```
função Ninja () {
      this.swung = verdade ;
 }

 const ninja1 = novo Ninja ();
```

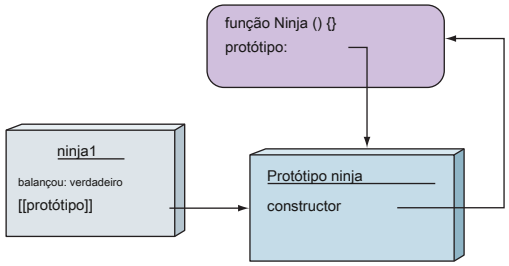


Figura 7.7 Após a construção, **ninja1** tem a propriedade **balançado,** e seu protótipo é o **Ninja** protótipo que tem apenas um **construtor** propriedade.

Aqui, novamente definimos um Ninja construtor e continue a usá-lo para criar uma instância de objeto. O estado da aplicação neste momento é mostrado na figura 7.7.

Depois de a instância foi criada, nós adicionamos um swingSword método para o protótipo. Em seguida, executamos um teste para mostrar que a alteração que fizemos no protótipo depois que o objeto foi construído entra em vigor. O estado atual do aplicativo é mostrado na figura 7.8.

```
function Ninja(){
    this.swung = true ;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = função () {
    Retorna this.swung;
};
```

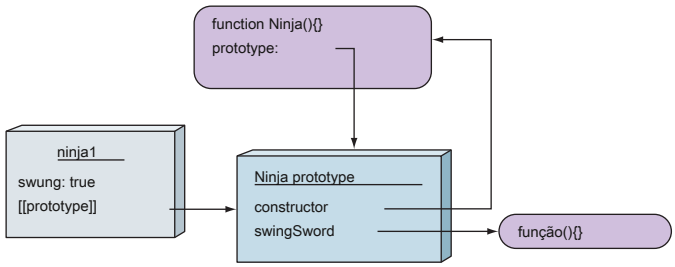


Figura 7.8 Porque o **ninja1** instância faz referência a **Ninja** protótipo, mesmo as alterações feitas depois que a instância foi construída são acessíveis.

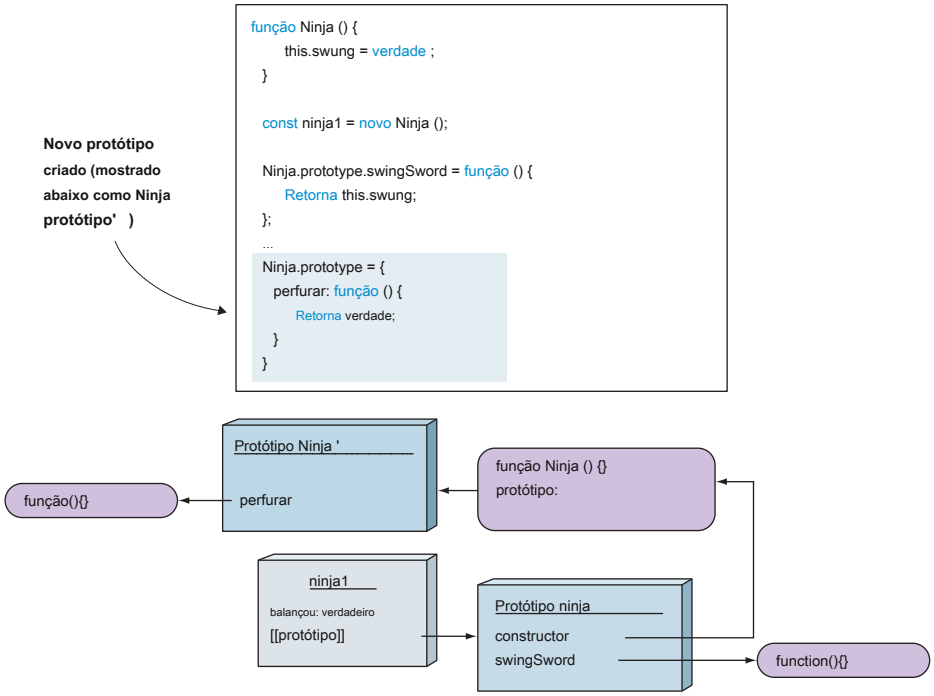


Figura 7.9 O protótipo da função pode ser substituído à vontade. As instâncias já construídas fazem referência ao antigo protótipo!

Mais tarde, substituímos o Ninja protótipo da função atribuindo-o a um objeto completamente novo que tem um pierce método. Isso resulta no estado do aplicativo mostrado na figura 7.9.

Como você pode ver, embora o Ninja função não faz referência ao antigo Ninja protótipo, o antigo protótipo ainda é mantido vivo pelo ninja1 instância, que ainda pode, através da cadeia de protótipos, acessar o swingSword método. Mas se criarmos novos objetos após essa troca de protótipo, o estado da aplicação será como mostrado na figura 7.10.

A referência entre um objeto e o protótipo da função é estabelecida no momento da instanciação do objeto. Objetos recém-criados terão uma referência ao novo protótipo e terão acesso ao perfurar , enquanto os objetos antigos pré-mudança de protótipo mantêm seu protótipo original, balançando suas espadas alegremente.

Exploramos como os protótipos funcionam e como estão relacionados à instanciação de objetos. Bem feito! Agora, respire rapidamente, para que possamos continuar aprendendo mais sobre a natureza desses objetos.

```
função Ninja () {
      this.swung = true;
 }

 const ninja1 = novo Ninja ();

 Ninja.prototype.swingSword = função () {
     Retorna this.swung;
 };
 ...
 Ninja.prototype = {
   perfurar: função () {
       Retorna verdade;
   }
 }
 ...
 const ninja2 = novo Ninja ();
```

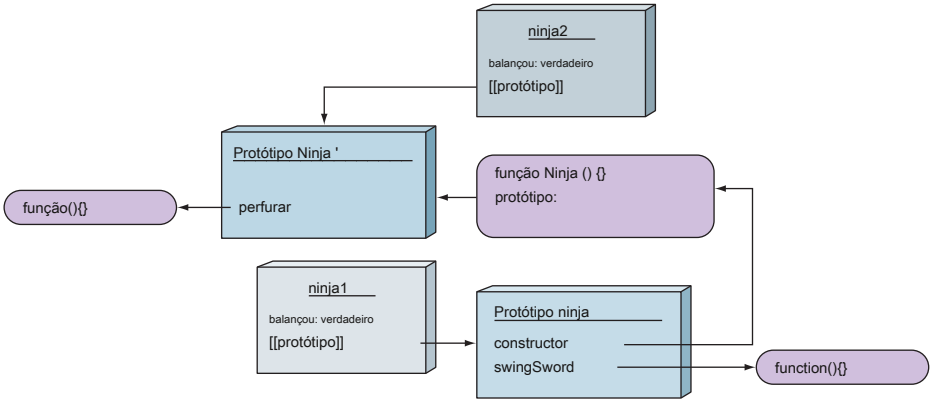


Figura 7.10 Todas as instâncias recém-criadas fazem referência ao novo protótipo.

7.2.3 *Digitação de objetos por meio de construtores*

Embora seja ótimo saber como JavaScript usa o protótipo para encontrar as referências de propriedade corretas, também é útil saber qual função construiu uma instância de objeto. Como você viu anteriormente, o construtor de um objeto está disponível por meio da propriedade constructor do protótipo da função do construtor. Por exemplo, figura

7.11 mostra o estado do aplicativo quando instanciamos um objeto com o Ninja construtor.

```
função Ninja () {}
const ninja1 = new Ninja();
```

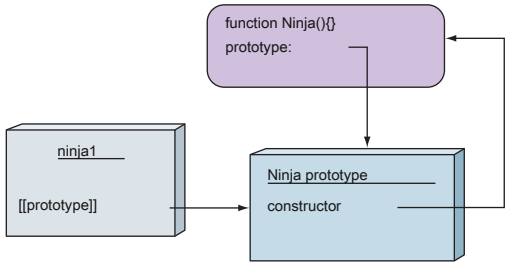


Figura 7.11 O objeto de protótipo de cada função tem uma propriedade de construtor que faz referência à função.

Usando o construtor propriedade, podemos acessar a função que foi usada para criar o objeto. Essas informações podem ser usadas como forma de verificação de tipo, conforme mostrado na listagem a seguir.

Listagem 7.5 Examinando o tipo de uma instância e seu construtor

```
função Ninja () {}
const ninja = novo Ninja ();

assert (typeof ninja === "objeto",
        "O tipo da instância é objeto."); assert (instância ninja de Ninja,
        "instanceof identifica o construtor." ); assert (ninja.constructor === Ninja,
        "O objeto ninja foi criado pela função Ninja.");
```

Testa o tipo de ninja por meio de typeof. Isso nos diz que é um objeto, mas não muito mais.

Testa o tipo de ninja por meio de instanceof. Isso fornece mais informações - que foi construído a partir de Ninja.

Testa o tipo de ninja por meio da referência do construtor. Isso fornece uma referência à função do construtor.

Definimos um construtor e criamos uma instância de objeto usando-o. Em seguida, examinamos o tipo da instância usando o tipo de operador. Isso não revela muito, pois todas as instâncias serão objetos, portanto, sempre retornando objeto como resultado. Muito mais interessante é o instancia de operador, que nos dá uma maneira de determinar se uma instância foi criada por um construtor de função particular. Você aprenderá mais sobre como o instancia de operador trabalhará posteriormente neste capítulo.

Além disso, podemos usar o construtor propriedade, que agora sabemos que é acessível a todas as instâncias, como uma referência à função original que a criou. Podemos usar isso para verificar a origem da instância (tanto quanto podemos com o instancia de operador).

Além disso, como esta é apenas uma referência ao construtor original, podemos instanciar um novo Ninja objeto usando-o, conforme mostrado na próxima lista.

Listagem 7.6 Instanciar um novo objeto usando uma referência a um construtor

```
função Ninja () {}

const ninja = novo Ninja ();
const ninja2 = novo ninja.constructor ();

assert (instância ninja2 de Ninja, "É um Ninja!"); afirmar (ninja! == ninja2, "Mas não o mesmo Ninja!");
```

Constrói um segundo Ninja desde o primeiro

Prova o novo Ninja-ness do objeto

Eles não são o mesmo objeto, mas duas instâncias distintas.

Aqui, definimos um construtor e criamos uma instância usando esse construtor. Então usamos o construtor propriedade da instância criada para construir uma segunda instância. O teste mostra que um segundo Ninja foi construído e que a variável não aponta meramente para a mesma instância.

O que é especialmente interessante é que podemos fazer isso sem nem mesmo ter acesso à função original; podemos usar a referência completamente nos bastidores, mesmo se o construtor original não estiver mais no escopo.

NOTA Apesar de construtor propriedade de um objeto pode ser alterada, fazer isso não tem nenhum propósito construtivo imediato ou óbvio (embora possamos pensar em alguns maliciosos). A razão de ser da propriedade é indicar de onde o objeto foi construído. Se o

construtor propriedade é substituída, o valor original é perdido.

Tudo isso é útil, mas acabamos de arranhar a superfície dos superpoderes que os protótipos nos conferem. Agora as coisas ficam interessantes.

7,3 Atingindo a herança

Herança é uma forma de reutilização na qual novos objetos têm acesso às propriedades dos objetos existentes. Isso nos ajuda a evitar a necessidade de repetir código e dados em nossa base de código. Em JavaScript, a herança funciona de maneira um pouco diferente do que em outras linguagens orientadas a objetos populares. Considere a seguinte lista, na qual tentamos obter herança.

Listagem 7.7 Tentando obter herança com protótipos

```
função Person () {}
Person.prototype.dance = function () {};

função Ninja () {}
Ninja.protótipo = {dança: Person.prototype.dance};

const ninja = novo Ninja (); assert (instância ninja de
Ninja,
        "ninja recebe funcionalidade do protótipo Ninja"); assert (instância ninja de Pessoa, "... e o protótipo da
Pessoa"); assert (ninja instanceof Object, "... and the Object prototype");
```

Define uma pessoa que dança por meio de um construtor e seu protótipo

Define um Ninja

Tentativas de fazer do Ninja uma pessoa dançarina copiando a dança método do Protótipo de pessoa

Como o protótipo de uma função é um objeto, existem várias maneiras de copiar a funcionalidade (como propriedades ou métodos) para efetuar a herança. Neste código, definimos um Pessoa e então Ninja. E porque uma Ninja é claramente uma pessoa, nós queremos Ninja para herdar os atributos de Pessoa. Tentamos fazer isso copiando o dança propriedade do Pessoa método do protótipo para uma propriedade com nome semelhante no Ninja protótipo.

A execução de nosso teste revela que, embora possamos ter ensinado o ninja a dançar, não conseguimos fazer o Ninja uma Pessoa, como mostrado na figura 7.12. Nós ensinamos o Ninja imitar a dança

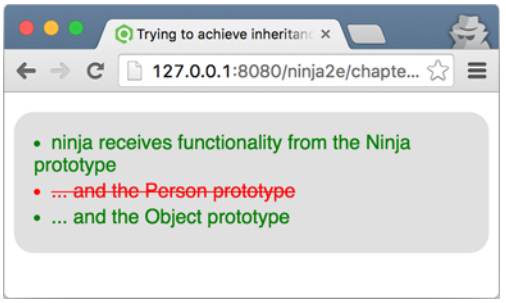


Figura 7.12 Nosso **Ninja** não é realmente um **Pessoa.** Sem dança feliz!

de uma pessoa, mas isso não *fez* a Ninja uma Pessoa. Isso não é herança - é apenas cópia.

Além do fato de que essa abordagem não está exatamente funcionando, também precisaríamos copiar cada propriedade de Pessoa ao Ninja protótipo individualmente. Isso não é maneira de fazer herança. Vamos continuar explorando.

O que realmente queremos alcançar é um *cadeia de protótipo* para que um Ninja posso *estar* uma Pessoa, e um Pessoa pode ser um Mamífero, e um Mamífero pode ser um Animal, e assim por diante, todo o caminho para Objeto. A melhor técnica para criar essa cadeia de protótipo é usar uma instância de um objeto como o protótipo do outro objeto:

SubClass. protótipo = novo ***SuperClass ();***

Por exemplo:

```
Ninja.protótipo = nova pessoa ();
```

Isso preserva a cadeia de protótipo, porque o protótipo do Subclasse instância será uma instância do SuperClass, que tem um protótipo com todas as propriedades de SuperClass, e que, por sua vez, terá umotótipo que aponta para uma instância de *Está* superclasse e assim por diante. Na próxima listagem, alteramos ligeiramente a listagem 7.7 para usar essa técnica.

Listagem 7.8 Obtendo herança com protótipos

```
função Person () {}
Person.prototype.dance = function () {};

função Ninja () {}
Ninja.prototype = nova pessoa();

const ninja = novo Ninja (); assert (instância ninja de
Ninja,
        "ninja recebe funcionalidade do protótipo Ninja"); assert (instância ninja de Pessoa, "... e o protótipo da
Pessoa"); assert (ninja instanceof Object, "... and the Object prototype"); assert (typeof ninja.dance === "function", "... and
can dance!")
```

Torna um Ninja uma Pessoa, tornando o protótipo Ninja uma instância de Pessoa

A única mudança no código é usar uma instância de Pessoa como o protótipo para Ninja. A execução dos testes mostra que fomos bem-sucedidos, conforme mostrado na figura 7.13. Agora vamos dar uma olhada mais de perto no funcionamento interno, observando o estado do aplicativo depois de criarmos o novo ninja objeto, conforme mostrado na figura 7.14.

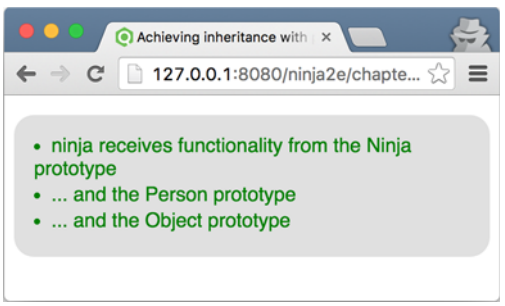


Figura 7.13 Nosso **Ninja** é um **Pessoa!** Deixe a dança da vitória começar.

A Figura 7.14 mostra que quando definimos um Pessoa função, um Pessoa

```
função Pessoa(){}
Person.prototype.dance = function () {};

função Ninja () {}
Ninja.prototype = novo Pessoa();
const ninja = novo Ninja ();
```

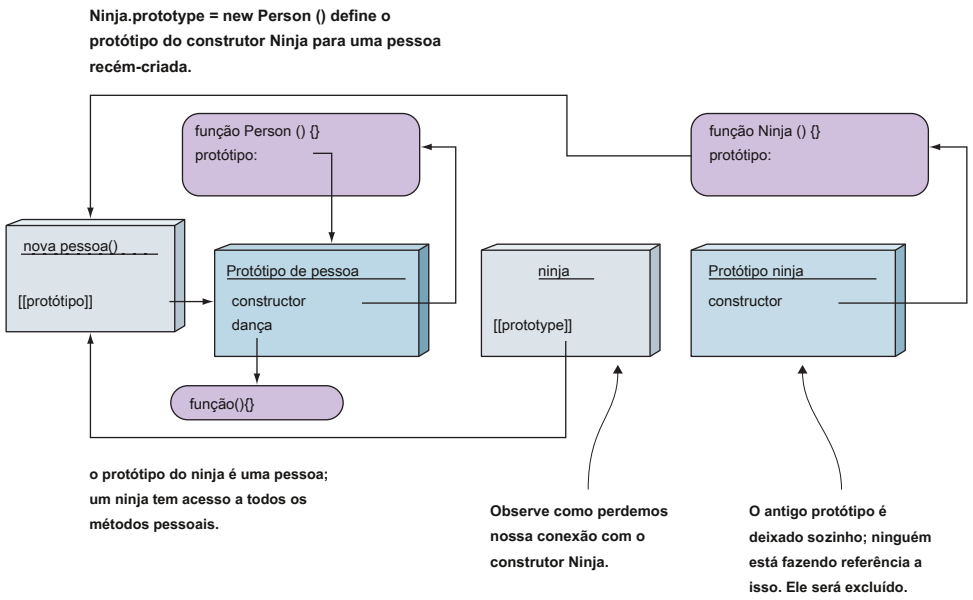


Figura 7.14 Conseguimos herança definindo o protótipo do **Ninja** construtor para uma nova instância de um **Pessoa** objeto.

protótipo também é criado que faz referência ao Pessoa função através de seu constructor propriedade. Normalmente, podemos estender o Person protótipo com propriedades adicionais e, neste caso, especificamos que cada pessoa, criada com o Person construtor, tem acesso ao dança método:

```
função Person () {}
Person.prototype.dance = function () {};
```

Também definimos um Ninja função que obtém seu próprio objeto de protótipo com um constructor propriedade referenciando o Ninja função: função Ninja () {}.

Em seguida, a fim de alcançar a herança, substituímos o protótipo do Ninja funcionar com um novo Pessoa instância. Agora, quando criamos um novo Ninja objeto, a propriedade protótipo interno do recém-criado ninja objeto será definido como o objeto para o qual o Ninja propriedade de protótipo aponta para, o previamente construído Pessoa instância:

```
função Ninja () {}
Ninja.protótipo = nova pessoa (); var ninja = novo Ninja ();
```

Quando tentamos acessar o dança método através do ninja objeto, o tempo de execução do JavaScript primeiro verificará o ninja objeto em si. Porque não tem o dança propriedade, seu protótipo, o pessoa objeto, é pesquisado. O pessoa objeto também não tem o
dança propriedade, então seu protótipo é pesquisado e a propriedade finalmente encontrada. É assim que se obtém herança em JavaScript!

Aqui está a implicação importante: quando executamos um instancia de operação, podemos determinar se a função herda a funcionalidade de qualquer objeto em sua cadeia de protótipo.

> NOTA Outra técnica que pode ter ocorrido com você, e que recomendamos fortemente contra, é usar o Pessoa objeto de protótipo diretamente como o Ninja
> protótipo, assim: Ninja.prototype = Person.prototype. Quaisquer mudanças para
> a Ninja protótipo também mudará o Pessoa protótipo (porque eles são o mesmo objeto), e isso provavelmente terá efeitos colaterais indesejáveis.

Um efeito colateral adicional feliz de fazer a herança do protótipo dessa maneira é que todos os protótipos de função herdada continuarão a atualizar ao vivo. Os objetos que herdam do protótipo sempre têm acesso às propriedades do protótipo atual.

7.3.1 *O problema de substituir a propriedade do construtor*

Se olharmos mais de perto a figura 7.14, veremos isso definindo o novo Pessoa objeto como um protótipo do Ninja construtor, perdemos nossa conexão com o Ninja construtor que foi anteriormente mantido pelo original Ninja protótipo. Isso é um problema, porque o construtor propriedade pode ser usada para determinar a função com a qual o objeto foi criado. Alguém usando nosso código pode fazer uma suposição perfeitamente razoável de que o seguinte teste será aprovado:

```
assert (ninja.constructor === Ninja,
        "O objeto ninja foi criado pelo construtor Ninja");
```

Mas, no estado atual do aplicativo, esse teste falha. Como mostra a figura 7.14, se pesquisarmos o ninja objeto para o construtor propriedade, não vamos encontrar. Então vamos ao seu protótipo, que também não tem uma propriedade constructor, e novamente, seguimos o protótipo e terminamos no objeto de protótipo de Pessoa, que tem um constructor
propriedade referenciando o Pessoa função. Na verdade, recebemos a resposta errada: se perguntarmos o ninja objeto cuja função o construiu, vamos obter Pessoa como a resposta. Isso pode ser a origem de alguns bugs graves.

Cabe a nós consertar essa situação! Mas antes de fazermos isso, precisamos fazer um desvio e ver como o JavaScript nos permite configurar propriedades.

CONFIGURANDO PROPRIEDADES DOS OBJETOS

Em JavaScript, cada propriedade de objeto é descrita com um *descritor de propriedade* através do qual podemos configurar as seguintes chaves:

- configurável —Se definido como verdade, o descritor da propriedade pode ser alterado e a propriedade pode ser excluída. Se definido para falso, não podemos fazer nenhuma dessas coisas.
- enumerável —Se definido como verdade, a propriedade aparece durante um para dentro loop sobre as propriedades do objeto (chegaremos ao para dentro loop em breve).
- valor —Especifica o valor da propriedade. Padrões para Indefinido. gravável —Se definido como verdade, o
- valor da propriedade pode ser alterado usando uma atribuição.

- obter —Define o *getter* função, que será chamada quando acessarmos a propriedade. Não pode ser definido em conjunto com valor e gravável. definir —Define o *normatizador* , que será chamada sempre que for feita
- uma atribuição à propriedade. Também não pode ser definido em conjunto com valor e

  gravável.

Digamos que criamos uma propriedade por meio de uma atribuição simples, por exemplo:

```
ninja.name = "Yoshi";
```

Esta propriedade será configurável, enumerável e gravável, seu valor será definido como Yoshi, e funções obter e definir seria Indefinido.

Quando quisermos ajustar a configuração de nossa propriedade, podemos usar o Object.defineProperty método, que pega um objeto no qual a propriedade será definida, o nome da propriedade e um objeto descritor de propriedade. Como exemplo, dê uma olhada no código a seguir.

Listagem 7.9 Configurando propriedades

```
var ninja = {};
ninja.name = "Yoshi";
ninja.weapon = "kusarigama";
```

Cria um objeto vazio; usa atribuições para adicionar duas propriedades

```
Object.defineProperty (ninja, "sorrateiro", {
    configurável: falso,
    enumerável: falso,
    valor: verdadeiro,
    gravável: verdadeiro
});
```

O Object.defineProperty integrado método é usado para ajustar os detalhes de configuração da propriedade.

```
assert ("sorrateiro" em ninja, "Podemos acessar a nova propriedade");

para (deixe prop no ninja) {
    assert (prop! == undefined, "Uma propriedade enumerada:" + prop);
}
```

Usa o loop for-in para iterar propriedades enumeráveis do ninja

Começamos com a criação de um objeto vazio, ao qual adicionamos duas propriedades:
nome e arma, à boa e velha maneira, usando atribuições. Em seguida, usamos o integrado Objeto

. defineProperty método para definir o
propriedade sorrateira, que não é configurável, não é enumerável, e tem o seu
valor definido como verdade. Esta valor pode ser mudado porque é gravável.

Finalmente, testamos se podemos acessar o recém-criado sorrateira propriedade, e usamos o para dentro loop para percorrer todas as propriedades enumeráveis do objeto. A Figura 7.15 mostra o resultado.

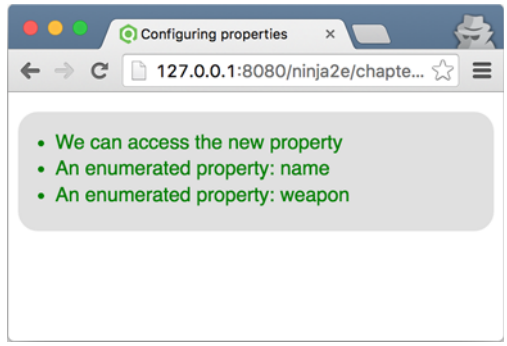


Figura 7.15 Propriedades **nome** e **arma** será visitado no **para dentro** loop, enquanto nosso especialmente adicionado **sorrateira** propriedade não (embora possamos acessá-la normalmente).

Pela configuração enumerável para falso, podemos ter certeza de que a propriedade não aparecerá ao usar o para dentro ciclo. Para entender por que queremos fazer algo assim, vamos voltar ao problema original.

RESOLVENDO FINALMENTE O PROBLEMA DE SUBSTITUIR A PROPRIEDADE DO CONSTRUTOR

Ao tentar estender Pessoa com Ninja (ou para fazer Ninja uma subclasse de Pessoa), encontramos o seguinte problema: quando definimos um novo Pessoa objeto como um protótipo para o
Ninja construtor, perdemos o original Ninja protótipo que mantém nosso constructor
propriedade. Não queremos perder o construtor , porque é útil para determinar a função usada para criar nossas instâncias de objeto e pode ser esperada por outros desenvolvedores que trabalham em nossa base de código.

Podemos resolver esse problema usando o conhecimento que acabamos de obter. Vamos definir um novo constructor propriedade no novo Ninja.prototype usando o
Object.defineProperty método. Veja a lista a seguir.

Listagem 7.10 Corrigindo o problema da propriedade do construtor

```
função Person () {}
Person.prototype.dance = function () {};

função Ninja () {}
Ninja.prototype = nova pessoa();

Object.defineProperty (Ninja.prototype, "construtor", {
    enumerável: falso,
    valor: Ninja,
    gravável: verdadeiro
});

var ninja = novo Ninja ();
```

Definimos um novo construtor não enumerável propriedade apontando de volta para Ninja.

```
assert (ninja.constructor === Ninja,
        "Conexão de instâncias ninja ao construtor Ninja
            restabelecido! ");
for (let prop in Ninja.prototype) {
    assert (prop === "dance", "A única propriedade enumerável é dance!");
}
```

Nós temos restabelecido a conexão.

Não adicionamos nenhuma propriedade enumerável ao Ninja.prototype.

Agora, se executarmos o código, veremos que está tudo ótimo. Nós restabelecemos a conexão entre ninja instâncias e o Ninja função, para que possamos saber que eles foram construídos pela Ninja função. Além disso, se alguém tentar percorrer as propriedades do Ninja.prototype objeto, certificamo-nos de que nossa propriedade corrigida construtor não será visitado. Essa é a marca de um verdadeiro ninja; entramos, fizemos nosso trabalho e saímos, sem ninguém perceber nada de fora!

7.3.2 O operador instanceof

Na maioria das linguagens de programação, a abordagem direta para verificar se um objeto faz parte de uma hierarquia de classes é usar o instancia de operador. Por exemplo, em Java, o instancia de operador funciona verificando se o objeto do lado esquerdo é a mesma classe ou uma subclasse do tipo de classe à direita.

Embora certos paralelos possam ser feitos com a forma como o instancia de operador funciona em JavaScript, há uma pequena diferença. Em JavaScript, o instancia de operador trabalha na cadeia de protótipo do objeto. Por exemplo, digamos que temos a seguinte expressão:

```
instância ninja de Ninja
```

O instancia de operador trabalha verificando se o *atual* protótipo do
Ninja função está na cadeia de protótipo do ninja instância. Voltemos às nossas pessoas e ninjas, para um exemplo mais concreto.

Listagem 7.11 Estudando o **instancia de** operador

```
função Person () {}
função Ninja () {}

Ninja.protótipo = nova pessoa ();

const ninja = novo Ninja ();

assert (instância ninja de Ninja, "Nosso ninja é um Ninja!"); assert (instância ninja de Pessoa, "Um ninja também é uma
pessoa.");
```

Uma instância ninja é tanto um Ninja e uma pessoa.

Como esperado, um ninja é, ao mesmo tempo, um Ninja e um Pessoa. Mas, para esclarecer esse ponto, a figura 7.16 mostra como a coisa toda funciona nos bastidores.

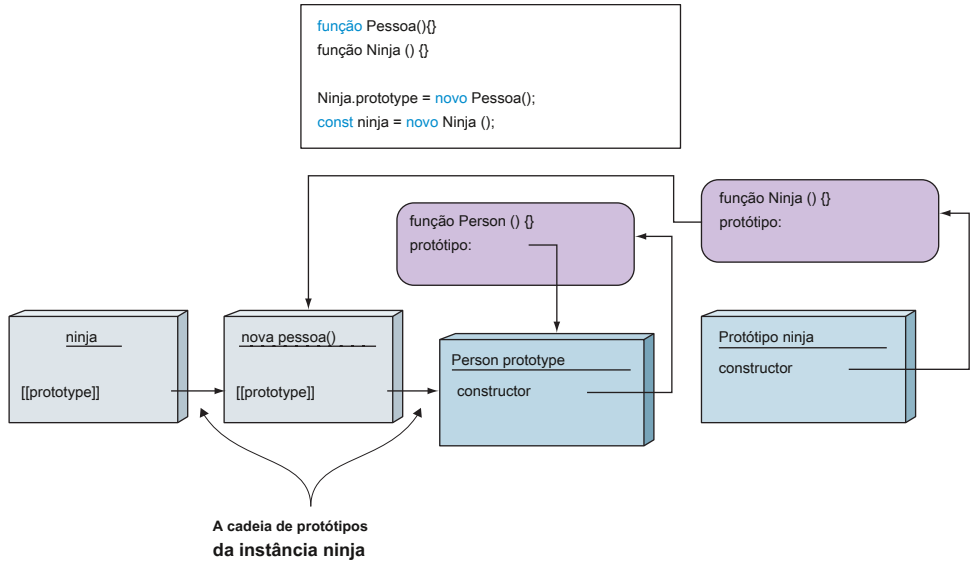


Figura 7.16 A cadeia de protótipo de um **ninja** instância é composta por um **nova pessoa()** objeto e o **Protótipo de pessoa.**

A cadeia de protótipo de uma instância ninja é composta por um new Person() objeto, através do qual alcançamos a herança, e o Protótipo de pessoa. Ao avaliar a expressão instância ninja do Ninja, o motor JavaScript pega o protótipo do
Ninja função, o new Person() objeto e verifica se ele está na cadeia de protótipos do ninja instância. Porque o nova pessoa() objeto é um protótipo direto do
ninja exemplo, o resultado é verdade.

No segundo caso, onde verificamos instância ninja de Pessoa, o motor JavaScript pega o protótipo do Pessoa função, o Protótipo de pessoa, e verifica se ele pode ser encontrado na cadeia de protótipos do ninja instância. Mais uma vez, pode, porque é o protótipo do nosso nova pessoa() objeto, que, como já vimos, é o protótipo do ninja instância.

E isso é tudo que há para saber sobre o instancia de operador. Embora seu uso mais comum seja fornecer uma maneira clara de determinar se uma instância foi criada por um construtor de função específico, ele não funciona exatamente assim. Em vez disso, ele verifica se o protótipo da função do lado direito está na cadeia de protótipos do objeto à esquerda. Portanto, há uma advertência com a qual devemos ter cuidado.

A INSTÂNCIA DA CAVEAT

Como você viu várias vezes ao longo deste capítulo, JavaScript é uma linguagem dinâmica na qual podemos modificar um *muito* de coisas durante a execução do programa. Por exemplo, não há nada que nos impeça de alterar o protótipo de um construtor, conforme mostrado na listagem a seguir.

Listagem 7.12 Cuidado com as mudanças nos protótipos do construtor

```
função Ninja () {}

const ninja = new Ninja();

assert (instância ninja de Ninja, "Nosso ninja é um Ninja!");

Ninja.prototype = {};

afirmar (! (instância do ninja Ninja), "O ninja agora não é um Ninja !?");
```

Nós mudamos o protótipo do Ninja constructor função.

Mesmo que nossa instância ninja tenha sido criada pelo construtor Ninja, o operador instanceof agora diz aquele ninja não é mais uma instância de Ninja!

Neste exemplo, repetimos novamente todas as etapas básicas para fazer um ninja instância, e nosso primeiro teste vai bem. Mas se mudarmos o protótipo do Ninja função construtora *depois de* a criação do ninja instância, e novamente teste se ninja é um
instância de Ninja, veremos que a situação mudou. Isso nos surpreenderá apenas se nos apegarmos à suposição imprecisa de que o instanceof operador nos diz se uma instância foi criada por um construtor de função particular. Se, por outro lado, tomarmos a semântica real do instanceof operador - que verifica apenas se o protótipo da função do lado direito está na cadeia de protótipos do objeto do lado esquerdo - não ficaremos surpresos. Essa situação é mostrada na figura 7.17.

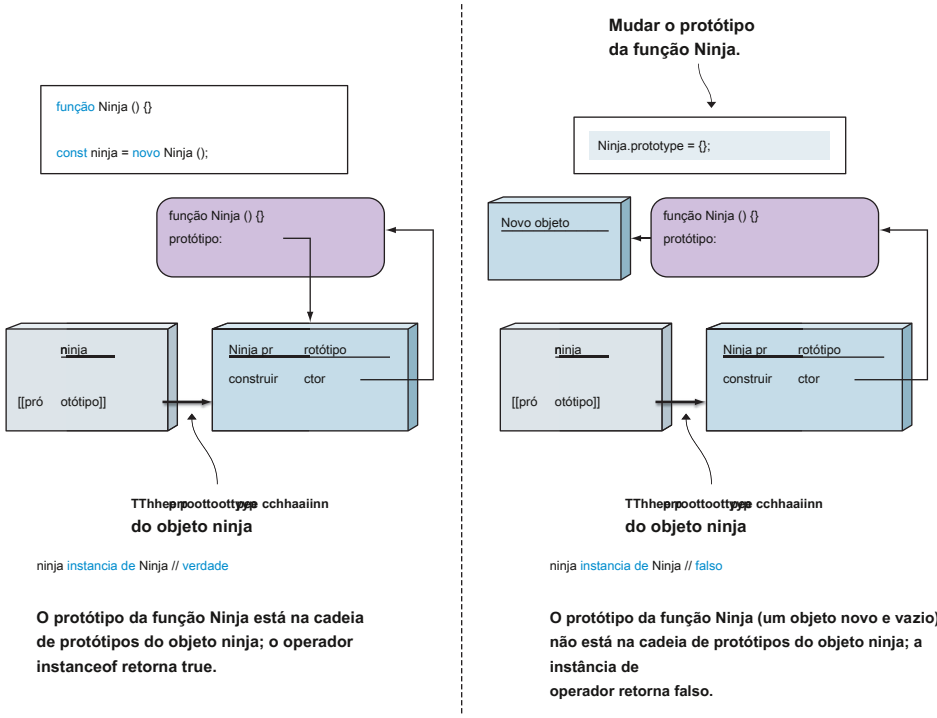


Figura 7.17 O **instancia de** operador verifica se o protótipo da função à direita está na cadeia de protótipos do objeto à esquerda. Tome cuidado; o protótipo da função pode ser alterado a qualquer momento!

Agora que entendemos como os protótipos funcionam em JavaScript e como usar protótipos em conjunto com funções de construtor para implementar herança, vamos passar para uma nova adição na versão ES6 de JavaScript: classes.

7,4 *Usando "classes" de JavaScript no ES6*

É ótimo que o JavaScript nos permita usar uma forma de herança por meio de protótipos. Mas muitos desenvolvedores, especialmente aqueles com experiência clássica em orientação a objetos, prefeririam uma simplificação ou abstração do sistema de herança do JavaScript em um com o qual estejam mais familiarizados.

Isso leva inevitavelmente ao reino das classes, embora o JavaScript não suporte herança clássica nativamente. Em resposta a essa necessidade, surgiram várias bibliotecas JavaScript que simulam a herança clássica. Como cada biblioteca implementa classes em sua própria maneira, o comitê ECMAScript padronizou a sintaxe para simular a herança baseada em classes. Observe como dissemos *simulando.* Mesmo que agora possamos usar o aula palavra-chave em JavaScript, a implementação subjacente ainda é baseada na herança do protótipo!

NOTA O aula a palavra-chave foi adicionada à versão ES6 do JavaScript e nem todos os navegadores a implementam (consulte http://mng.bz/3ykA para suporte atual).

Vamos começar estudando a nova sintaxe.

7.4.1 Usando a palavra-chave class

ES6 apresenta um novo aula palavra-chave que fornece uma maneira muito mais elegante de criar objetos e implementar herança do que implementá-la manualmente com protótipos. Usando o aula palavra-chave é fácil, conforme mostrado na lista a seguir.

Listagem 7.13 Criando uma classe no ES6

Usa a classe palavra-chave para começar especificando um ES6 aula

```
aula Ninja {
    construtor (nome) {
        this.name = nome;
    }

    swingSword () {
        return true;
    }
}

var ninja = novo Ninja ("Yoshi");

assert (instância ninja de Ninja, "Nosso ninja é um Ninja"); assert (ninja.name === "Yoshi", "nomeado
Yoshi"); assert (ninja.swingSword (), "e ele pode brandir uma espada");
```

Define uma função construtora que será chamada quando chamarmos a classe com a palavra-chave new

Define um adicional método acessível para todas as instâncias Ninja

Instancia um novo ninja objeto com o palavra-chave nova

Testes para o comportamento esperado

Listagem 7.13 mostra que podemos criar um Ninja classe usando o aula palavra-chave. Ao criar classes ES6, podemos definir explicitamente um construtor função que será invocada ao instanciar um Ninja instância. No corpo do construtor, podemos acessar a instância recém-criada com o esta palavra-chave, e podemos facilmente adicionar novas propriedades, como a nome propriedade. Dentro do corpo da classe, também podemos definir métodos que serão acessíveis a todos Ninja instâncias. Neste caso, definimos um swingSword

método que retorna verdade:

```
aula Ninja {
    construtor (nome) {
        this.name = nome;
    }

    swingSword () {
        return true;
    }
}
```

Em seguida, podemos criar um Ninja instância chamando o Ninja classe com a palavra-chave novo, assim como faríamos se Ninja era uma função construtora simples (como no início do capítulo):

```
var ninja = novo Ninja ("Yoshi");
```

Finalmente, podemos testar se o ninja instância se comporta conforme o esperado, que é um instância de Ninja, tem um nome propriedade, e tem acesso ao swingSword método:

```
assert (instância ninja de Ninja, "Nosso ninja é um Ninja"); assert (ninja.name === "Yoshi", "nomeado
Yoshi"); assert (ninja.swingSword (), "e ele pode brandir uma espada");
```

AS AULAS SÃO AÇÚCAR SINTÁTICO

Conforme mencionado anteriormente, embora ES6 tenha introduzido o aula palavra-chave, sob o capô ainda estamos lidando com os bons e velhos protótipos; classes são açúcares sintáticos projetados para tornar nossas vidas um pouco mais fáceis ao imitar classes em JavaScript.

Nosso código de classe da listagem 7.13 pode ser traduzido para código ES5 funcionalmente idêntico:

```
function Ninja (nome) {
    this.name = nome;
}
Ninja.prototype.swingSword = function () {
    return true;
};
```

Como você pode ver, não há nada de especialmente novo nas classes ES6. O código é mais elegante, mas os mesmos conceitos são aplicados.

MÉTODOS ESTÁTICOS

Nos exemplos anteriores, você viu como definir métodos de objeto (métodos de protótipo), acessíveis a todas as instâncias de objeto. Além de tais métodos, as linguagens clássicas orientadas a objetos, como Java, usam métodos estáticos, métodos definidos em nível de classe. Veja o seguinte exemplo.

Listagem 7.14 Métodos estáticos em ES6

```
class Ninja {
    construtor (nome, nível) {
        this.name = nome;
        this.level = level;
    }

    swingSword () {
        return true;
    }

    estático compare (ninja1, ninja2) {
        retornar ninja1.level - ninja2.level;
    }
}

var ninja1 = novo Ninja ("Yoshi", 4); var ninja2 = novo Ninja ("Hattori",
3);

assert (! ("compare" no ninja1) &&! ("compare" no ninja2),
            "Uma instância ninja não sabe como comparar");

assert (Ninja.compare (ninja1, ninja2)> 0,
            "A classe Ninja pode fazer a comparação!");

assert (! ("swingSword" em Ninja),
            "A classe Ninja não pode brandir uma espada");
```

Usa a estática palavra-chave para fazer um método estático

instâncias ninja não tem acesso comparar.

A classe Ninja tem acesso ao método de comparação.

Nós novamente criamos um Ninja classe que tem um swingSword método acessível de todos ninja instâncias. Também definimos um método estático, comparar, prefixando o nome do método com a palavra-chave estático.

```
estático compare (ninja1, ninja2) {
        retornar ninja1.level - ninja2.level;
}
```

O comparar método, que compara os níveis de habilidade de dois ninjas, é definido no nível da classe, e não no nível da instância! Mais tarde, testamos se isso significa efetivamente que o comparar método não é acessível de ninja instâncias, mas está acessível a partir do Ninja aula:

```
assert (! ("compare" no ninja1) &&! ("compare" no ninja2),
            "A instância ninja não sabe como comparar"); assert (Ninja.compare (ninja1, ninja2)> 0,

            "A classe Ninja pode fazer a comparação!");
```

Também podemos ver como os métodos “estáticos” podem ser implementados no código pré-ES6. Para isso, devemos lembrar apenas que as classes são implementadas por meio de funções. Como os métodos estáticos são métodos de nível de classe, podemos implementá-los tirando proveito de funções como objetos de primeira classe e adicionando uma propriedade de método à nossa função de construtor, como no exemplo a seguir:

```
função Ninja () {}
Ninja.compare = função (ninja1, ninja2) {...}
```

Estende a função do construtor com um método para imitar métodos estáticos no código pré-ES6

Agora, vamos prosseguir para a herança.

7.4.2 Implementando herança

Para ser honesto, executar a herança no código pré-ES6 pode ser uma dor. Vamos voltar aos nossos ninjas confiáveis, exemplo de Pessoas:

```
função Person () {}
Person.prototype.dance = function () {};

função Ninja () {}
Ninja.prototype = nova pessoa();

Object.defineProperty (Ninja.prototype, "construtor", {
    enumerável: falso,
    valor: Ninja,
    gravável: verdadeiro
});
```

Há muito para se manter em mente aqui: métodos acessíveis a todas as instâncias devem ser adicionados diretamente ao protótipo da função do construtor, como fizemos com o dança
método e o Pessoa construtor. Se quisermos implementar herança, temos que definir o protótipo da “classe” derivada para a instância da “classe” base. Neste caso, atribuímos uma nova instância de Pessoa para Ninja.prototype. Infelizmente, isso bagunça o construtor propriedade, então temos que restaurá-la manualmente com o

Object.defineProperty método. É muito importante ter isso em mente ao tentar obter um recurso relativamente simples e comumente usado (herança). Felizmente, com o ES6, tudo isso é significativamente simplificado.

Vamos ver como isso é feito na lista a seguir.

Listagem 7.15 Herança em ES6

```
class Person {
    construtor (nome) {
        this.name = nome;
    }

    dança(){
        return true;
    }
```

```
}

classe Ninja extends Person {
    construtor (nome, arma) {
        super (nome);
        this.weapon = arma;
    }

    wieldWeapon () {
        return true;
    }
}

var pessoa = nova pessoa ("Bob");

assert (person instanceof Person, "Uma pessoa é uma pessoa"); assert (person.dance (), "Uma pessoa pode dançar."); assert
(person.name === "Bob", "Podemos chamá-lo pelo nome."); assert (! (person instanceof Ninja), "Mas não é um Ninja"); assert (!
("wieldWeapon" pessoalmente), "E não pode empunhar uma arma");



var ninja = novo Ninja ("Yoshi", "Wakizashi"); assert (instância ninja de Ninja, "A ninja's a ninja"); assert
(ninja.wieldWeapon (), "Isso pode empunhar uma arma"); assert (instância ninja de Person, "Mas também é
uma pessoa"); assert (ninja.name === "Yoshi", "Isso tem um nome"); assert (ninja.dance (), "E gosta de
dançar");
```

Usa a palavra-chave extends para herdar de outra classe

Usa a palavra-chave super para chamar o construtor da classe base

A Listagem 7.15 mostra como obter herança no ES6; nós usamos o estende palavra-chave para herdar de outra classe:

```
classe Ninja estende Pessoa
```

Neste exemplo, criamos um Pessoa classe com um construtor que atribui um nome para cada Pessoa instância. Também definimos um dança método que será acessível a todos Pessoa instâncias:

```
class Person {
    construtor (nome) {
        this.name = nome;
    }
    dança(){
        return true;
    }
}
```

Em seguida, definimos um Ninja classe que estende o Pessoa aula. Tem um adicional arma propriedade, e um empunhar arma método:

```
classe Ninja extends Person {
    construtor (nome, arma) {
        super (nome);
```

```
        this.weapon = arma;
    }

    wieldWeapon () {
        return true;
    }
}
```

No construtor do derivado, Ninja classe, há uma chamada para o construtor da base, Pessoa classe, por meio da palavra-chave super. Isso deve ser familiar, se você já trabalhou com qualquer linguagem baseada em classe.

Continuamos criando um pessoa instância e verificando se é uma instância do Pessoa classe que tem um nome e pode dança. Só para ter certeza, também verificamos se uma pessoa que *não é* um Ninja não pode empunhar uma arma:

```
var pessoa = nova pessoa ("Bob");

assert (person instanceof Person, "Uma pessoa é uma pessoa"); assert (person.dance (), "Uma pessoa pode dançar."); assert (person.name === "Bob", "Podemos chamá-lo pelo nome."); assert (! (person instanceof Ninja), "Mas não é um Ninja"); assert (! ("wieldWeapon" pessoalmente), "E não pode empunhar uma arma");
```

Nós também criamos um ninja instância e verifique se é uma instância de Ninja e pode empunhar uma arma. Porque todo ninja também é um Pessoa, verificamos se um ninja é uma instância de Pessoa, que tem um nome, e que também, no ínterim da luta, gosta de dançar:

```
var ninja = novo Ninja ("Yoshi", "Wakizashi"); assert (instância ninja de Ninja, "A ninja's a ninja"); assert (ninja.wieldWeapon (), "Isso pode empunhar uma arma"); assert (instância ninja de Person, "Mas também é uma pessoa"); assert (ninja.name === "Yoshi", "Isso tem um nome"); assert (ninja.dance (), "E gosta de dançar");
```

Veja como isso é fácil? Não há necessidade de pensar sobre protótipos ou os efeitos colaterais de certas propriedades substituídas. Definimos classes e especificamos seu relacionamento usando o estende palavra-chave. Finalmente, com o ES6, hordas de desenvolvedores vindos de linguagens como Java ou C # podem ficar em paz.

E é isso. Com o ES6, construímos hierarquias de classes quase tão facilmente quanto em qualquer outra linguagem orientada a objetos mais convencional.

7,5 Resumo

- Os objetos JavaScript são coleções simples de propriedades nomeadas com valores. JavaScript usa
- protótipos.
- Cada objeto pode ter uma referência a um *protótipo,* um objeto ao qual delegamos a busca por uma propriedade particular, se o próprio objeto não tiver a propriedade procurada. O protótipo de um objeto pode ter seu próprio protótipo, e assim por diante, formando um *cadeia de protótipo*.

- Podemos definir o protótipo de um objeto usando o Object.setPrototypeOf método.
- Os protótipos estão intimamente ligados às funções do construtor. Cada função tem um protótipo propriedade que é definida como o protótipo dos objetos que ela instancia. Uma função protótipo objeto
- tem um construtor propriedade apontando de volta para a própria função. Esta propriedade é acessível a todos os objetos instanciados com aquela função e, com certas limitações, pode ser usada para descobrir se um objeto foi criado por uma função específica.

- Em JavaScript, quase tudo pode ser alterado em tempo de execução, incluindo os protótipos de um objeto e os protótipos de uma função!
- Se quisermos as instâncias criadas por um Ninja função de construtor para "herdar" (mais precisamente, ter acesso a) propriedades acessíveis a instâncias criadas pelo Pessoa função de construtor, defina o protótipo do Ninja construtor para uma nova instância do Pessoa aula.

- Em JavaScript, as propriedades têm atributos (configuráveis, enumeráveis, graváveis). Essas propriedades podem ser definidas usando o built-in Object.defineProperty método. JavaScript ES6 adiciona suporte para um aula
- palavra-chave que nos permite simular classes mais facilmente. Nos bastidores, os protótipos ainda estão em jogo!
- O estende palavra-chave habilita herança elegante.

7,6 Exercícios

1 Qual das seguintes propriedades aponta para um objeto que será pesquisado se o objeto de destino não tem a propriedade procurada?

uma aula

b instância

c protótipo

d aponta para

2 Qual é o valor da variável a1 depois que o código a seguir é executado?

```
função Ninja () {}
Ninja.prototype.talk = function () {
    retornar "Olá";
};

const ninja = novo Ninja (); const a1 = ninja.talk ();
```

3 Qual é o valor de a1 depois de executar o seguinte código?

```
função Ninja () {}
Ninja.message = "Olá";

const ninja = novo Ninja ();

const a1 = ninja.mensagem;
```

4 Explique a diferença entre o getFullName método nestes dois códigos fragmentos:

```
// Primeiro fragmento
função Pessoa (primeiroNome, últimoNome) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.getFullName = function () {
        retorna this.firstName + "" + this.lastName;
    }
}

// segundo fragmento
função Pessoa (primeiroNome, últimoNome) {
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.getFullName = function () {
    retorna this.firstName + "" + this.lastName;
}
```

5 Depois de executar o código a seguir, o que ninja.constructor aponta para?

```
function Person () {} function Ninja () {}


const ninja = novo Ninja ();
```

6 Depois de executar o código a seguir, o que ninja.constructor aponta para?

```
function Person () {} function Ninja () {}

Ninja.protótipo = nova pessoa (); const ninja = novo Ninja ();
```

7 Explique como o instancia de operador funciona no exemplo a seguir.

```
função Guerreiro () {}

função Samurai () {}
Samurai.prototype = novo Guerreiro ();

var samurai = novo Samurai ();

exemplo de samurai do guerreiro; //Explicar
```

8 Traduza o seguinte código ES6 para o código ES5.

```
class Warrior {
    construtor (arma) {
```

```
        this.weapon = arma;
    }

    manejar() {
        return "Wielding" + this.weapon;
    }

    duelo estático (guerreiro1, guerreiro2) {
        return warrior1.wield () + "" + warrior2.wield ();
    }
}
```

8 Controle de acesso a objetos

Este capítulo cobre

- Usando getters e setters para controlar o acesso às propriedades do objeto
- Controlar o acesso a objetos por meio de proxies
- Usando proxies para questões transversais

No capítulo anterior, você viu que os objetos JavaScript são coleções dinâmicas de propriedades. Podemos facilmente adicionar novas propriedades, alterar os valores das propriedades e, eventualmente, remover completamente as propriedades existentes. Em muitas situações (por exemplo, ao validar valores de propriedade, registrar ou exibir dados na IU), precisamos ser capazes de monitorar exatamente o que está acontecendo com nossos objetos. Portanto, neste capítulo, você aprenderá técnicas para controlar o acesso e monitorar todas as mudanças que ocorrem em seus objetos.

Começaremos com getters e setters, métodos que controlam o acesso a propriedades específicas do objeto. Você viu esses métodos em ação pela primeira vez nos capítulos 5 e 7. Neste capítulo, você verá alguns de seus suportes de linguagem embutidos e como usá-los para registro, execução de validação de dados e definição de propriedades computadas.

Continuaremos com proxies, um tipo de objeto completamente novo introduzido no ES6. Esses objetos controlam o acesso a outros objetos. Você aprenderá como eles funcionam e como usá-los com grande eficácia para expandir facilmente seu código com questões transversais

como medição de desempenho ou registro e como evitar exceções nulas ao preencher automaticamente as propriedades do objeto. Vamos começar a jornada com algo que já sabemos até certo ponto: getters e setters.

Você sabe?

Quais são alguns dos benefícios de acessar uma propriedade valor por meio de getters e setters?

Qual é a principal diferença entre proxies e getters e setters?

O que são armadilhas de proxy? Cite três tipos de armadilha.

8,1 Controle de acesso a propriedades com getters e setters

Em JavaScript, os objetos são coleções relativamente simples de propriedades. A principal forma de acompanhar o estado do nosso programa é modificando essas propriedades. Por exemplo, considere o seguinte código:

```
função Ninja (nível) {
    this.skillLevel = nível;
}
const ninja = novo Ninja (100);
```

Aqui nós definimos um Ninja construtor que cria ninja objetos com uma propriedade nível de habilidade. Mais tarde, se quisermos alterar o valor dessa propriedade, podemos escrever o seguinte
código ing: ninja.skillLevel = 20.

Tudo isso é bom e conveniente, mas o que acontece nos casos a seguir?

- Queremos nos proteger contra erros acidentais, como atribuição de dados imprevistos. Por exemplo, queremos nos impedir de fazer algo como atribuir um valor de um tipo errado: ninja.skillLevel = "alto".

- Queremos registrar todas as alterações no nível de habilidade propriedade. Precisamos mostrar o valor do nosso nível de
- habilidade propriedade em algum lugar na interface do usuário de nossa página da web. Naturalmente, queremos apresentar o último valor atualizado da propriedade, mas como podemos fazer isso facilmente?

Podemos lidar com todos esses casos elegantemente com os métodos getter e setter.

No capítulo 5, você viu os getters e setters como um meio de imitar propriedades de objetos privados em JavaScript por meio de fechamentos. Vamos revisitar o material que você aprendeu até agora, trabalhando com ninjas que têm um nível de habilidade propriedade acessível apenas por meio de getters e setters, conforme mostrado na listagem a seguir.

Listagem 8.1 Usando getters e setters para proteger propriedades privadas

```
function Ninja () {
    let skillLevel;                          <-- Define uma variável privada skillLevel
```

```
    this.getSkillLevel = () => skillLevel;

    this.setSkillLevel = value => {
        skillLevel = value;
    };
}

const ninja = novo Ninja (); ninja.setSkillLevel
(100);
assert (ninja.getSkillLevel () === 100,
            "Nosso ninja está no nível 100!");
```

O método getter controla o acesso à nossa variável particular skillLevel.

O método setter controla os valores que podemos atribuir a skillLevel.

Define um novo valor de skillLevel por meio do método setter

Recupera o valor de skillLevel com o método getter

Nós definimos um Ninja construtor que cria ninjas com um "privado" nível de habilidade variável acessível apenas através de nosso getSkillLevel e setSkillLevel métodos: O valor da propriedade pode ser obtido apenas através do getSkillLevel método, enquanto um novo valor de propriedade pode ser definido apenas por meio do setSkillLevel método (lembra do capítulo 5 sobre fechamentos?).

Agora, se quisermos registrar todas as tentativas de leitura do nível de habilidade propriedade, nós expandimos o getSkillLevel método; e se quisermos reagir a todas as tentativas de gravação, expandimos o setSkillLevel método, como no seguinte snippet:

```
function Ninja () {
    let skillLevel;

    this.getSkillLevel = () => {
        relatório ("Obtendo valor do nível de habilidade");
        return skillLevel;
    };

    this.setSkillLevel = value => {
        report ("Modificando a propriedade skillLevel de:",
                    nível de habilidade, " para: ", valor); skillLevel =
        value;
    }
}
```

Usando getters, podemos saber sempre que o código acessa uma propriedade.

Usando setters, podemos saber sempre que o código deseja definir um novo valor para uma propriedade.

Isso é ótimo. Podemos reagir facilmente a todas as interações com nossas propriedades, conectando, por exemplo, registro, validação de dados ou outros efeitos colaterais, como modificações da IU.

Mas uma preocupação persistente pode estar surgindo em sua mente. O nível de habilidade propriedade é uma propriedade de valor; faz referência a dados (o número 100) e não a uma função. Infelizmente, para tirar proveito de todos os benefícios do acesso controlado, todas as nossas interações com a propriedade devem ser feitas chamando explicitamente os métodos associados, o que é, para ser honesto, um pouco estranho.

Felizmente, o JavaScript tem suporte integrado para getters e setters verdadeiros: propriedades que são acessadas como propriedades de dados normais (por exemplo, ninja.skillLevel), mas esses são métodos que podem calcular o valor de uma propriedade solicitada, validar o valor passado ou qualquer outra coisa que precisemos que eles façam. Vamos dar uma olhada neste suporte integrado.

8.1.1 Definindo getters e setters

Em JavaScript, os métodos getter e setter podem ser definidos de duas maneiras:

- Especificando-os em literais de objeto ou nas definições de classe ES6
- Usando o integrado Object.defineProperty método

O suporte explícito para getters e setters já existe há algum tempo, desde os dias de ES5. Como sempre, vamos explorar a sintaxe por meio de um exemplo. Nesse caso, temos um objeto que armazena uma lista de ninjas e queremos ser capazes de obter e definir o primeiro ninja da lista.

Listagem 8.2 Definindo getters e setters em literais de objeto

```
const ninjaCollection = {
    ninjas: ["Yoshi", "Kuma", "Hattori"],
    obter firstNinja () {
        report ("Getting firstNinja");
        retornar this.ninjas [0];
    },
    definir firstNinja (valor) {
        report ("Configurando firstNinja");
        this.ninjas [0] = valor;
    }
};

assert (ninjaCollection.firstNinja === "Yoshi",
            "Yoshi é o primeiro ninja");

ninjaCollection.firstNinja = "Hachi";

assert (ninjaCollection.firstNinja === "Hachi"
        && ninjaCollection.ninjas [0] === "Hachi",
            "Agora Hachi é o primeiro ninja");
```

Define um método getter para a propriedade firstNinja que retorna o primeiro ninja em nossa coleção e registra uma mensagem

Define um método setter para a propriedade firstNinja que modifica o primeiro ninja em nossa coleção e registra uma mensagem

Acessa a propriedade firstNinja como se fosse uma propriedade de objeto padrão

Modifica a propriedade firstNinja como se fosse uma propriedade de objeto padrão

Testa se a modificação da propriedade é armazenada

Este exemplo define um ninjaCollection objeto que possui uma propriedade padrão, ninjas, que faz referência a uma matriz de ninjas e um getter e um setter para a propriedade firstNinja. A sintaxe geral para getters e setters é mostrada na figura 8.1.

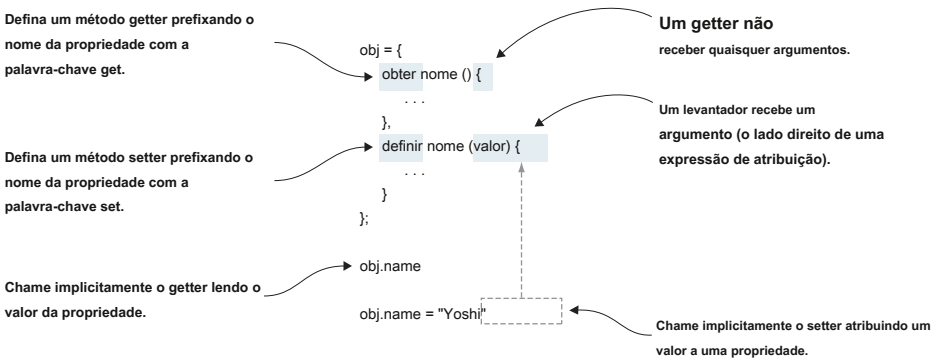


Figura 8.1 A sintaxe para definir getters e setters. Prefixe o nome da propriedade com o **obter** ou o **definir** palavra-chave.

Como você pode ver, definimos uma propriedade getter prefixando o nome com um obter palavra-chave e uma propriedade setter com um definir palavra-chave.

Na Listagem 8.2, tanto o getter quanto o setter registram uma mensagem. Além disso, o getter retorna o valor do ninja no índice 0, e o setter atribui um novo valor ao ninja no mesmo índice:

```
obter firstNinja () {
    report ("Getting firstNinja");
    retornar this.ninjas [0];
},
definir firstNinja (valor) {
    report ("Configurando firstNinja");
    this.ninjas [0] = valor;
}
```

Em seguida, testamos se o acesso à propriedade getter retorna o primeiro ninja, Yoshi:

```
afirmar( ninjaCollection.firstNinja === " Yoshi ",
            "Yoshi é o primeiro ninja");
```

Observe que a propriedade getter é acessada como se fosse uma propriedade de objeto padrão (e não como o método que é).

Depois de acessar uma propriedade getter, o método getter associado é implicitamente chamado, a mensagem Pegando o primeiro Ninja é registrado, e o valor do ninja no índice 0 é devolvido.

Continuamos tirando proveito de nosso método setter e escrevendo para o firstNinja propriedade, novamente, da mesma forma que atribuiríamos um novo valor a uma propriedade de objeto normal:

```
ninjaCollection.firstNinja = "Hachi";
```

Semelhante ao caso anterior, porque o firstNinja A propriedade tem um método setter, sempre que atribuímos um valor a essa propriedade, o método setter é chamado implicitamente. Isso registra a mensagem Configurando o firstNinja e modifica o valor do ninja no índice 0

Finalmente, podemos testar se nossa modificação fez o trabalho e se o novo valor do ninja no índice 0 pode ser acessado através do ninjas coleção e por meio de nosso método getter:

```
assert (ninjaCollection.firstNinja === "Hachi"
        && ninjaCollection.ninjas [0] === "Hachi",
            "Agora Hachi é o primeiro ninja");
```

A Figura 8.2 mostra a saída gerada pela Listagem 8.2. Quando acessamos uma propriedade com um getter (por exemplo, por meio de ninjaCollection.firstNinja), o método getter é chamado imediatamente e, neste caso, a mensagem Pegando o primeiro Ninja é registrado. Mais tarde, testamos se a saída é Yoshi e que a mensagem Yoshi é o primeiro ninja
é registrado. Procedemos de forma semelhante, atribuindo um novo valor ao firstNinja propriedade,

Figura 8.2 A saída da listagem 8.2: se uma propriedade tem um método getter e um método setter, o método getter é implicitamente chamado sempre que lemos o valor da propriedade, e o método setter é chamado sempre que atribuímos um novo valor à propriedade.

e como podemos ver na saída, isso aciona implicitamente a execução do método setter, que gera a mensagem Configurando firstNinja.

Um ponto importante a tirar de tudo isso é que getters e setters nativos nos permitem especificar propriedades que são acessadas como propriedades padrão, mas que são métodos cuja execução é acionada imediatamente quando a propriedade é acessada. Isso é enfatizado ainda mais na figura 8.3.

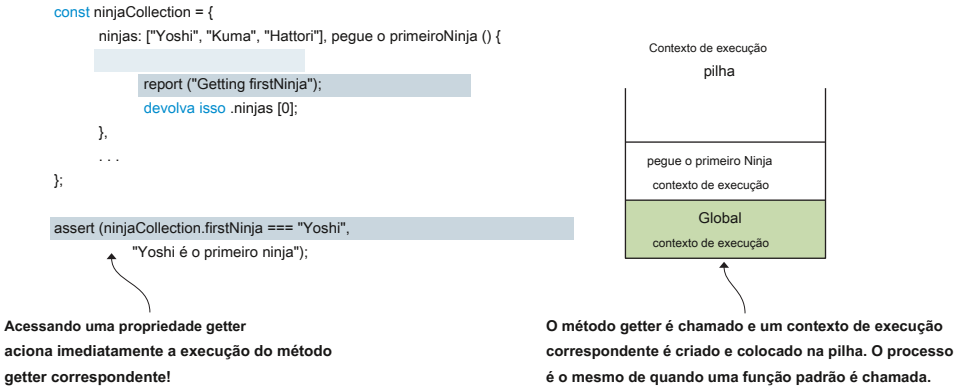


Figura 8.3 Acessar uma propriedade com um método getter chama implicitamente o getter correspondente. O processo é o mesmo como se fosse uma chamada de método padrão e o método getter é executado. Algo semelhante acontece quando atribuímos um valor a uma propriedade por meio de um método setter.

Essa sintaxe para definir um getter e um setter é direta, portanto, não é de se admirar que possamos usar exatamente a mesma sintaxe para definir getters e setters em outras situações. O exemplo a seguir usa classes ES6.

Listagem 8.3 Usando getters e setters com classes ES6

```
class NinjaCollection {
    construtor(){
        this.ninjas = ["Yoshi", "Kuma", "Hattori"];
    }
```

```
    obter firstNinja () {
        report ("Getting firstNinja");
        retornar this.ninjas [0];
    }
    definir firstNinja (valor) {
        report ("Configurando firstNinja");
        this.ninjas [0] = valor;
    }
}
const ninjaCollection = new NinjaCollection ();

assert (ninjaCollection.firstNinja === "Yoshi",
            "Yoshi é o primeiro ninja");

ninjaCollection.firstNinja = "Hachi";

assert (ninjaCollection.firstNinja === "Hachi"
        && ninjaCollection.ninjas [0] === "Hachi",
            "Agora Hachi é o primeiro ninja");
```

Define um getter e um setter dentro de uma classe ES6

Isso modifica o código da listagem 8.2 para incluir classes ES6. Guardamos todos os testes para verificar se o exemplo ainda funciona conforme o esperado.

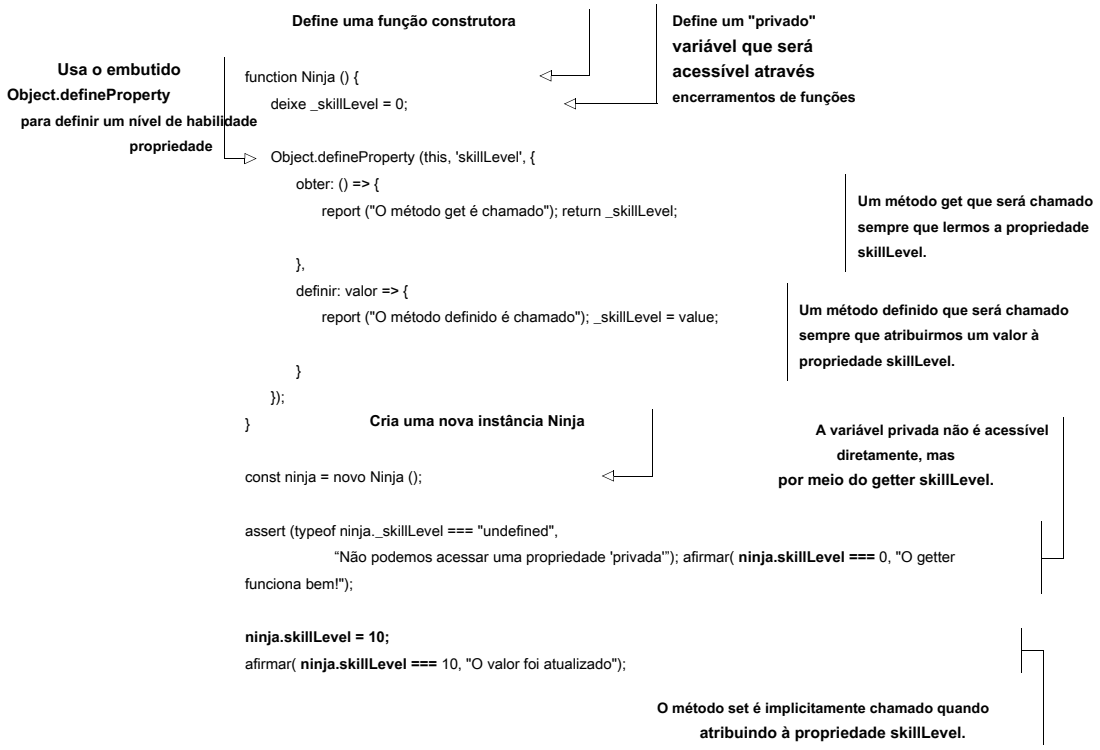
NOTA Nem sempre precisamos definir um getter e um setter para uma determinada propriedade. Por exemplo, muitas vezes queremos fornecer apenas um getter. Se, nesse caso, ainda tentarmos escrever um valor para essa propriedade, o comportamento exato depende se o código está no modo estrito ou não estrito. Se o código estiver no modo não estrito, atribuir um valor a uma propriedade com apenas um getter não leva a nada; o mecanismo JavaScript irá ignorar silenciosamente nossa solicitação. Se, por outro lado, o código estiver no modo estrito, o mecanismo JavaScript emitirá um erro de tipo, indicando que estamos tentando atribuir um valor a uma propriedade que possui um getter, mas nenhum setter.

Embora especificar getters e setters por meio de classes ES6 e literais de objeto seja fácil, você provavelmente percebeu que algo está faltando. Tradicionalmente, getters e setters são usados para controlar o acesso às propriedades do objeto privado, como na Listagem 8.1. Infelizmente, como já sabemos no capítulo 5, o JavaScript não tem propriedades de objeto privado. Em vez disso, podemos imitá-los por meio de fechamentos, definindo variáveis e especificando métodos de objeto que fecharão sobre essas variáveis. Como com literais e classes de objeto, nossos métodos getter e setter não são criados no mesmo escopo de função que as variáveis que poderíamos usar para propriedades de objeto privado, não podemos fazer isso. Felizmente, há uma maneira alternativa, por meio do Object.defineProper método.

No capítulo 7, você viu que o Object.defineProperty O método pode ser usado para definir novas propriedades passando em um objeto descritor de propriedade. Entre outras coisas, o descritor de propriedade pode incluir um obter e um definir propriedade que define os métodos getter e setter da propriedade.

Usaremos esse recurso para modificar a listagem 8.1 para implementar getters e setters integrados que controlam o acesso a uma propriedade de objeto “privada”, conforme mostrado na listagem a seguir.

Listagem 8.4 Definindo getters e setters com **Object.defineProperty**



Neste exemplo, primeiro definimos um Ninja função construtora com _ nível de habilidade variável que usaremos como uma variável privada, assim como na listagem 8.1.

Em seguida, no objeto recém-criado, referenciado pelo esta palavra-chave, definimos um nível de habilidade propriedade usando o embutido Object.defineProperty método:

```
Object.defineProperty (this, 'skillLevel', {
    obter: () => {
        report ("O método get é chamado"); return _skillLevel;

    },
    definir: valor => {
        report ("O método definido é chamado"); _skillLevel = value;

    }
});
```

Porque queremos o nível de habilidade propriedade para controlar o acesso a uma variável privada, especificamos um obter e um definir método que será chamado sempre que a propriedade for acessada.

Observe que, ao contrário de getters e setters especificados em literais e classes de objeto, o obter e definir métodos definidos através de Object.defineProperty são criados no mesmo escopo que o "privado" nível de habilidade variável. Ambos os métodos criam um fechamento

em torno da variável privada, e podemos acessar essa variável privada apenas por meio desses dois métodos.

O resto do código funciona exatamente como nos exemplos anteriores. Nós criamos um novo Ninja instância e verifique se não podemos acessar a variável privada diretamente. Em vez disso, todas as interações devem passar por getter e setter, que agora usamos como se fossem propriedades de objeto padrão:

```
ninja.skillLevel === 0      // Ativa o método getter
ninja.skillLevel = 10       // Ativa o método setter
```

Como você pode ver, a abordagem com Object.defineProperty é mais detalhado e complicado do que getters e setters em literais e classes de objeto. Mas em certos casos, quando precisamos de propriedades de objetos privados, vale a pena.

Independentemente da forma como os definimos, getters e setters nos permitem definir propriedades de objeto que são usadas como propriedades de objeto padrão, mas são métodos que podem executar código adicional sempre que lemos ou gravamos em uma propriedade particular. Este é um recurso incrivelmente útil que nos permite realizar o registro, validar valores de atribuição e até mesmo notificar outras partes do código quando certas alterações ocorrem. Vamos explorar alguns desses aplicativos.

8.1.2 Usando getters e setters para validar valores de propriedade

Como estabelecemos até agora, um setter é um método executado sempre que gravamos um valor na propriedade correspondente. Podemos tirar proveito dos setters para executar uma ação sempre que o código tentar atualizar o valor de uma propriedade. Por exemplo, uma das coisas que podemos fazer é validar o valor passado. Dê uma olhada no código a seguir, que garante que nosso nível de habilidade propriedade pode ser atribuída apenas a valores inteiros.

Listagem 8.5: Validando atribuições de valor de propriedade com setters

```
function Ninja () {
    deixe _skillLevel = 0;

    Object.defineProperty (this, 'skillLevel', {
        get: () => _skillLevel, set: value => {

            if (! Number.isInteger (value)) {
                lance novo TypeError ("Nível de habilidade deve ser um número");
            }
            _skillLevel = value;
        }
    });
}

const ninja = novo Ninja ();

ninja.skillLevel = 10;
afirmar( ninja.skillLevel === 10, "O valor foi atualizado");
```

Verifica se o valor passado é um número inteiro. Se não for, uma exceção é lançada.

Podemos atribuir um valor inteiro para a propriedade.

```
tentar {
    ninja.skillLevel = "Ótimo";
    fail ("Não deveria estar aqui"); } catch (e) {

    pass ("Definir um valor não inteiro lança uma exceção");
}
```

Tentando atribuir um valor não inteiro (em neste caso, uma string) resulta em uma exceção lançada do método setter.

Este exemplo é uma extensão direta da listagem 8.4. A única grande diferença é que agora, sempre que um novo valor é atribuído ao nível de habilidade , verificamos se o valor passado é um número inteiro. Se não for, uma exceção é lançada e o _ nível de habilidade variável não será modificada. Se tudo correr bem e um valor inteiro for recebido, terminaremos com um novo valor de _ nível de habilidade variável:

```
definir: valor => {
    if (! Number.isInteger (value)) {
        lance novo TypeError ("Nível de habilidade deve ser um número");
    }
    _skillLevel = value;
}
```

Ao testar este código, primeiro verificamos se tudo vai bem se atribuirmos um inteiro:

```
ninja.skillLevel = 10;
assert (ninja.skillLevel === 10, "O valor foi atualizado");
```

E então testamos a situação em que atribuímos erroneamente um valor de outro tipo, como uma string. Nesse caso, devemos terminar com uma exceção.

```
tentar {
    ninja.skillLevel = "Ótimo";
    fail ("Não deveria estar aqui"); } catch (e) {

    pass ("Definir um valor não inteiro lança uma exceção");
}
```

É assim que você evita todos aqueles bichinhos bobos que acontecem quando um valor do tipo errado acaba em uma determinada propriedade. Claro, isso adiciona sobrecarga, mas esse é um preço que às vezes temos que pagar para usar com segurança uma linguagem altamente dinâmica como JavaScript.

Este é apenas um exemplo da utilidade dos métodos setter; há muitos mais que não exploraremos explicitamente. Por exemplo, você pode usar o mesmo princípio para rastrear o histórico de valores, fazer o registro, fornecer notificação de mudança e muito mais.

8.1.3 *Usando getters e setters para definir propriedades computadas*

Além de ser capaz de controlar o acesso a certas propriedades do objeto, getters e setters podem ser usados para definir *propriedades computadas,* propriedades cujo valor é calculado por solicitação. Propriedades computadas não armazenam um valor; eles fornecem um obter e / ou um definir método para recuperar e definir outras propriedades indiretamente. No exemplo a seguir, o objeto tem duas propriedades, nome e clã, que usaremos para calcular a propriedade título completo.

Listagem 8.6 Definindo propriedades computadas

```
const shogun = {
    nome: "Yoshiaki",
    clã: "Ashikaga",
    get fullTitle () {
        return this.name + "" + this.clan;
    },
    set fullTitle (value) {
        const segmentos = valor.split (""); this.name = segmentos [0];

        this.clan = segmentos [1];
    }
};

assert (shogun.name === "Yoshiaki", "Nosso shogun Yoshiaki"); assert (shogun.clan === "Ashikaga", "Do clã
Ashikaga"); assert (shogun.fullTitle === "Yoshiaki Ashikaga",

            "O nome completo agora é Yoshiaki Ashikaga");

shogun.fullTitle = "Ieyasu Tokugawa";
assert (shogun.name === "Ieyasu", "Nosso shogun Ieyasu"); assert (shogun.clan === "Tokugawa", "Do clã
Tokugawa"); assert (shogun.fullTitle === "Ieyasu Tokugawa",

            "O nome completo agora é Ieyasu Tokugawa");
```

Define um método getter em uma propriedade fullTitle de um literal de objeto que calcula o valor concatenando duas propriedades de objeto

Define um método setter em uma propriedade fullTitle de um literal de objeto que divide o valor passado e atualiza duas propriedades padrão

O nome e as propriedades do clã são normais propriedades cujo valores são diretamente obtido. Acessando a propriedade fullTitle chama o método get, que calcula o valor.

Atribuir um valor à propriedade fullTitle chama o método set, que calcula e atribui novos valores às propriedades de nome e clã.

Aqui nós definimos um Shogun objeto, com duas propriedades padrão, nome e clã. Além disso, especificamos um método getter e um método setter para uma propriedade computada, título completo:

```
const shogun = {
    nome: "Yoshiaki",
    clã: "Ashikaga",
    get fullTitle () {
        return this.name + "" + this.clan;
    },
    set fullTitle (value) {
        const segmentos = valor.split (""); this.name = segmentos [0];

        this.clan = segmentos [1];
    }
};
```

O obter método calcula o valor do título completo propriedade, a pedido, concatenando o nome e clã propriedades. O definir método, por outro lado, usa o embutido dividir método, disponível para todas as strings, para dividir a string atribuída em segmentos pelo caractere de espaço. O primeiro segmento representa o nome e é atribuído ao nome propriedade, enquanto o segundo segmento representa o clã e é atribuído ao clã propriedade.

Isso cuida de ambas as rotas: Lendo o título completo propriedade calcula seu valor, e a gravação no título completo propriedade modifica as propriedades que constituem o valor da propriedade.

Para ser honesto, não precisamos usar propriedades computadas. Um método chamado getFullTitle poderia ser igualmente útil, mas as propriedades computadas podem melhorar a clareza conceitual de nosso código. Se um determinado valor (neste caso, o título completo valor) depende *só* no estado interno do objeto (neste caso, no nome e clã propriedades), faz todo o sentido representá-lo como um campo de dados, uma propriedade, em vez de uma função.

Isso conclui nossa exploração de getters e setters. Você viu que eles são uma adição útil à linguagem que pode nos ajudar a lidar com o registro, validação de dados e detecção de mudanças nos valores das propriedades. Infelizmente, às vezes isso não é suficiente. Em certos casos, precisamos controlar todos os tipos de interações com nossos objetos e, para isso, podemos usar um tipo de objeto completamente novo: a *proxy*.

8,2 Usando proxies para controlar o acesso

UMA *procuração* é um substituto por meio do qual controlamos o acesso a outro objeto. Ele nos permite definir ações personalizadas que serão executadas quando um objeto está sendo interagido - por exemplo, quando um valor de propriedade é lido ou definido, ou quando um método é chamado. Você pode pensar em proxies como quase uma generalização de getters e setters; mas com cada getter e setter, você controla o acesso a apenas uma única propriedade do objeto, enquanto os proxies permitem que você manipule genericamente todas as interações com um objeto, incluindo até mesmo chamadas de método.

Podemos usar proxies quando tradicionalmente usamos getters e setters, como para registro, validação de dados e propriedades computadas. Mas os proxies são ainda mais poderosos. Eles nos permitem adicionar facilmente perfis e medições de desempenho ao nosso código, preencher automaticamente as propriedades do objeto para evitar exceções nulas incômodas e agrupar objetos de host, como o DOM, a fim de reduzir incompatibilidades entre navegadores.

NOTA Os proxies são introduzidos pelo ES6. Para suporte de navegador atual, consulte http://mng.bz/9uEM .

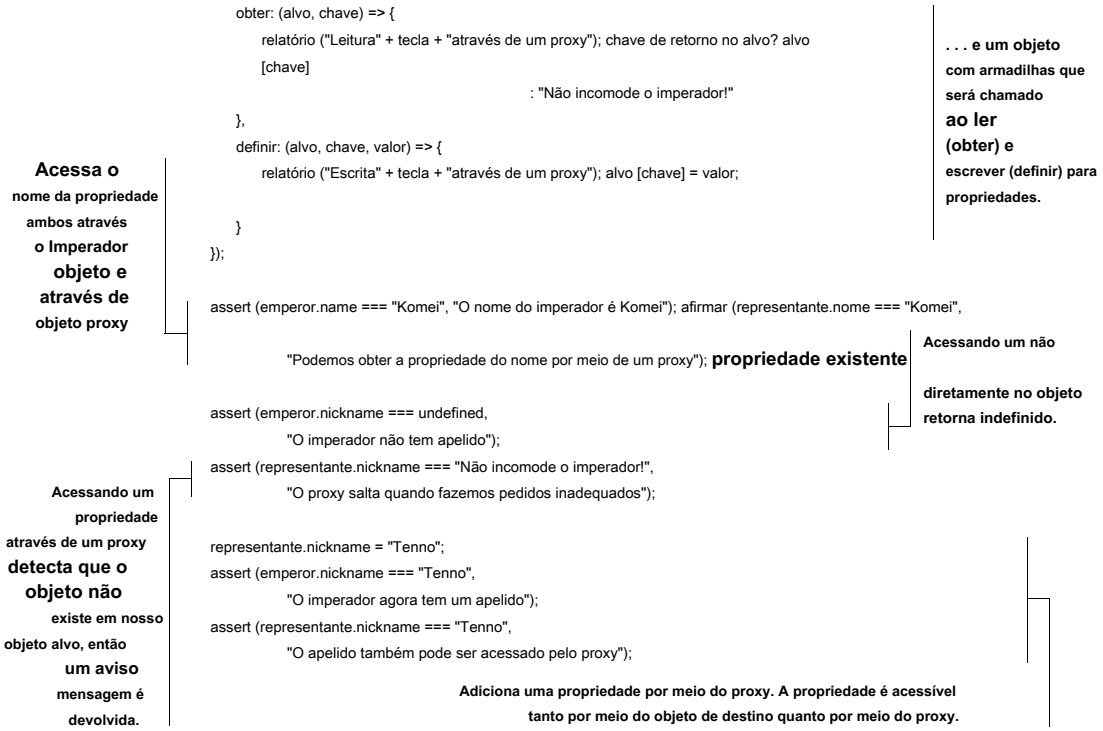
Em JavaScript, podemos criar proxies usando o built-in Proxy construtor. Vamos começar de forma simples, com um proxy que intercepta todas as tentativas de leitura e gravação nas propriedades de um objeto.

Listagem 8.7 Criando proxies com o **Proxy** constructor

O imperador é nosso objeto alvo.

```
imperador const = {nome: "Komei"}; representante const = novo Proxy (imperador,
{
```

Cria um proxy com o construtor Proxy que recebe o objeto que o proxy envolve ...



Primeiro criamos nossa base imperador objeto que tem apenas um nome propriedade. Em seguida, usando o integrado Proxy construtor, nós envolvemos nosso imperador objeto (ou *alvo* objeto, como é comumente chamado) em um objeto proxy denominado representante. Durante a construção do proxy, como segundo argumento, também enviamos um objeto que especifica *armadilhas,* funções que serão chamadas quando certas ações forem realizadas em um objeto:

```
representante const = novo Proxy (imperador, {
    obter: (alvo, chave) => {
        relatório ("Leitura" + tecla + "através de um proxy"); chave de retorno no alvo? alvo
        [chave]
                                                : "Não incomode o imperador!"
    },
    definir: (alvo, chave, valor) => {
        relatório ("Escrita" + tecla + "através de um proxy"); alvo [chave] = valor;

    }
});
```

Nesse caso, especificamos duas armadilhas: a obter armadilha que será chamada sempre que tentarmos ler um valor de uma propriedade por meio do proxy, e um definir trap que será chamado sempre que definirmos um valor de propriedade por meio do proxy. O *obter* trap executa a seguinte funcionalidade: Se o objeto de destino tiver uma propriedade, essa propriedade será retornada; e se o

objeto não tem uma propriedade, retornamos uma mensagem avisando nosso usuário para não incomodar o imperador com detalhes frívolos.

```
obter: (alvo, chave) => {
    relatório ("Leitura" + tecla + "através de um proxy"); chave de retorno no alvo? alvo
    [chave]
                                                : "Não incomode o imperador!"
}
```

Em seguida, testamos se podemos acessar o nome propriedade tanto diretamente através do alvo imperador objeto, bem como por meio de nosso objeto proxy:

```
assert (emperor.name === "Komei", "O nome do imperador é Komei"); afirmar (representante.nome === "Komei",

        "Podemos obter a propriedade do nome por meio de um proxy");
```

Se acessarmos o nome propriedade diretamente através do imperador objeto, o valor Komei é devolvido. Mas se acessarmos o nome propriedade através do procuração objeto, o obter trap é implicitamente chamado. Porque o nome propriedade é encontrada no alvo imperador objeto, o valor Komei também é retornado. Veja a figura 8.4.

> NOTA É importante enfatizar que as armadilhas de proxy são ativadas da mesma maneira que getters e setters. Assim que executamos uma ação (por exemplo, acessar um valor de propriedade em um proxy), a armadilha correspondente é chamada implicitamente e o mecanismo JavaScript passa por um processo semelhante, como se tivéssemos chamado explicitamente uma função.

Por outro lado, se acessarmos um não existente apelido propriedade diretamente no alvo imperador objeto, obteremos, sem surpresa, um Indefinido valor. Mas se tentarmos acessá-lo por meio de nosso procuração objeto, o obter manipulador será ativado. Porque o alvo

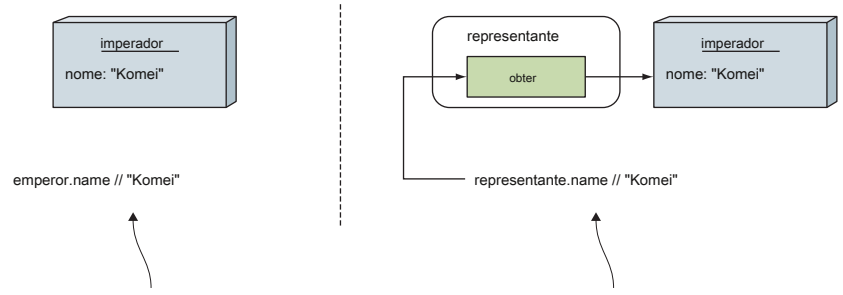


Figura 8.4 Acessando o **nome** propriedade direta (à esquerda) e indiretamente, por meio de um proxy (à direita)

imperador objeto não tem um apelido propriedade, o proxy obter armadilha irá retornar o
Não incomode o imperador! mensagem.

Continuaremos o exemplo atribuindo uma nova propriedade por meio de nosso objeto proxy: representante.nickname = "Tenno". Como a atribuição é feita por meio de um proxy, e não diretamente, o definir trap, que registra uma mensagem e atribui uma propriedade ao nosso alvo imperador objeto, está ativado:

```
definir: (alvo, chave, valor) => {
        relatório ("Escrita" + tecla + "através de um proxy"); alvo [chave] = valor;

}
```

Naturalmente, a propriedade recém-criada pode ser acessada por meio do objeto proxy e do objeto de destino:

```
assert (emperor.nickname === "Tenno",
            "O imperador agora tem um apelido");
assert (representante.nickname === "Tenno",
            "O apelido também pode ser acessado pelo proxy");
```

Esta é a essência de como usar proxies: por meio do Proxy construtor, criamos um objeto proxy que controla o acesso ao objeto de destino ativando certas armadilhas, sempre que uma operação é realizada diretamente em um proxy.

Neste exemplo, usamos o obter e definir armadilhas, mas muitas outras armadilhas integradas nos permitem definir manipuladores para várias ações de objeto (consulte http://mng.bz/ba55) Por exemplo:

- O *Aplique* trap será ativado ao chamar uma função, e o *construir* armadilha ao usar o novo operador.
- O *obter* e *definir* armadilhas serão ativadas ao ler / gravar em uma propriedade. O *enumerar* armadilha
- será ativada para para dentro declarações.
- getPrototypeOf e setPrototypeOf será ativado para obter e definir o valor do protótipo.

Podemos interceptar muitas operações, mas passar por todas elas está fora do escopo deste livro. Por enquanto, voltamos nossa atenção para algumas operações que não podemos substituir: igualdade (== ou ===), instancia de, e a tipo de operador.

Por exemplo, a expressão x == y (ou um mais estrito x === y) é usado para verificar se
x e y referem-se a objetos idênticos (ou têm o mesmo valor). Este operador de igualdade tem algumas suposições. Por exemplo, comparar dois objetos deve sempre retornar o mesmo valor para os mesmos dois objetos, o que não é algo que possamos garantir se esse valor for determinado por uma função especificada pelo usuário. Além disso, o ato de comparar dois objetos não deve dar acesso a um desses objetos, o que seria o caso se a igualdade pudesse ser interceptada. Por razões semelhantes, o instancia de e a tipo de os operadores não podem ser presos.

Agora que sabemos como os proxies funcionam e como criá-los, vamos explorar alguns de seus aspectos práticos, como usar proxies para registro, medição de desempenho, propriedades de preenchimento automático e implementação de matrizes que podem ser acessadas com índices negativos. Começaremos com o registro.

8.2.1 *Usando proxies para registro*

Uma das ferramentas mais poderosas ao tentar descobrir como o código funciona ou ao tentar chegar à raiz de um bug desagradável é *exploração madeireira,* o ato de enviar informações que consideramos úteis em um determinado momento. Podemos, por exemplo, querer saber quais funções são chamadas, há quanto tempo estão em execução, quais propriedades são lidas ou gravadas e assim por diante.

Infelizmente, ao implementar o log, geralmente espalhamos as instruções de log por todo o código. Dê uma olhada no Ninja exemplo usado anteriormente neste capítulo.

Listagem 8.8 Log sem proxies

```
function Ninja () {
        deixe _skillLevel = 0;

        Object.defineProperty (this, 'skillLevel', {
            obter: () => {
                report ("o método de obtenção skillLevel é chamado"); return _skillLevel;

            },
            definir: valor => {
                report ("o método de conjunto skillLevel é chamado"); _skillLevel = value;

            }
        });
    }

    const ninja = novo Ninja (); ninja.skillLevel;

    ninja.skillLevel = 4;
```

Registramos sempre que o
A propriedade skillLevel foi lida ...

· sempre que a propriedade
skillLevel é gravada.

Lê o skillLevel
propriedade e gatilhos
o método get

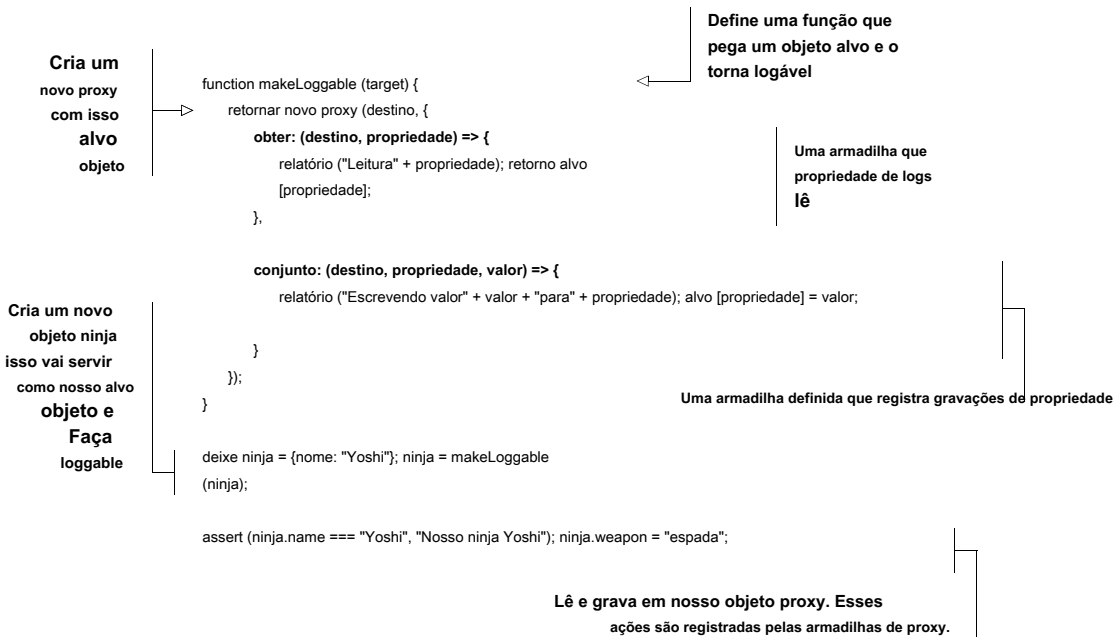
Grava na propriedade skillLevel e
aciona o método set

Nós definimos um Ninja função construtora que adiciona um getter e um setter ao nível de habilidade , que registra todas as tentativas de leitura e gravação nessa propriedade.

Observe que esta não é uma solução ideal. Desordenamos nosso código de domínio que lida com a leitura e gravação de uma propriedade de objeto com código de registro. Além disso, se no futuro precisarmos de mais propriedades no ninja objeto, temos que ter cuidado para não esquecer de adicionar instruções de registro adicionais a cada nova propriedade.

Felizmente, um dos usos diretos dos proxies é habilitar o registro sempre que lermos ou escrevermos em uma propriedade, mas de uma maneira muito mais agradável e limpa. Considere o seguinte exemplo.

Listagem 8.9 O uso de proxies torna mais fácil adicionar registro a objetos



Aqui nós definimos um makeLoggable função que leva um alvo objeto e retorna um novo Proxy que tem um manipulador com um obter e um definir armadilha. Essas armadilhas, além de ler e gravar na propriedade, registram as informações sobre a propriedade lida ou gravada.

Em seguida, criamos um ninja objeto com um nome propriedade, e passamos para o makeLoggable , na qual será usado como destino para um proxy recém-criado. Em seguida, atribuímos o proxy de volta ao ninja identificador, substituindo-o. (Não se preocupe, nosso original ninja objeto é mantido ativo como o objeto de destino do nosso proxy.)

Sempre que tentamos ler uma propriedade (por exemplo, com ninja.name), a obter trap será chamado e as informações sobre qual propriedade foi lida serão registradas. Algo semelhante acontecerá ao gravar em uma propriedade: ninja.weapon = "espada".

Observe como isso é muito mais fácil e transparente quando comparado à maneira padrão de usar getters e setters. Não precisamos misturar nosso código de domínio com nosso código de registro e não há necessidade de adicionar registros separados para cada propriedade de objeto. Em vez disso, todas as leituras e gravações de propriedades passam por nossos métodos de armadilha de objeto proxy. O registro foi especificado em apenas um lugar e é reutilizado quantas vezes forem necessárias, em quantos objetos forem necessários.

8.2.2 Usando proxies para medir o desempenho

Além de ser usado para registrar acessos de propriedade, os proxies podem ser usados para medir o desempenho de invocações de função, sem mesmo modificar o código-fonte de um

função. Digamos que queremos medir o desempenho de uma função que calcula se um número é primo, conforme mostrado na lista a seguir.

Listagem 8.10 Medindo o desempenho com proxies

```
function isPrime (número) {
    if (número <2) {return false; }

    para (deixe i = 2; i <número; i ++) {
        if (número% i === 0) {return false; }
    }

    return true;
}

isPrime = new Proxy (isPrime, {
    aplicar: (destino, esteArg, args) => {
        console.time (" isPrime ");

        const result = target.apply (thisArg, args);

        console.timeEnd (" isPrime ");

        resultado de retorno;
    }
});

isPrime (1299827);
```

Define um primitivo implementação do função isPrime

Envolve o isPrime função dentro de um procuração

Fornece uma armadilha de aplicação que será chamada sempre que um proxy for chamado como uma função

Inicia um cronômetro chamado isPrime

Invoca o alvo função

Para o cronômetro e produz o resultado

Chama a função. A chamada funciona da mesma forma como se tivéssemos chamado a função original.

Neste exemplo, temos um simples isPrime função. (A função exata não importa; estamos usando-a como um exemplo de uma função cuja execução pode durar um período de tempo não trivial.)

Agora imagine que precisamos medir o desempenho do isPrime função, mas sem modificar seu código. Poderíamos envolver a função em um proxy que tenha uma armadilha que será chamada sempre que a função for chamada:

```
isPrime = new Proxy (isPrime, {
    aplicar: (destino, esteArg, args) => {
. . .
    }
});
```

Nós usamos o isPrime functionas o objeto de destino de um proxy recém-construído. Além disso, fornecemos um manipulador com um Aplique trap que será executado na invocação da função.

Da mesma forma, como no exemplo anterior, atribuímos o proxy recém-criado ao isPrime identificador. Dessa forma, não temos que alterar nenhum código que chama a função cujo tempo de execução queremos medir; o resto do código do programa é completamente alheio às nossas mudanças. (O que acha disso para uma ação furtiva ninja?)

Sempre que isPrime função é chamada, essa chamada é redirecionada para o nosso proxy Aplique armadilha, que iniciará um cronômetro com o console.time método (lembre-se do capítulo 1), chame o original isPrime função, registre o tempo decorrido e, finalmente, retorne o resultado do isPrime invocação.

8.2.3 Usando proxies para preencher automaticamente as propriedades

Além de simplificar o registro, os proxies podem ser usados para preencher automaticamente as propriedades. Por exemplo, imagine que você precise modelar a estrutura de pastas do seu computador, na qual um objeto de pasta pode ter propriedades que também podem ser pastas. Agora imagine que você precise modelar um arquivo no final de um longo caminho, como este:

```
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Para criar isso, você pode escrever algo com as seguintes linhas:

```
const rootFolder = nova pasta (); rootFolder.ninjasDir = nova pasta ();

rootFolder.ninjasDir.firstNinjaDir = nova pasta (); rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Parece um pouco mais tedioso do que o necessário, não é? É aqui que as propriedades de preenchimento automático entram em jogo; basta dar uma olhada no exemplo a seguir.

Listagem 8.11 Preenchendo automaticamente propriedades com proxies

```
function Folder () {
    retornar novo Proxy ({}, {
        obter: (destino, propriedade) => {
            relatório ("Leitura" + propriedade);

            if (! (propriedade no destino)) {
                destino [propriedade] = nova pasta ();
            }

            retorno alvo [propriedade];
        }
    });
}

const rootFolder = nova pasta ();

tentar {
    rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt"; pass ("Uma exceção não foi levantada");

}
pegar (e) {
    fail ("Ocorreu uma exceção");
}
```

Registra todas as leituras para o nosso objeto

Se o acessado propriedade não existir, nós o criamos.

Sempre que uma propriedade é acessada, o get trap, que cria uma propriedade se não existe, está ativado.

Nenhuma exceção será levantada.

Normalmente, se considerarmos apenas o código a seguir, esperaríamos que uma exceção fosse levantada:

```
const rootFolder = nova pasta ();
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Estamos acessando uma propriedade, firstNinjaDir, de uma propriedade indefinida, ninjasDir, do pasta raiz objeto. Mas se executarmos o código, você verá que está tudo bem, como mostra a figura 8.5.

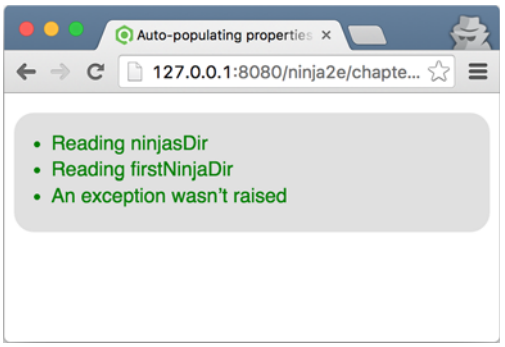


Figura 8.5 A saída da execução do código da listagem 8.11

Isso acontece porque estamos usando proxies. Cada vez que acessamos uma propriedade, o proxy obter armadilha é ativada. Se nosso objeto de pasta já contém a propriedade solicitada, seu valor é retornado e, caso não contenha, uma nova pasta é criada e atribuída à propriedade. É assim que duas de nossas propriedades, ninjasDir e firstNinjaDir, são criados. Solicitando um valor de uma propriedade não inicializada aciona sua criação.

Finalmente, temos uma ferramenta para nos livrar de alguns casos da incômoda exceção nula!

8.2.4 Usando proxies para implementar índices de array negativos

Em nossa programação do dia a dia, geralmente trabalhamos com *muito* de matrizes. Vamos explorar como tirar proveito dos proxies para tornar o tratamento com arrays um pouco mais agradável.

Se o seu histórico de programação vem de linguagens como Python, Ruby ou Perl, você pode estar acostumado com índices de array negativos, que permitem usar índices negativos para acessar itens de array na parte de trás, conforme mostrado no seguinte snippet:

```
const ninjas = ["Yoshi", "Kuma", "Hattori"];

ninjas [0]; // "Yoshi"
ninjas [1]; // "Kuma"
ninjas [2]; // "Hattori"

ninjas [-1]; // "Hattori"
ninjas [-2]; // "Kuma"
ninjas [-3]; // "Yoshi"
```

Acesso padrão a itens da matriz, com índices de matriz

Os índices de array negativos nos permitem acessar itens de array de trás, começando com –1, que acessa o último item de array.

Agora compare o código que normalmente usamos para acessar o último item da matriz, ninjas [ninjas.length-1], com o código que podemos usar se a nossa linguagem de escolha suportar índices de array negativos, ninjas [-1]. Veja como isso é muito mais elegante?

Infelizmente, o JavaScript não oferece suporte integrado para índices de array negativos, mas podemos imitar essa capacidade por meio de proxies. Para explorar esse conceito, veremos um

versão ligeiramente simplificada do código escrito por Sindre Sorhus (https://github.com/ sindresorhus / negative-array), conforme mostrado na lista a seguir.

Listagem 8.12 Índices de array negativos com proxies

```
function createNegativeArrayProxy (array) {
    if (! Array.isArray (array)) {
        lançar novo TypeError ('Esperado um array');
    }

    retornar novo Proxy (array, {
        obter: (destino, índice) => {
            índice = + índice;
            objetivo de retorno [índice <0? comprimento.alvo + índice: índice];
        },
        conjunto: (destino, índice, val) => {índice = + índice;

          objetivo de retorno [índice <0? comprimento.alvo + índice: índice] = val;
        }
    });
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy (ninjas);

afirmar (ninjas [0] === "Yoshi" && ninjas [1] === "Kuma"
        && ninjas [2] === "Hattori",
            "Itens da matriz acessados   por meio de índices positivos");

assert (proxiedNinjas [0] === "Yoshi" && proxiedNinjas [1] === "Kuma"
        && proxiedNinjas [2] === "Hattori",
            "Itens da matriz acessados   por meio de índices positivos em um proxy");

assert (typeof ninjas [-1] === "undefined"
        && typeof ninjas [-2] === "undefined" && typeof ninjas [-3] ===
        "undefined",
            "Os itens não podem ser acessados   por meio de índices negativos em uma matriz");

assert (proxiedNinjas [-1] === "Hattori"
        && proxiedNinjas [-2] === "Kuma" && proxiedNinjas [-3] ===
        "Yoshi",
            “Mas eles podem ser acessados   através de índices negativos”);

proxiedNinjas [-1] = "Hachi";
assert (proxiedNinjas [-1] === "Hachi" && ninjas [2] === "Hachi",
            “Os itens podem ser alterados através de índices negativos”);
```

Retorna um novo proxy que leva em a matriz e usa isso como um proxy alvo

Se nosso objeto de destino não for um array, lance uma exceção.

A armadilha é ativado sempre que um o índice da matriz é lido.

Vira a propriedade nome em um número com o operador mais unário

Se o índice de leitura é um negativo numero, leia da parte de trás da matriz, e se é um positivo número, acesse normalmente.

O set trap é ativado sempre que um índice de array é gravado.

Cria uma matriz padrão

Verifica se podemos acessar os itens da matriz através do original array, bem como através do proxy

Passa para nossa função isso vai criar um proxy para isso variedade

Verifica se não podemos acessar itens da matriz por meio de índices negativos em uma matriz padrão ...

. . . mas podemos fazer isso por meio de nosso proxy, porque fornecemos uma armadilha get que trata do caso.

Também podemos modificar os itens da matriz na parte de trás, mas apenas se passarmos pelo proxy.

Neste exemplo, definimos uma função que criará um proxy para um array passado. Como não queremos que nosso proxy funcione com outros tipos de objetos, lançamos uma exceção caso o argumento não seja um array:

```
if (! Array.isArray (array)) {
        lançar novo TypeError ('Esperado um array');
}
```

Continuamos criando e retornando um novo proxy com duas armadilhas, um obter armadilha que será ativada sempre que tentarmos ler um item do array, e um definir armadilha que será ativada sempre que escrevermos em um item da matriz:

```
retornar novo Proxy (array, {
    obter: (destino, índice) => {
        índice = + índice;
        objetivo de retorno [índice <0? comprimento.alvo + índice: índice];
    },
    conjunto: (alvo, índice, val) => {
        índice = + índice;
        objetivo de retorno [índice <0? comprimento.alvo + índice: índice] = val;
    }
});
```

Os corpos das armadilhas são semelhantes. Primeiro, transformamos a propriedade em um número usando o operador unário mais (índice = + índice). Então, se o índice solicitado for menor que 0, acessamos os itens da matriz na parte de trás, ancorando ao comprimento da matriz, e se for maior ou igual a 0, acessamos o item do array de uma maneira padrão.

Por fim, realizamos vários testes para verificar se em arrays normais só podemos acessar itens de array por meio de índices de array positivos e se, se usarmos um proxy, podemos acessar e modificar itens de array por meio de índices negativos.

Agora que você viu como usar proxies para obter alguns recursos interessantes, como autopreencher propriedades de objetos e acessar índices de array negativos, que são totalmente impossíveis sem proxies, vamos explorar a desvantagem mais significativa dos proxies: problemas de desempenho.

8.2.5 *Custos de desempenho de proxies*

Como já sabemos, um proxy é um objeto substituto por meio do qual controlamos o acesso a outro objeto. Um proxy pode definir traps, funções que serão executadas sempre que uma determinada operação for realizada em um proxy. E, como você também viu, podemos usar essas armadilhas para implementar funcionalidades úteis, como registro, medições de desempenho, propriedades de preenchimento automático, índices de array negativos e assim por diante. Infelizmente, também há uma desvantagem. O fato de todas as nossas operações terem que passar pelo proxy adiciona uma camada de indireção que nos permite implementar todos esses recursos interessantes, mas ao mesmo tempo introduz uma quantidade significativa de processamento adicional que afeta o desempenho.

Para testar esses problemas de desempenho, podemos construir sobre o exemplo de índices de array negativos da listagem 8.12 e comparar o tempo de execução ao acessar itens em um array normal versus acessar itens por meio de um proxy, conforme mostrado na listagem a seguir.

Listagem 8.13 Verificando as limitações de desempenho de proxies

```
medida de função (itens) {
    const startTime = new Date (). getTime (); para (seja i = 0; i <500000; i ++) {
        itens [0] === "Yoshi";
        itens [1] === "Kuma";
        itens [2] === "Hattori";
    }
    retorna novo Date (). getTime () - startTime;
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy (ninjas);

console.log ("Proxies estão por aí",
                Math.round (medir (proxiedNinjas) / medir (ninjas)),
                "vezes mais lento");
```

Acessa os itens em nossa coleção em um ciclo de longa duração

Obtém a hora atual antes de executar uma operação de longa duração

Mede o tempo que demorou por muito tempo código em execução executar

Compara o tempo de execução ao acessar o padrão array versus ao acessar através de um proxy

Cria uma matriz padrão e um proxy para essa matriz

Como uma única operação do código acontece muito rapidamente para ser medida de maneira confiável, o código deve ser executado muitas vezes para obter um valor mensurável. Freqüentemente, essa contagem pode chegar a dezenas de milhares, ou mesmo milhões, dependendo da natureza do código que está sendo medido. Uma pequena tentativa e erro nos permite escolher um valor razoável: neste caso, 500.000.

Também precisamos agrupar a execução do código com dois new Date (). getTime () timestamps: um antes de começarmos a executar o código de destino e um depois. A diferença deles nos diz quanto tempo o código levou para ser executado. Finalmente, podemos comparar os resultados chamando o medir função na matriz com proxy e na matriz padrão.

Em nossa máquina, os resultados não são bons para proxies. Acontece que, no Chrome, os proxies são cerca de 50 vezes mais lentos; no Firefox, eles são cerca de 20 vezes mais lentos.

Por enquanto, recomendamos que você tome cuidado ao usar proxies. Embora eles permitam que você seja criativo ao controlar o acesso aos objetos, essa quantidade de controle vem com problemas de desempenho. Você pode usar proxies com código que não é sensível ao desempenho, mas tenha cuidado ao usá-los em código que é executado muito. Como sempre, recomendamos que você teste exaustivamente o desempenho do seu código.

8,3 Resumo

- Podemos monitorar nossos objetos com getters, setters e proxies.
- Usando métodos de acesso (getters e setters), podemos controlar o acesso às propriedades do objeto.

- As propriedades do acessador podem ser definidas usando o Object.defineProperty método ou com um especial obter e definir sintaxe como partes de literais de objeto ou classes ES6.

- UMA obter método é implicitamente chamado sempre que tentamos ler, enquanto um definir método é chamado sempre que atribuímos um valor à propriedade do objeto correspondente. Os métodos
- getter podem ser usados para definir propriedades computadas, propriedades cujo valor é calculado por solicitação, enquanto os métodos setter podem ser usados para obter validação de dados e registro.

- Proxies são uma adição ES6 ao JavaScript e são usados para controlar outros objetos.
  - Os proxies nos permitem definir as opções que serão executadas quando um objeto for interagido com (por exemplo, quando uma propriedade é lida ou uma função é chamada).
  - Todas as interações devem passar pelo proxy, que possui armadilhas que são acionadas quando ocorre uma ação específica.
- Use proxies para obter elegância
  - Exp
  - Medidas de desempenho
  - Data de validade
  - Preenchimento automático de propriedades de objetos (evitando assim exceções nulas incômodas)
  - índices de matriz negativos
- Os proxies não são rápidos, então tome cuidado ao usá-los em códigos que são muito executados. Recomendamos que você faça testes de desempenho.

8,4 Exercícios

1 Depois de executar o código a seguir, qual das seguintes expressões lançará uma exceção e por quê?

```
const ninja = {
        obter nome () {
            retornar "Akiyama";
        }
}
```

uma ninja.name ();
b nome const = ninja.name;

2 No código a seguir, qual mecanismo permite que os getters acessem um objeto privado variável?

```
function Samurai () {
    const _weapon = "katana";
    Object.defineProperty (this, "weapon", {
        obter: () => _weapon});

}
Const samurai = novo Samurai ();
assert (samurai.weapon === "katana", "Um samurai empunhando uma katana");
```

3 Qual das afirmações a seguir será aprovada?

```
const daimyo = {nome: "Matsu", clã: "Takasu"}; const proxy = new Proxy (daimyo, {

    obter: (alvo, chave) => {
        if (chave === "clã") {
            retornar "Tokugawa";
        }
    }
});

assert (daimyo.clan === "Takasu", "Matsu do clã Takasu"); assert (proxy.clan === "Tokugawa", "Matsu do clã
Tokugawa?");

proxy.clan = "Tokugawa";

assert (daimyo.clan === "Takasu", "Matsu do clã Takasu"); assert (proxy.clan === "Tokugawa", "Matsu do clã
Tokugawa?");
```

4 Qual das afirmações a seguir será aprovada?

```
const daimyo = {nome: "Matsu", clã: "Takasu", armySize: 10000}; const proxy = new Proxy (daimyo, {

    definir: (alvo, chave, valor) => {
        if (key === "armySize") {
            número const = Number.parseInt (valor); if (! Number.isNaN (number)) {

                alvo [chave] = número;
            }
        } outro {
            alvo [chave] = valor;
        }
    },
});

assert (daimyo.armySize === 10000, "Matsu tem 10.000 homens armados"); assert (proxy.armySize === 10000, "Matsu tem
10.000 homens armados");

proxy.armySize = "grande";
assert (daimyo.armySize === "grande", "Matsu tem um grande exército");

daimyo.armySize = "grande";
assert (daimyo.armySize === "grande", "Matsu tem um grande exército");
```

9 Lidando com coleções

Este capítulo cobre

- Criação e modificação de matrizes
- Usando e reutilizando funções de array
- Criação de dicionários com mapas
- Criação de coleções de objetos únicos com conjuntos

Agora que passamos algum tempo discutindo as particularidades da orientação a objetos em JavaScript, passaremos para um tópico intimamente relacionado: coleções de itens. Começaremos com matrizes, o tipo mais básico de coleção em JavaScript, e examinaremos algumas peculiaridades de matrizes que você não pode esperar se sua experiência em programação estiver em outra linguagem de programação. Continuaremos explorando alguns dos métodos de array integrados que o ajudarão a escrever um código de manipulação de array mais elegante.

A seguir, discutiremos duas novas coleções ES6: mapas e conjuntos. Usando mapas, você pode criar dicionários de um tipo que carregam mapeamentos entre chaves e valores - uma coleção que é extremamente útil em certas tarefas de programação. Conjuntos, por outro lado, são coleções de itens exclusivos em que cada item não pode ocorrer mais de uma vez. Vamos começar nossa exploração com a mais simples e mais comum de todas as coleções: arrays.

Você sabe?

Quais são algumas das armadilhas comuns de usar objetos como dicionários ou mapas?

Quais tipos de valor um par chave / valor pode ter em um Mapa ?

Os itens devem estar em um Definir ser do mesmo tipo?

9,1 Arrays

Arrays são um dos tipos de dados mais comuns. Usando-os, você pode lidar com coleções de itens. Se o seu histórico de programação está em uma linguagem fortemente tipada como C, você provavelmente pensa em matrizes como pedaços sequenciais de memória que hospedam itens do mesmo tipo, onde cada pedaço de memória é de tamanho fixo e tem um índice associado através do qual você pode acessá-lo facilmente.

Mas, como acontece com muitas coisas em JavaScript, os arrays têm uma peculiaridade: eles são apenas objetos. Embora isso leve a alguns efeitos colaterais desagradáveis, principalmente em termos de desempenho, também tem alguns benefícios. Por exemplo, arrays podem acessar métodos, como outros objetos - métodos que tornarão nossas vidas muito mais fáceis.

Nesta seção, veremos primeiro as maneiras de criar arrays. Em seguida, exploraremos como adicionar e remover itens de diferentes posições em um array. Finalmente, examinaremos os métodos de array integrados que tornarão nosso código de manipulação de array muito mais elegante.

9.1.1 Criação de matrizes

Existem duas maneiras fundamentais de criar novos arrays:

- Usando o integrado Variedade constructor
- Usando literais de matriz []

Vamos começar com um exemplo simples em que nós crie uma série de ninjas e uma série de samurais.

Listagem 9.1 Criando arrays

Para criar uma matriz, podemos usar um literal de matriz [] ...

... ou o Array embutido constructor.

```
const ninjas = ["Kuma", "Hattori", "Yagyu"]; const samurai = novo Array ("Oda",
"Tomoe");

assert (ninjas.length === 3, "Existem três ninjas"); assert (samurai.length === 2, "E apenas dois
samurais");

assert (ninjas [0] === "Kuma", "Kuma é o primeiro ninja"); assert (samurai [samurai.length-1] === "Tomoe",
                "Tomoe é o último samurai");

afirmar (ninjas [4] === indefinido,
            "Ficamos indefinidos se tentarmos acessar um índice fora dos limites");
```

Nós acessamos a matriz itens com índice notação: o primeiro o item está indexado com 0, e o último com array.length - 1.

O comprimento propriedade diz nós do tamanho de a matriz.

Lendo itens fora da matriz limites resulta em Indefinido.

```
ninjas [4] = "Ishi";
assert (ninjas.length === 5,
        "As matrizes são expandidas automaticamente");

ninjas.length = 2;
assert (ninjas.length === 2, "Existem apenas dois ninjas agora"); assert (ninjas [0] === "Kuma" && ninjas [1] ===
"Hattori",
        "Kuma e Hattori");
assert (ninjas [2] === undefined, "Mas nós perdemos Yagyu");
```

Gravar em índices fora dos limites do array estende o array.

Substituir manualmente a propriedade de comprimento por um valor inferior exclui os itens em excesso.

Na Listagem 9.1, começamos criando dois arrays. O ninjas array é criado com um literal de array simples:

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
```

É imediatamente preenchido com três ninjas: Kuma, Hattori e Yagyu. O samurai array é criado usando o Variedade construtor:

```
const samurai = novo Array ("Oda", "Tomoe");
```

Literais de matriz versus o construtor de matriz

Usar literais de array para criar arrays é preferível em vez de criar arrays com o Variedade construtor. O principal motivo é a simplicidade: [] versus novo Array () (2 personagens contra 11 personagens - dificilmente um concurso justo). Além disso, como o JavaScript é altamente dinâmico, nada impede que alguém substitua o código integrado Variedade construtor, o que significa chamar novo Array () não precisa necessariamente criar um array. Portanto, recomendamos que você geralmente se atenha aos literais de array.

Independentemente de como o criamos, cada array tem um comprimento propriedade que especifica o tamanho da matriz. Por exemplo, o comprimento do ninjas array é 3, e contém 3 ninjas. Podemos testar isso com as seguintes afirmações:

```
assert (ninjas.length === 3, "Existem três ninjas"); assert (samurai.length === 2, "E apenas dois
samurais");
```

Como você provavelmente sabe, você acessa itens de matriz usando notação de índice, onde o primeiro item é posicionado no índice 0 e o último item no índice array.length - 1. Mas se tentarmos acessar um índice fora desses limites - por exemplo, com ninjas [4] (lembre-se de que temos apenas três ninjas!), não obteremos a assustadora exceção “Array index out of bounds” que recebemos na maioria das outras linguagens de programação. Em vez de, Indefinido é retornado, sinalizando que não há nada lá:

```
afirmar (ninjas [4] === indefinido,
        "Ficamos indefinidos se tentarmos acessar um índice fora dos limites");
```