escapar de nossa atenção (a menos que tenhamos bons testes), mas não há como errar no modo estrito. Este é um bom exemplo de por que o modo estrito foi introduzido.

Como os construtores são geralmente codificados e usados de maneira diferente de outras funções, e não são tão úteis a menos que sejam chamados como construtores, uma convenção de nomenclatura surgiu para distinguir construtores de funções e métodos comuns. Se você tem prestado atenção, talvez já tenha percebido.

Funções e métodos são geralmente nomeados começando com um verbo que descreve o que eles fazem (esquivar, rastejar, esgueirar-se, fazer algo maravilhoso, e assim por diante) e comece com uma letra minúscula. Os construtores, por outro lado, geralmente são nomeados como um substantivo que descreve o objeto que está sendo construído e começam com um caractere maiúsculo: Ninja, Samurai, Imperador, Ronin, e assim por diante.

É fácil ver como um construtor torna mais elegante criar vários objetos que estejam em conformidade com o mesmo padrão sem ter que repetir o mesmo código indefinidamente. O código comum é escrito apenas uma vez, como o corpo do construtor. No capítulo 7, você verá mais sobre como usar construtores e sobre os outros mecanismos orientados a objetos que o JavaScript fornece para tornar ainda mais fácil configurar padrões de objetos.

Mas não terminamos com as invocações de função ainda. Há ainda outra maneira de o JavaScript nos permitir invocar funções que fornecem um grande controle sobre os detalhes da invocação.

4.2.4 Invocação com os métodos apply e call

Até agora, você viu que uma das principais diferenças entre os tipos de invocação de função é o objeto que termina como o contexto de função referenciado pelo implícito esta parâmetro que é passado para a função em execução. Para métodos, é o objeto proprietário do método; para funções de nível superior, é janela ou Indefinido (dependendo do rigor atual); para construtores, é uma instância de objeto recém-criada.

Mas e se quisermos tornar o contexto da função o que quisermos? E se quisermos defini-lo explicitamente? E se ... bem, por que queremos fazer uma coisa dessas?

Para ter uma ideia de por que nos importamos com essa capacidade, veremos um exemplo prático que ilustra um bug surpreendentemente comum relacionado ao tratamento de eventos. Por enquanto, considere que, quando um manipulador de eventos é chamado, o contexto da função é definido para o objeto ao qual o evento foi vinculado. (Não se preocupe se isso parecer vago; você aprenderá sobre o tratamento de eventos em detalhes no capítulo 13.) Dê uma olhada na lista a seguir.

Listagem 4.10 Vinculando um contexto específico a uma função

Uma função construtora que cria objetos que retêm o estado em relação ao nosso botão. Com ele, rastrearemos se o botão foi clicado.

```
<button id = "test"> Clique aqui! </button>
<script>
    function Button () {
        this.clicked = false;
        this.click = function () {
```

Um elemento de botão ao qual atribuiremos um manipulador de eventos

Declara o método que usaremos como manipulador de cliques. Por ser um método do objeto, usamos isso dentro da função para obter um referência ao objeto.

```
        this.clicked = true;
        assert (button.clicked, "O botão foi clicado");
    };
}
botão var = novo Botão ();
var elem = document.getElementById ("teste"); elem.addEventListener ("click",
button.click);
</script>
```

Cria uma instância Isso rastreará se o botão foi clicado

Estabelece o manipulador de clique no botão

Dentro do método, testamos se o estado do botão tem foi corretamente alterado após um clique.

Neste exemplo, temos um botão, < button id = "test"> Clique aqui! </button>, e nós deseja saber se alguma vez foi clicado. Para reter essas informações de estado, usamos uma função construtora para criar um objeto de apoio denominado botão, no qual armazenaremos o estado clicado:

```
function Button () {
    this.clicked = false;
    this.click = function () {
        this.clicked = true;
        assert (button.clicked, "O botão foi clicado");
    };
}
botão var = novo Botão ();
```

Nesse objeto, também definimos um clique método que servirá como um manipulador de eventos que dispara quando o botão é clicado. O método define a propriedade clicada como verdade e então testa se o estado foi devidamente registrado no objeto de apoio (usamos intencionalmente o botão identificador em vez do esta palavra-chave - afinal, eles devem se referir à mesma coisa, ou não?). Finalmente, estabelecemos o button.click

método como um manipulador de cliques para o botão:

```
var elem = document.getElementById ("teste"); elem.addEventListener ("click",
button.click);
```

Quando carregamos o exemplo em um navegador e clicamos no botão, vemos pela exibição da figura 4.2 que algo está errado; o texto marcado indica que o teste falhou. O código na lista

4.10 falha porque o contexto do clique função *não é* referindo-se ao botão objeto como pretendíamos.

Relembrando as lições do início do capítulo, se tivéssemos chamado a função via button.click (), o contexto *seria* foi o botão, porque a função seria invocada como um método no

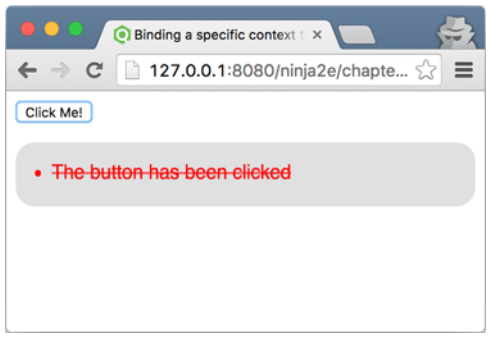


Figura 4.2 Por que nosso teste falhou? Para onde foi a mudança de estado? Normalmente, o contexto do retorno de chamada do evento é o objeto que gera o evento (neste caso, o botão HTML, e não o objeto do botão).

botão objeto. Mas, neste exemplo, o sistema de manipulação de eventos do navegador define o contexto da invocação como o elemento de destino do evento, o que faz com que o contexto seja **<** botão**>** elemento, não o botão objeto. Então, nós definimos nosso clique estado no objeto errado!

Este é um problema surpreendentemente comum e, posteriormente neste capítulo, você verá técnicas para evitá-lo completamente. Por enquanto, vamos explorar como lidar com isso examinando como definir explicitamente o contexto da função usando o Aplique e chamar métodos.

USANDO OS MÉTODOS DE APLICAÇÃO E CHAMADA

JavaScript fornece um meio para invocarmos uma função e especificarmos explicitamente qualquer objeto que desejamos como o contexto da função. Fazemos isso com o uso de um dos dois métodos existentes para cada função: Aplique e chamar.

Sim, dissemos métodos de funções. Como objetos de primeira classe (criados, aliás, pelo built-in Função construtor), as funções podem ter propriedades como qualquer outro tipo de objeto, incluindo métodos.

Para invocar uma função usando seu Aplique método, passamos dois parâmetros para Aplique: o objeto a ser usado como o contexto da função e uma matriz de valores a ser usada como os argumentos de invocação. O chamar método é usado de maneira semelhante, exceto que os argumentos são passados diretamente na lista de argumentos em vez de como uma matriz.

A lista a seguir mostra os dois métodos em ação.

Listagem 4.11 Usando o **Aplique** e **chamar** métodos para fornecer o contexto da função

```
function juggle () {
    resultado var = 0;
    para (var n = 0; n <argumentos.comprimento; n ++) {
        resultado + = argumentos [n];
    }
    this.result = resultado;
}

var ninja1 = {}; var ninja2 = {};

juggle.apply (ninja1, [1,2,3,4]);
juggle.call (ninja2, 5,6,7,8);

assert (ninja1.result === 10, "malabarismo via aplicar"); assert (ninja2.result === 26, "feito
malabarismo por meio de chamada");
```

A função "faz malabarismo" os argumentos e coloca o resultado em qualquer objeto que seja o contexto da função.

Esses objetos são inicialmente vazio e vai servir como nosso assuntos de teste.

Usa o método apply, passando ninja1 e uma série de argumentos

Usa a chamada método, passando ninja2 e uma lista de argumentos

Os testes mostram como o resultado do juggle é colocado nos objetos passados para os métodos.

Neste exemplo, configuramos uma função chamada malabarismo, em que definimos malabarismo como somar todos os argumentos e armazená-los como uma propriedade chamada resultado no contexto da função (referenciado pelo esta palavra-chave). Essa pode ser uma definição um tanto esfarrapada de malabarismo, mas *vontade* nos permitem determinar se os argumentos foram passados para a função corretamente e qual objeto acabou como o contexto da função.

Em seguida, configuramos dois objetos, ninja1 e ninja2, que usaremos como contextos de função, passando o primeiro para o Aplique método, junto com um *variedade* de argumentos, e passando o segundo para o chamar método, junto com um *Lista* de outros argumentos:

```
juggle.apply (ninja1, [1,2,3,4]);
juggle.call (ninja2, 5,6,7,8);
```

Observe que a única diferença entre Aplique e chamar é como os argumentos são fornecidos. No caso de Aplique, usamos uma matriz de argumentos e, no caso de chamar, nós os listamos como argumentos de chamada, após o contexto da função. Veja a figura 4.3.

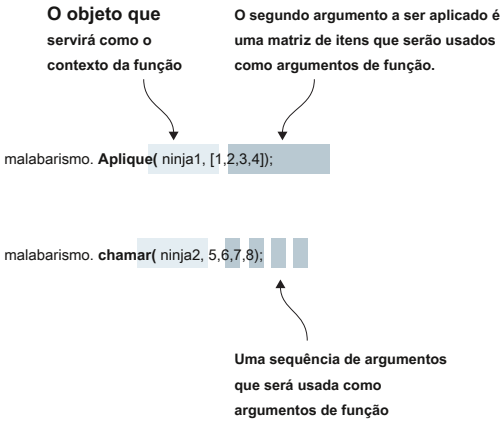


Figura 4.3 Como primeiro argumento, tanto o **chamar** e **Aplique** métodos pegam o objeto que será usado como o contexto da função. A diferença está nos seguintes argumentos.
Aplique leva apenas um argumento adicional, uma matriz de valores de argumento; **chamar** recebe qualquer número de argumentos, que serão usados como argumentos de função.

Depois de fornecer os contextos e argumentos de nossa função, continuamos testando! Primeiro, verificamos se ninja1, que foi chamado via Aplique, recebeu um resultado propriedade que é o resultado da adição de todos os valores de argumento (1, 2, 3, 4) na matriz passada. Então fazemos o mesmo para ninja2, que foi chamado via chamar, onde verificamos o resultado em busca de argumentos 5, 6, 7, e 8:

```
assert (ninja1.result === 10, "malabarismo via aplicar"); assert (ninja2.result === 26, "feito
malabarismo por meio de chamada");
```

A Figura 4.4 fornece uma visão mais detalhada do que está acontecendo na Listagem 4.11.

Esses dois métodos, chamar e Aplique, pode ser útil sempre que for conveniente usurpar o que normalmente seria o contexto da função com um objeto de nossa própria escolha - algo que pode ser particularmente útil ao invocar funções de retorno de chamada.

FORÇANDO O CONTEXTO DE FUNÇÃO EM CALLBACKS

Vamos considerar um exemplo concreto de forçar o contexto da função a ser um objeto de nossa própria escolha. Usaremos uma função simples para realizar uma operação em cada entrada de um array.

```
função juggle () {
    var resultado = 0;
    pra ( var n = 0; n <argumentos.comprimento; n ++) {
        resultado + = argumentos [n];
    }
    this.result = resultado;
}

var ninja1 = {};
var ninja2 = {};
```

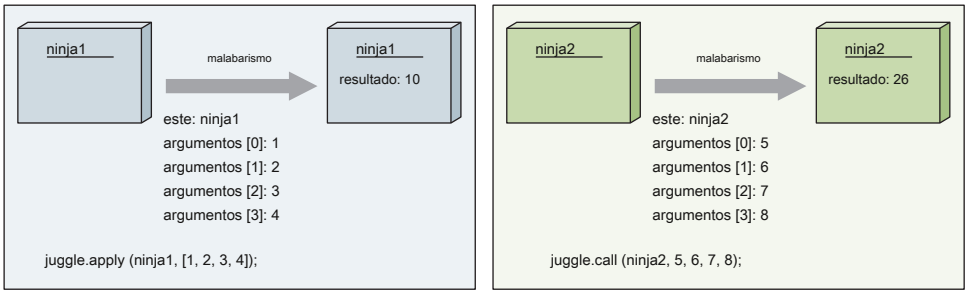


Figura 4.4 Configurando manualmente um contexto de função usando **chamar** e **Aplique** da listagem 4.11 resulta nessas combinações de contextos de função (o **esta** parâmetro) e **argumentos.**

Na programação imperativa, é comum passar o array para um método e usar um pra loop para iterar sobre cada entrada, realizando a operação em cada entrada:

```
função (coleção) {
    para (var n = 0; n <comprimento da coleção; n ++) {
        / * fazer algo para a coleção [n] * /
    }
}
```

Em contraste, a abordagem funcional é criar uma função que opera em um único elemento e passa cada entrada para essa função:

```
função (item) {
    / * fazer algo para o item * /
}
```

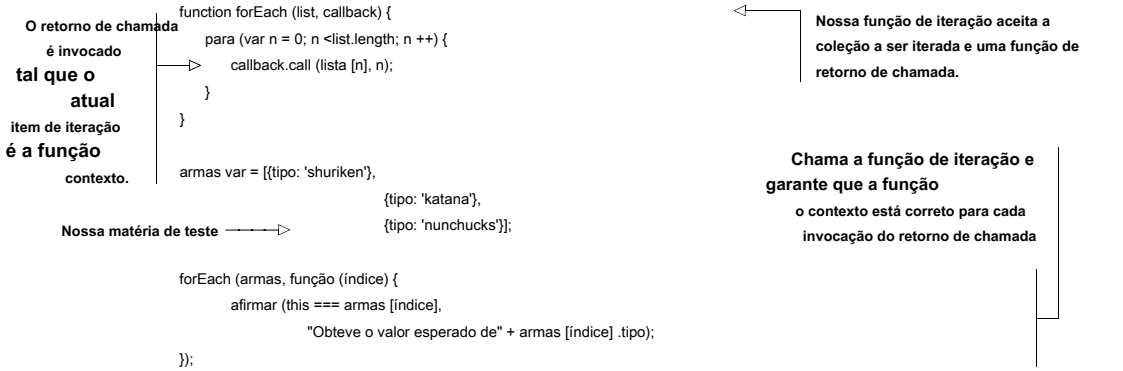
A diferença está em pensar em um nível em que as funções são os principais blocos de construção do programa. Você pode pensar que é discutível e que tudo o que você está fazendo é mover o pra loop out um nível, mas não terminamos de massagear este exemplo ainda.

Para facilitar um estilo mais funcional, todos os objetos de matriz têm acesso a um para cada função que invoca um retorno de chamada em cada elemento de uma matriz. Muitas vezes, é mais sucinto, e este estilo é preferido ao tradicional pra declaração por aqueles familiarizados com a programação funcional. Seus benefícios organizacionais se tornarão ainda mais evidentes (*tosse,* reutilização de código, *tosse)* depois de cobrir os fechamentos no capítulo 5. Tal iteração

função *poderia* passe o elemento atual para o retorno de chamada como um parâmetro, mas a maioria torna o elemento atual o contexto da função do retorno de chamada.

Embora todos os motores JavaScript modernos agora suportem um para cada em matrizes, construiremos nossa própria versão (simplificada) de tal função na próxima listagem.

Listagem 4.12 Construindo um **para cada** função para demonstrar a configuração de um contexto de função




```
function forEach (list, callback) {
    para (var n = 0; n <list.length; n ++) {
        callback.call (lista [n], n);
    }
}

armas var = [{tipo: 'shuriken'},
                {tipo: 'katana'},
                {tipo: 'nunchucks'}];

forEach (armas, função (índice) {
        afirmar (this === armas [índice],
                "Obteve o valor esperado de" + armas [índice] .tipo);
});
```

A função de iteração exibe uma assinatura simples que espera que a matriz de objetos seja iterada como o primeiro argumento e uma função de retorno de chamada como o segundo. A função itera sobre as entradas da matriz, invocando a função de retorno de chamada para cada entrada:

```
function forEach (list, callback) {
    para (var n = 0; n <list.length; n ++) {
        callback.call (lista [n], n);
    }
}
```

Nós usamos o chamar método da função de retorno de chamada, passando a entrada de iteração atual como o primeiro parâmetro e o índice de loop como o segundo. Esta *deve* faz com que a entrada atual se torne o contexto da função e o índice seja passado como o único parâmetro para o retorno de chamada.

Agora, para testar isso! Configuramos um simples armas variedade. Então chamamos o para cada função, passando a matriz de teste e um retorno de chamada dentro do qual testamos se a entrada esperada é definida como o contexto da função para cada invocação do retorno de chamada:

```
forEach (armas, função (índice) {
    afirmar (this === armas [índice],
                "Obteve o valor esperado de" + armas [índice] .tipo);
});
```

A Figura 4.5 mostra que nossa função funciona esplendidamente.

Em uma implementação pronta para produção de tal função, haveria muito mais trabalho a fazer. Por exemplo, e se o primeiro argumento não for um array? E se o segundo

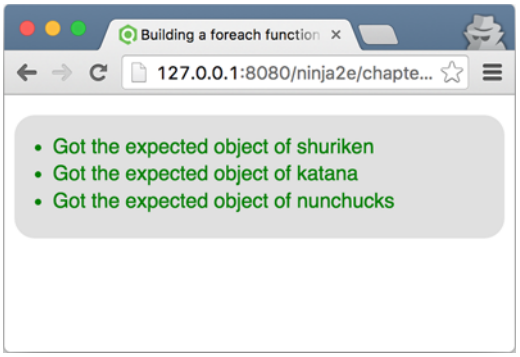


Figura 4.5 Os resultados do teste mostram que temos a capacidade de tornar qualquer objeto que desejamos no contexto de função de uma chamada de retorno.

argumento não é uma função? Como você permitiria que o autor da página encerrasse o loop a qualquer momento? Como exercício, você pode aumentar a função para lidar com essas situações. Outro exercício que você pode fazer é aprimorar a função para que o autor da página possa passar um número arbitrário de argumentos para o retorno de chamada, além do índice de iteração.

Dado que Aplique e chamar fazer praticamente a mesma coisa, aqui está algo que você pode estar se perguntando neste momento: Como decidimos qual usar? A resposta de alto nível é a mesma para muitas dessas perguntas: Usamos aquela que melhora a clareza do código. Uma resposta mais prática é usar aquele que melhor corresponda aos argumentos de que dispomos. Se tivermos um monte de valores não relacionados em variáveis ou especificados como literais,
chamar nos permite listá-los diretamente em sua lista de argumentos. Mas se já temos os valores do argumento em uma matriz, ou se é conveniente coletá-los como tal, Aplique poderia ser a melhor escolha.

4,3 Corrigindo o problema de contextos de função

Na seção anterior, você viu alguns dos problemas que podem ocorrer ao lidar com o contexto de função em JavaScript. Em funções de retorno de chamada (como manipuladores de eventos), o contexto da função pode não ser exatamente o que esperamos, mas podemos usar o chamar
e Aplique métodos para contorná-lo. Nesta seção, você verá duas outras opções: funções de seta e o ligar método, que pode, em certos casos, obter o mesmo efeito, mas de uma forma muito mais elegante.

4.3.1 Usando funções de seta para contornar contextos de função

Além de nos permitir criar funções de uma maneira mais elegante do que as declarações de função padrão e expressões de função, as funções de seta introduzidas no capítulo anterior têm um recurso que as torna particularmente boas como funções de retorno de chamada: as funções de seta não têm as suas próprias esta valor. Em vez disso, eles se lembram do valor do esta parâmetro no momento da sua definição. Vamos revisitar nosso problema com retornos de chamada de clique de botão na lista a seguir.

Listagem 4.13 Usando funções de seta para contornar contextos de função de retorno de chamada

Um elemento de botão para o qual nós vamos atribuir um evento manipulador

Uma função construtora que cria objetos que retêm o estado em relação ao nosso botão. Com isso, vamos rastrear se o botão foi clicado.

Declara a flecha função que usaremos como o manipulador de cliques. Por ser um método do objeto, usamos isso dentro da função para obter uma referência ao objeto.

Dentro do método, nós testamos que o botão estado foi alterado corretamente após um clique.

Estabelece o manipulador de clique no botão

```
<button id = "test"> Clique aqui! </button>
<script>
    function Button () {
        this.clicked = false;
        this.click = () => {
            this.clicked = true;
            assert (button.clicked, "O botão foi clicado"); //
        };
    }
    botão var = novo Botão ();
    var elem = document.getElementById ("teste"); elem.addEventListener ("click",
    button.click);
</script>
```

A única mudança, quando comparada à listagem 4.10, é que a listagem 4.13 usa uma função de seta:

```
this.click = () => {
    this.clicked = true;
    assert (button.clicked, "O botão foi clicado");
};
```

Agora, se executarmos o código, obteremos a saída mostrada na figura 4.6.

Como você pode ver, está tudo bem agora. O objeto de botão rastreia o clicado Estado. O que aconteceu é que nosso manipulador de cliques foi criado dentro do Botão construtor como uma função de seta:

```
function Button () {
        this.clicked = false;
        this.click = () => {
            this.clicked = true;
            assert (button.clicked, "O botão foi clicado");
        };
}
```

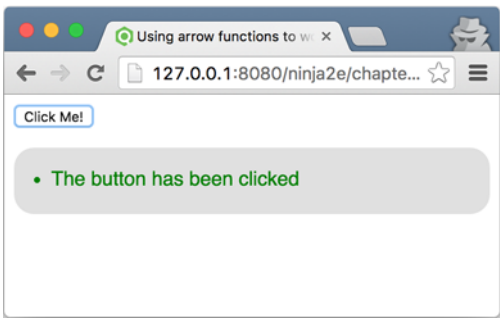


Figura 4.6 As funções de seta não têm contexto próprio. Em vez disso, o contexto é herdado da função na qual foram definidos. O **esta** parâmetro em nosso retorno de chamada de função de seta se refere ao objeto de botão.

Como já mencionamos, as funções de seta não têm seus próprios implícitos esta parâmetro quando os chamamos; em vez disso, eles se lembram do valor do esta parâmetro no momento em que foram criados. No nosso caso, o clique função de seta foi criada dentro de uma função de construtor, onde o esta parâmetro é o objeto recém-construído, então sempre que nós (ou o navegador) chamamos o clique função, o valor do esta parâmetro sempre será vinculado ao objeto de botão recém-construído.

CAVEAT: FUNÇÕES DE SETA E LITERAIS DE OBJETOS

Porque o valor do esta parâmetro é escolhido no momento em que a função de seta é criada, alguns comportamentos aparentemente estranhos podem resultar. Vamos voltar ao nosso exemplo de manipulador de clique de botão. Digamos que chegamos à conclusão de que não precisamos de uma função construtora, porque temos apenas um botão. Nós o substituímos por um literal de objeto simples, da seguinte maneira.

Listagem 4.14 Funções de seta e literais de objeto

O botão objeto é definido como um objeto literal.

O valor deste parâmetro é global code é o objeto global da janela.

```
<button id = "test"> Clique aqui! </button>
<script>
    afirmar (esta janela === "esta janela ==="); botão var = {

        clicado: falso,
        clique: () => {
            this.clicked = true;
            assert (button.clicked, "O botão foi clicado"); assert (esta janela ===, "Na função de seta esta janela ==="); assert
            (window.clicked, "clicked is stored in window");

        }
    }

    var elem = document.getElementById ("teste"); elem.addEventListener ("click",
    button.click);
</script>
```

Nossa função de seta é uma propriedade de um objeto literal.

Teste se o botão foi clicado.

clicado é armazenado na janela.

O valor de isso em nosso função de seta é o global objeto de janela.

Se executarmos a listagem 4.14, ficaremos novamente desapontados, porque o botão objeto mais uma vez não conseguiu rastrear o clicado Estado. Veja a figura 4.7.

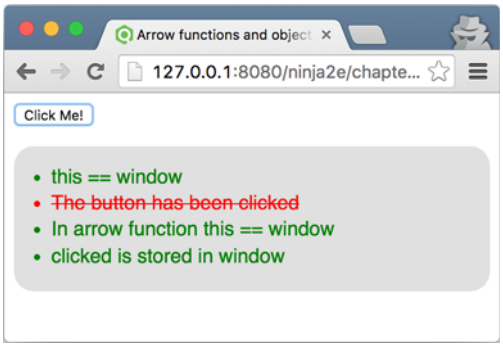


Figura 4.7 Se uma função de seta for definida em um literal de objeto definido no código global, o valor do **esta** parâmetro associado à função de seta é o global **janela** objeto.

Felizmente, espalhamos algumas afirmações em nosso código que ajudarão. Por exemplo, colocamos o seguinte diretamente no código global, a fim de verificar o valor do esta parâmetro:

```
afirmar (esta janela ===, "esta janela ===");
```

Como a afirmação é aprovada, podemos ter certeza de que no código global esta refere-se ao global janela objeto.

Seguimos especificando que o botão literal de objeto tem um clique propriedade da função de seta:

```
botão var = {
    clicado: falso,
    clique: () => {
        this.clicked = true;
        assert (button.clicked, "O botão foi clicado"); assert (esta janela ===, "Na função de seta esta janela ==="); assert
        (window.clicked, "Clicked is stored in window");

    };
}
```

Agora, vamos revisitar novamente nossa pequena regra: *As funções de seta captam o valor do* esta *parâmetro no momento da sua criação*. Porque o clique a função de seta é criada como um valor de propriedade em um literal de objeto, e o literal de objeto é criado em código global, o esta
valor da função de seta será o esta valor do código global. E, como vimos desde a primeira afirmação colocada em nosso código global

```
afirmar (esta janela ===, "esta janela ===");
```

o valor do esta parâmetro no código global é o global janela objeto. Portanto, nosso clicado propriedade será definida no global janela objeto, e não em nosso
botão objeto. Só para ter certeza, no final, verificamos se o janela objeto foi atribuído a um clicado propriedade:

```
assert (window.clicked, "Clicked is stored in window");
```

Como você pode ver, não se lembrar de todas as consequências das funções das setas pode levar a alguns erros sutis, então tome cuidado!

Agora que exploramos como as funções de seta podem ser usadas para contornar o problema de contextos de função, vamos continuar com outro método para corrigir o mesmo problema.

4.3.2 Usando o método bind

Neste capítulo, você já conheceu dois métodos aos quais toda função tem acesso, chamar
e Aplique, e você viu como usá-los para maior controle sobre o contexto e os argumentos de nossas invocações de função.

Além desses métodos, cada função tem acesso ao ligar método que, em suma, cria uma nova função. Esta função tem o mesmo corpo, mas seu contexto é *sempre*

ligado a um certo objeto, *independentemente* da maneira como o invocamos.

Vamos revisitar nosso pequeno problema com manipuladores de clique de botão uma última vez.

Listagem 4.15 Vinculando um contexto específico a um manipulador de eventos

```
<button id = "test"> Clique aqui! </button>
<script>
    botão var = {
        clicado: falso,
        click: function () {
            this.clicked = true;
            assert (button.clicked, "O botão foi clicado");
        }
    };
    var elem = document.getElementById ("teste"); elem.addEventListener ("click", botão.clique.bind (botão));

    var boundFunction = button.click.bind (button); assert (boundFunction! = button.click,

                "Chamar bind cria uma função completamente nova");
</script>
```

Usa a função de ligação para criar uma nova função vinculado ao objeto de botão

O molho secreto adicionado aqui é o ligar() método:

```
elem.addEventListener ("click", botão.clique.bind (botão));
```

O ligar método está disponível para todas as funções e é projetado para *Criar* e devolver um *novo* função que está ligada ao objeto passado (neste caso, o botão objeto). O valor do esta parâmetro é sempre definido para esse objeto, independentemente da forma como a função vinculada foi chamada. Além disso, a função associada se comporta como a função de origem, pois possui o mesmo código em seu corpo.

Sempre que o botão é clicado, essa função associada será invocada com o botão objeto como seu contexto, porque usamos isso botão objeto como um argumento para ligar.

Observe que chamar o ligar método não modifica a função original. Ele cria uma função completamente nova, um fato afirmado no final do exemplo:

```
var boundFunction = button.click.bind (button); assert (boundFunction! = button.click,

            "Chamar bind cria uma função completamente nova");
```

Com isso, encerraremos nossa exploração do contexto da função. Descanse por enquanto, porque no próximo capítulo, lidaremos com um dos conceitos mais importantes em JavaScript: fechamentos.

4,4 Resumo

- Ao invocar uma função, além dos parâmetros explicitamente declarados na definição da função, as invocações da função são passadas em dois parâmetros implícitos: argumentos e esta:
  - O argumentos parâmetro é uma coleção de argumentos transmitidos à função. Tem um comprimento propriedade que indica quantos argumentos foram passados e nos permite acessar os valores dos argumentos que não possuem parâmetros correspondentes. No modo não estrito, o argumentos objeto aliases os parâmetros da função (alterar o argumento altera o valor do parâmetro e vice-versa). Isso pode ser evitado usando o modo estrito. O esta parâmetro representa o contexto da função, um objeto ao qual a invocação da função está associada. Quão esta é determinado pode depender de como uma função é
  - definida, bem como de como ela é chamada.

- UMA A função pode ser chamada de quatro maneiras: Como uma
  - função: skulk ()
  - Como método: ninja.skulk ()
  - Como construtor: novo Ninja ()
  - Via seu Aplique e chamar métodos: skulk.call (ninja) ou skulk.apply (ninja)
- A forma como uma função é chamada influencia o valor do esta parâmetro:
  - Se uma função for invocada como uma função, o valor do esta parâmetro geralmente é o global janela objeto em modo não estrito, e Indefinido no modo estrito. Se uma função é chamada como um método, o valor do esta parâmetro geralmente é o objeto no qual a função foi chamada.
  -
  - Se uma função for invocada como um construtor, o valor do esta parâmetro é o objeto recém-construído.
  - Se uma função é chamada por meio de chamar e Aplique, o valor do esta parâmetro é o primeiro argumento fornecido para chamar e Aplique.
- As funções de seta não têm seu próprio valor de esta parâmetro. Em vez disso, eles o pegam no momento de sua criação.
- Use o ligar método, disponível para todas as funções, para criar uma nova função que está sempre ligada ao argumento do ligar método. Em todos os outros aspectos, a função vinculada se comporta como a função original.

4,5 Exercícios

1 A função a seguir calcula a soma dos argumentos transmitidos usando a argumentos objeto:

```
function sum () {
    var sum = 0;
    para (var i = 0; i <argument.length; i ++) {
        soma + = argumentos [i];
    }
```

```
    soma de retorno;
}

assert (sum (1, 2, 3) === 6, 'A soma dos três primeiros números é 6'); assert (sum (1, 2, 3, 4) === 10, 'A soma dos quatro primeiros números é 10');
```

Usando os parâmetros restantes introduzidos no capítulo anterior, reescreva o soma função para que não use o argumentos objeto.

2 Depois de executar o código a seguir, quais são os valores das variáveis ninja e samurai?

```
function getSamurai (samurai) {
    "use estrito"

    argumentos [0] = "Ishida";

    retorno samurai;
}

function getNinja (ninja) {
    argumentos [0] = "Fuma";
    retornar ninja;
}

var samurai = getSamurai ("Toyotomi"); var ninja = getNinja ("Yoshi");
```

3 Ao executar o código a seguir, qual das afirmações será aprovada?

```
function whoAmI1 () {
    "use estrito";
    devolva isso;
}

function whoAmI2 () {
    devolva isso;
}

assert (whoAmI1 () === window, "Window?"); assert (whoAmI2 () ===
window, "Window?");
```

4 Ao executar o código a seguir, qual das afirmações será aprovada?

```
var ninja1 = {
      whoAmI: function () {
          devolva isso;
      }
};

var ninja2 = {
    whoAmI: ninja1.whoAmI
};
```

```
var identificar = ninja2.whoAmI;

assert (ninja1.whoAmI () === ninja1, "ninja1?"); assert (ninja2.whoAmI () === ninja1, "ninja1
novamente?");

assert (identifique () === ninja1, "ninja1 novamente?");

assert (ninja1.whoAmI.call (ninja2) === ninja2, "ninja2 aqui?");
```

5 Ao executar o código a seguir, qual das afirmações será aprovada?

```
function Ninja () {
    this.whoAmI = () => isso;
}

var ninja1 = novo Ninja (); var ninja2 = {

    whoAmI: ninja1.whoAmI
};


assert (ninja1.whoAmI () === ninja1, "ninja1 aqui?"); assert (ninja2.whoAmI () === ninja2, "ninja2
aqui?");
```

6 Qual das afirmações a seguir será aprovada?

```
function Ninja () {
    this.whoAmI = function () {
        devolva isso;
    } .bind (este);
}

var ninja1 = novo Ninja (); var ninja2 = {

    whoAmI: ninja1.whoAmI
};


assert (ninja1.whoAmI () === ninja1, "ninja1 aqui?"); assert (ninja2.whoAmI () === ninja2, "ninja2
aqui?");
```

5 Funções para o mestre: fechamentos e escopos

Este capítulo cobre

- Usando fechamentos para simplificar o desenvolvimento
- Acompanhar a execução de programas JavaScript com contextos de execução
- Rastreamento de escopos variáveis com ambientes lexicais
- Compreendendo os tipos de variáveis Explorando como
- funcionam os fechamentos

Intimamente vinculado às funções que aprendemos nos capítulos anteriores, os encerramentos são um recurso definidor do JavaScript. Embora muitos desenvolvedores de JavaScript possam escrever código sem compreender os benefícios dos encerramentos, seu uso pode não apenas nos ajudar a reduzir a quantidade e a complexidade do código necessária para adicionar recursos avançados, mas também nos permitir fazer coisas que de outra forma não seriam possíveis, ou seria muito complexo para ser viável. Por exemplo, quaisquer tarefas envolvendo retornos de chamada, como manipulação de eventos ou animações, seriam significativamente mais complexas sem encerramentos. Outros, como fornecer suporte para variáveis de objeto privado, seriam totalmente

impossível. A paisagem da linguagem e a maneira como escrevemos nosso código é para sempre moldada pela inclusão de fechamentos.

Tradicionalmente, os fechamentos têm sido uma característica das linguagens de programação puramente funcionais. Vê-los passar para o desenvolvimento dominante é encorajador. É comum encontrar fechamentos que permeiam as bibliotecas JavaScript, junto com outras bases de código avançadas, devido à sua capacidade de simplificar drasticamente operações complexas.

Os fechamentos são um efeito colateral de como os escopos funcionam em JavaScript. Por esse motivo, exploraremos as regras de escopo do JavaScript, com um foco especial nas adições recentes. Isso ajudará você a entender como os fechamentos funcionam nos bastidores. Vamos começar!

Você sabe?

- **Quantos escopos diferentes pode uma variável ou método tem, e quais são eles?**
- **Como os identificadores e seus valores são rastreados?**
- **O que é uma variável mutável e como você define uma em JavaScript?**

5,1 Entendendo fechamentos

UMA *fecho* permite que uma função acesse e manipule variáveis externas a essa função. Os fechamentos permitem que uma função acesse todas as variáveis, bem como outras funções, que estão no escopo quando a própria função é definida.

NOTA Você provavelmente está familiarizado com o conceito de escopos, mas apenas no caso, um *escopo* refere-se à visibilidade de identificadores em certas partes de um programa. Um escopo é uma parte do programa em que um certo nome está vinculado a uma determinada variável.

Isso pode parecer intuitivo até que você se lembre de que uma função declarada pode ser chamada a qualquer momento posterior, mesmo *depois de* o escopo em que foi declarado desapareceu. Este conceito é provavelmente melhor explicado por meio de código. Mas antes de entrarmos em exemplos concretos que irão ajudá-lo a desenvolver animações mais elegantes em código ou para definir propriedades de objetos privados, vamos começar pequeno, com a seguinte listagem.

Listagem 5.1 Um fechamento simples

```
var outerValue = "ninja"; function outerFunction
() {
    afirmar( outerValue === " ninja "," posso ver o ninja. ");
}
outerFunction ();
```

Define um valor em escopo global

Declara uma função em escopo global

Executa a função

Neste exemplo de código, declaramos uma variável outerValue e uma função outerFunction no mesmo escopo - neste caso, o escopo global. Depois, ligamos outerFunction.

Como você pode ver na figura 5.1, a função é capaz de "ver" e acessar o outerValue variável. Provavelmente, você escreveu um código como este centenas de vezes sem perceber que estava criando um encerramento!

Não estou impressionado? Acho que isso não é surpreendente. Porque ambos outerValue e outerFunction são declarados em escopo global, esse escopo (que é um encerramento) nunca vai embora (enquanto nosso aplicativo estiver em execução). Não é surpreendente que a função possa acessar a variável, porque ela ainda está no escopo e é viável.

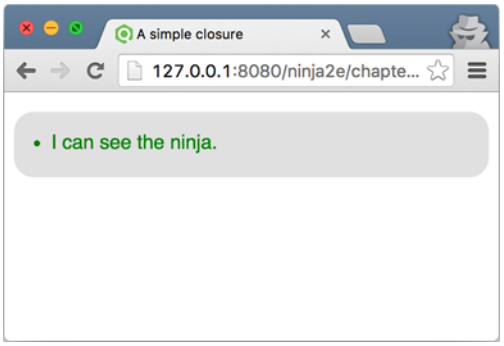


Figura 5.1 Nossa função encontrou o ninja, que estava escondido à vista de todos.

Mesmo que o fechamento exista, seus benefícios ainda não são claros. Vamos incrementar na próxima lista.

Listagem 5.2 Outro exemplo de fechamento

```
var outerValue = "samurai"; var depois;

function outerFunction () {
    var innerValue = "ninja";

    function innerFunction () {
        assert (outerValue === "samurai", "Eu posso ver o samurai."); assert (innerValue === "ninja", "Eu posso ver o
        ninja.")
    }

    posterior = innerFunction;
}

outerFunction ();

mais tarde();
```

Uma variável vazia que usaremos mais tarde

Declara um valor dentro do função. Esta escopo da variável é limitado a a função e não pode ser acessado de fora da função.

Declara uma função interna dentro de outerFunction. innerValue está no escopo quando criamos esta função.

Armazena uma referência a innerFunction na variável posterior. Como o mais tarde está no escopo global, isso nos permitirá chamar a função mais tarde.

Invoca outerFunction, que causa innerFunction para ser criado e sua referência atribuído para mais tarde

Invoca innerFunction mais tarde. Não podemos invocá-lo diretamente porque seu escopo (junto com innerValue) é limitado a outerFunction.

Vamos analisar demais o código em innerFunction e ver se podemos prever o que pode acontecer:

- O primeiro afirmar está certo de passar; outerValue está no âmbito global e é visível para tudo. Mas e quanto ao segundo afirmar?
- Estamos executando innerFunction *depois de* outerFunction foi executado através do truque de copiar uma referência à função para a variável global mais tarde.

- Quando innerFunction executa, o escopo dentro da função externa há muito se foi e não é visível no ponto em que estamos invocando a função por meio

  mais tarde.
- Portanto, poderíamos muito bem esperar afirmar falhar, como innerValue com certeza será Indefinido. Direita?

Mas quando executamos o teste, vemos a tela da figura 5.2.

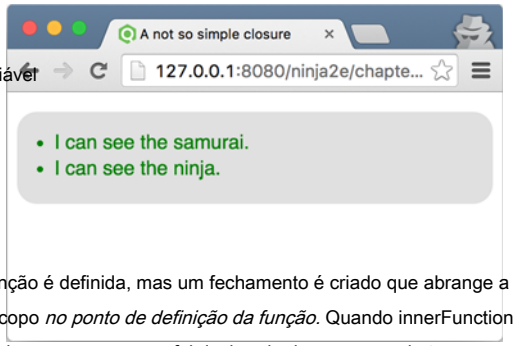


Figura 5.2 Apesar de tentar se esconder dentro de uma função, o ninja foi detectado!

Como pode ser? Que magia permite o innerValue variável ainda estar "viva" quando executamos a função interna, muito depois de o escopo em que foi criada ter desaparecido? A resposta é encerramentos.

Quando declaramos innerFunction dentro da função externa, não apenas a declaração da função é definida, mas um fechamento é criado que abrange a definição da função, bem como todas as variáveis no escopo *no ponto de definição da função.* Quando innerFunction eventualmente executa, mesmo se for executado *depois de* o escopo em que foi declarado desaparece, ele tem acesso ao escopo original em que foi declarado por meio de seu fechamento, conforme mostrado na figura 5.3.

Isso é o que significa encerramentos. Eles criam uma "bolha de segurança" da função e das variáveis em escopo no ponto da definição da função, de modo que a função tenha tudo o que precisa para ser executada. Esta bolha, contendo a função e suas variáveis, permanece enquanto a função permanece.

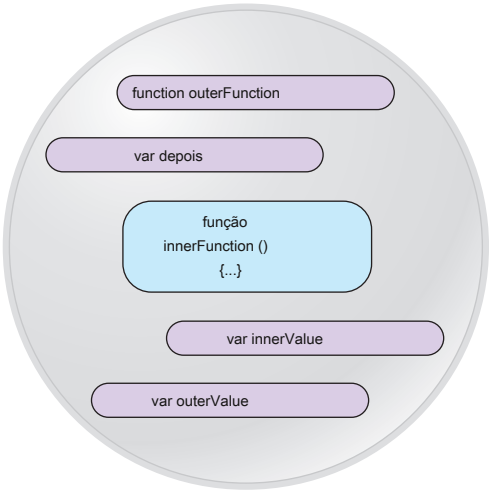


Figura 5.3 Como uma bolha protetora, o fechamento para **innerFunction** mantém as variáveis no escopo da função vivas enquanto a função existir.

Embora toda essa estrutura não seja prontamente visível (não há nenhum objeto de "fechamento" contendo todas as informações que você pode inspecionar), armazenar e referenciar informações dessa forma tem um custo direto. É importante lembrar que cada função que acessa informações por meio de um fecho tem uma "bola e uma corrente" anexada a ela, carregando essas informações. Portanto, embora os fechamentos sejam incrivelmente úteis, eles não são isentos de sobrecarga. Todas essas informações precisam ser mantidas na memória até que seja

absolutamente claro para o mecanismo JavaScript que não é mais necessário (e é seguro coletar lixo) ou até que a página seja descarregada.

Não se preocupe; isso não é tudo o que temos a dizer sobre como funcionam os fechamentos. Mas antes de explorar os mecanismos que permitem fechamentos, vamos dar uma olhada em seus usos práticos.

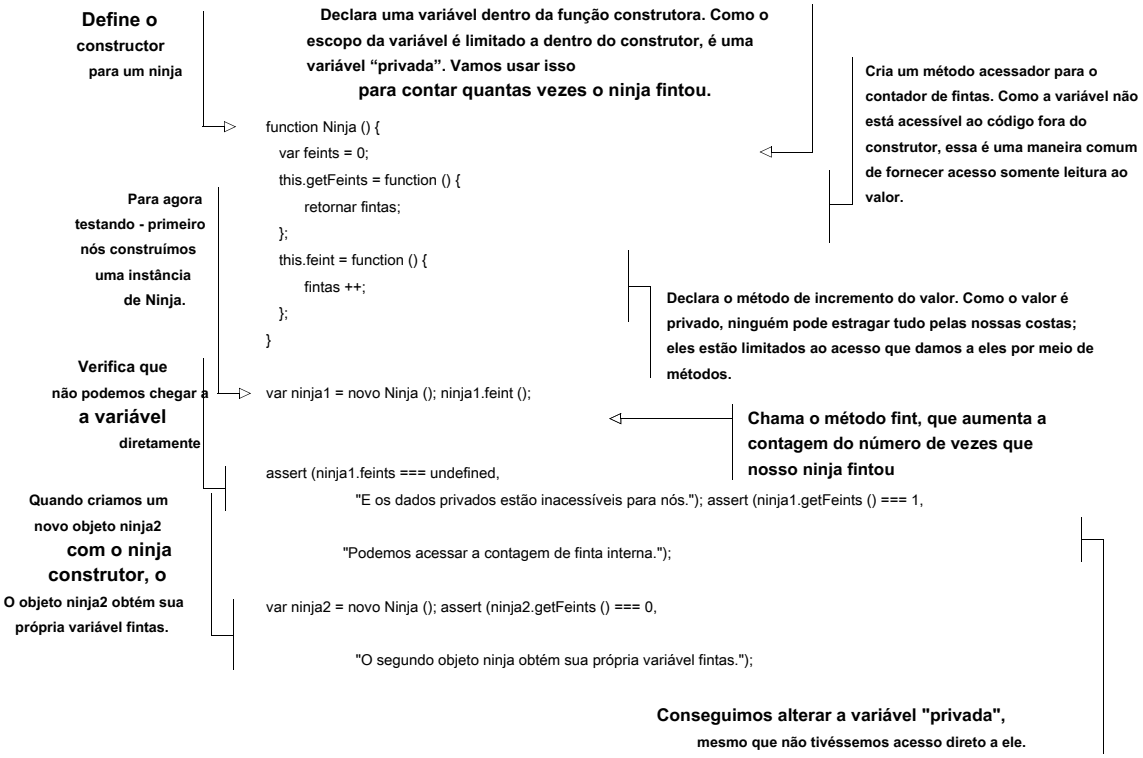
5,2 Colocando fechamentos para funcionar

Agora que temos um entendimento de alto nível sobre fechamentos, vamos ver como colocá-los para funcionar em nossos aplicativos JavaScript. Por enquanto, vamos nos concentrar em seus aspectos práticos e benefícios. Posteriormente neste capítulo, revisitaremos os mesmos exemplos para ver exatamente o que está acontecendo nos bastidores.

5.2.1 Imitando variáveis privadas

Muitas linguagens de programação usam *variáveis privadas* —Propriedades de um objeto que estão ocultas de terceiros. Este é um recurso útil, porque não queremos sobrecarregar os usuários de nossos objetos com detalhes de implementação desnecessários ao acessar esses objetos de outras partes do código. Infelizmente, JavaScript não tem suporte nativo para variáveis privadas. Mas, usando um encerramento, podemos obter uma aproximação aceitável, conforme demonstrado pelo código a seguir.

Listagem 5.3 Usando fechamentos para aproximar variáveis privadas



Aqui criamos uma função, Ninja, para servir como um construtor. Introduzimos o uso de uma função como construtor no capítulo 3 (e daremos uma olhada em profundidade no capítulo 7). Por enquanto, lembre-se de que ao usar o novo em uma função, uma nova instância de objeto é criada e a função é chamada com esse novo objeto como seu contexto, para servir como um construtor para esse objeto. Então esta dentro da função se refere a um objeto recém-instanciado.

Dentro do construtor, definimos uma variável para manter o estado, fintas. As regras de escopo do JavaScript para esta variável limitam sua acessibilidade a *dentro de* o construtor. Para dar acesso ao valor da variável do código que está fora do escopo, definimos um
acessor método: getFeints, que pode ser usado para ler a variável privada. (Os métodos de acesso são frequentemente chamados *getters.)*

```
function Ninja () {
    var feints = 0;
    this.getFeints = function () {
        retornar fintas;
    };
    this.feint = function () {
        fintas ++;
    };
}
```

Um método de implementação, finta, é então criado para nos dar controle sobre o valor da variável. Em um aplicativo do mundo real, isso pode ser um método de negócios, mas neste exemplo, ele apenas incrementa o valor de fintas.

Depois que o construtor tiver cumprido sua função, podemos chamar o finta método no recém-criado ninja1 objeto:

```
var ninja1 = novo Ninja (); ninja1.feint ();
```

Nossos testes mostram que podemos usar o método acessador para obter o valor da variável privada, mas não podemos acessá-lo diretamente. Isso nos impede de fazer alterações não controladas no valor da variável, como se fosse uma verdadeira variável privada. Essa situação é ilustrada na figura 5.4.

O uso de encerramentos permite que o estado do ninja seja mantido dentro de um método, sem permitir que ele seja acessado diretamente por um usuário do método - porque a variável está disponível para os métodos internos por meio de seus encerramentos, mas não para o código que fica fora do construtor .

Este é um vislumbre do mundo do JavaScript orientado a objetos, que exploraremos com mais profundidade no capítulo 7. Por enquanto, vamos nos concentrar em outro uso comum de encerramentos.

5.2.2 Usando fechamentos com callbacks

Outro uso comum de encerramentos ocorre ao lidar com retornos de chamada - quando uma função é chamada em um momento posterior não especificado. Muitas vezes, dentro de tais funções, frequentemente

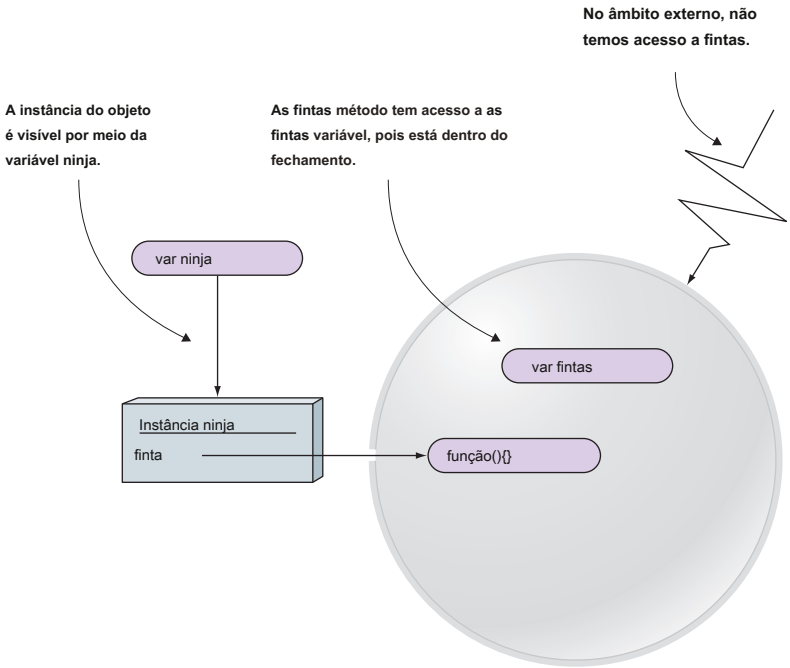
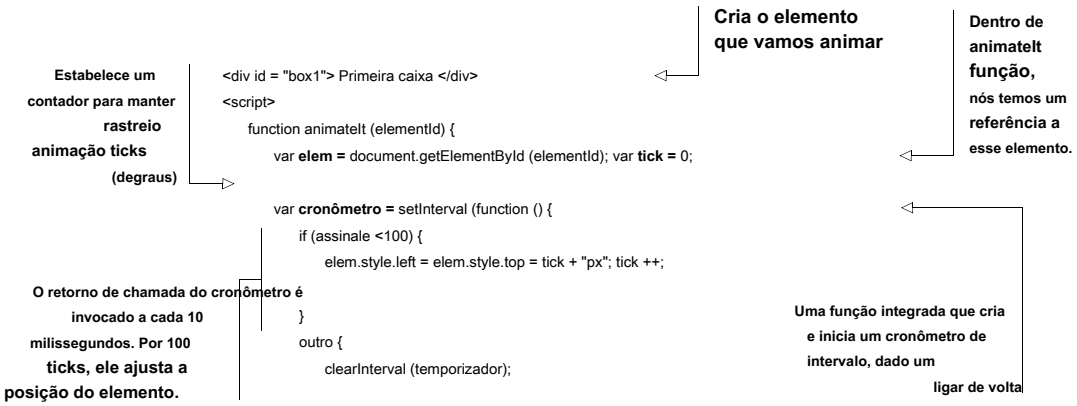
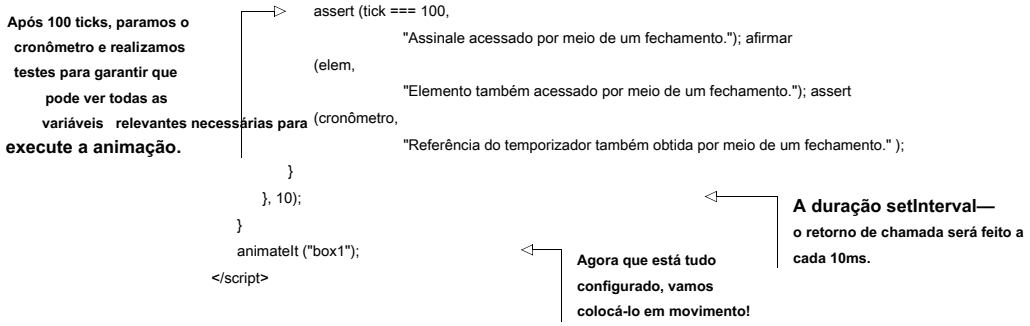


Figura 5.4 Ocultar a variável dentro do construtor a mantém invisível para o escopo externo, mas onde é importante, a variável está viva e bem, protegida pelo fechamento.

precisa acessar dados externos. A lista a seguir mostra um exemplo que cria uma animação simples com cronômetros de retorno de chamada.

Listagem 5.4 Usando um fechamento em um retorno de chamada de intervalo do cronômetro





O que é especialmente importante sobre este código é que ele usa uma única função anônima, colocada como um setInterval argumento, para realizar a animação do alvo
div elemento. Essa função acessa três variáveis: elem, tick, e cronômetro, por meio de um fechamento, para controlar o processo de animação. As três variáveis (a referência ao elemento DOM, elem; o contador de carrapatos, marcação; e a referência do cronômetro, cronômetro) tudo deve ser mantido *entre* as etapas da animação. E precisamos mantê-los fora do escopo global.

Mas o exemplo ainda funcionará bem se movermos as variáveis para fora do animateIt
função e no âmbito global. Então, por que todo o braço se debatendo sobre não poluir o escopo global?

Vá em frente e mova as variáveis para o escopo global e verifique se o exemplo ainda funciona. Agora modifique o exemplo para animar dois elementos: Adicione outro elemento com um ID exclusivo e chame o animateIt função com esse ID logo após a chamada original.

O problema se torna imediatamente óbvio. Se mantivermos as variáveis no escopo global, precisamos de um conjunto de três variáveis para *cada* animação. Caso contrário, eles vão pisar uns nos outros, tentando usar o mesmo conjunto de variáveis para manter o controle de vários estados.

Definindo as variáveis *lado de dentro* a função, e contando com fechamentos para torná-los disponíveis para as invocações de retorno de chamada do temporizador, cada animação obtém sua própria "bolha" privada de variáveis, conforme mostrado na Figura 5.5.

Sem encerramentos, fazer várias coisas ao mesmo tempo, seja manipulação de eventos, animações ou até mesmo solicitações do servidor, seria incrivelmente difícil. Se você estava esperando por um motivo para se preocupar com os fechamentos, é isso!

Este exemplo é particularmente bom para demonstrar como os encerramentos são capazes de produzir algum código surpreendentemente intuitivo e conciso. Incluindo as variáveis no animateIt função, criamos um encerramento implícito sem a necessidade de qualquer sintaxe complexa.

Há outro conceito importante que este exemplo deixa claro. Não apenas podemos ver os valores que essas variáveis tinham no momento em que o encerramento foi criado, mas podemos atualizá-los dentro do encerramento enquanto a função dentro do encerramento é executada. O fechamento não é apenas um instantâneo do estado do escopo no momento da criação, mas um encapsulamento ativo desse estado que podemos modificar enquanto o fechamento existir.

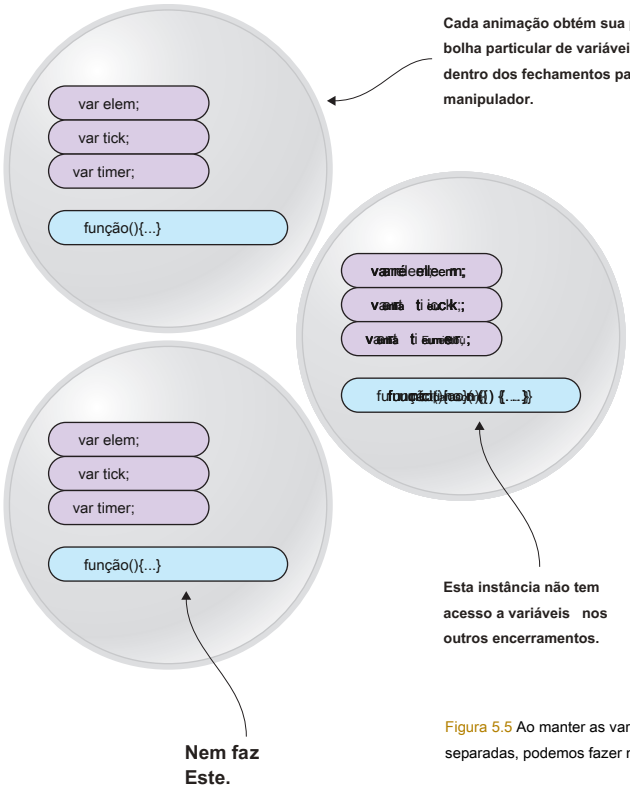


Figura 5.5 Ao manter as variáveis para várias instâncias da função separadas, podemos fazer muitas coisas ao mesmo tempo.

Os fechamentos estão intimamente relacionados aos escopos, portanto, passaremos uma boa parte deste capítulo explorando as regras de escopo em JavaScript. Mas, primeiro, começaremos com os detalhes de como a execução do código é rastreada em JavaScript.

5,3 Acompanhar a execução do código com contextos de execução

Em JavaScript, a unidade fundamental de execução é uma função. Nós os usamos o tempo todo, para calcular algo, para realizar efeitos colaterais, como alterar a IU, para obter a reutilização de código ou para tornar nosso código mais fácil de entender. Para cumprir seu propósito, uma função pode chamar outra função, que por sua vez pode chamar outra função e assim por diante. E quando uma função faz seu trabalho, a execução de nosso programa deve retornar à posição de onde a função foi chamada. Mas você já se perguntou como o mecanismo JavaScript mantém o controle de todas essas funções em execução e posições de retorno?

Como mencionamos no capítulo 2, existem dois tipos principais de código JavaScript: *código global,* colocado fora de todas as funções, e *código de função,* contido em funções. Quando nosso código está sendo executado pelo motor JavaScript, cada instrução é executada em um determinado *contexto de execução.*

E assim como temos dois tipos de código, temos dois tipos de contextos de execução: a *contexto de execução global* e um *contexto de execução da função.* Aqui está a diferença significativa:

Há apenas *1* contexto de execução global, criado quando nosso programa JavaScript começa a ser executado, enquanto um *novo* o contexto de execução da função é criado em *cada* invocação de função.

NOTA Você deve se lembrar do capítulo 4 que *contexto de função* é o objeto no qual nossa função é invocada, que pode ser acessado por meio do esta palavra-chave. Um contexto de execução, embora tenha um nome semelhante, é uma coisa completamente diferente. É um conceito interno de JavaScript que o mecanismo JavaScript usa para rastrear a execução de nossas funções.

Como mencionamos no capítulo 2, o JavaScript é baseado em um modelo de execução de thread único: apenas uma parte do código pode ser executada por vez. Cada vez que uma função é chamada, o contexto de execução atual deve ser interrompido e um novo contexto de execução da função, no qual o código da função será avaliado, deve ser criado. Depois que a função realiza sua tarefa, o contexto de execução da função geralmente é descartado e o contexto de execução do chamador é restaurado. Portanto, é necessário manter o controle de todos esses contextos de execução - tanto o que está em execução quanto os que estão esperando pacientemente. A maneira mais fácil de fazer isso é usando um *pilha,* Chamou o *pilha de contexto de execução (* ou frequentemente chamado de *pilha de chamadas).*

NOTA Uma pilha é uma estrutura de dados fundamental na qual você pode colocar novos itens apenas no topo e pode pegar os itens existentes apenas no topo. Pense em uma pilha de bandejas em uma cafeteria. Quando você quer pegar um, você escolhe um do topo. E um funcionário do refeitório que tem um novo limpo também o coloca no topo.

Isso pode parecer vago, então vamos dar uma olhada no código a seguir, que relata a atividade de dois ninjas furtivos.

Listagem 5.5 A criação de contextos de execução

```
function skulk (ninja) {
    relatório (ninja + "esquiva");
}

relatório de função (mensagem) {
    console.log (mensagem);
}

skulk ("Kuma");
skulk ("Yoshi");
```

Uma função que chama outra função

Uma função que relata uma mensagem por meio da função interna console.log

Duas chamadas de função do código global

Este código é direto; nós definimos o esconder-se função, que chama o relatório função, que produz uma mensagem. Então, a partir do código global, fazemos duas chamadas separadas para o esconder-se função: skulk ("Kuma") e skulk ("Yoshi"). Usando esse código como base, exploraremos a criação de contextos de execução, conforme mostrado na figura 5.6.

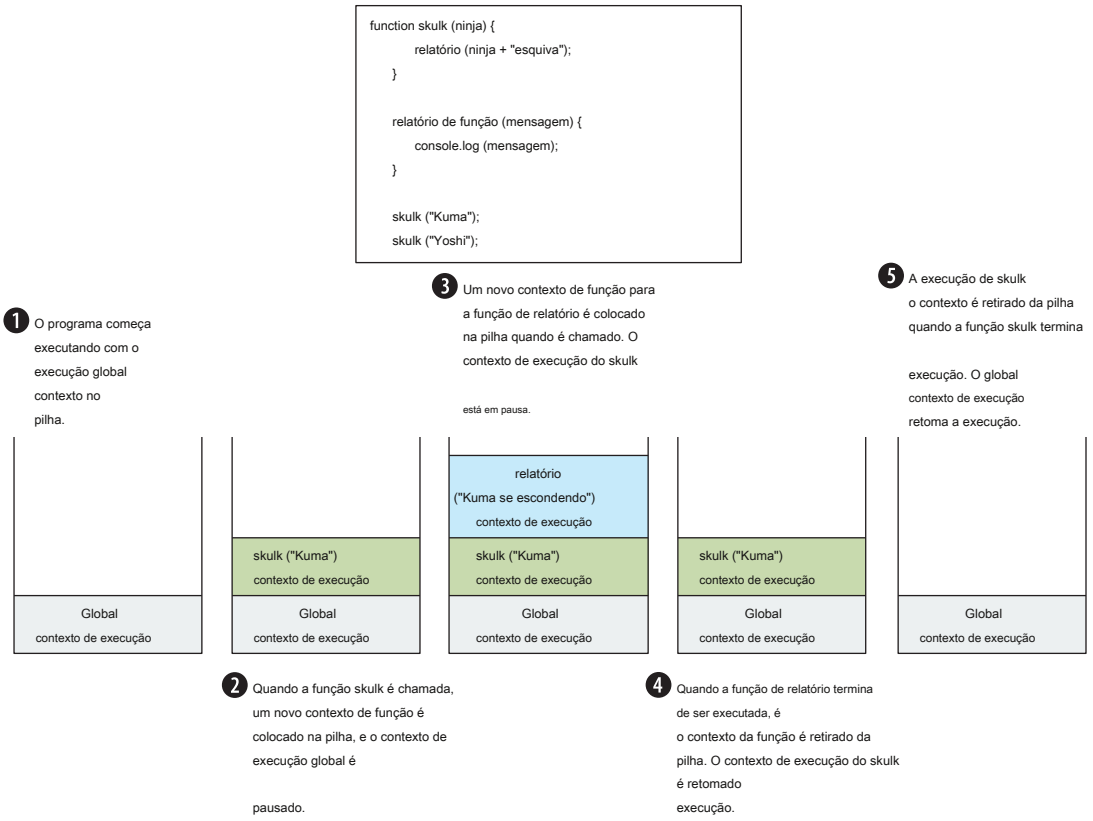


Figura 5.6 O comportamento da pilha de contexto de execução

Ao executar o código de exemplo, o contexto de execução se comporta da seguinte maneira:

1. A pilha de contexto de execução começa com o contexto de execução global que é criado apenas uma vez por programa JavaScript (uma vez por página, no caso de páginas da web). O contexto de execução global é o contexto de execução ativo ao executar o código global.

2. No código global, o programa primeiro define duas funções: esconder-se e relatório, e então chama o esconder-se funcionar com skulk ("Kuma"). Como apenas uma parte do código pode ser executada de uma vez, o mecanismo JavaScript pausa a execução do código global e vai executar o esconder-se código de função com Kuma como um argumento. Isso é feito criando um *novo* contexto de execução da função e colocá-lo no topo da pilha.

3. O esconder-se função, por sua vez, chama o relatório função com o argumento Kuma se escondendo. Novamente, porque apenas um trecho de código pode ser executado por vez, o esconder-se contexto de execução é pausado, e um novo contexto de execução de função para o relatório função, com o argumento Kuma se escondendo, é criado e colocado na pilha.

4 Depois de relatório função registra a mensagem usando o integrado console.log
função (ver apêndice C) e termina sua execução, temos que voltar ao
esconder-se função. Isso é feito abrindo o relatório contexto de execução da função da pilha. O esconder-se contexto
de execução da função é então reativado, e a execução do esconder-se a função continua. Algo
semelhante acontece quando o esconder-se função termina sua execução: O contexto de execução da
5 função do esconder-se a função é removida da pilha e o contexto de execução global, que esperou
pacientemente todo esse tempo, é restaurado como o contexto de execução ativo. A execução do código
JavaScript global é restaurada.

Todo este processo é repetido de forma semelhante para a segunda chamada para o esconder-se função, agora com o argumento Yoshi. Dois novos contextos de execução de função são criados
e empurrado para a pilha, skulk ("Yoshi") e relatório ("Yoshi se escondendo"), quando o
respectivas funções são chamadas. Esses contextos de execução também são retirados da pilha quando o programa retorna da função de correspondência.

Mesmo que a pilha de contexto de execução seja um conceito interno de JavaScript, você pode explorá-la em qualquer depurador de JavaScript, onde é referido como um *pilha de chamadas.* A Figura 5.7 mostra a pilha de chamadas no Chrome DevTools.

NOTA O Apêndice C oferece uma visão mais detalhada das ferramentas de depuração disponíveis em vários navegadores.

Além de acompanhar a posição na execução do aplicativo, o contexto de execução é vital em *resolução do identificador,* o processo de descobrir a qual variável um determinado identificador se refere. O contexto de execução faz isso por meio do *ambiente lexical.*

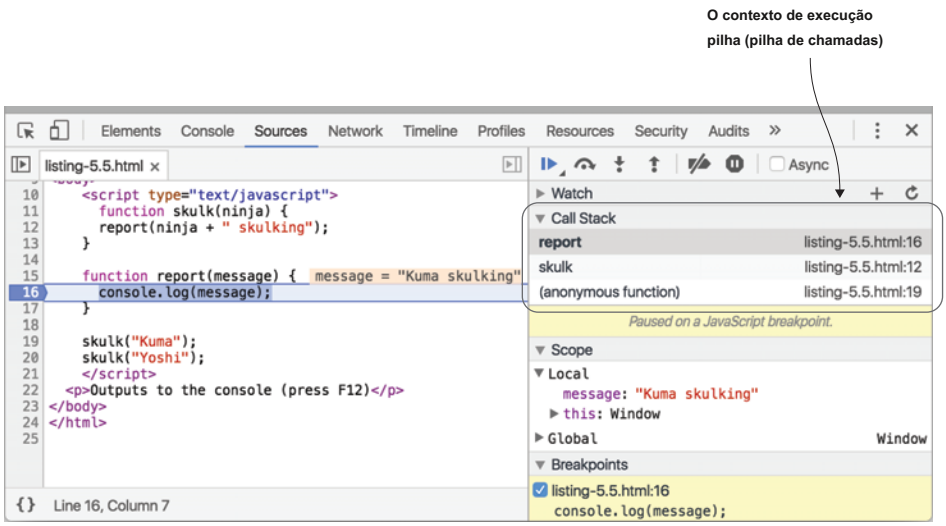


Figura 5.7 O estado atual da pilha de contexto de execução no Chrome DevTools

5,4 Acompanhar identificadores com ambientes lexicais

UMA *ambiente lexical* é uma construção de mecanismo JavaScript interno usada para rastrear o mapeamento de identificadores para variáveis específicas. Por exemplo, considere o seguinte código:

```
var ninja = "Hattori"; console.log (ninja);
```

O ambiente lexical é consultado quando o ninja variável é acessada no console.log demonstração.

NOTA Ambientes lexicais são uma implementação interna do mecanismo de escopo do JavaScript, e as pessoas costumam se referir a eles como *escopos*.

Normalmente, um ambiente léxico é associado a uma estrutura específica de código JavaScript. Ele pode ser associado a uma função, um bloco de código ou a pegar parte de um tentar pegar demonstração. Cada uma dessas estruturas (funções, blocos e pegar partes) podem ter seus próprios mapeamentos de identificador separados.

NOTA Em versões pré-ES6 de JavaScript, um ambiente léxico pode ser associado apenas a uma função. Variáveis podem ter apenas escopo de função. Isso causou muita confusão. Como o JavaScript é uma linguagem semelhante a C, as pessoas vindas de outras linguagens semelhantes a C (como C ++, C # ou Java) naturalmente esperavam que alguns conceitos de baixo nível, como a existência de escopos de bloco, fossem os mesmos. Com o ES6, isso finalmente foi corrigido.

5.4.1 Aninhamento de código

Ambientes lexicais são fortemente baseados em *aninhamento de código,* que permite que uma estrutura de código seja contida em outra. A Figura 5.8 mostra vários tipos de aninhamento de código.

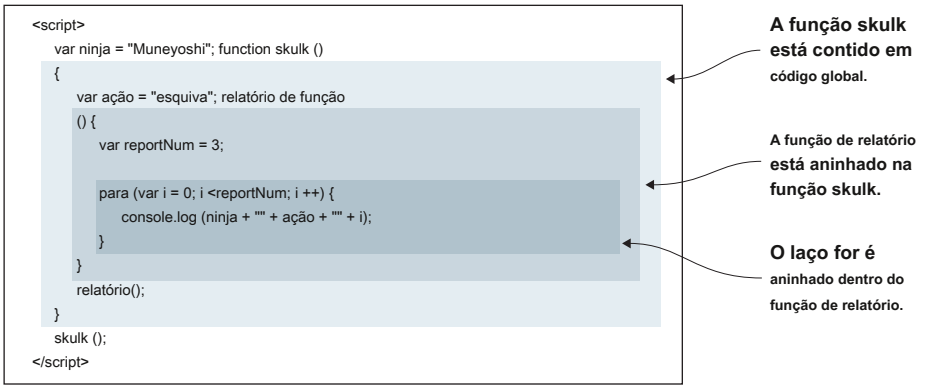


Figura 5.8 Tipos de aninhamento de código

Neste exemplo, podemos ver o seguinte:

- O pra loop está aninhado dentro do relatório função.
- O relatório função está aninhada no esconder-se função.
- O esconder-se função está aninhada no código global.

Em termos de escopos, cada uma dessas estruturas de código obtém um ambiente léxico associado *cada* hora em que o código é avaliado. Por exemplo, em cada invocação do esconder-se
função, um novo ambiente léxico de função é criado.

Além disso, é importante enfatizar que uma estrutura de código interna tem acesso às variáveis definidas em estruturas de código externas; por exemplo, o pra loop pode acessar variáveis do relatório função, o esconder-se função e o código global; a
relatório função pode acessar variáveis do esconder-se função e o código global; e a esconder-se função pode acessar apenas variáveis adicionais do código global.

Não há nada de especial nessa forma de acessar variáveis; todos nós provavelmente já o fizemos muitas vezes. Mas como o mecanismo JavaScript rastreia todas essas variáveis e o que é acessível de onde? É aqui que entram os ambientes lexicais.

5.4.2 Aninhamento de código e ambientes lexicais

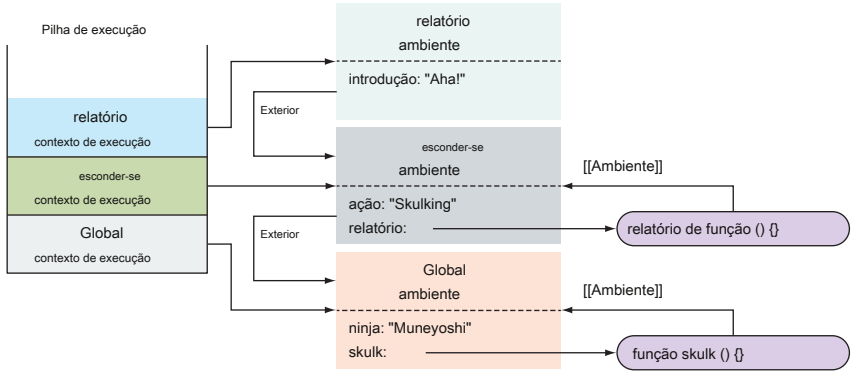
Além de manter o controle de variáveis locais, declarações de função e parâmetros de função, cada ambiente léxico deve manter o controle de seus *exterior (* pai) ambiente léxico. Isso é necessário porque temos que ser capazes de acessar variáveis definidas em estruturas de código externas; se um identificador não puder ser encontrado no ambiente atual, o ambiente externo é pesquisado. Isso para quando a variável correspondente é encontrada ou com um erro de referência se alcançamos o ambiente global e não há sinal do identificador procurado. A Figura 5.9 mostra um exemplo; você pode ver como os identificadores introdução, ação, e ninja são resolvidos ao executar o relatório função.

Neste exemplo, o relatório função é chamada pelo esconder-se função, que por sua vez é chamada pelo código global. Cada contexto de execução possui um ambiente léxico associado a ele que contém o mapeamento para todos os identificadores definidos diretamente naquele contexto. Por exemplo, o ambiente global mantém o mapeamento para identificadores ninja e
esquivar-se, a esconder-se ambiente mantém o mapeamento para os identificadores açao e
relatório, e a relatório ambiente mantém o mapeamento para o introdução identificador (lado direito da figura 5.9).

Em um contexto de execução particular, além de acessar identificadores definidos diretamente no ambiente léxico correspondente, nossos programas freqüentemente acessam outras variáveis definidas em ambientes externos. Por exemplo, no corpo do relatório função, acessamos a variável açao do exterior esconder-se função, bem como o global ninja variável. Para fazer isso, precisamos de alguma forma acompanhar esses ambientes externos. JavaScript faz isso aproveitando as funções como objetos de primeira classe.

Sempre que uma função é criada, uma referência ao ambiente léxico no qual a função foi criada é armazenada em uma propriedade interna (o que significa que você não pode acessá-la ou manipulá-la diretamente) chamada [[Ambiente]]; colchetes duplos é a notação que usaremos para marcar essas propriedades internas. No nosso caso, o esconder-se função vai

```
< roteiro >
    var ninja = "Muneyoshi";

    função skulk () {
        var action = "Skulking";

        função relatório() {
            var intro = "Aha!";
            assert (intro === "Aha!", "Local"); assert (action === "Skulking",
            "Outer"); afirmar (ninja === "Muneyoshi", "Global");

        }

        relatório();
    }

    skulk ();
</ roteiro >
```



Achar introdução

1. Verifique relatório ambiente -> encontrado

Achar açao

1. Verifique relatório ambiente -> não encontrado
2. Verifique relatório ambiente externo de: esconder-se
   Verificar esconder-se ambiente -> encontrado

Achar ninja

1. Verifique relatório ambiente -> não encontrado
2. Verifique relatório ambiente externo de: esconder-se
   Verificar esconder-se ambiente -> não encontrado
3. Verifique esconder-se ambiente externo de: global
   Verificar global ambiente -> encontrado

Figura 5.9 Como os mecanismos JavaScript resolvem os valores das variáveis

manter uma referência ao meio ambiente global, e o relatório função manterá uma referência ao esconder-se ambiente, porque esses foram os ambientes em que as funções foram criadas.

NOTA Isso pode parecer estranho à primeira vista. Por que simplesmente não percorremos toda a pilha de contextos de execução e procuramos em seus ambientes correspondentes por mapeamentos de identificadores? Tecnicamente, isso funcionaria em nosso exemplo. Mas lembre-se, uma função JavaScript pode ser passada como qualquer outro objeto, então a posição da definição da função e a posição de onde a função é chamada geralmente não estão relacionadas (lembre-se dos fechamentos).

Sempre que uma função é chamada, um novo contexto de execução de função é criado e colocado na pilha de contexto de execução. Além disso, um novo ambiente léxico associado é criado. Agora vem a parte crucial: para o ambiente externo do ambiente léxico recém-criado, o mecanismo JavaScript coloca o ambiente referenciado pelo interno da função chamada [[Ambiente]] propriedade, o ambiente no qual a função agora chamada foi criada!

No nosso caso, quando o esconder-se função é chamada, o ambiente externo do recém-criado esconder-se ambiente torna-se o ambiente global (porque é o ambiente no qual o esconder-se função foi criada). Da mesma forma, ao chamar o relatório
função, o ambiente externo do recém-criado relatório ambiente está definido para o esconder-se ambiente.

Agora vamos dar uma olhada no relatório função:

```
relatório de função () {
    var intro = "Aha!";
    assert (intro === "Aha!", "Local"); assert (action === "Skulking", "Outer");
    afirmar (ação === "Muneyoshi", "Global");

}
```

Quando o primeiro afirmar declaração está sendo avaliada, temos que resolver o introdução identificador. Para fazer isso, o mecanismo JavaScript começa verificando o ambiente do contexto de execução atualmente em execução, o relatório ambiente. Porque o relatório
ambiente contém uma referência a introdução, o identificador é resolvido.

Próximo, o segundo afirmar declaração tem que resolver o açao identificador. Novamente, o ambiente do contexto de execução atualmente em execução é verificado. Mas o relatório
ambiente não contém uma referência ao açao identificador, portanto, o mecanismo JavaScript deve verificar o ambiente externo do relatório ambiente: o esconder-se
ambiente. Felizmente, o esconder-se ambiente contém uma referência ao açao
identificador e o identificador é resolvido. Um processo semelhante é seguido ao tentar resolver o ninja identificador (uma pequena dica: o identificador pode ser encontrado no ambiente global).

Agora que você entende os fundamentos da resolução de identificadores, vamos examinar as várias maneiras como uma variável pode ser declarada.

5,5 Noções básicas sobre tipos de variáveis JavaScript

Em JavaScript, podemos usar três palavras-chave para definir variáveis: var, deixe, e const. Eles diferem em dois aspectos: *mutabilidade* e sua relação com o ambiente lexical.

NOTA A palavra-chave var faz parte do JavaScript desde o início, enquanto deixei e const são adições ES6. Você pode verificar se o seu navegador suporta deixei e const nos seguintes links:
http://mng.bz/CGJ6 e http://mng.bz/uUIT .

5.5.1 Mutabilidade variável

Se fôssemos dividir palavras-chave de declaração de variável por mutabilidade, colocaríamos const de um lado e var e deixei por outro lado. Todas as variáveis definidas com const são imutáveis, o que significa que seu valor pode ser definido apenas uma vez. Por outro lado, variáveis definidas com palavras-chave var e deixei são variáveis comuns típicas, cujo valor podemos alterar quantas vezes forem necessárias.

Agora, vamos nos aprofundar em como const variáveis funcionam e se comportam.

VARIÁVEIS CONST

UMA const “Variável” é semelhante a uma variável normal, com a exceção de que temos que fornecer um valor de inicialização quando ela é declarada e não podemos atribuir um valor completamente novo a ela posteriormente. Hmmm não muito *variável,* é isso?

Const variáveis são frequentemente usadas para dois propósitos ligeiramente diferentes:

- Especificar variáveis que não devem ser reatribuídas (e, no resto do livro, as usaremos principalmente a esse respeito).
- Referenciar um valor fixo, por exemplo, o número máximo de ronins em um esquadrão, MAX_RONIN_COUNT, pelo nome, em vez de usar um número literal como 234. Isso torna nossos programas mais fáceis de entender e manter. Nosso código não está cheio de literais aparentemente arbitrários (234), mas com nomes significativos (MAX_RONIN_COUNT) cujos valores são especificados em apenas um lugar.

Em qualquer caso, porque const as variáveis não devem ser reatribuídas durante a execução do programa, protegemos nosso código contra modificações indesejadas ou acidentais e até possibilitamos que o mecanismo JavaScript faça algumas otimizações de desempenho.

A lista a seguir ilustra o comportamento de const variáveis.

Listagem 5.6 O comportamento de **const** variáveis

```
const firstConst = "samurai";
assert (firstConst === "samurai", "firstConst é um samurai");

tentar{
    firstConst = "ninja";
    fail ("Não deveria estar aqui");
} catch (e) {
    pass ("Ocorreu uma exceção");
}

assert (firstConst === "samurai",
            "firstConst ainda é um samurai!");

const secondConst = {};

secondConst.weapon = "wakizashi";
assert (secondConst.weapon === "wakizashi",
            "Podemos adicionar novas propriedades");
```

Define um const variável e verifica que o valor era atribuído

Tentando atribuir um novo valor para uma variável const lança uma exceção.

Cria uma nova variável const e atribui um novo objeto a ela

Não podemos atribuir um objeto completamente novo ao variável secondConst, mas não há nada que pare de modificarmos o que já temos.

```
const thirdConst = [];
assert (thirdConst.length === 0, "Nenhum item em nossa matriz");

thirdConst.push ("Yoshi");

assert (thirdConst.length === 1, "O array mudou");
```

Exatamente o mesmo coisa vale para matrizes.

Aqui, primeiro definimos um const variável chamada firstConst com um valor samurai e teste se a variável foi inicializada, conforme esperado:

```
const firstConst = "samurai";
assert (firstConst === "samurai", "firstConst é um samurai");
```

Continuamos tentando atribuir um valor completamente novo, ninja, para nosso firstConst variável:

```
tentar{
    firstConst = "ninja";
    fail ("Não deveria estar aqui");
} catch (e) {
    pass ("Ocorreu uma exceção");
}
```

Porque o firstConst variável é, bem, uma constante, não podemos atribuir um novo valor a ela, portanto, o mecanismo JavaScript lança uma exceção sem modificar o valor da variável. Observe que estamos usando duas funções que não usamos até agora: falhou e passar. Esses dois métodos se comportam de forma semelhante ao afirmar método, exceto falhou sempre falha e passar
sempre passa. Aqui, nós os usamos para verificar se ocorreu uma exceção: Se ocorrer uma exceção, o pegar declaração é ativada e o passar método é executado. Se não houver exceção, o falhou método é executado e seremos notificados de que algo não está como deveria estar. Podemos verificar para ver-

se acontecer a exceção, conforme mostrado na figura 5.10.

Em seguida, definimos outro const variável, desta vez inicializando-a para um objeto vazio:

```
const secondConst = {};
```

Agora vamos discutir um recurso importante do const variáveis. Como você já viu, não podemos atribuir um valor completamente novo a um const variável. Mas não há nada nos impedindo de *modificando* o atual. Por exemplo, podemos adicionar novas propriedades ao objeto atual:

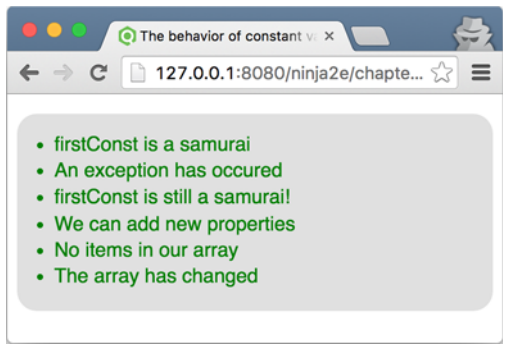


Figura 5.10 Verificando o comportamento de **const** variáveis. Uma exceção ocorre quando tentamos atribuir um valor completamente novo a um **const** variável.

```
secondConst.weapon = "wakizashi";
assert (secondConst.weapon === "wakizashi",
            "Podemos adicionar novas propriedades");
```

Ou, se nosso const variável se refere a uma matriz, podemos modificar essa matriz em qualquer grau:

```
const thirdConst = [];
assert (thirdConst.length === 0, "Nenhum item em nossa matriz");

thirdConst.push ("Yoshi");

assert (thirdConst.length === 1, "O array mudou");
```

E é só isso. const variáveis não são tão complicadas para começar. Você só precisa lembrar que o valor de um const variável pode ser definida apenas na inicialização e não podemos atribuir um valor completamente novo posteriormente. Ainda podemos modificar o valor existente; simplesmente não podemos substituí-lo completamente.

Agora que exploramos a mutabilidade de variáveis, vamos considerar os detalhes das relações entre vários tipos de variáveis e ambientes lexicais.

5.5.2 Palavras-chave de definição de variável e ambientes lexicais

Os três tipos de definições de variáveis - var, deixe, e const - também pode ser categorizado por sua relação com o ambiente lexical (em outras palavras, por seu escopo). Nesse caso, podemos colocar var de um lado, e deixei e const no outro.

USANDO A PALAVRA-CHAVE VAR

Quando usamos o var palavra-chave, a variável é definida na função mais próxima ou ambiente léxico global. (Observe que os blocos são ignorados!) Este é um detalhe antigo do JavaScript que confundiu muitos desenvolvedores vindos de outras linguagens.

Considere a seguinte lista.

Listagem 5.7 usando o **var** palavra chave

```
var globalNinja = "Yoshi";

function reportActivity () {
    var functionActivity = "pular";

    para (var i = 1; i <3; i ++) {
            var forMessage = globalNinja + "" + functionActivity; assert (forMessage === "Yoshi jumping",

                    "Yoshi está pulando dentro do bloco for"); assert (i, "Contador de loop atual:" +
            i);
    }

    assert (i === 3 && forMessage === "Yoshi pulando",
                "Variáveis de loop acessíveis fora do loop");
    }

reportActivity ();
```

Define um global variável, usando var

Define uma variável local de função, usando var

Define dois variáveis no loop for, usando var

Dentro de para loop, nós podemos acessar o bloco variáveis, função variáveis, e variáveis globais- nada surpreendente lá.

Mas as variáveis do loop for também estão acessíveis fora do loop for.

```
assert (typeof functionActivity === "undefined"
        && typeof i === "undefined" && typeof forMessage === "undefined", "Não podemos ver variáveis   de função fora de uma
        função");
```

Normalmente, nenhuma das variáveis de função são acessíveis fora da função.

Começamos definindo uma variável global, globalNinja, que é seguido pela definição de um reportActivity função que faz um loop duas vezes e nos notifica sobre a atividade de salto de nosso globalNinja. Como você pode ver, dentro do corpo do pra loop, podemos normalmente acessar ambas as variáveis de bloco (eu e forMessage), as variáveis de função (functionActivity), e as variáveis globais (globalNinja).

Mas o que é estranho com JavaScript, e o que confunde muitos desenvolvedores vindos de outras linguagens, é que podemos acessar as variáveis definidas com blocos de código mesmo fora desses blocos:

```
assert (i === 3 && forMessage === "Yoshi pulando",
            "Variáveis   de loop acessíveis fora do loop");
```

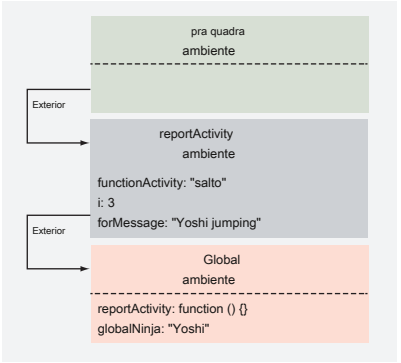
Isso decorre do fato de que as variáveis declaradas com a palavra-chave var são sempre registrados na função mais próxima ou no ambiente léxico global, sem prestar atenção aos blocos. A Figura 5.11 mostra essa situação, mostrando o estado dos ambientes lexicais após a segunda iteração do pra loop no reportActivity função.

```
var globalNinja = "Yoshi";

function reportActivity () {
  var functionActivity = "pular";

  para (var i = 1; i <3; i ++) {
      var forMessage = globalNinja + "" + functionActivity; assert (forMessage = "Yoshi jumping",
                  "Yoshi está pulando dentro do bloco for"); assert (i, "Contador de loop
      atual:" + i);
  }
    assert (i == 3 && forMessage == "Yoshi jumping",
              "Variáveis   de loop acessíveis fora do loop");
}

  reportActivity ();
    . . .
```

Global ambiente

Função ambiente

Quadra ambiente



O estado dos ambientes lexicais ao terminar a segunda iteração do loop for

Figura 5.11 Ao definir variáveis com a palavra-chave **var,** uma variável é definida na função mais próxima ou ambiente global (enquanto ignora ambientes de bloco). No nosso caso, as variáveis **forMessage** e **eu** estão registrados no **reportActivity** ambiente (o ambiente funcional mais próximo), mesmo que estejam contidos em um **pra** ciclo.

Aqui temos três ambientes lexicais:

- O ambiente global em que o globalNinja variável é registrada (porque esta é a função mais próxima ou ambiente léxico global) O reportActivity ambiente, criado no reportActivity invocação de função, que contém
- o functionActivity, i, e forMessage variáveis, porque são definidas com a palavra-chave var, e este é o ambiente funcional mais próximo

- O pra ambiente de bloco, que está vazio, porque var- variáveis definidas ignoram blocos (mesmo quando contidos neles)

Como esse comportamento é um pouco estranho, a versão ES6 do JavaScript oferece duas novas palavras-chave de declaração de variável: deixei e const.

USANDO LET E CONST PARA ESPECIFICAR VARIÁVEIS COM ESCOPO DE BLOCO

diferente var, que define a variável na função mais próxima ou ambiente léxico global, o deixei e const palavras-chave são mais diretas. Eles definem variáveis no ambiente léxico mais próximo (que pode ser um ambiente de bloco, um ambiente de loop, um ambiente de função ou até mesmo o ambiente global). Podemos usar deixei e

const para definir variáveis com escopo de bloco, escopo de função e escopo global.

Vamos reescrever nosso exemplo anterior para usar const e deixei.

Listagem 5.8 Usando **const** e **deixei** palavras-chave

Define uma variável global, usando const. Variáveis globais const são geralmente escrito em maiúsculas.

Define um função local variável, usando const

```
const GLOBAL_NINJA = "Yoshi";

function reportActivity () {
    const functionActivity = "pulando";
```

Dentro do loop for, podemos acessar as variáveis de bloco, variáveis de função e variáveis globais - nada de surpreendente aí.

Define dois variáveis em o loop for, usando let

```
    para (deixe i = 1; i <3; i ++) {
        deixe forMessage = GLOBAL_NINJA + "" + functionActivity; assert (forMessage === "Yoshi jumping",
                "Yoshi está pulando dentro do bloco for"); assert (i, "Contador de loop atual:" +
        i);
    }
```

Agora o variáveis de o laço for não são acessível fora da para loop.

```
    assert (typeof i === "undefined" && typeof forMessage === "undefined",
            "Variáveis de loop não acessíveis fora do loop");
}

reportActivity ();
assert (typeof functionActivity === "undefined"
        && typeof i === "undefined" && typeof forMessage === "undefined", "Não podemos ver variáveis de função fora de uma
        função");
```

Normalmente, nenhuma das variáveis de função são acessível fora da função.

A Figura 5.12 ilustra a situação atual, ao finalizar a execução da segunda iteração do pra loop no reportActivity função. Mais uma vez, temos três ambientes lexicais: o ambiente global (para código global fora de todas as funções e blocos), o reportActivity ambiente ligado ao reportActivity função e o ambiente de bloco para o pra corpo do laço. Mas porque estamos usando deixei

e const palavras-chave, as variáveis são definidas em seu ambiente lexical mais próximo; a GLOBAL_NINJA variável é definida no ambiente global, o functionActivity variável no reportActivity ambiente, e o eu e forMessage variáveis no pra ambiente de bloco.

Agora isso const e deixei foram introduzidos, muitos novos desenvolvedores de JavaScript que vieram recentemente de outras linguagens de programação podem ficar em paz. JavaScript

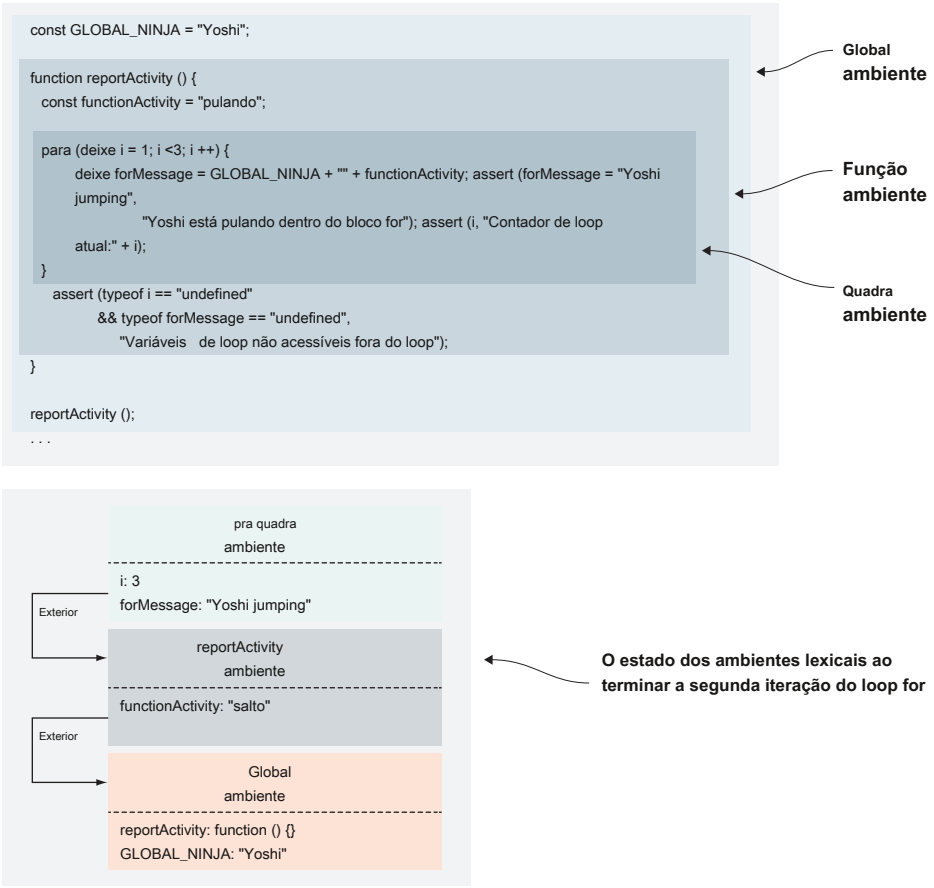


Figura 5.12 Ao definir variáveis com palavras-chave **deixei** e **const,** uma variável é definida no ambiente mais próximo. Em nosso caso, variáveis **forMessage** e **eu** estão registrados no **pra** ambiente de bloco, a variável **functionActivity** no **reportActivity** ambiente, e o **GLOBAL_NINJA** variável no ambiente global (em todos os casos, o ambiente mais próximo da respectiva variável).

finalmente suporta as mesmas regras de escopo que outras linguagens semelhantes a C. Por esse motivo, a partir deste ponto deste livro, quase sempre usamos const e deixei ao invés de var.

Agora que entendemos como os mapeamentos de identificadores são mantidos em ambientes lexicais e como os ambientes lexicais estão vinculados à execução do programa, vamos discutir o processo exato pelo qual os identificadores são definidos em ambientes lexicais. Isso nos ajudará a entender melhor alguns bugs comuns.

5.5.3 Registrando identificadores em ambientes lexicais

Um dos princípios básicos por trás do design de JavaScript como linguagem de programação foi sua facilidade de uso. Esse é um dos principais motivos para não especificar tipos de retorno de função, tipos de parâmetro de função, tipos de variável e assim por diante. E você já sabe que o código JavaScript é executado linha por linha, de maneira direta. Considere o seguinte:

```
firstRonin = "Kiyokawa";
secondRonin = "Kondo";
```

O valor que Kiyokawa é atribuído ao identificador firstRonin, e então o valor Kondo é atribuído ao identificador secondRonin. Não há nada de estranho nisso, certo? Mas dê uma olhada em outro exemplo:

```
const firstRonin = "Kiyokawa"; verificar (firstRonin);

verificação de função (ronin) {
    assert (ronin === "Kiyokawa", "O ronin foi verificado!");
}
```

Neste caso, atribuímos o valor Kiyokawa para o identificador firstRonin, e então chamamos o Verifica função com o identificador firstRonin como um parâmetro. Mas espere um segundo - se o código for executado linha por linha, devemos ser capazes de chamar o Verifica função? A execução do nosso programa não atingiu sua declaração, então o mecanismo JavaScript nem deveria saber disso.

Mas se verificarmos, como mostrado na figura 5.13, você verá que está tudo bem e bem. JavaScript não é muito exigente sobre onde definimos nossas funções. Podemos escolher colocar declarações de função antes ou mesmo depois de suas respectivas chamadas. Isso não é algo que o desenvolvedor precise se preocupar.

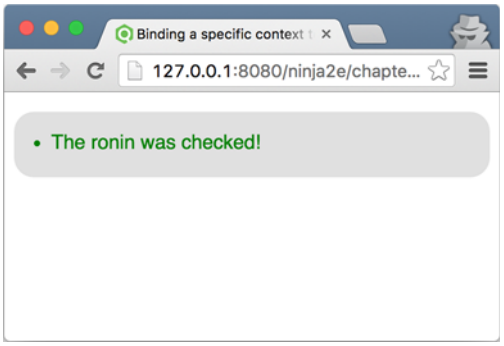


Figura 5.13 A função é de fato visível, mesmo antes que a execução alcance sua definição.

O PROCESSO DE REGISTRO DE IDENTIFICADORES

Mas facilidade de uso à parte, se o código é executado linha por linha, como o motor JavaScript sabia que uma função chamada

Verifica existe? Acontece que o mecanismo JavaScript “trapaceia” um pouco e que a execução do código JavaScript ocorre em duas fases.

A primeira fase é ativada sempre que um novo ambiente léxico é criado. Nesta fase, o código não é executado, mas o mecanismo JavaScript visita e registra todas as variáveis e funções declaradas no ambiente léxico atual. A segunda fase, a execução do JavaScript, começa depois que isso é realizado; o comportamento exato depende do tipo de variável (let, var, const, declaração de função) e o tipo de ambiente (global, função ou bloco).

O processo é como se segue:

1. Se estamos criando um ambiente de função, o implícito argumentos identificador é criado, junto com todos os parâmetros de função formais e seus valores de argumento. Se estivermos lidando com um ambiente sem função, esta etapa será ignorada.
2. Se estivermos criando um ambiente global ou de função, o código atual será verificado (sem entrar no corpo de outras funções) em busca de declarações de função (mas não de expressões de função ou funções de seta!). Para cada declaração de função descoberta, uma nova função é criada e associada a um identificador no ambiente com o nome da função. Se esse nome de identificador já existir, seu valor será sobrescrito. Se estivermos lidando com ambientes de bloco, esta etapa será ignorada.
3. O código atual é verificado em busca de declarações de variáveis. Em ambientes funcionais e globais, todas as variáveis declaradas com a palavra-chave var e definidas fora de outras funções (mas podem ser colocadas dentro blocos!) são encontrados, e todas as variáveis declaradas com as palavras-chave deixei e const definido fora de outras funções e blocos são encontrados. Em ambientes de bloco, o código é verificado apenas para variáveis declaradas com as palavras-chave deixei e const, diretamente no bloco atual. Para cada variável descoberta, se o identificador não existir no ambiente, o identificador é registrado e seu valor inicializado para Indefinido. Mas se o identificador existir, ele será deixado com seu valor.

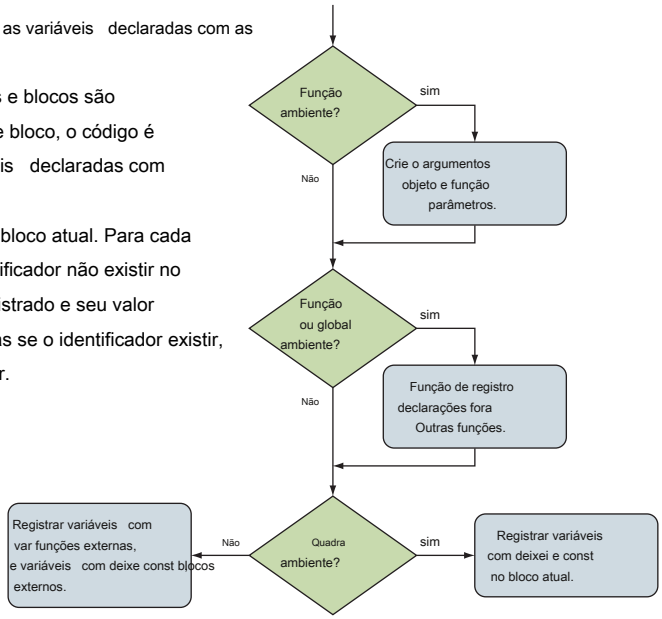


Figura 5.14 O processo de registro de identificadores, dependendo do tipo de ambiente

Essas etapas estão resumidas na figura 5.14.

Agora examinaremos as implicações dessas regras. Você verá alguns conun-

tambores que podem levar a bugs estranhos que são fáceis de

criar, mas difícil de entender. Vamos começar explicando por que podemos chamar uma função antes mesmo de ela ser declarada.

FUNÇÕES DE CHAMADA ANTES DE SUAS DECLARAÇÕES

Um dos recursos que tornam o JavaScript agradável de usar é que a ordem das definições das funções não importa. Aqueles que usaram Pascal podem não se lembrar com carinho de seus rígidos requisitos estruturais. Em JavaScript, podemos chamar uma função antes mesmo de ela ser declarada formalmente. Confira a lista a seguir.

Listagem 5.9 Acessando uma função antes de sua declaração

```
assert (typeof fun === "função",
    "diversão é uma função, embora sua definição ainda não seja alcançada!");

assert (typeof myFunExp === "undefined",
    "Mas não podemos acessar expressões de função");

assert (typeof myArrow === "undefined",
    "Nem funções de seta");

função diversão () {}

var myFunExpr = function () {}; var minhaArrow = (x) =>
x;
```

Podemos acessar uma função que ainda não foi definida, se a função for definida como um declaração de função.

Não podemos acessar funções que são definidas como função expressões ou funções de seta.

A função divertida é definida como uma declaração de função.

myFunExpr aponta para uma expressão de função e myArrow para uma função de seta.

Podemos acessar a função Diversão mesmo antes de defini-lo. Podemos fazer isso porque Diversão é definido como uma declaração de função, e a segunda etapa (listada anteriormente nesta seção) indica que as funções criadas com as declarações de função são criadas e seus identificadores registrados quando o ambiente léxico atual é criado, *antes* qualquer código JavaScript é executado. Então, mesmo antes de começarmos a executar nosso afirmar Ligar para Diversão função já existe.

O mecanismo JavaScript faz isso para facilitar as coisas para nós como desenvolvedores, permitindo-nos encaminhar funções de referência e não nos sobrecarregando com uma ordem exata para colocar funções. As funções já existem no momento em que nosso código começa a ser executado.

Observe que isso é válido apenas para declarações de funções. Expressões de função e funções de seta não fazem parte deste processo e são criadas quando a execução do programa atinge suas definições. É por isso que não podemos acessar o myFunExp e minha flecha funções.

FUNÇÕES DE SUBSTITUIÇÃO

O próximo enigma a ser resolvido é o problema de substituir os identificadores de função. Vamos dar uma olhada em outro exemplo.

Listagem 5.10 Substituindo identificadores de função

```
assert (typeof fun === "função", "Acessamos a função");

var fun = 3;

assert (typeof fun === "number", "Agora vamos acessar o número");

função diversão () {}

assert (typeof fun === "número", "Ainda um número");
```

diversão se refere a uma função.

Define uma variável divertida e atribui um número a ela

diversão se refere a um número.

Uma declaração de função divertida

diversão ainda se refere a um número.

Neste exemplo, uma declaração de variável e uma declaração de função têm o mesmo nome: Diversão. Se você executar este código, verá que ambos afirmar s passar. Em primeiro afirmar, o identificador Diversão refere-se a uma função; e no segundo e terceiro, Diversão refere-se a um número.

Esse comportamento é uma consequência direta das etapas executadas ao registrar os identificadores. Na segunda etapa do processo delineado, funções definidas com declarações de função são criadas e associadas a seus identificadores antes de qualquer código ser avaliado; e na terceira etapa, as declarações de variáveis são processadas, e o valor Indefinido está associado a identificadores que ainda não foram encontrados no ambiente atual.

Neste caso, porque o identificador Diversão foi encontrado na segunda etapa, quando as declarações de função são registradas, o valor Indefinido não é atribuído à variável Diversão. É por isso que a primeira afirmação, testando se Diversão é uma função, passa. Depois disso, temos uma declaração de atribuição, var fun = 3, que atribui o número 3 ao identificador Diversão. Ao fazer isso, perdemos a referência à função e, a partir de então, o identificador Diversão refere-se a um número.

Durante a execução do programa real, as declarações de função são ignoradas, de modo que a definição do Diversão função não tem nenhum impacto sobre o valor do Diversão identificador.

Içamento variável

Se você leu vários blogs ou livros sobre JavaScript explicando a resolução de identificadores, provavelmente encontrou o termo *elevação* —Por exemplo, declarações de variáveis e funções são içadas, ou *levantado,* para o topo de uma função ou escopo global.

Como você viu, porém, essa é uma visão simplista. Variáveis e declarações de funções tecnicamente não são “movidas” para lugar nenhum. Eles são visitados e registrados em ambientes lexicais antes de qualquer código ser executado. Embora *elevação,* como geralmente é definido, é o suficiente para fornecer uma compreensão básica de como o escopo do JavaScript funciona. Nós fomos muito mais fundo do que isso ao observar os ambientes lexicais, dando mais um passo no caminho para nos tornarmos um verdadeiro ninja do JavaScript.

Na próxima seção, todos os conceitos que exploramos até agora neste capítulo o ajudarão a entender melhor os encerramentos.

5,6 Explorando como funcionam os fechamentos

Começamos este capítulo com closures, um mecanismo que permite a uma função acessar todas as variáveis que estão no escopo quando a própria função é criada. Você também viu algumas maneiras pelas quais os encerramentos podem ajudá-lo - por exemplo, permitindo-nos imitar variáveis de objeto privado ou tornando nosso código mais elegante ao lidar com retornos de chamada.

Os fechamentos são irrevogavelmente acoplados aos escopos. Os fechamentos são um efeito colateral direto da maneira como as regras de escopo funcionam em JavaScript. Portanto, nesta seção, revisitaremos os exemplos de fechamento do início do capítulo. Mas desta vez, você aproveitará os contextos de execução e ambientes lexicais que permitirão que você entenda como os fechamentos funcionam nos bastidores.

5.6.1 Revisitando variáveis privadas imitando com fechamentos

Como você já viu, os fechamentos podem nos ajudar a imitar as variáveis privadas. Agora que temos um entendimento sólido de como as regras de escopo funcionam em JavaScript, vamos rever o exemplo de variáveis privadas. Desta vez, vamos nos concentrar em contextos de execução e ambientes lexicais. Só para facilitar, vamos repetir a listagem.

Listagem 5.11 Aproximar variáveis privadas com fechamentos

Declara uma variável dentro do construtor. Porque o escopo do variável é limitada a dentro do construtor, é uma variável "privada".

O método de incremento para o valor. Como o valor é privado, ninguém pode estragar tudo pelas nossas costas; eles estão limitados ao acesso que damos a eles por meio de métodos.

Um acessor método para as fintas contador

Chama a finta método, que incrementa o contagem do número de vezes que nosso ninja fintou

Verifica que nós não pode chegar na variável diretamente

Testa que o incremento foi realizado

Quando criamos um novo objeto ninja2 com o construtor Ninja, o objeto ninja2 obtém sua própria variável fintas.

```
function Ninja () {
    var feints = 0;
    this.getFeints = function () {
        retornar fintas;
    };
    this.feint = function () {
        fintas ++;
    };
}
var ninja1 = novo Ninja ();
assert (ninja1.feints === undefined,
        "E os dados privados estão inacessíveis para nós."); ninja1.feint ();

assert (ninja1.getFeints () === 1,
        "Podemos acessar a contagem de finta interna.");

var ninja2 = novo Ninja (); assert (ninja2.getFeints () === 0,

        "O segundo objeto ninja obtém sua própria variável fintas.");
```

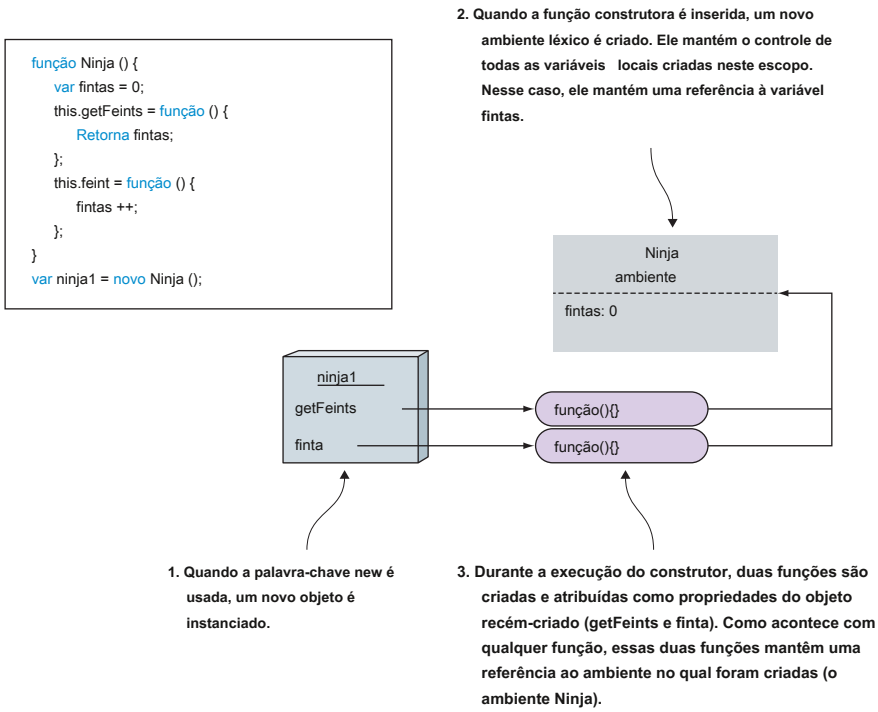


Figura 5.15 Variáveis privadas são realizadas como fechamentos que são criados por métodos de objeto definidos no construtor.

Agora vamos analisar o estado do aplicativo após o primeiro Ninja objeto foi criado, conforme mostrado na figura 5.15. Podemos usar nosso conhecimento das complexidades da resolução de identificadores para entender melhor como os fechamentos entram em jogo nessa situação. Construtores de JavaScript são funções invocadas com a palavra-chave novo. Portanto, *cada*
vez que invocamos uma função construtora, criamos um *novo* ambiente léxico, que controla as variáveis locais do construtor. Neste exemplo, um novo Ninja ambiente que mantém o controle do fintas variável é criada.

Além disso, sempre que uma função é criada, ela mantém uma referência ao ambiente léxico no qual foi criada (por meio de um [[Ambiente]] propriedade). Neste caso, dentro do Ninja função construtora, criamos duas novas funções: getFeints e finta, que obtêm uma referência ao Ninja ambiente, porque este é o ambiente em que foram criados.

O getFeints e finta funções são atribuídas como métodos do recém-criado
ninja objeto (que, se você se lembra do capítulo anterior, é acessível através do esta palavra-chave). Portanto, getFeints e finta estará acessível de fora do Ninja função construtora, que por sua vez leva ao fato de que você efetivamente criou um fechamento em torno do fintas variável.

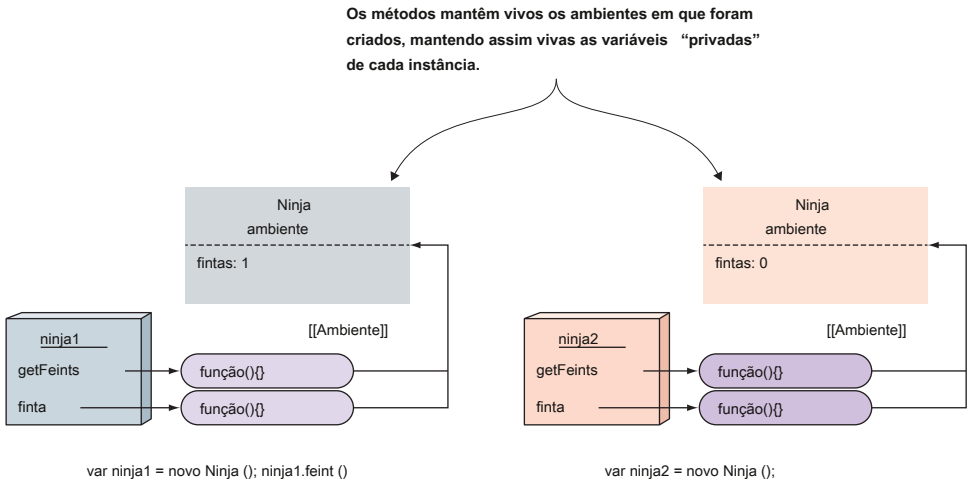


Figura 5.16 Os métodos de cada instância criam fechamentos em torno das variáveis de instância "privadas".

Quando criamos outro Ninja objeto, o ninja2 objeto, todo o processo é repetido. A Figura 5.16 mostra o estado do aplicativo após a criação do segundo
Ninja objeto.

Cada objeto criado com o Ninja construtor obtém seus próprios métodos (o ninja1.getFeints método é diferente do ninja2.getFeints método) que fecha em torno das variáveis definidas quando o construtor foi chamado. Essas variáveis "privadas" são acessíveis apenas por meio de métodos de objeto criados dentro do construtor, e não diretamente!

Agora vamos ver como as coisas acontecem ao fazer o ninja2.getFeints () chamar. A Figura 5.17 mostra os detalhes.

Antes de fazer o ninja2.getFeints () chamada, nosso mecanismo JavaScript está executando o código global. A execução do nosso programa está no contexto de execução global, que também é o único contexto na pilha de execução. Ao mesmo tempo, o único ambiente léxico ativo é o ambiente global, o ambiente associado ao contexto de execução global.

Ao fazer o ninja2.getFeints () ligar, estamos ligando para o getFeints método do ninja2 objeto. Porque cada chamada de função causa a criação de um novo contexto de execução, um novo getFeints o contexto de execução é criado e enviado para a pilha de execução. Isso também leva à criação de um novo getFeints Ambiente léxico, que normalmente é usado para rastrear variáveis definidas nesta função. Além disso, o getFeints ambiente léxico, como seu ambiente externo, obtém o ambiente no qual o getFeints função foi criada, o Ninja ambiente que estava ativo quando o ninja2 objeto foi construído.

```
função Ninja () {
    var fintas = 0;
    this.getFeints = função () {
        Retorna fintas;
    };
    this.feint = função () {
        fintas ++;
    };
}

var ninja1 = novo Ninja ();
var ninja1.feint ();

var ninja2 = novo Ninja (); ninja2.getFeints
();
```

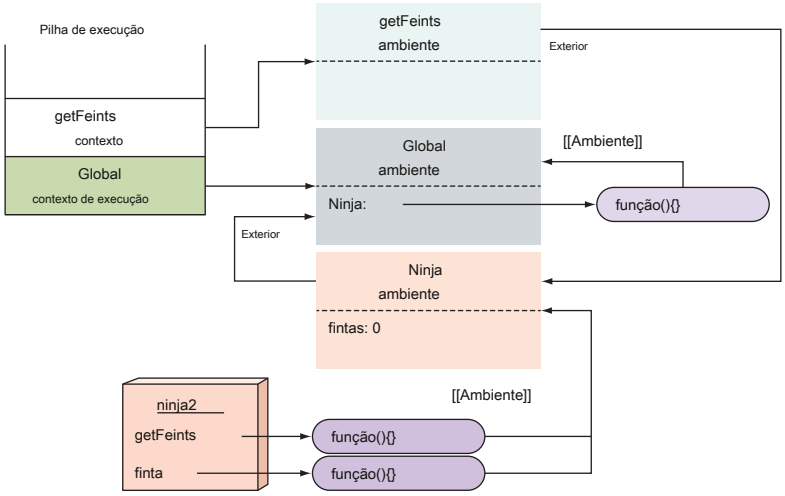


Figura 5.17 O estado dos contextos de execução e ambientes lexicais ao realizar o **ninja2.getFeints ()** chamar. Uma nova **getFeints** ambiente é criado que tem o ambiente da função de construtor em que **ninja2** foi criado como seu ambiente externo. **getFeints** pode acessar o "privado" **fintas** variável.

Agora vamos ver como as coisas acontecem quando tentamos obter o valor do fintas variável. Primeiro, o atualmente ativo getFeints ambiente lexical é consultado. Porque não definimos nenhuma variável no getFeints função, este ambiente léxico está vazio e nosso alvo fintas variável não será encontrada lá. Em seguida, a pesquisa continua no *exterior* ambiente do ambiente léxico atual - no nosso caso, o Ninja ambiente é ativo durante a construção do ninja2 objeto. Desta vez, o Ninja ambiente tem uma referência ao fintas variável, e a pesquisa é feita. É simples assim.

Agora que entendemos o papel que os contextos de execução e ambientes lexicais desempenham ao lidar com fechamentos, gostaríamos de voltar nossa atenção para variáveis "privadas"

e por que continuamos colocando aspas entre eles. Como você já deve ter percebido, essas variáveis "privadas" não são propriedades privadas do objeto, mas são variáveis mantidas vivas pelos métodos de objeto criados no construtor. Vamos dar uma olhada em um efeito colateral interessante disso.

5.6.2 Advertência sobre variáveis privadas

Em JavaScript, não há nada que nos impeça de atribuir propriedades criadas em um objeto para outro objeto. Por exemplo, podemos facilmente reescrever o código da listagem 5.11 em algo como o seguinte.

Listagem 5.12 Variáveis privadas são acessadas por meio de funções, não por meio de objetos!

```
function Ninja () {
    var feints = 0;
    this.getFeints = function () {
        retornar fintas;
    };
    this.feint = function () {
        fintas ++;
    };
}
var ninja1 = novo Ninja (); ninja1.feint ();


var impostor = {};
imposter.getFeints = ninja1.getFeints;

assert (imposter.getFeints () === 1,
        "O impostor tem acesso à variável fintas!");
```

Torna a função getFeints de ninja1 acessível por meio do impostor

Verifica se podemos acessar o suposto variável privada de ninja1

Esta lista modifica o código-fonte de uma forma que atribui o ninja1.getFeints
método para um completamente novo impostor
objeto. Então, quando chamamos o getFeints
funcionar no impostor objeto, testamos se podemos
acessar o valor da variável fintas
criado quando ninja1 foi instanciado, provando assim que estamos falsificando toda essa coisa de variável "privada". Veja a figura 5.18.

Este exemplo ilustra que não existem variáveis de objeto privado em JavaScript, mas que podemos usar closures criados por métodos de objeto para ter uma alternativa "boa o suficiente". Ainda assim, embora não seja a coisa real, muitos desenvolvedores acham útil essa forma de ocultar informações.

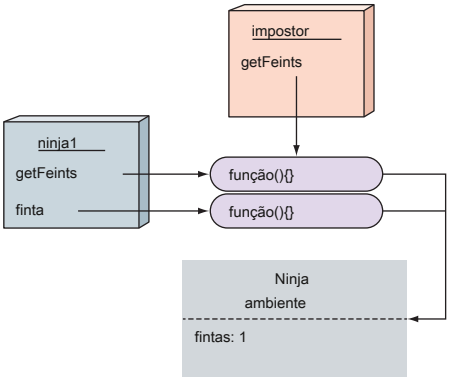


Figura 5.18 Podemos acessar as variáveis "privadas" por meio de funções, mesmo que essa função esteja anexada a outro objeto!

5.6.3 Revisitando o exemplo de encerramentos e retornos de chamada

Vamos voltar ao nosso exemplo de animações simples com cronômetros de retorno de chamada. Desta vez, vamos animar dois objetos, conforme mostrado na lista a seguir.

Listagem 5.13 Usando um fechamento em um **cronômetro** intervalo de retorno

```
<div id = "box1"> Primeira caixa </div>
<div id = "box2"> Segunda caixa </div>
<script>
    function animateIt (elementId) {
        var elem = document.getElementById (elementId); var tick = 0;

        var timer = setInterval (function () {
            if (assinale <100) {
                elem.style.left = elem.style.top = tick + "px"; tick ++;

            }
            outro {
                clearInterval (temporizador);
                assert (tick === 100,
                        "Assinale acessado por meio de um fechamento."); afirmar
                (elem,
                        "Elemento também acessado por meio de um fechamento."); assert
                (cronômetro,
                        "Referência do temporizador também obtida por meio de um fechamento." );
            }
        }, 10);
    }
    animateIt ("box1");
    animateIt ("box2");
</script>
```

Como você viu no início do capítulo, usamos fechamentos para simplificar a animação de vários objetos em nossas páginas. Mas agora vamos considerar os ambientes lexicais, como mostrado na figura 5.19.

Cada vez que ligamos para o animateIt função, um novo ambiente léxico de função é criado AC que mantém o controle do conjunto de variáveis importantes para a animação (o elementId; elem, o elemento que está sendo animado; marcação, o número atual de carrapatos; e cronômetro, o ID do cronômetro que faz a animação). Esse ambiente será mantido ativo enquanto houver pelo menos uma função que trabalhe com suas variáveis por meio de fechamentos. Neste caso, o navegador manterá vivo o setInterval callback até que chamemos o clearInterval função. Mais tarde, quando um intervalo expira, o navegador chama o retorno de chamada correspondente - e com ele, por meio dos fechamentos, vêm as variáveis definidas quando o retorno de chamada foi criado. Isso nos permite evitar o problema de

mapeando manualmente o retorno de chamada e as variáveis ativas DEF , assim, simplificando significativamente nosso código.

Isso é tudo o que temos a dizer sobre encerramentos e escopos. Agora recapitule este capítulo e nos encontre no próximo, onde exploraremos dois conceitos ES6 completamente novos, geradores e promessas, que podem ajudar ao escrever código assíncrono.

```
< id div = "box1"> Primeira caixa </ div >
< id div = "box2"> Segunda caixa </ div >
< roteiro >
função animateIt (elementId) {
      var elem = document.getElementById (elementId);
      var tick = 0;
        var timer = setInterval ( função () {
          E se (marque <100) {
              var posição = tick + "px"; elem.style.left =
              posição;
              elem.style.top = posição;
              tick ++;
          }
          outro {
            clearInterval (temporizador);
          }
      }, 10);
}
animateIt ("box1");
animateIt ("box2");
</ roteiro >
```

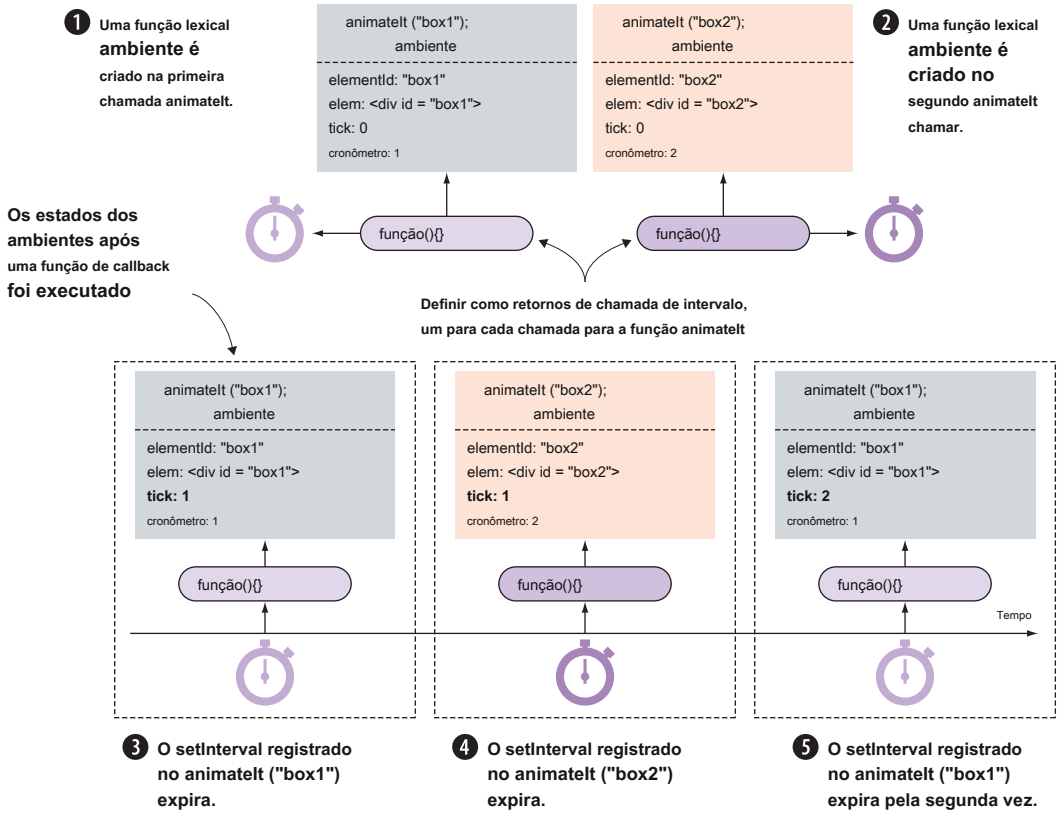


Figura 5.19 Ao criar vários encerramentos, fazemos muitas coisas ao mesmo tempo. Cada intervalo de tempo expira, a função de retorno de chamada reativa o ambiente que estava ativo no momento da criação do retorno de chamada. O encerramento de cada retorno de chamada controla automaticamente seu próprio conjunto de variáveis.

5,7 Resumo

- Os fechamentos permitem que uma função acesse todas as variáveis que estão no escopo quando a própria função foi definida. Eles criam uma "bolha de segurança" da função e das variáveis que estão no escopo no ponto de definição da função. Dessa forma, a função tem tudo o que precisa para ser executada, mesmo que o escopo em que a função foi criada já tenha se esgotado.

- Podemos usar encerramentos de função para esses usos avançados:

  - Imitar variáveis de objeto privado, fechando sobre variáveis de construtor por meio de encerramentos de método

  - Lidar com retornos de chamada, de uma forma que simplifica significativamente nosso código

- Os mecanismos JavaScript rastreiam a execução da função por meio de uma pilha de contexto de execução (ou uma pilha de chamadas). Cada vez que uma função é chamada, um novo contexto de execução de função é criado e colocado na pilha. Quando a execução de uma função termina, o contexto de execução correspondente é retirado da pilha.
- Os mecanismos JavaScript rastreiam identificadores com ambientes lexicais (ou coloquialmente, escopos).

- Em JavaScript, podemos definir variáveis com escopo global, com escopo de função e com escopo de bloco par.

- Para definir variáveis, usamos var, deixe, e const palavras-chave:

  - O var palavra-chave define uma variável na função mais próxima ou escopo global (enquanto ignora os blocos).

  - deixei e const palavras-chave definem uma variável no escopo mais próximo (incluindo blocos), permitindo-nos criar variáveis com escopo de bloco, algo que não era possível no JavaScript pré-ES6. Além disso, a palavra-chave const nos permite definir "variáveis" cujo valor pode ser atribuído apenas uma vez.

- Os fechamentos são apenas um efeito colateral das regras de escopo do JavaScript. Uma função pode ser chamada mesmo quando o escopo no qual ela foi criada já não existe mais.

5,8 Exercícios

1 Fechamentos permitem funções para

uma Acesse variáveis externas que estão no escopo quando a função é definida

b Acesse variáveis externas que estão no escopo quando a função é chamada

2 Fechamentos vêm com

uma Custos de tamanho de código

b Custos de memória

c Custos de processamento

3 No exemplo de código a seguir, marque os identificadores acessados por meio de fechamentos:

```
function Samurai (nome) {
    var arma = "katana";
```

```
    this.getWeapon = function () {
        arma de retorno;
    };

    this.getName = function () {
        nome de retorno;
    }

    esta.mensagem = nome + "empunhando uma" + arma;

    this.getMessage = function () {
        return this.message;
    }
}

var samurai = novo Samurai ("Hattori");

samurai.getWeapon ();
samurai.getName ();
samurai.getMessage ();
```

4 No código a seguir, quantos contextos de execução são criados e qual é o maior tamanho da pilha de contexto de execução?

```
função executar (ninja) {
    sneak (ninja);
    infiltrar (ninja);
}

função sneak (ninja) {
    retornar ninja + "esquiva";
}

função infiltrar (ninja) {
    retornar ninja + "infiltrando";
}

perfom ("Kuma");
```

5 Qual palavra-chave em JavaScript nos permite definir variáveis que não podem ser reassinado com um valor completamente novo?

6 Qual é a diferença entre var e deixei?

7 Onde e por que o código a seguir lançará uma exceção?

```
getNinja ();
getSamurai ();

function getNinja () {
    retornar "Yoshi";
}

var getSamurai = () => "Hattori";
```

6 Funções para o futuro: geradores e promessas

Este capítulo cobre

- Continuando a execução da função com geradores
- Lidando com tarefas assíncronas com promessas
- Alcançar código assíncrono elegante combinando geradores e promessas

Nos três capítulos anteriores, nos concentramos nas funções, especificamente em como definir funções e como usá-las com ótimo efeito. Embora tenhamos introduzido alguns recursos do ES6, como funções de seta e escopos de bloco, estivemos explorando principalmente os recursos que fazem parte do JavaScript há algum tempo. Este capítulo aborda a vanguarda do ES6 apresentando *geradores* e *promessas,* dois recursos de JavaScript completamente novos.

NOTA Geradores e promessas são introduzidos pelo ES6. Você pode verificar o suporte do navegador atual em http://mng.bz/sOs4 e http://mng.bz/Du38.