

No exemplo anterior, o tipo `Square` tem seu próprio método `toString()` que encobre o método `toString()` do protótipo. É bastante comum sobrescrever os métodos do supertipo com novas funcionalidades no subtipo; mas e se você quiser continuar acessando o método do supertipo? Em outras linguagens, provavelmente você poderá usar `super.toString()`, porém o JavaScript não tem nada semelhante. Em vez disso, é possível acessar diretamente o método do protótipo referente ao supertipo e usar tanto `call()` quanto `apply()` para executar o método no objeto referente ao subtipo. Por exemplo:

```
function Rectangle(length, width) {
    this.length = length;
```

```
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

Rectangle.prototype.toString = function() {
    return "[Rectangle " + this.length + "x" + this.width + "]";
};

// herda de Rectangle
function Square(size) {
    Rectangle.call(this, size, size);
}

Square.prototype = Object.create(Rectangle.prototype, {
                        constructor: {
                            configurable: true,
                            enumerable: true,
                            value: Square,
                            writable: true
                        }
                    });

// chama o método do supertipo
❶ Square.prototype.toString = function() {
    var text = Rectangle.prototype.toString.call(this);
    return text.replace("Rectangle", "Square");
};
```

Nessa versão, `Square.prototype.toString()` ❶ chama `Rectangle.prototype.toString()` usando `call()`. O método só precisa substituir `"Rectangle"` por `"Square"` antes de retornar o texto resultante. Essa abordagem pode parecer um pouco extensa para uma operação simples como essa, mas é a única maneira de acessar um método do supertipo.

Sumário

O JavaScript suporta a herança por meio de cadeia de protótipos. Uma cadeia de protótipos é criada entre os objetos quando o `[[Prototype]]` de um objeto é definido como o outro objeto. Todos os objetos genéricos herdam automaticamente de `Object.prototype`. Se você deseja criar um

objeto que herde de outro objeto, `Object.create()` poderá ser usado para especificar o valor de `[[Prototype]]` do novo objeto.

A herança é implementada entre tipos personalizados por meio da criação de uma cadeia de protótipos no construtor. Ao definir a propriedade `prototype` do construtor com outro valor, a herança é implementada entre as instâncias do tipo personalizado e o protótipo desse outro valor. Todas as instâncias desse construtor compartilham o mesmo protótipo, portanto todas herdam do mesmo objeto. Essa técnica funciona bem para a herança de métodos de outros objetos, mas não se podem herdar propriedades próprias usando protótipos.

Para herdar propriedades próprias corretamente, o furto de construtor (constructor stealing) pode ser usado, o que consiste simplesmente em chamar uma função construtora com `call()` ou `apply()` para que qualquer inicialização seja feita no objeto referente ao subtipo. Combinar furto de construtor com cadeia de protótipos é a maneira mais comum de implementar a herança entre tipos personalizados em JavaScript. Essa combinação é frequentemente chamada de herança pseudoclássica devido a sua semelhança com a herança em linguagens baseadas em classes.

Você pode acessar os métodos de um supertipo ao acessar diretamente o seu protótipo. Para isso, use `call()` ou `apply()` para executar o método do supertipo no objeto referente ao subtipo.

---

1 Consulte "Code Conventions for the JavaScript Programming Language" (Convenções de código para a linguagem de programação JavaScript) de Douglas Crockford (*http://JavaScript.crockford.com/code.html*).

CAPÍTULO 6

Padrões de objeto

O JavaScript tem muitos padrões para criar objetos e, normalmente, há mais de uma maneira de fazer o mesmo. Você pode definir seus próprios tipos personalizados ou seus próprios objetos genéricos sempre que quiser. A herança pode ser usada para compartilhar comportamentos entre objetos, ou outras técnicas podem ser empregadas, por exemplo, o uso de mixins. Você também pode tirar proveito dos recursos avançados do JavaScript para evitar que a estrutura de um objeto seja modificada. Os padrões discutidos neste capítulo representam maneiras eficientes de criar e de manipular objetos, todos baseados em casos de uso.

Membros privados e privilegiados

Todas as propriedades de objetos em JavaScript são públicas, e não há nenhuma maneira explícita de indicar que uma propriedade não deva ser acessível de fora de um objeto em particular. Em algum momento, porém, você pode querer que os dados não sejam públicos. Por exemplo, quando um objeto utiliza um valor para determinar algum tipo de estado, modificar esse valor sem que o objeto tenha conhecimento transformaria o processo de gerenciamento desse estado em um caos. Uma maneira de evitar isso é usar convenções de nomenclatura. Por exemplo, é muito comum prefixar as propriedades com um underline (como em `this._name`) quando não queremos que elas sejam públicas. Entretanto há maneiras de ocultar dados que não dependem de convenções e, desse modo, são mais "à prova de balas" para evitar a modificação de informações privadas.

O padrão de módulo

O *padrão de módulo* é um padrão de criação de objetos concebido para criar objetos únicos (singleton) com dados privados. A abordagem básica consiste em usar uma IIFE (Immediately Invoked Function Expression, ou expressão de função imediatamente invocada) que retorna um objeto. Uma IIFE é uma expressão de função definida e chamada imediatamente para gerar um resultado. Essa expressão de função pode conter qualquer número de variáveis locais que não sejam acessíveis de fora dessa função. Como o objeto retornado é definido dentro dessa função, os métodos do objeto têm acesso aos dados. (Todos os objetos definidos em uma IIFE têm acesso às mesmas variáveis locais.) Os métodos que acessam dados privados dessa maneira são chamados de métodos *privilegiados*. Aqui está o formato básico do padrão de módulo:

```
var yourObject = (function() {

    // variáveis referentes a dados privados

    return {
    // métodos e propriedades públicas
        };
❶}());
```

Nesse padrão, uma função anônima é criada e executada imediatamente. (Note os parênteses extras no final da função ❶. Você pode executar funções anônimas imediatamente usando essa sintaxe.) Isso significa que a função só existe por um momento, é executada e, em seguida, é destruída. A IIFE é um padrão bem popular em JavaScript, em parte por causa de seu uso no padrão de módulo.

O padrão de módulo permite usar variáveis normais como propriedades de objetos que não são expostas publicamente. Isso é feito por meio da criação de funções de *closure* como métodos do objeto. As closures são simplesmente funções que têm acesso a dados fora de seu escopo. Por exemplo, sempre que um objeto global é acessado em uma função, por exemplo, `window` em um web browser, essa função está acessando uma variável fora de seu próprio escopo. A diferença em relação às funções do padrão de módulo está no fato de que as variáveis são declaradas na IIFE,

e uma função que também é declarada na IIFE acessa essas variáveis. Por exemplo:

```
var person = (function() {
❶    var age = 25;
     return {
         name: "Nicholas",
❷        getAge: function() {
             return age;
         },
❸       growOlder: function() {
            age++;
        }
    };
})();
console.log(person.name); // "Nicholas"
console.log(person.GetAge()); // 25
person.age = 100;
console.log(person.getAge()); // 25
person.growOlder();
console.log(person.getAge()); // 26
```

Esse código cria o objeto `person` usando o padrão de módulo. A variável `age` ❶ atua como uma propriedade privada do objeto. Ela não pode ser acessada diretamente de fora do objeto, mas pode ser usada pelos métodos do objeto. Há dois métodos privilegiados no objeto: `getAge()` ❷, que lê o valor da variável `age`, e `growOlder()` ❸, que incrementa `age`. Ambos os métodos podem acessar a variável `age` diretamente porque ela está definida na função mais externa em relação ao local em que os métodos estão definidos.

Há uma variação do padrão de módulo chamada *revealing module pattern* (padrão de módulo revelador), que organiza todas as variáveis e os métodos no início da IIFE e simplesmente os atribui ao objeto retornado. O exemplo anterior pode ser escrito usando esse padrão desta maneira:

```
var person = (function() {
    var age = 25;
```

```
    function getAge() {
        return age;
    }

    function growOlder() {
        age++;
    }

    return {
        name: "Nicholas",
❶       getAge: getAge,
        growOlder: growOlder
    }

}());
```

No *revealing module pattern*, `age`, `getAge()` e `growOlder()` são todos definidos como locais na IIFE. As funções `getAge()` e `growOlder()` são então atribuídas ao objeto retornado ❶, efetivamente "revelando-as" para fora da IIFE. Esse código é essencialmente o mesmo do exemplo anterior que usa o padrão de módulo tradicional; porém algumas pessoas preferem esse padrão porque ele mantém todas as variáveis e as declarações de funções juntas.

Membros privados de construtores

O padrão de módulo é ótimo para definir objetos individuais que tenham propriedades privadas, mas e os tipos personalizados que também exigem propriedades privadas próprias? Você pode usar um padrão semelhante ao padrão de módulo no construtor para criar dados privados específicos das instâncias. Por exemplo:

```
function Person(name) {

    // define uma variável acessível somente no construtor Person
    var age = 25;
    this.name = name;

❶   this.getAge = function() {
        return age;
    };

❷   this.growOlder = function() {
        age++;
    };
```

```
}
var person = new Person("Nicholas");

console.log(person.name); // "Nicholas"
console.log(person.getAge()); // 25

person.age = 100;
console.log(person.getAge()); // 25

person.growOlder();
console.log(person.getAge()); // 26
```

Nesse código, o construtor `Person` tem uma variável local `age`. Essa variável é usada como parte dos métodos `getAge()` ❶ e `growOlder()` ❷. Quando uma instância de `Person` é criada, essa instância recebe sua própria variável `idade` e seus próprios métodos `getAge()` e `growOlder()`. Em vários aspectos, isso é semelhante ao padrão de módulo, em que o construtor cria um escopo local e retorna o objeto `this`. Como discutido no capítulo 4, colocar métodos na instância de um objeto é menos eficiente que fazer isso no protótipo, mas essa é a única abordagem possível quando você quer ter dados privados e específicos da instância.

Se quiser que dados privados sejam compartilhados entre todas as instâncias (como se eles estivessem no protótipo), uma abordagem híbrida que se parece com o padrão de módulo, embora use um construtor, poderá ser usada:

```
var Person = (function() {

    // todos compartilham a mesma idade
❶     var age = 25;

❷     function InnerPerson(name) {
        this.name = name;
    }

    InnerPerson.prototype.getAge = function() {
        return age;
    };

    InnerPerson.prototype.growOlder = function() {
        age++;
    };

    return InnerPerson;

}());
```

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name); // "Nicholas"
console.log(person1.getAge()); // 25

console.log(person2.name); // "Greg"
console.log(person2.getAge()); // 25

person1.growOlder();
console.log(person1.getAge()); // 26
console.log(person2.getAge()); // 26
```

Nesse código, o construtor `InnerPerson` ❷ é definido em uma IIFE. A variável `age` ❶ é definida fora do construtor, mas é usada em dois métodos do protótipo. O construtor `InnerPerson` então é retornado e se torna o construtor `Person` no escopo global. Todas as instâncias de `Person` compartilham a variável `age`, portanto mudar o seu valor em uma instância automaticamente afetará a outra instância.

Mixins

Embora a herança pseudoclássica e a herança por protótipos sejam usadas frequentemente em JavaScript, há também um tipo de pseudo-herança implementada por meio de mixins. Os *mixins* ocorrem quando um objeto adquire as propriedades de outro sem modificar a cadeia de protótipos. O primeiro objeto (o *receiver* [receptor]) recebe as propriedades do segundo objeto (o *supplier* [fornecedor]) ao copiar diretamente essas propriedades. Tradicionalmente, os mixins são criados por meio de funções como esta:

```
function mixin(receiver, supplier) {
    for (var property in supplier) {
        if (supplier.hasOwnProperty(property)) {
            receiver[property] = supplier[property]
        }
    }

    return receiver;
}
```

A função `mixin()` aceita dois argumentos: o `receiver` (receptor) e o `supplier`

(fornecedor). O objetivo da função é copiar todas as propriedades enumeráveis do fornecedor para o receptor. Isso é feito por meio de um loop `for-in` que efetua a iteração pelas propriedades de `supplier` e atribui o valor de cada propriedade a uma propriedade de mesmo nome no `receiver`. Tenha em mente que essa é uma cópia rasa (shallow copy): se uma propriedade contiver um objeto, então tanto o receptor quanto o fornecedor apontarão para o mesmo objeto. Esse padrão é usado frequentemente em JavaScript para adicionar novos comportamentos já existentes em um objeto a outro.

Por exemplo, um suporte a eventos pode ser adicionado a um objeto por meio de um mixin em vez de usar a herança. Inicialmente, suponha que você já tenha definido um tipo personalizado para usar eventos:

```
function EventTarget() {

}

EventTarget.prototype = {

    constructor: EventTarget,

❶   addListener: function(type, listener) {

        // cria um array se ele não existir
        if (!this.hasOwnProperty("_listeners")) {
            this._listeners = [];
        }

        if (typeof this._listeners[type] == "undefined") {
            this._listeners[type] = [];
        }

        this._listeners[type].push(listener);
    },

❷   fire: function(event) {

        if (!event.target) {
            event.target = this;
        }

        if (!event.type) { // falsy
            throw new Error("Event object missing 'type' property");
        }

        if (this._listeners && this._listeners[event.type] instanceof Array) {
            var listeners = this._listeners[event.type];
```

```
            for (var i=0, len=listeners.length; i < len; i++) {
                listeners[i].call(this, event);
            }
        }
    },

❸   removeListener: function(type, listener) {
        if (this._listeners && this._listeners[type] instanceof Array) {
            var listeners = this._listeners[type];
            for (var i=0, len=listeners.length; i < len; i++) {
                if (listeners[i] === listener) {
                    listeners.splice(i, 1);
                    break;
                }
            }
        }
    }
};
```

O tipo `EventTarget` oferece uma manipulação básica de eventos para qualquer objeto. Você pode adicionar ❶ ou remover ❸ listeners assim como disparar eventos ❷ diretamente no objeto. Os listeners de evento são armazenados na propriedade `_listeners`, criada somente quando `addListener()` é chamada pela primeira vez (isso facilita efetuar a combinação do mixin). Instâncias de `EventTarget` podem ser usadas desta maneira:

```
var target = new EventTarget();
target.addListener("message", function(event) {
    console.log("Message is " + event.data);
})

target.fire({
    type: "message",
    data: "Hello world!"
});
```

O suporte a eventos é útil para os objetos em JavaScript. Se você quiser ter um tipo diferente de objeto que também suporte eventos, há algumas opções. Em primeiro lugar, podemos criar uma nova instância de `EventTarget` e adicionar as propriedades desejadas:

```
var person = new EventTarget();
```

```
person.name = "Nicholas";
person.sayName = function() {
    console.log(this.name);
    this.fire({ type: "namesaid", name: this.name});
};
```

Nesse código, uma nova variável chamada `person` é criada como uma instância de `EventTarget` e, em seguida, as propriedades relacionadas à `person` são adicionadas. Infelizmente, isso significa que `person` é uma instância de `EventTarget` e não de `Object` ou de um tipo personalizado. Há também o overhead de ter de adicionar várias propriedades novas manualmente. Seria melhor se houvesse uma maneira mais organizada de fazer isso.

A segunda maneira de resolver esse problema é usar a herança pseudoclássica:

```
function Person(name) {
    this.name = name;
}
❶ Person.prototype = Object.create(EventTarget.prototype);
Person.prototype.constructor = Person;

Person.prototype.sayName = function() {
    console.log(this.name);
    this.fire({type: "namesaid", name: this.name});
};

var person = new Person("Nicholas");

console.log(person instanceof Person); // true
console.log(person instanceof EventTarget); // true
```

Nesse caso, há um novo tipo `Person` que herda de `EventTarget` ❶. Você pode adicionar quaisquer outros métodos necessários ao protótipo de `Person` depois disso. Entretanto isso não é tão sucinto quanto poderia ser, e você poderia argumentar que o relacionamento não faz sentido: uma pessoa é um tipo de event target? Ao usar um mixin no lugar dessa solução, é possível reduzir a quantidade de código necessária para atribuir essas novas propriedades ao protótipo:

```
function Person(name) {
    this.name = name;
```

```
}
❶ mixin(Person.prototype, EventTarget.prototype);
mixin(Person.prototype, {
    constructor: Person,
   sayName: function() {
        console.log(this.name);
        this.fire({ type: "namesaid", name: this.name});
    }
});
var person = new Person("Nicholas");
console.log(person instanceof Person); // true
console.log(person instanceof EventTarget); // false
```

Nesse caso, `Person.prototype` é combinado com `EventTarget.prototype` ❶ para que tenha o comportamento do evento. Em seguida, `Person.prototype` é combinada com `constructor` e `sayName()` para completar a composição do protótipo. Instâncias de `Person` não são instâncias de `EventTarget` nesse exemplo porque não há herança.

É claro que você pode decidir que, ao mesmo tempo que você quer usar as propriedades de um objeto, você não quer ter um construtor com herança pseudoclássica. Nesse caso, um mixin pode ser usado diretamente ao criar o seu novo objeto:

```
var person = mixin(new EventTarget(), {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
        this.fire({ type: "namesaid", name: this.name});
    }
});
```

Nesse exemplo, uma nova instância de `EventTarget` é combinada com algumas propriedades novas para criar o objeto `person` sem afetar a cadeia de protótipos de `person`.

Um aspecto a se ter em mente sobre o uso de mixins dessa maneira é que as propriedades de acesso do fornecedor se tornam propriedades de dados no receptor, o que significa que se você não for cuidadoso, poderá

sobrescrevê-las. Isso acontece porque as propriedades do receptor estão sendo criadas por atribuição e não pelo método `Object.defineProperty()`, o que significa que o valor da propriedade atual no fornecedor é lido e, em seguida, atribuído a uma propriedade de mesmo nome no receptor. Por exemplo:

```
var person = mixin(new EventTarget(), {
❶    get name() {
       return "Nicholas"
    },
    sayName: function() {
        console.log(this.name);
        this.fire({ type: "namesaid", name: name});
    }
});
console.log(person.name); // "Nicholas"
❷ person.name = "Greg";
console.log(person.name); // "Greg"
```

Nesse código, `name` é definido como uma propriedade de acesso somente com um getter ❶. Isso significa que atribuir um valor à propriedade não deve ter nenhum efeito. Entretanto, como a propriedade de acesso se torna uma propriedade de dados no objeto `person`, é possível sobrescrever `name` com um novo valor. ❷ Durante a chamada a `mixin()`, o valor de `name` é lido do fornecedor e atribuído à propriedade chamada `name` no receptor. Em nenhum momento, durante esse processo, uma propriedade nova de acesso foi definida, fazendo com que a propriedade `name` no receptor seja uma propriedade de dado.

Se você quiser que propriedades de acesso sejam copiadas como propriedades de acesso, será necessário ter uma função `mixin()` diferente, como esta:

```
function mixin(receiver, supplier) {
❶    Object.keys(supplier).forEach(function(property) {
         var descriptor = Object.getOwnPropertyDescriptor(supplier,
             property);
❷        Object.defineProperty(receiver, property, descriptor);
     });
```

```
        return receiver;
    }

    var person = mixin(new EventTarget(), {

        get name() {
            return "Nicholas"
        },

        sayName: function() {
            console.log(this.name);
            this.fire({ type: "namesaid", name: name});
        }
    });

    console.log(person.name); // "Nicholas"

    person.name = "Greg";
    console.log(person.name); // "Nicholas"
```

Essa versão de `mixin()` usa `Object.keys()` ❶ para obter um array com todas as propriedades próprias enumeráveis de `supplier`. O método `forEach()` é usado para efetuar uma iteração por essas propriedades. O descritor de cada propriedade em `supplier` é obtido e, em seguida, é adicionado ao `receiver` por meio de `Object.defineProperty()` ❷. Essa operação garante que todas as informações relevantes da propriedade sejam transferidas para o `receiver`, e não apenas o valor. Isso quer dizer que o objeto `person` tem uma propriedade de acesso chamada `name`, portanto ela não poderá ser sobrescrita.

É claro que essa versão de `mixin()` só funciona com as engines de JavaScript compatíveis com o ECMAScript 5. Se for preciso que seu código funcione com engines mais antigas, você deverá combinar as duas abordagens de `mixin()` em uma única função:

```
function mixin(receiver, supplier) {
❶    if (Object.getOwnPropertyDescriptor) {

        Object.keys(supplier).forEach(function(property) {
            var descriptor = Object.getOwnPropertyDescriptor(supplier,
                property);
            Object.defineProperty(receiver, property, descriptor);
        });
    } else {
```

```
❷          for (var property in supplier) {
               if (supplier.hasOwnProperty(property)) {
                   receiver[property] = supplier[property]
               }
           }
       }
    return receiver;
}
```

Nesse caso, `mixin()` verifica se `Object.getOwnPropertyDescriptor()` ❶ existe para determinar se a engine do JavaScript suporta ECMAScript 5. Em caso afirmativo, a versão para ECMAScript 5 é utilizada. Do contrário, a versão para ECMAScript 3 é usada ❷. Essa função é segura tanto para as engines modernas quanto para as mais antigas, pois a estratégia de mixin mais apropriada será aplicada.

NOTA Tenha em mente que `Object.keys()` retorna somente as propriedades enumeráveis. Se você também quiser copiar as propriedades não enumeráveis, use `Object.getOwnPropertyNames()`.

Construtores com escopo seguro

Como todos os construtores são apenas funções, eles podem ser chamados sem o uso do operador `new` e, sendo assim, podem afetar o valor de `this`. Fazer isso pode levar a resultados inesperados, pois `this` acaba referenciando o objeto global em modo não restrito ou o construtor lançará um erro em modo restrito. No capítulo 4, vimos este exemplo:

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayName = function() {
    console.log(this.name);
};

❶ var person1 = Person("Nicholas"); // nota: falta "new"

console.log(person1 instanceof Person); // false
console.log(typeof person1); // "undefined"
console.log(name); // "Nicholas"
```

Nesse caso, `name` é criado como uma variável global porque o construtor

`Person` é chamado sem o operador `new` ❶. Tenha em mente que esse código está sendo executado em modo não restrito, pois deixar de usar `new` fará um erro ser lançado em modo restrito. O fato de o nome do construtor começar com uma letra maiúscula normalmente indica que ele deve ser precedido por um `new`, mas e se você quiser permitir esse tipo de uso e fazer a função operar sem `new`? Muitos construtores prontos, como `Array` e `RegExp`, também funcionam sem `new` porque eles foram escritos para terem *escopo seguro*. Um construtor de escopo seguro pode ser chamado com ou sem o operador `new` e retorna o mesmo tipo de objeto em qualquer caso.

Quando `new` é chamado com uma função, o objeto recém-criado representado por `this` já é uma instância do tipo personalizado representado pelo construtor. Portanto `instanceof` pode ser utilizado para determinar se `new` foi usado na chamada da função:

```
function Person(name) {
    if (this instanceof Person) {
        // chamado com "new"
    } else {
        // chamado sem "new"
    }
}
```

Usar um padrão como esse permite controlar o que uma função faz de acordo com o fato de ela ter sido chamada ou não com `new`. Pode ser que você queira tratar cada circunstância de modo diferente, mas, normalmente, irá querer que a função se comporte da mesma maneira (frequentemente, para se proteger contra omissões acidentais de `new`). Uma versão de escopo seguro de `Person` tem o seguinte aspecto:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;
    } else {
        return new Person(name);
    }
}
```

Para esse construtor, a propriedade `name` é atribuída como sempre, quando `new` é usado. Se `new` não for usado, o construtor será chamado

recursivamente por meio de `new` para criar uma instância apropriada do objeto. Dessa maneira, ambas as opções a seguir são equivalentes:

```
var person1 = new Person("Nicholas");
var person2 = Person("Nicholas");

console.log(person1 instanceof Person); // true
console.log(person2 instanceof Person); // true
```

Criar novos objetos sem usar o operador `new` está se tornando mais comum, como um esforço para coibir erros causados pela omissão de `new`. O próprio JavaScript tem diversos tipos de referência com construtores de escopo seguro, como `Object`, `Array`, `RegExp` e `Error`.

Sumário

Há diversas maneiras de criar e compor objetos em JavaScript. Embora o JavaScript não tenha um conceito formal de propriedades privadas, você pode criar dados ou funções que sejam acessíveis somente de dentro de um objeto. Para objetos únicos (singleton), o padrão de módulo pode ser usado para ocultar os dados do mundo externo. Uma IIFE (função de expressão imediatamente invocada) pode ser usada para definir variáveis locais e funções que sejam acessíveis somente pelo objeto recém-criado. Métodos privilegiados são métodos de um objeto que têm acesso a dados privados. Você também pode criar construtores que tenham dados privados, seja definindo variáveis na função construtora, seja usando uma IIFE para criar dados privados que serão compartilhados por todas as instâncias.

Os mixins são uma maneira eficiente de adicionar funcionalidade a objetos ao mesmo tempo que se evita a herança. Um mixin copia propriedades de um objeto para outro, de modo que o objeto receptor obtenha as funcionalidades do fornecedor sem que haja herança. De modo diferente da herança, os mixins não permitem identificar a procedência das funcionalidades depois que o objeto for criado. Por essa razão, os mixins são mais bem empregados com propriedades de dados ou com pequenas funcionalidades. A herança continua sendo preferível quando queremos obter mais funcionalidades e conhecemos a sua

procedência.

Os construtores de escopo seguro são construtores que podem ser chamados com ou sem o operador `new` para criar uma nova instância de um objeto. Esse padrão tira vantagem do fato de que `this` é uma instância do tipo personalizado assim que o construtor começa a ser executado, o que permite alterar o comportamento do construtor de acordo com o fato de o operador `new` ter sido ou não utilizado.


UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES
Lógica de Programação e Algoritmos com JavaScript
novatec
Edécio Fernando Iepsen

Lógica de Programação e Algoritmos com JavaScript

Iepsen, Edécio Fernando

9788575226575

320 páginas

Compre agora e leia

Os conteúdos abordados em Lógica de Programação e Algoritmos são fundamentais a todos aqueles que desejam ingressar no universo da Programação de Computadores. Esses conteúdos, no geral, impõem algumas dificuldades aos iniciantes. Neste livro, o autor utiliza sua experiência de mais de 15 anos em lecionar a disciplina de Algoritmos em cursos de graduação, para trabalhar o assunto passo a passo. Cada capítulo foi cuidadosamente planejado a fim de evitar a sobrecarga de informações ao leitor, com exemplos e exercícios de fixação para cada assunto. Os exemplos e exercícios são desenvolvidos em JavaScript, linguagem amplamente utilizada no desenvolvimento de páginas para a internet. Rodar os programas JavaScript não exige nenhum software adicional; é preciso apenas abrir a página em seu navegador favorito. Como o aprendizado de Algoritmos ocorre a partir do estudo das técnicas de programação e da prática de exercícios, este livro pretende ser uma importante fonte de conhecimentos para você ingressar nessa fascinante área da programação de computadores. Assuntos abordados no livro: Fundamentos de Lógica de

Programação■Programas■de■entrada,■processamento■e■saída Integração■do■código■JavaScript■com■as■páginas■HTML■Estruturas condicionais■e■de■repetição■Depuração■de■Programas■JavaScript Manipulação■de■listas■de■dados■(vetores)■Operações■sobre■cadeias de■caracteres■(strings)■e■datas■Eventos■JavaScript■e■funções■com passagem■de■parâmetros■Persistência■dos■dados■de■um■programa com■localStorage■Inserção■de■elementos■HTML■via■JavaScript■com referência■ao■DOM■No■capítulo■final,■um■jogo■em■JavaScript■e■novos exemplos■que■exploram■os■recursos■discutidos■ao■longo■do■livro■são apresentados■com■comentários■e■dicas,■a■fim■de■incentivar■o■leitor■a prosseguir■nos■estudos■sobre■programação.

Compre■agora■e■leia

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

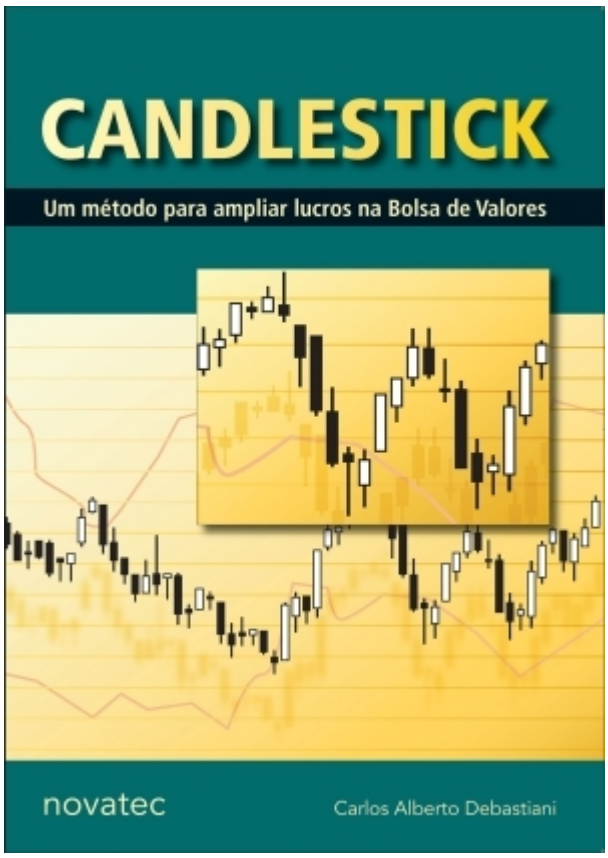
272 páginas

Compre agora e leia

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

Compre agora e leia


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick - Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Compre agora e leia

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

Compre agora e leia