

2.3.3 Tratamento de eventos

A ideia principal por trás do tratamento de eventos é que, quando um evento ocorre, o navegador chama o tratador de eventos associado. Como já mencionamos, devido ao modelo de execução de thread único, apenas um único manipulador de eventos pode ser executado por vez. Todos os eventos a seguir são processados somente após a execução do manipulador de eventos atual estar totalmente concluída!

Vamos voltar ao aplicativo da listagem 2.1. A Figura 2.9 mostra um exemplo de execução em que um usuário rápido moveu e clicou com o mouse.

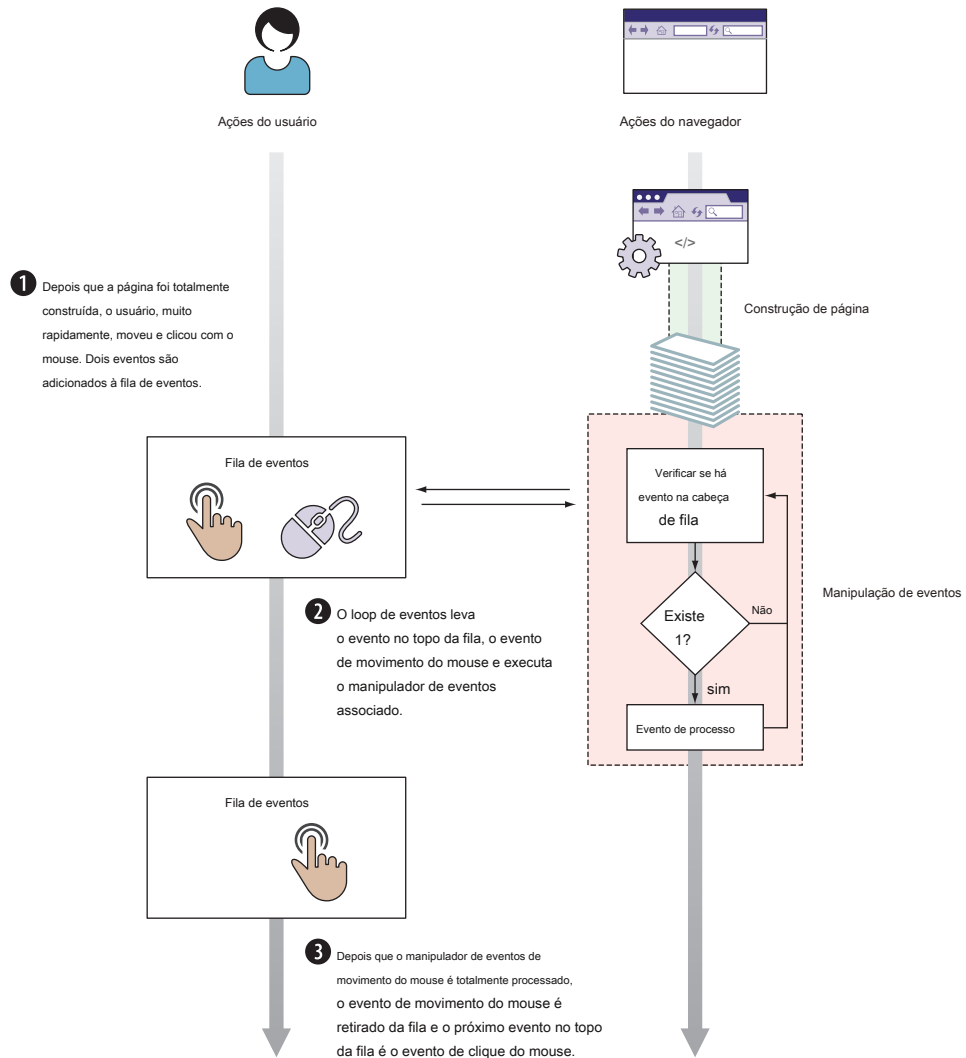


Figura 2.9 Exemplo de uma fase de tratamento de eventos em que dois eventos— **mousemove** e **click** — são manuseados

Vamos examinar o que está acontecendo aqui. Como resposta a essas ações do usuário, o navegador coloca o mousemove e clique eventos na fila de eventos na ordem em que eles ocorreram: primeiro o mousemove evento e então o clique evento **B**.

Na fase de tratamento de eventos, o loop de eventos verifica a fila, vê que há um mousemove evento na frente da fila e executa o evento associado manipulador **C**. Enquanto o mousemove manipulador está sendo processado, o clique evento espera na fila por sua vez. Quando a última linha do mousemove função de manipulador tem fim-executado e o mecanismo JavaScript sai da função de manipulador, o mousemove evento está totalmente processado **D**, e o loop de eventos verifica novamente a fila. Desta vez, na frente da fila, o loop de eventos encontra o clique evento e processa-o. Uma vez a clique handler terminou a execução, não há novos eventos na fila e o loop de eventos continua em loop, esperando que novos eventos sejam tratados. Este loop continuará em execução até que o usuário feche o aplicativo da web.

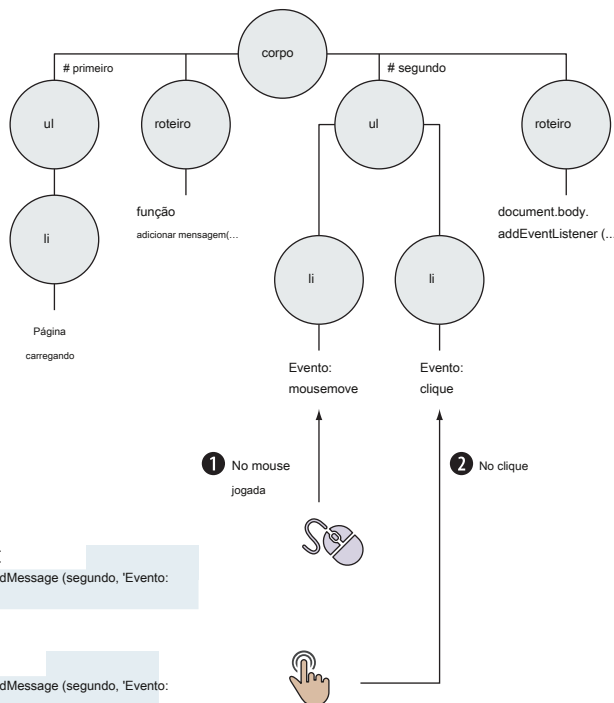
Agora que temos uma noção das etapas gerais que acontecem na fase de tratamento de eventos, vamos ver como essa execução influencia o DOM (figura 2.10). A execução de

```
<ul id = "first"> </ul>
```

```
<script>
```

```
function addMessage (elemento, mensagem) {
  var messageElement = document.createElement ("li"); messageElement.textContent =
  mensagem;
  element.appendChild (messageElement);
}
```

```
</script>
```



```
<ul id = "second"> </ul>
```

```
<script>
```

```
document.body.addEventListener ('mousemove', function () {
  var segundo = document.getElementById ('segundo'); addMessage (segundo, 'Evento:
  mousemove');
});
```

```
document.body.addEventListener ('click', function () {
  var segundo = document.getElementById ('segundo'); addMessage (segundo, 'Evento:
  clique');
});
```

```
</script>
```

Figura 2.10 O DOM do aplicativo de exemplo após lidar com o **mousemove** e **clique** eventos

a mousemove handler seleciona o segundo elemento da lista com ID segundo e, usando a adicionar mensagem função, adiciona um novo elemento de item de lista B com o texto " Evento: mousemove ". Uma vez que a execução do mousemove o manipulador está concluído, o loop de eventos executa o clique manipulador, o que leva à criação de outro elemento de item de lista

C , que também é anexado ao segundo elemento da lista com o ID segundo.

Armado com esta compreensão sólida do ciclo de vida dos aplicativos da web do lado do cliente, na próxima parte do livro, começaremos a nos concentrar na linguagem JavaScript, aprendendo os meandros das funções.

2,4 Resumo

- O código HTML recebido pelo navegador é usado como um blueprint para criar o DOM, uma representação interna da estrutura de um aplicativo da web do lado do cliente. Usamos o código JavaScript para modificar
- dinamicamente o DOM para trazer comportamento dinâmico para aplicativos da web.
- A execução de aplicativos da web do lado do cliente é realizada em duas fases:
 - *Construção de página* - o código HTML é processado para criar o DOM e o código JavaScript global é executado quando os nós de script são encontrados. Durante essa execução, o código JavaScript pode modificar o DOM atual em qualquer grau e pode até registrar manipuladores de eventos - funções que são executadas quando um evento específico ocorre (por exemplo, um clique do mouse ou um pressionamento de teclado). Registrar manipuladores de eventos é fácil: use o addEventListener método.
 - *Manipulação de eventos* —Vários eventos são processados um de cada vez, na ordem em que foram gerados. A fase de tratamento de eventos depende muito da fila de eventos, na qual todos os eventos são armazenados na ordem em que ocorreram. O loop de eventos sempre verifica se há eventos no topo da fila e, se um evento for encontrado, a função manipuladora de eventos correspondente é chamada.

2,5 Exercícios

- 1 Quais são as duas fases do ciclo de vida de um aplicativo da web do lado do cliente? Qual é a principal vantagem de
- 2 usar o addEventListener método para registrar um manipulador de eventos versus atribuir um manipulador a uma propriedade de elemento específico? Quantos eventos podem ser processados de uma vez?
- 3
- 4 Em que ordem os eventos da fila de eventos são processados?

Funções de compreensão


Nomo que você está mentalmente preparado e entende o ambiente no qual o código JavaScript é executado, você está pronto para aprender os fundamentos dos recursos JavaScript disponíveis para você.

No capítulo 3, você aprenderá tudo sobre o conceito básico mais importante de JavaScript: não, não o objeto, mas a função. Este capítulo ensinará por que entender as funções do JavaScript é a chave para desvendar os segredos da linguagem.

O Capítulo 4 continua nossa exploração detalhada das funções, estudando como as funções são chamadas, juntamente com todos os detalhes dos parâmetros implícitos que são acessíveis durante a execução do código da função.

O Capítulo 5 leva as funções para o próximo nível com fechamentos - provavelmente um dos aspectos mais mal compreendidos (e até mesmo desconhecidos) da linguagem JavaScript. Como você verá em breve, os fechamentos estão intimamente ligados aos escopos. Neste capítulo, além dos encerramentos, colocamos um foco especial nos mecanismos de escopo em JavaScript.

Nossa exploração das funções será concluída no capítulo 6, onde discutimos um tipo de função completamente novo - a função geradora - que possui algumas propriedades especiais e é especialmente útil ao lidar com código assíncrono.



Primeira classe

funções para o novato: definições e argumentos

Este capítulo cobre

- Por que entender as funções é tão crucial
- Como as funções são objetos de primeira classe
- As formas de definir uma função
- Os segredos de como os parâmetros são atribuídos

Você pode se surpreender, ao consultar esta parte do livro dedicada aos fundamentos do JavaScript, para ver que o primeiro tópico de discussão é *funções* ao invés de *objetos*. Certamente estaremos prestando muita atenção aos objetos na parte 3 do livro, mas quando se trata de detalhes, a principal diferença entre escrever código JavaScript como a Jill (ou Joe) comum e escrevê-lo como um ninja JavaScript é entender JavaScript como um *linguagem funcional*. O nível de sofisticação de todo o código que você escreverá em JavaScript depende dessa compreensão.

Se você está lendo este livro, você não é um iniciante. Estamos assumindo que você conhece fundamentos de objetos suficientes para sobreviver (e vamos dar uma olhada em

conceitos de objetos mais avançados no capítulo 7), mas *mesmo* compreender as funções em JavaScript é a arma mais importante que você pode usar. Tão importante, na verdade, que este e os três capítulos seguintes são dedicados a uma compreensão completa das funções em JavaScript.

Mais importante, em JavaScript, as funções são *objetos de primeira classe*, ou *cidadãos de primeira classe* como são frequentemente chamados. Eles coexistem e podem ser tratados como qualquer outro objeto JavaScript. Assim como os tipos de dados JavaScript mais comuns, eles podem ser referenciados por variáveis, declarados com literais e até mesmo passados como parâmetros de função. Neste capítulo, vamos primeiro dar uma olhada na diferença que esta orientação para funções traz, e você verá como isso pode nos ajudar a escrever um código mais compacto e fácil de entender, permitindo-nos definir funções exatamente onde Nós precisamos deles. Também exploraremos como tirar proveito das funções como objetos de primeira classe para escrever funções de melhor desempenho. Você verá várias maneiras de definir funções, incluindo alguns novos tipos, como *seta* funções, que o ajudarão a escrever um código mais elegante. Finalmente, veremos a diferença entre parâmetros de função e argumentos de função, com um foco especial nas adições ES6, como o resto e os parâmetros padrão.

Vamos começar examinando alguns dos benefícios da programação funcional.

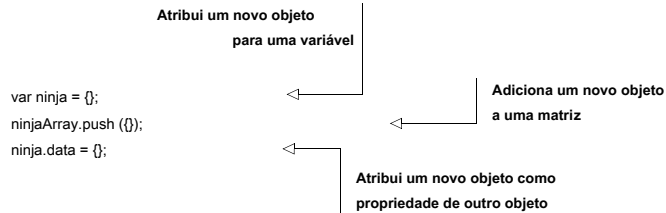
Você sabe?	Em quais situações as funções de retorno de chamada podem ser usadas sincronicamente? De forma assíncrona?
	Qual é a diferença entre uma função de seta e um expressão de função?
	Por que você pode precisar usar valores de parâmetro padrão em um função?

3.1 *Qual é a diferença funcional?*

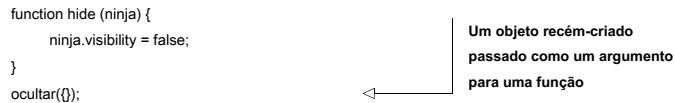
Uma das razões pelas quais funções e conceitos funcionais são tão importantes em JavaScript é que as funções são unidades modulares primárias de execução. Exceto pelo código JavaScript global executado na fase de construção de página, todo o código de script que escreveremos para nossas páginas estará dentro de uma função.

Como a maior parte do nosso código será executado como resultado de uma invocação de função, você verá que ter funções que são construções versáteis e poderosas nos dá uma grande flexibilidade e controle ao escrever código. Partes significativas deste livro explicam como a natureza das funções como objetos de primeira classe pode ser explorada para nosso grande benefício. Mas primeiro, vamos dar uma olhada em algumas das ações que podemos realizar com objetos. Em JavaScript, os objetos desfrutam de certos recursos:

- Eles podem ser criados por meio de literais: {}
- Eles podem ser atribuídos a variáveis, entradas de array e propriedades de outros objetos:



- Eles podem ser passados como argumentos para funções:



- Eles podem ser retornados como valores de funções:



- Eles podem possuir propriedades que podem ser criadas e atribuídas dinamicamente:



Acontece que, ao contrário de muitas outras linguagens de programação, em JavaScript podemos fazer quase exatamente as mesmas coisas com funções também.

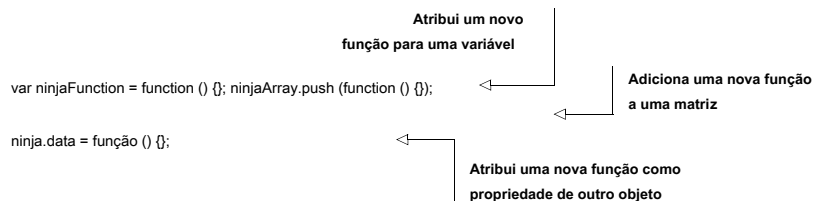
3.1.1 Funciona como objetos de primeira classe

As funções em JavaScript possuem todos os recursos de objetos e, portanto, são tratadas como qualquer outro objeto na linguagem. Dizemos que as funções são *primeira classe* objetos, que também podem ser

- Criado por meio de literais

```
function ninjaFunction () {}
```

- Atribuído a variáveis, entradas de array e propriedades de outros objetos



- Passado como argumentos para outras funções

```
chamada de função (ninjaFunction) {
  ninjaFunction ();
}
chamar (função () {});
```

Uma função recém-criada
passada como um argumento
para uma função

- Retornado como valores de funções

```
function returnNewNinjaFunction () {
  função de retorno () {};
}
```

Retorna um novo
função

- Eles podem possuir propriedades que podem ser criadas e atribuídas dinamicamente:

```
var ninjaFunction = function () {}; ninjaFunction.name = "Hanzo";
```

Adiciona uma nova
propriedade a uma função

O que quer que possamos fazer com objetos, também podemos fazer com funções. Funções *estamos* objetos, apenas com uma capacidade especial adicional de ser *invokable*: As funções podem ser chamadas ou invocadas para executar uma ação.

Programação funcional em JavaScript

Ter funções como objetos de primeira classe é o primeiro passo para *programação funcional*, um estilo de programação que se concentra na solução de problemas por meio da composição de funções (em vez de especificar sequências de etapas, como na programação mais convencional e imperativa). A programação funcional pode nos ajudar a escrever códigos mais fáceis de testar, estender e modularizar. Mas é um grande tópico, e neste livro nós apenas concordamos com ele (por exemplo, no capítulo 9). Se você estiver interessado em aprender como tirar proveito dos conceitos de programação funcional e aplicá-los aos seus programas JavaScript, recomendamos *Programação Funcional em JavaScript* por Luis Atencio (Manning, 2016), disponível em www.manning.com/books/functional-programming-in-javascript

Uma das características dos objetos de primeira classe é que eles podem ser passados para funções como argumentos. No caso de funções, isso significa que passamos uma função como um argumento para outra função que pode, em um ponto posterior da execução do aplicativo, chamar a função passada. Este é um exemplo de um conceito mais geral conhecido como um *função de retorno de chamada*. Vamos explorar este importante conceito.

3.1.2 Funções de retorno de chamada

Sempre que configuramos uma função para ser chamada posteriormente, seja pelo navegador na fase de manipulação de eventos ou por outro código, estamos configurando um *ligar de volta*. O termo deriva do fato de que estamos estabelecendo uma função que outro código irá "chamar de volta" posteriormente em um ponto apropriado de execução.

Callbacks são uma parte essencial do uso eficaz do JavaScript, e estamos dispostos a apostar que você já os usa muito em seu código - seja executando código com um clique de botão, recebendo dados de um servidor ou animando partes de sua IU.

Nesta seção, vamos ver como usar callbacks para manipular eventos ou para classificar coleções facilmente - exemplos típicos do mundo real de como callbacks são usados. Mas é um pouco complexo, então antes de mergulhar, vamos despir o conceito de callback completamente e examiná-lo em sua forma mais simples. Começaremos com um exemplo esclarecedor de uma função completamente inútil que aceita uma referência a outra função como parâmetro e chama essa função como retorno de chamada:

```
function useless (ninjaCallback) {  
    return ninjaCallback ();  
}
```

Por mais inútil que seja, essa função demonstra a capacidade de passar uma função como um argumento para outra função e, posteriormente, invocar essa função por meio do parâmetro passado.

Podemos testar essa função inútil com o código da listagem a seguir.

Listagem 3.1 Um exemplo simples de retorno de chamada

```
var text = "Domo arigato!";  
relatório ("Antes de definir funções");
```

```
function useless (ninjaCallback) {  
    relatório ("Em função inútil");  
    return ninjaCallback ();  
}
```

Define uma função que recebe uma função de retorno de chamada e a invoca imediatamente

```
function getText () {  
    report ("Na função getText");  
    texto de retorno;  
}
```

Define um simples função que retorna uma variável global

```
relatório ("Antes de fazer todas as ligações");
```

```
assert (useless (getText) === text,  
        "A função inútil funcional" + Texto);
```

Chama nossa função inútil com a função getText como um retorno de chamada

```
relatório ("Após as ligações terem sido feitas");
```

Nesta lista, usamos um personalizado relatório função para produzir várias mensagens conforme nosso código está sendo executado, para que possamos acompanhar a execução de nosso programa. Nós também usamos o

afirmar função de teste que mencionamos no capítulo 1. O afirmar A função geralmente leva dois argumentos. O primeiro argumento é uma expressão cuja premissa é afirmada. Neste caso, queremos estabelecer se o resultado de invocar nosso sem utilidade função com o argumento getText retorna um valor que é igual ao valor da variável

texto (inútil (getText) === texto). Se o primeiro argumento for avaliado como verdade, a afirmação passa; caso contrário, é considerado um fracasso. O segundo argumento é a mensagem associada, que geralmente é registrada com um indicador de aprovação / reprovação apropriado. (Apêndice C

discute os testes em geral, bem como nossa pequena implementação do afirmar e relatório funções).

Quando executamos esse código, chegamos ao resultado mostrado na figura 3.1. Como você pode ver, chamando o sem utilidade funcionar com o nosso `getText` a função de retorno de chamada como um argumento retorna o valor esperado.

Também podemos dar uma olhada em como exatamente este exemplo simples de callback é executado. Como mostra a figura 3.2, passamos no `getText` função para o sem utilidade funcionar como um argumento. Isso significa que dentro do corpo do sem utilidade função, o `getText` função pode ser referenciada através do ligar de volta

parâmetro. Então, fazendo o ligar de volta() chamada, nós causamos a execução do `getText` função; a `getText` função, que passamos como um argumento, é chamada de volta pelo sem utilidade função.

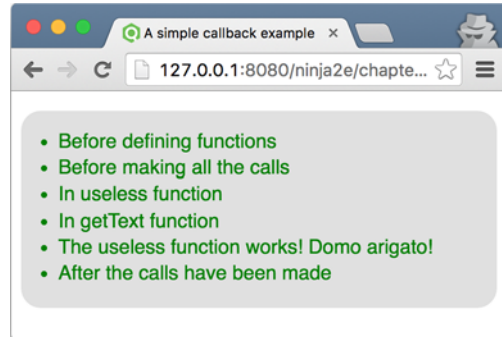


Figura 3.1 O resultado da execução do código da listagem 3.1

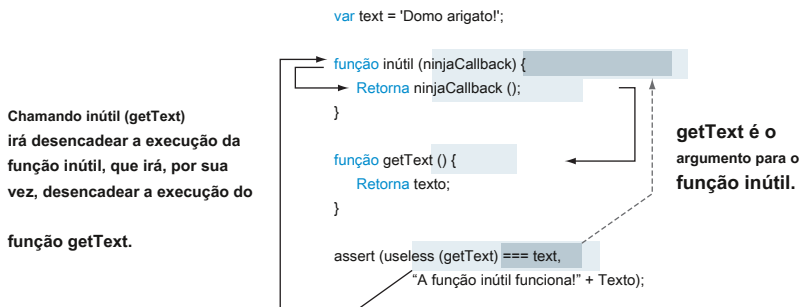


Figura 3.2 O fluxo de execução ao fazer o inútil (getText) chamar. O sem utilidade função é chamada com `getText` como um argumento. No corpo do sem utilidade função é uma chamada para a função passada, que neste caso dispara a execução do `getText` função (nós "chamamos de volta" para o `getText` função).

Isso é fácil, porque a natureza funcional do JavaScript nos permite lidar com funções como objetos de primeira classe. Podemos até dar um passo adiante, reescrevendo nosso código da seguinte maneira:

```

var text = 'Domo arigato!';

function useless (ninjaCallback) {
  return ninjaCallback ();
}

afirmar (inútil ( função () {texto de retorno;} ) === texto,
        "A função inútil funciona!" + Texto);
  
```

Define um retorno de chamada funcionar diretamente como um argumento

Um dos recursos mais importantes do JavaScript é a capacidade de criar funções no código em qualquer lugar em que uma expressão possa aparecer. Além de tornar o código mais compacto e fácil de entender (colocando as definições de função perto de onde são usadas), esse recurso também pode eliminar a necessidade de poluir o namespace global com nomes desnecessários quando uma função não for referenciada de vários lugares dentro do código.

No exemplo anterior de um retorno de chamada, chamamos nosso próprio retorno de chamada. Mas os retornos de chamada também podem ser chamados pelo navegador. Pense no capítulo 2, que tem um exemplo com o seguinte snippet:

```
document.body.addEventListener("mousemove", function () {  
    var segundo = document.getElementById("segundo"); addMessage(segundo, "Evento:  
    mousemove");  
});
```

Essa também é uma função de retorno de chamada, definida como um manipulador de eventos para o mousemove evento, e que será chamado pelo navegador quando esse evento ocorrer.

NOTA Esta seção apresenta retornos de chamada como funções que outro código “chamará de volta” posteriormente em um ponto apropriado de execução. Você viu um exemplo em que nosso próprio código chama imediatamente o retorno de chamada fornecido (o sem utilidade exemplo de função), bem como um exemplo em que o navegador faz a chamada (o mousemove exemplo) sempre que um determinado evento acontece. É importante observar que, ao contrário de nós, algumas pessoas acreditam que um retorno de chamada deve ser chamado de forma assíncrona e, portanto, o primeiro exemplo não é realmente um retorno de chamada. Mencionamos isso apenas no caso de você tropeçar em alguma discussão acalorada.

Agora vamos considerar o uso de callbacks que simplificarão muito a forma como classificamos as coleções.

CLASSIFICANDO COM UM COMPARADOR

Quase assim que nós *ter* uma coleção de dados, provavelmente precisaremos classificá-la. Digamos que temos uma matriz de números em ordem aleatória: 0, 3, 2, 5, 7, 4, 8, 1. Essa ordem pode ser adequada, mas é provável que, mais cedo ou mais tarde, queiramos para reorganizá-lo.

Normalmente, implementar algoritmos de classificação não é a mais trivial das tarefas de programação; temos que selecionar o melhor algoritmo para o trabalho em questão, implementá-lo, adaptá-lo às nossas necessidades atuais (de modo que os itens sejam classificados em uma ordem específica) e ter cuidado para não introduzir bugs. Destas tarefas, a única que é específica do aplicativo é a ordem de classificação. Felizmente, todas as matrizes de JavaScript têm acesso ao organizar método que exige que definamos apenas um algoritmo de comparação que diga ao algoritmo de classificação como os valores devem ser ordenados.

É aqui que entram os callbacks! Em vez de deixar o algoritmo de classificação decidir quais valores vão antes de outros valores, *Nós vamos* fornecer uma função que executa a comparação. Forneceremos ao algoritmo de classificação acesso a essa função como um retorno de chamada, e o algoritmo chamará o retorno de chamada sempre que precisar fazer uma comparação. O retorno de chamada deve retornar um número positivo se a ordem dos valores passados for

invertido, um número negativo se não, e zero se os valores forem iguais; subtrair os valores comparados produz o valor de retorno desejado para classificar a matriz:

```
valores var = [0, 3, 2, 5, 7, 4, 8, 1];
```

```
values.sort ( função (valor1, valor2) {
    retorno valor1 - valor2;
});
```

Não há necessidade de pensar sobre os detalhes de baixo nível de um algoritmo de classificação (ou mesmo que algoritmo de classificação escolher). Fornecemos um retorno de chamada que o mecanismo JavaScript chamará sempre que precisar comparar dois itens.

O *funcional* abordagem nos permite criar uma função como uma entidade autônoma, assim como podemos qualquer outro tipo de objeto, e passá-la como um argumento para um método, assim como qualquer outro tipo de objeto, que pode aceitá-la como um parâmetro, assim como qualquer outro tipo de objeto. É esse status de primeira classe entrando em jogo.

3,2 *Diversão com funções como objetos*

Nesta seção, examinaremos maneiras de explorar as semelhanças que as funções compartilham com outros tipos de objetos. Um recurso que pode ser surpreendente é que nada nos impede de anexar propriedades às funções:

```
var ninja = {};
ninja.name = "hitsuke";
```

Cria um objeto e atribui uma nova propriedade a ele

```
var wieldSword = function () {}; wieldSword.swordType =
"katana";
```

Cria uma função e atribui uma nova propriedade a ela

Vejamos algumas das coisas mais interessantes que podem ser feitas com esse recurso:

- *Armazenamento de funções em uma coleção* nos permite gerenciar facilmente funções relacionadas - por exemplo, retornos de chamada que precisam ser invocados quando algo de interesse ocorre.
- *Memoização* permite que a função lembre-se de valores calculados anteriormente, melhorando assim o desempenho de invocações subsequentes.

Vamos começar.

3.2.1 *Armazenando funções*

Em certos casos (por exemplo, quando precisamos gerenciar coleções de callbacks que devem ser invocadas quando um certo evento ocorre), queremos armazenar coleções de funções únicas. Ao adicionar funções a tal coleção, um desafio que podemos enfrentar é determinar quais funções são novas para a coleção e devem ser adicionadas e quais já são residentes e não devem ser adicionadas. Em geral, ao gerenciar coleções de retorno de chamada, não queremos duplicatas, porque um único evento resultaria em várias chamadas para o mesmo retorno de chamada.

Uma técnica óbvia, mas ingênua, é armazenar todas as funções em um array e percorrer o array, verificando se há funções duplicadas. Infelizmente, o desempenho é ruim e, como ninja, queremos fazer as coisas funcionarem *Nós vamos*, não apenas trabalhar. Podemos usar propriedades de função para conseguir isso com um nível apropriado de sofisticação, conforme mostrado na próxima listagem.

Listagem 3.2 Armazenando uma coleção de funções exclusivas

```

var store = {
  nextId: 1,
  cache: {},
  add: function (fn) {
    if (!fn.id) {
      fn.id = this.nextId++;
      this.cache [fn.id] = fn;
      return true;
    }
  }
};

função ninja () {}
assert (store.add (ninja),
        "A função foi adicionada com segurança."); assert (!
store.add (ninja),
        "Mas foi adicionado apenas uma vez.");

```

Mantém o controle do próximo ID disponível a ser atribuído

Cria um objeto para servir como um cache no qual iremos armazenar funções

Adiciona funções a o cache, mas apenas se eles forem únicos

Testa que tudo funciona como planejado

Nesta listagem, criamos um objeto atribuído à variável `armazenar`, no qual armazenaremos um conjunto único de funções. Este objeto possui duas propriedades de dados: uma que armazena um próximo disponível eu ia valor, e dentro do qual iremos esconderijo as funções armazenadas. As funções são adicionadas a este cache por meio do `adicionar()` método:

```

add: function (fn) {
  if (!fn.id) {
    fn.id = this.nextId++;
    this.cache [fn.id] = fn;
    return true;
  }
  ...

```

Dentro de `adicionar`, primeiro verificamos se a função já foi adicionada à coleção, procurando a existência do eu ia propriedade. Se a função atual tiver um eu ia propriedade, assumimos que a função já foi processada e a ignoramos. Caso contrário, atribuímos um eu ia propriedade para a função (incrementando o `nextId` propriedade ao longo do caminho) e adicione a função como uma propriedade do cache, usando o eu ia valor como o nome da propriedade. Em seguida, devolvemos o valor verdade, para que possamos saber quando a função foi adicionada após uma chamada para `adicionar()`.

Executar a página no navegador mostra que quando nossos testes tentam adicionar o `ninja()` função duas vezes, a função é adicionada apenas uma vez, conforme mostrado na figura 3.3. O Capítulo 9 mostra uma técnica ainda melhor para trabalhar com coleções de itens exclusivos que utilizam conjuntos, um novo tipo de objeto disponível no ES6.

Outro truque útil para tirar de nossas mangas ao usar as propriedades da função é dar a uma função a capacidade de se modificar. Esta técnica pode ser usada para lembrar valores calculados anteriormente, economizando tempo durante cálculos futuros.

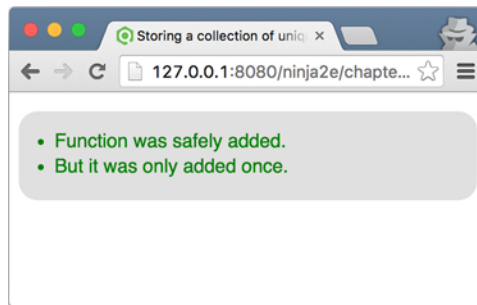


Figura 3.3 Ao adicionar uma propriedade a uma função, podemos controlá-la. Dessa forma, podemos ter certeza de que nossa função foi adicionada apenas uma vez.

3.2.2 *Funções self-memoizing*

Como observado anteriormente, *memoização* (não, isso não é um erro de digitação) é o processo de construção de uma função que é capaz de lembrar seus valores calculados anteriormente. Resumindo, sempre que uma função calcula seu resultado, armazenamos esse resultado junto com os argumentos da função. Dessa forma, quando outra chamada ocorre para o mesmo conjunto de argumentos, podemos retornar o resultado armazenado anteriormente, em vez de calculá-lo novamente. Isso pode aumentar significativamente o desempenho, evitando cálculos complexos desnecessários que já foram executados. A memorização é particularmente útil ao realizar cálculos para animações, pesquisar dados que não mudam com tanta frequência ou qualquer matemática demorada.

Como exemplo, vamos examinar um algoritmo simplista (e certamente não particularmente eficiente) para calcular números primos. Embora este seja um exemplo simples de um cálculo complexo, essa técnica é facilmente aplicável a outros cálculos caros (como derivar o hash MD5 para uma string) que são muito complexos para apresentar aqui.

Do lado de fora, a função parece ser como qualquer função normal, mas vamos construir secretamente em um cache de resposta no qual a função salvará as respostas para os cálculos que realiza. Examine o código a seguir.

Listagem 3.3 Memorizando valores calculados anteriormente

```
function isPrime (value) {
  if (!isPrime.answers) {
    isPrime.answers = {};
  }

  if (isPrime.answers [value] !== undefined) {
    return isPrime.answers [value];
  }

  var prime = valor! == 1; // 1 não é primo
```

Cria o cache

Verifica os valores em cache


```

para (var i = 2; i < valor; i++) {
  if (valor % i === 0) {
    prime = false;
    intervalo;
  }
}
return isPrime.answers [value] = prime;
}

```

← Armazena o valor calculado

```

assert (isPrime (5), "5 é primo!");
assert (isPrime.answers [5], "A resposta foi armazenada em cache!");

```

Testa que isso
tudo funciona

Dentro do `isPrime` função, começamos verificando se o respostas propriedade que usaremos como cache foi criada e, se não, nós a criamos:

```

if (! isPrime.answers) {
  isPrime.answers = {};
}

```

A criação desse objeto inicialmente vazio ocorrerá apenas na primeira chamada para a função; depois disso, o cache existirá.

Em seguida, verificamos se o resultado do valor passado já foi armazenado em cache no respostas:

```

if (isPrime.answers [value] !== undefined) {
  return isPrime.answers [value];
}

```

Dentro desse cache, vamos armazenar a resposta computada (verdade ou falso) usando o argumento valor como a chave de propriedade. Se encontrarmos uma resposta em cache, a devolvemos.

Se nenhum valor em cache for encontrado, prosseguimos e realizamos os cálculos necessários para determinar se o valor é melhor (que pode ser uma operação cara para valores maiores) e armazenar o resultado no cache à medida que o retornamos:

```

return isPrime.answers [value] = prime;

```

Nosso cache é uma propriedade da própria função, portanto, é mantido ativo enquanto a própria função estiver ativa.

Por fim, testamos se a memoização está funcionando!

```

assert (isPrime (5), "5 é primo!");
assert (isPrime.answers [5], "A resposta foi armazenada em cache!");

```

Essa abordagem tem duas vantagens principais:

- O usuário final desfruta de benefícios de desempenho para chamadas de função que solicitam um valor calculado anteriormente.
- Isso acontece perfeitamente e nos bastidores; nem o usuário final nem o autor da página precisam realizar quaisquer solicitações especiais ou fazer qualquer inicialização extra para fazer tudo funcionar.

Mas nem tudo são rosas e violinos; suas desvantagens podem precisar ser pesadas contra suas vantagens:

- Qualquer tipo de cache certamente sacrificará a memória em favor do desempenho. Os puristas podem considerar que o armazenamento em cache é uma preocupação que não deve ser misturada com a lógica de negócios; uma função ou método deve fazer uma coisa e fazê-lo bem. Mas não se preocupe; no capítulo 8, você verá como lidar com essa reclamação.
- É difícil testar a carga ou medir o desempenho de um algoritmo como este, porque nossos resultados dependem das entradas anteriores para a função.

Agora que você viu alguns dos casos de uso prático de funções de primeira classe, vamos explorar as várias maneiras de definir funções.

3.3 Definindo funções

As funções JavaScript são geralmente definidas usando um *literal de função* que cria um valor de função da mesma maneira que, por exemplo, um literal numérico cria um valor numérico. Lembre-se de que, como objetos de primeira classe, as funções são valores que podem ser usados na linguagem como outros valores, como strings e números. E quer você perceba ou não, você tem feito isso o tempo todo.

JavaScript fornece algumas maneiras de definir funções, que podem ser divididas em quatro grupos:

- *Declarações de função e expressões de função* —As duas formas mais comuns, embora sutilmente diferentes, de definir funções. Muitas vezes, as pessoas nem mesmo os consideram separados, mas como você verá, estar ciente de suas diferenças pode nos ajudar a entender quando nossas funções estão disponíveis para invocação:

```
function myFun () {return 1;}
```

- *Funções de seta* (frequentemente chamado *funções lambda*) —Uma adição recente do ES6 ao padrão JavaScript que nos permite definir funções com muito menos confusão sintática. Eles até resolvem um problema comum com funções de retorno de chamada, mas mais sobre isso mais tarde:

```
myArg => myArg * 2
```

- *Construtores de função* —Uma maneira não tão usada de definir funções que nos permite construir dinamicamente uma nova função a partir de uma string que também pode ser gerada dinamicamente. Este exemplo cria dinamicamente uma função com dois parâmetros, *a* e *b*, que retorna a soma desses dois parâmetros:

```
nova função ('a', 'b', 'retornar a + b')
```

- *Funções geradoras* —Esta adição ES6 ao JavaScript nos permite criar funções que, ao contrário das funções normais, podem ser encerradas e reinseridas posteriormente na execução do aplicativo, enquanto mantemos os valores de suas variáveis entre essas reentradas. Podemos definir versões geradoras de *declarações de função*, *expressões de função*, e *construtores de função*:

```
função * myGen () {rendimento 1; }
```

É importante que você entenda essas diferenças, porque a maneira como uma função é definida influencia significativamente quando a função está disponível para ser chamada e como ela se comporta, bem como em qual objeto a função pode ser chamada.

Neste capítulo, exploraremos as declarações de funções, expressões de funções e funções de seta. Você aprenderá sua sintaxe e como funcionam, e voltaremos a ela várias vezes ao longo do livro para explorar suas especificidades. As funções do gerador, por outro lado, são bastante peculiares e são significativamente diferentes das funções padrão. Iremos revisá-los em detalhes no capítulo 6.

Isso nos deixa com construtores de função, um recurso JavaScript que ignoraremos completamente. Embora tenha alguns aplicativos interessantes, especialmente ao criar e avaliar código dinamicamente, nós o consideramos um recurso fundamental da linguagem JavaScript. Se você quiser saber mais sobre construtores de função, visite <http://mng.bz/ZN8e>.

Vamos começar com as maneiras mais simples e tradicionais de definir funções: *declarações de função* e *expressões de função*.

3.3.1 Declarações de funções e expressões de funções

As duas maneiras mais comuns de definir funções em JavaScript são usando declarações e expressões de função. Essas duas técnicas são tão semelhantes que muitas vezes nem fazemos distinção entre elas, mas, como você verá nos próximos capítulos, existem diferenças sutis.

DECLARAÇÕES DE FUNÇÃO

A maneira mais básica de definir uma função em JavaScript é usando declarações de função (consulte a figura 3.4). Como você pode ver, cada declaração de função começa com um obrigatório **função** palavra-chave, seguida por um nome de função obrigatório e uma lista de nomes de parâmetros separados por vírgulas opcionais entre parênteses obrigatórios. O corpo da função, que é uma lista potencialmente vazia de instruções, deve ser colocado entre uma chave de abertura e uma chave de fechamento. Além deste formulário, que cada declaração de função deve satisfazer, há mais uma condição: Uma declaração de função deve ser colocada por conta própria, como uma instrução JavaScript separada (mas pode ser

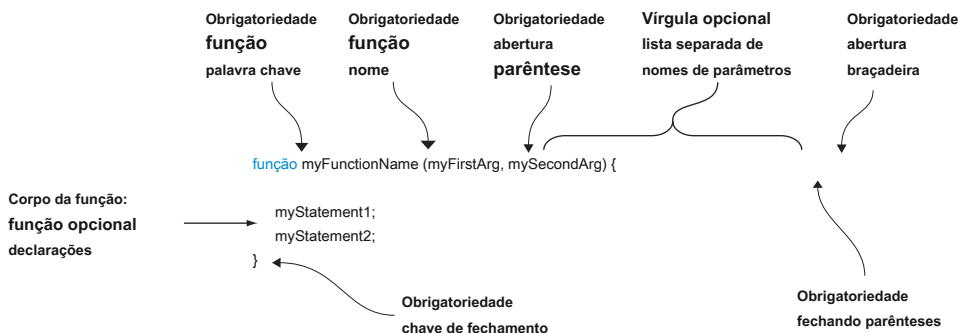


Figura 3.4 A declaração da função é independente, como um bloco separado de código JavaScript! (Pode estar contido em outras funções.)

contido em outra função ou bloco de código; você verá exatamente o que queremos dizer com isso na próxima seção).

Alguns exemplos de declaração de função são mostrados na lista a seguir.

Listagem 3.4 Exemplos de declarações de funções

<pre>function samurai () { retornar "samurai aqui"; } function ninja () { function hiddenNinja () { retornar "ninja aqui"; } return hiddenNinja (); }</pre>	<div style="border-left: 1px solid black; padding-left: 10px; margin-bottom: 20px;"> Define a função samurai no código global </div> <div style="border-left: 1px solid black; padding-left: 10px;"> Define a função ninja no código global </div> <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 20px;"> Define a função hiddenNinja dentro a função ninja </div>
--	--

Se você olhar mais de perto, verá algo com o qual talvez não esteja acostumado, se não teve muita experiência com linguagens funcionais: uma função definida dentro de outra função!

```
function ninja () {
    function hiddenNinja () {
        retornar "ninja aqui";
    }
    return hiddenNinja ();
}
```

Em JavaScript, isso é perfeitamente normal, e o usamos aqui para enfatizar novamente a importância das funções em JavaScript.

NOTA Ter funções contidas em outras funções pode levantar algumas questões complicadas em relação ao escopo e à resolução do identificador, mas guarde-as por enquanto, porque revisaremos esse caso em detalhes no capítulo 5.

EXPRESSÕES DE FUNÇÃO

Como já mencionamos várias vezes, as funções em JavaScript são objetos de primeira classe, o que, entre outras coisas, significa que podem ser criados por meio de literais, atribuídos a variáveis e propriedades e usados como argumentos e valores de retorno de e para outras funções. Como as funções são construções fundamentais, o JavaScript nos permite tratá-las como quaisquer outras expressões. Então, assim como podemos usar literais de número, por exemplo

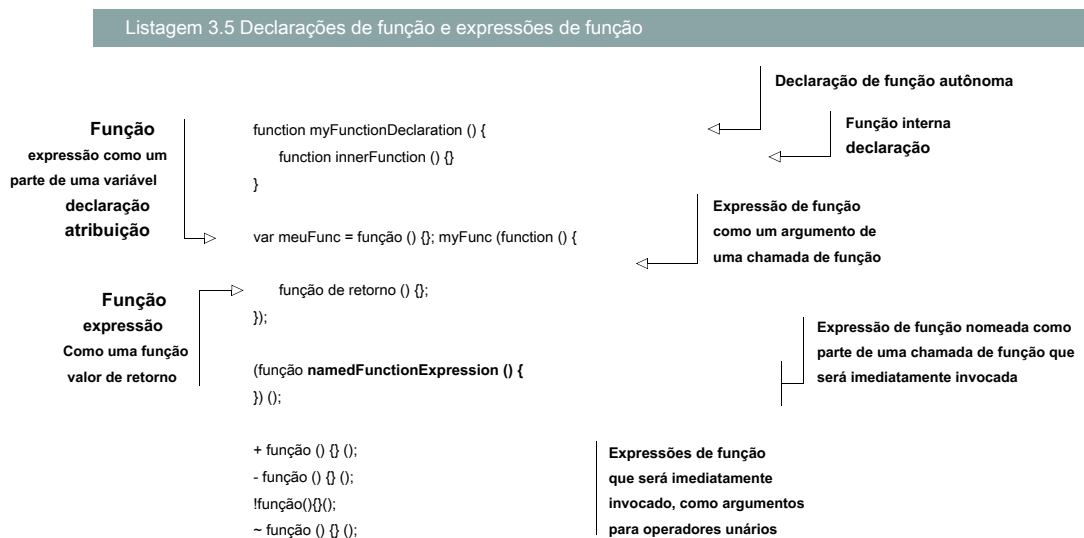
```
var a = 3;
minhaFunção (4);
```

assim também podemos usar literais de função, nos mesmos locais

```
var a = função () {}; minhaFunção (função () {});
```

Essas funções que sempre fazem parte de outra instrução (por exemplo, como o lado direito de uma expressão de atribuição ou como um argumento para outra função) são chamadas *expressões de função*. As expressões de função são ótimas porque nos permitem definir funções exatamente onde precisamos, no processo tornando nosso código mais fácil de entender.

A lista a seguir mostra as diferenças entre as declarações e expressões de função.



Este código de exemplo começa com uma declaração de função padrão que contém outra declaração de função interna:

```
function myFunctionDeclaration () {
    function innerFunction () {}
}
```

Aqui você pode ver como as declarações de função são instruções separadas do código JavaScript, mas podem estar contidas no corpo de outras funções.

Em contraste, há expressões de função, que sempre fazem parte de outra instrução. Eles são colocados no nível da expressão, como o lado direito de uma declaração de variável (ou uma atribuição):

```
var meuFunc = função () {};
```

Ou como um argumento para outra chamada de função ou como um valor de retorno de função:

```
myFunc (function () {  
    função de retorno () {};  
});
```

Além da posição no código onde são colocados, há mais uma diferença entre as declarações de função e expressões de função: Para declarações de função, o nome da função é *obrigatoriedade*, enquanto para expressões de função é completamente *opcional*.

As declarações de funções devem ter um nome definido porque são independentes. Como um dos requisitos básicos para uma função é que ela deve ser invocável, devemos ter uma maneira de referenciá-la, e a única maneira de fazer isso é por meio de seu nome.

As expressões de função, por outro lado, são partes de outras expressões JavaScript, portanto, temos maneiras alternativas de invocá-las. Por exemplo, se uma expressão de função for atribuída a uma variável, podemos usar essa variável para invocar a função:

```
var doNothing = function () {}; fazer nada();
```

Ou, se for um argumento para outra função, podemos invocá-lo dentro dessa função por meio do nome do parâmetro correspondente:

```
function doSomething (action) {  
    ação();  
}
```

FUNÇÕES IMEDIATAS

As expressões de função podem até mesmo ser colocadas em posições onde parecem um pouco estranhas no início, como em um local onde normalmente esperaríamos um identificador de função. Vamos parar e examinar mais de perto essa construção (veja a figura 3.5).

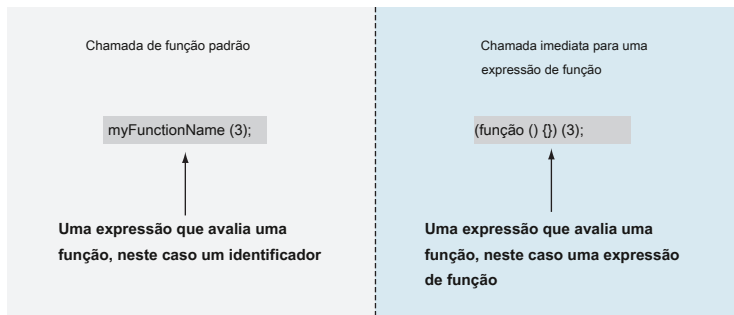


Figura 3.5 Uma comparação de uma chamada de função padrão e uma chamada imediata para uma expressão de função

Quando queremos fazer uma chamada de função, usamos uma expressão que avalia uma função, seguida por um par de parênteses de chamada de função, que pode conter argumentos. Na chamada de função mais básica, colocamos um identificador que avalia uma função, como no lado esquerdo da figura 3.5. Mas a expressão à esquerda do parêntese de chamada não precisa ser um identificador simples; pode ser *nenhum* expressão que avalia uma função. Por exemplo, uma maneira simples de especificar uma expressão avaliada como uma função é usar uma expressão de função. Portanto, no lado direito da figura 3.5, primeiro criamos uma função e, em seguida, invocamos imediatamente essa função recém-criada. A propósito, isso é chamado de *expressão de função imediatamente invocada* (IIFE), ou *função imediata* para resumir, e é um conceito importante no desenvolvimento de JavaScript porque nos permite imitar módulos em JavaScript. Vamos nos concentrar nesta aplicação de IIFEs no capítulo 11.

Parênteses em torno de expressões de função

Mais uma coisa pode estar incomodando você sobre a maneira como chamamos imediatamente nossa expressão de função: os parênteses ao redor da própria expressão de função. Por que ainda precisamos disso? O motivo é puramente sintático. O analisador JavaScript deve ser capaz de diferenciar facilmente entre declarações de função e expressões de função. Se deixarmos os parênteses em torno da expressão da função e colocarmos nossa chamada imediata como uma instrução separada função () {} (3), o analisador JavaScript começará a processá-lo e concluirá, porque é uma instrução separada que começa com a palavra-chave função, que está lidando com uma declaração de função. Como cada declaração de função deve ter um nome (e aqui não especificamos um), um erro será gerado. Para evitar isso, colocamos a expressão da função entre parênteses, sinalizando ao analisador JavaScript que está lidando com uma expressão, não com uma instrução.

Há também uma forma alternativa, ainda mais simples (embora, estranhamente, usada com menos frequência) de atingir o mesmo objetivo: (função () {} (3)). Envolvendo a definição de função imediata e chamada entre parênteses, você também pode notificar o analisador JavaScript que está lidando com uma expressão.

As últimas quatro expressões na listagem 3.5 são variações do mesmo tema de expressões de função imediatamente invocadas, frequentemente encontradas em várias bibliotecas JavaScript:

```
+ função () {} ();  
- função () {} ();  
!função(){}();  
~ função () {} ();
```

Desta vez, em vez de usar parênteses em torno das expressões de função para diferenciá-las das declarações de função, podemos usar operadores unários: +, -, ! E ~. Fazemos isso para sinalizar ao mecanismo JavaScript que está lidando com expressões e não com instruções. Observe como os resultados da aplicação desses operadores unários não são armazenados em lugar nenhum; de uma perspectiva computacional, eles realmente não importam; apenas as chamadas para nossos IIFEs importam.

Agora que estudamos os prós e contras das duas maneiras mais básicas de definir funções em JavaScript (declarações de função e expressões de função), vamos explorar uma nova adição ao padrão JavaScript: *funções de seta*.

3.3.2 Funções de seta



NOTA As funções de seta são uma adição ES6 ao padrão JavaScript (para compatibilidade de navegador, consulte <http://mng.bz/8bnH>)

Porque em nosso JavaScript usamos *muito* de funções, faz sentido adicionar algum açúcar sintático que nos permite criar funções de uma forma mais curta e sucinta, tornando assim nossas vidas como desenvolvedores mais agradáveis.

De várias maneiras, as funções de seta são uma simplificação das expressões de função. Vamos revisar nosso exemplo de classificação da primeira seção deste capítulo:

```
valores var = [0, 3, 2, 5, 7, 4, 8, 1]; valores.sort( função (valor1, valor2) {  
  
    retorno valor1 - valor2; });
```

Este exemplo usa uma expressão de função de retorno de chamada enviada ao método de classificação do objeto de matriz; esse retorno de chamada será invocado pelo mecanismo JavaScript para classificar os valores da matriz em ordem decrescente.

Agora vamos ver como fazer exatamente a mesma coisa com as funções de seta:

```
valores var = [0, 3, 2, 5, 7, 4, 8, 1]; values.sort (( valor1, valor2) => valor1 - valor2);
```

Veja como isso é muito mais sucinto?

Não há desordem causada pelo função palavra-chave, as chaves ou o Retorna demonstração. De uma maneira muito mais simples do que uma expressão de função pode, a função de seta afirma: aqui está uma função que recebe dois argumentos e retorna sua diferença. Observe a introdução de um novo operador, `=>`, o assim chamado *flecha gorda* operador (um sinal de igual imediatamente seguido por um sinal de maior que), que está no centro da definição de uma função de seta.

Agora vamos desconstruir a sintaxe de uma função de seta, começando da maneira mais simples possível:

```
param => expressão
```

Esta função de seta recebe um parâmetro e retorna o valor de uma expressão. Podemos usar essa sintaxe conforme mostrado na lista a seguir.

Listagem 3.6 Comparando uma função de seta e uma expressão de função

```
var greet = name => "Saudações" + nome;
assert (greet ("Oishi") === "Greetings Oishi", "Oishi é devidamente saudado");

var anotherGreet = function (name) {
    retornar "Saudações" + nome;
};
assert (anotherGreet ("Oishi") === "Saudações Oishi",
        "Novamente, Oishi é devidamente saudado");
```

Define uma função de seta

Define uma função
expressão

Pare um pouco para apreciar como as funções de seta tornam o código mais sucinto, sem sacrificar a clareza. Essa é a versão mais simples da sintaxe da função seta, mas, em geral, a função seta pode ser definida de duas maneiras, conforme mostrado na figura 3.6.

Como você pode ver, a definição da função de seta começa com uma lista opcional separada por vírgulas de nomes de parâmetros. Se não houver parâmetros, ou mais de um parâmetro, esta lista deve ser colocada entre parênteses. Mas se tivermos apenas um único parâmetro, os parênteses são opcionais. Essa lista de parâmetros é seguida por um operador de seta grande obrigatório, que nos informa e ao mecanismo JavaScript que estamos lidando com uma função de seta.

Depois do operador fat-arrow, temos duas opções. Se for uma função simples, colocamos uma expressão lá (uma operação matemática, outra chamada de função, qualquer coisa), e o resultado da chamada de função será o valor dessa expressão. Por exemplo, nosso primeiro exemplo de função de seta tem a seguinte função de seta:

```
var greet = name => "Saudações" + nome;
```

Parênteses obrigatórios
para 0 ou mais de 1
parâmetro; opcional
para 1 parâmetro.

Obrigatoriedade
flecha gorda
operador (igual a
e maior que).

Se o corpo da função de seta for uma
expressão, o valor de retorno da
função será o valor dessa expressão.

(param1, param2) => expressão

```
{
  myStatement1;
  myStatement2;
}
```

Separado por vírgula opcional
lista de nomes de parâmetros.

Se o corpo da função de seta for um bloco
de código, o retorno
o valor é como seria em uma função padrão
(indefinido se não houver instrução de
retorno e o valor da expressão de retorno se
houver).

Figura 3.6 A sintaxe de uma função de seta

O valor de retorno da função é uma concatenação da string " Saudações " com o valor do nome parâmetro.

Em outros casos, quando nossas funções de seta não são tão simples e exigem mais código, podemos incluir um bloco de código após o operador de seta. Por exemplo:

```
var greet = name => {
  var helloString = 'Saudações'; return helloString + name;
};
```

Nesse caso, o valor de retorno da função de seta se comporta como em uma função padrão. Se não houver instrução de retorno, o resultado da invocação da função será Indefinido, e se houver, o resultado será o valor da expressão de retorno.

Iremos revisitar as funções de seta várias vezes ao longo deste livro. Entre outras coisas, apresentaremos recursos adicionais de funções de seta que nos ajudarão a evitar erros sutis que podem ocorrer com funções mais padrão.

As funções de seta, como todas as outras funções, podem receber argumentos para usá-los para realizar sua tarefa. Vamos ver o que acontece com os valores que passamos para uma função.

3.4 Argumentos e parâmetros de função

Ao discutir funções, costumamos usar os termos *argumento* e *parâmetro* quase indistintamente, como se fossem mais ou menos a mesma coisa. Mas agora, vamos ser mais formais:

- Uma *parâmetro* é uma variável que listamos como parte da definição de uma função.
- A *argumento* é um valor que passamos para a função quando a chamamos.

A Figura 3.7 ilustra a diferença.

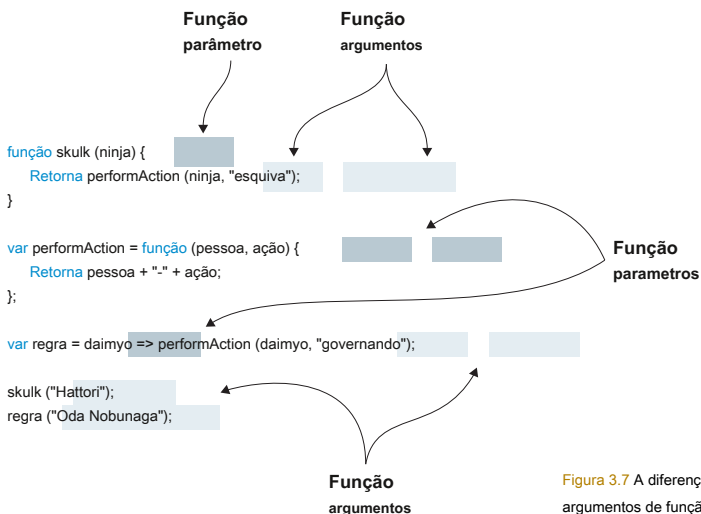


Figura 3.7 A diferença entre parâmetros de função e argumentos de função

Como você pode ver, um parâmetro de função é especificado com a definição da função, e todos os tipos de funções podem ter parâmetros:

- Declarações de funções (o ninja parâmetro para o esconder-se função) Expressões de função (o pessoa e ação parâmetros para o executar a ação função)
- Funções de seta (o daimyo parâmetro)

Os argumentos, por outro lado, estão vinculados à invocação da função; são valores passados para uma função no momento de sua invocação:

- A corda Hattori é passado como um argumento para o esconder-se função.
- A corda Oda Nobunaga é passado como um argumento para o regra função.
- O parâmetro ninja do esconder-se função é passada como um argumento para o executar a ação função.

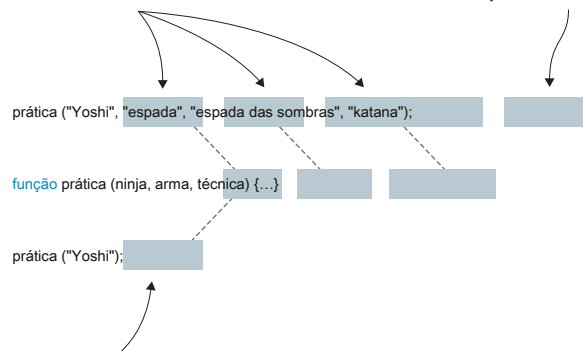
Quando uma lista de argumentos é fornecida como parte de uma chamada de função, esses argumentos são atribuídos aos parâmetros na definição da função na ordem especificada. O primeiro argumento é atribuído ao primeiro parâmetro, o segundo argumento ao segundo parâmetro e assim por diante.

Se tivermos um número diferente de argumentos do que parâmetros, nenhum erro será gerado. JavaScript está perfeitamente bem com essa situação e lida com ela da seguinte maneira. Se forem fornecidos mais argumentos do que parâmetros, os argumentos "em excesso" não serão atribuídos aos nomes dos parâmetros. Por exemplo, veja a figura 3.8.

A Figura 3.8 mostra que se chamássemos o prática funcionar com prática ("Yoshi", "espada", "espada sombra", "katana"), os argumentos Yoshi, espada, e espada sombra seria atribuído aos parâmetros ninja, arma, e técnica,

O argumento "Yoshi" é atribuído ao parâmetro ninja. O argumento "espada" é atribuído ao parâmetro arma. O argumento "espada da sombra" é atribuído ao parâmetro técnica.

Excesso de argumentos não são atribuídos a parâmetros.



O argumento "Yoshi" é atribuído ao parâmetro ninja. undefined é atribuído ao parâmetro arma. undefined é atribuído ao parâmetro técnica.

Figura 3.8 Os argumentos são atribuídos aos parâmetros da função na ordem especificada. Excesso de argumentos não são atribuídos a nenhum parâmetro.

respectivamente. O argumento katana é um argumento em excesso e não seria atribuído a nenhum parâmetro. No próximo capítulo, você verá que embora alguns argumentos não sejam atribuídos a nomes de parâmetros, ainda temos uma maneira de acessá-los.

Por outro lado, se tivermos mais parâmetros do que argumentos, os parâmetros que não têm argumento correspondente são definidos como Indefinido. Por exemplo, se fizéssemos a chamada prática ("Yoshi"), o parâmetro ninja seria atribuído o valor Yoshi, enquanto os parâmetros arma e técnica seria definido para Indefinido.

Lidar com argumentos e parâmetros de função é tão antigo quanto o próprio JavaScript, mas agora vamos explorar dois novos recursos de JavaScript oferecidos pelo ES6: resto e parâmetros padrão.

3.4.1 Parâmetros de descanso



NOTA Os parâmetros de descanso são adicionados pelo padrão ES6 (para compatibilidade do navegador, consulte <http://mng.bz/3go1>)

Para nosso próximo exemplo, vamos construir uma função que multiplica o primeiro argumento pelo maior dos argumentos restantes. Isso provavelmente não é algo particularmente aplicável em nossos aplicativos, mas é um exemplo de ainda mais técnicas para lidar com argumentos dentro de uma função.

Isso pode parecer bastante simples: pegaremos o primeiro argumento e o multiplicaremos pelo maior dos valores de argumento restantes. Nas versões antigas do JavaScript, isso exigiria algumas soluções alternativas (que veremos no próximo capítulo). Felizmente, no ES6, não precisamos passar por nenhum obstáculo. Podemos usar *parâmetros de descanso*, conforme mostrado na lista a seguir.

Listagem 3.7 Usando parâmetros de descanso

```
function multiMax (primeiro, ... restantesNúmeros) {
  var classificado = restantesNúmeros.sort (função (a, b) {
    retorno b - a;
  });
  retorna primeiro * classificado [0];
}
afirmar (multiMax (3, 1, 2, 3) == 9,
  "3 * 3 = 9 (Primeiro argumento, pelo maior.);");
```

← Parâmetros de descanso são prefixado com ...

← Classifique o restante números, decrescentes.

← A função é chamada como qualquer outra função.

Ao prefixar o último argumento nomeado de uma função com reticências (...), nós o transformamos em uma matriz chamada *os demais parâmetros*, que contém os argumentos passados restantes.

```
function multiMax (primeiro, ... restantesNúmeros) {
  ...
}
```

Por exemplo, neste caso, o multiMax função é chamada com quatro argumentos: multiMax (3, 1, 2, 3). No corpo do multiMax função, o valor do primeiro argumento, 3, é atribuído ao primeiro multiMax parâmetro de função, primeiro. Como o segundo parâmetro da função é o parâmetro restante, todos os argumentos restantes (1, 2, 3) são colocados em uma nova matriz: restantesNúmeros. Em seguida, obtemos o maior número classificando a matriz em ordem decrescente (observe como é simples mudar a ordem de classificação) e escolhendo o maior número, que está no primeiro lugar de nossa matriz classificada. (Essa está longe de ser a maneira mais eficiente de determinar o maior número, mas por que não aproveitar as habilidades que adquirimos no início do capítulo?)

NOTA Apenas o último parâmetro de função pode ser um parâmetro de descanso. Tentar colocar reticências na frente de qualquer parâmetro que não seja o último nos trará apenas sor-linha, na forma de SyntaxError: parâmetro após o parâmetro rest.

Na próxima seção, continuaremos adicionando ao nosso cinto de ferramentas JavaScript com funcionalidade ES6 adicional: parâmetros padrão.

3.4.2 Parâmetros padrão



NOTA Os parâmetros padrão são adicionados pelo padrão ES6 (para compatibilidade do navegador, consulte <http://mng.bz/wl8w>)

Muitos componentes de IU da web (especialmente plug-ins jQuery) podem ser configurados. Por exemplo, se estivermos desenvolvendo um componente de controle deslizante, podemos dar aos nossos usuários a opção de especificar um intervalo de tempo após o qual um item é substituído por outro, bem como uma animação que será usada conforme a mudança ocorre. Ao mesmo tempo, talvez alguns usuários não se importem e fiquem felizes em usar quaisquer configurações que oferecemos. Os parâmetros padrão são ideais para tais situações!

Nosso pequeno exemplo com as configurações do componente deslizante é apenas um caso específico de uma situação em que *quase* todas as chamadas de função usam o mesmo valor para um parâmetro específico (observe a ênfase em *quase*). Considere um caso mais simples em que a maioria de nossos ninjas está acostumada a se esgueirar, mas não Yagyu, que se preocupa apenas com uma simples espreita:

```
function performAction (ninja, ação) {  
  retornar ninja + "" + ação;  
}  
performAction ("Fuma", "esquiva");  
performAction ("Yoshi", "esquiva");  
performAction ("Hattori", "esquiva");  
performAction ("Yagyu", "sorrateiro");
```

Não parece tedioso ter que repetir sempre o mesmo argumento, esquivando-se, simplesmente porque Yagyu é obstinado e se recusa a agir como um ninja adequado?

Em outras linguagens de programação, esse problema é mais frequentemente resolvido com sobrecarga de função (especificando funções adicionais com o mesmo nome, mas um conjunto diferente de parâmetros). Infelizmente, o JavaScript não oferece suporte à sobrecarga de funções, portanto, quando confrontados com essa situação no passado, os desenvolvedores frequentemente recorriam a algo como a listagem a seguir.

Listagem 3.8 Lidando com parâmetros padrão antes de ES6

```
function performAction (ninja, ação) {
  action = typeof action === "undefined"? "esquiva": ação; retornar ninja + "" + ação;
}

assert (performAction ("Fuma") === "Fuma se escondendo",
  "O valor padrão é usado para Fuma");

assert (performAction ("Yoshi") === "Yoshi se escondendo",
  "O valor padrão é usado para Yoshi");

assert (performAction ("Hattori") === "Hattori se escondendo",
  "O valor padrão é usado para Hattori");

assert (performAction ("Yagyu", "sneaking") === "Yagyu sneaking",
  "Yagyu pode fazer o que quiser, até esgueirar-se!");
```

Se o parâmetro de ação for indefinido, usamos um valor padrão, skulking, e se estiver definido, mantemos o valor passado.

Não passamos em um segundo argumento, o valor da ação parâmetro; depois de executar a primeira função, a instrução do corpo assumirá como padrão skulking.

Passe uma string como o valor do parâmetro de ação; esse valor será usado em todo o corpo da função.

Aqui nós definimos um executar a ação função, que verifica se o valor do ação parâmetro é Indefinido (usando o tipo de operador), e se for, a função define o valor do ação variável para escondida. Se o ação parâmetro é enviado através de uma chamada de função (não é Indefinido), nós mantemos o valor.

NOTA O tipo de operador retorna uma string que indica o tipo do operando. Se o operando não estiver definido (por exemplo, se não tivermos fornecido um argumento correspondente para um parâmetro de função), o valor de retorno é a string Indefinido.

Este é um padrão que ocorre comumente e é tedioso de escrever, então o padrão ES6 adicionou suporte para *parâmetros padrão*, conforme mostrado na lista a seguir.

Listagem 3.9 Lidando com parâmetros padrão no ES6

```
function performAction (ninja, action = "skulking") {  
  retornar ninja + "" + ação;  
}
```

```
assert (performAction ("Fuma") === "Fuma se escondendo",  
  "O valor padrão é usado para Fuma");
```

```
assert (performAction ("Yoshi") === "Yoshi se escondendo",  
  "O valor padrão é usado para Yoshi");
```

```
assert (performAction ("Hattori") === "Hattori se escondendo",  
  "O valor padrão é usado para Hattori");
```

```
assert (performAction ("Yagyu", "sneaking") === "Yagyu sneaking",  
  "Yagyu pode fazer o que quiser, até esgueirar-se!");
```

← No ES6, é possível atribuir um valor a um parâmetro de função.

Se o valor não for passado, o valor padrão é usado.

O valor passado é usado.

Aqui você pode ver a sintaxe dos parâmetros de função padrão em JavaScript. Para criar um parâmetro padrão, atribuímos um valor a um parâmetro de função:

```
função performAction (ninja, action = "skulking") {  
  retornar ninja + "" + ação;  
}
```

Então, quando fazemos uma chamada de função e o valor do argumento correspondente é deixado de fora, como acontece com Fuma, Yoshi, e Hattori, o valor padrão (neste caso, esquivando), é usado:

```
assert (performAction ("Fuma") === "Fuma se escondendo",  
  "O valor padrão é usado para Fuma");
```

```
assert (performAction ("Yoshi") === "Yoshi se escondendo",  
  "O valor padrão é usado para Yoshi");
```

```
assert (performAction ("Hattori") === "Hattori se escondendo",  
  "O valor padrão é usado para Hattori");
```

Se, por outro lado, especificarmos o valor, o valor padrão será substituído:

```
assert (performAction ("Yagyu", "sneaking") === "Yagyu sneaking",  
  "Yagyu pode fazer o que quiser, até esgueirar-se!");
```

Podemos atribuir qualquer valor aos parâmetros padrão: valores simples e primitivos, como números ou strings, mas também tipos complexos, como objetos, matrizes e até funções. Os valores são avaliados em cada chamada de função, da esquerda para a direita e, ao atribuir valores a parâmetros padrão posteriores, podemos fazer referência a parâmetros anteriores, como na listagem a seguir.

Listagem 3.10 Referenciando parâmetros padrão anteriores

```

função performAction (ninja, ação = "esquiva",
                    mensagem = ninja + "" + ação) {
    mensagem de retorno;
}

assert (performAction (" Yoshi ") === " Yoshi está se escondendo ", " Yoshi está se escondendo ");

```

Podemos colocar expressões arbitrárias como padrão valores de parâmetro, em o processo até referenciando anterior parâmetros de função.

Embora o JavaScript permita que você faça algo assim, recomendamos cautela. Em nossa opinião, isso não melhora a legibilidade do código e deve ser evitado, sempre que possível. Mas o uso moderado de parâmetros padrão - como meio de evitar valores nulos ou como sinalizadores relativamente simples que configuram os comportamentos de nossas funções - pode levar a um código muito mais simples e elegante.

3,5 *Resumo*

- Escrever código sofisticado depende do aprendizado de JavaScript como uma linguagem funcional.
- Funções são objetos de primeira classe tratados como quaisquer outros objetos em JavaScript. Semelhante a qualquer outro tipo de objeto, eles podem ser
 - Criado por meio de literais
 - Atribuído a variáveis ou propriedades
 - Passado como parâmetros
 - Retornado como resultados de função
 - Propriedades e métodos atribuídos
- Funções de retorno de chamada são funções que outro código irá "chamar de volta" posteriormente e são frequentemente usadas, especialmente com tratamento de eventos.
- Podemos tirar vantagem do fato de que as funções podem ter propriedades e que essas propriedades podem ser usadas para armazenar qualquer informação; por exemplo
 - Podemos armazenar funções em propriedades de função para referência posterior e invocação.
 - Podemos usar as propriedades da função para criar um cache (memoização), evitando cálculos desnecessários.
- Existem diferentes tipos de funções: declarações de função, expressões de função, funções de seta e geradores de função.
- Declarações e expressões de funções são os dois tipos mais comuns de funções. As declarações de função devem ter um nome e devem ser colocadas como instruções separadas em nosso código. As expressões de função não precisam ser nomeadas, mas precisam ser parte de outra instrução de código.
- As funções de seta são uma nova adição ao JavaScript, permitindo-nos definir funções de uma forma muito mais sucinta do que com funções padrão.
- Um parâmetro é uma variável que listamos como parte da definição de uma função, enquanto um argumento é um valor que passamos para a função quando a invocamos.

- A lista de parâmetros de uma função e sua lista de argumentos podem ter comprimentos diferentes:
 - Parâmetros não atribuídos avaliados como `Indefinido`.
 - Argumentos extras não estão vinculados a nomes de parâmetros.
- Parâmetros de descanso e parâmetros padrão são novas adições ao JavaScript:
 - Os parâmetros Rest nos permitem fazer referência aos argumentos restantes que não têm nomes de parâmetros correspondentes.
 - Os parâmetros padrão nos permitem especificar os valores dos parâmetros padrão que serão usados se nenhum valor for fornecido durante a chamada da função.

3,6 Exercícios

- 1 No trecho de código a seguir, quais funções são funções de retorno de chamada?

```
numbers.sort (função sortAsc ( a, b) {
    retornar a - b;
});

função ninja () {}
ninja ();

var myButton = document.getElementById ("myButton"); myButton.addEventListener ("click", função handleClick
() {
    alerta ("clicado");
});
```

- 2 No snippet a seguir, categorize as funções de acordo com seu tipo (função declaração, expressão de função ou função de seta).

```
numbers.sort (função sortAsc ( a, b) {
    retornar a - b;
});

numbers.sort (( a, b) => b - a);

( função(){}());

function outer () {
    função interna () {}
    retorno interno;
}

( função(){}());

()=> " Yoshi " ();
```

- 3 Depois de executar o seguinte trecho de código, quais são os valores das variáveis `samurai` e `ninja`?

```
var samurai = (() => "Tomoe") (); var ninja = (() => {"Yoshi"})
();
```

- 4 Dentro do corpo do teste função, quais são os valores dos parâmetros a, b, e c para as duas chamadas de função?

```
teste de função (a, b, ... c) { / * a, b, c * / }
```

```
teste (1, 2, 3, 4, 5); teste();
```

- 5 Depois de executar o seguinte snippet de código, quais são os valores do mensagem1 e mensagem2 variáveis?

```
function getNinjaWieldingWeapon (ninja, weapon = "katana") {  
    retornar ninja + " " + katana;  
}
```

```
var message1 = getNinjaWieldingWeapon ("Yoshi");  
var message2 = getNinjaWieldingWeapon ("Yoshi", "wakizashi");
```

4

Funções para o jornaleiro: *função de compreensão* *invocação*

Este capítulo cobre

- Dois parâmetros de função implícitos: argumentos e este
- Maneiras de invocar funções
- Lidando com problemas de contextos funcionais

No capítulo anterior, você viu que JavaScript é uma linguagem de programação com características orientadas funcionalmente significativas. Exploramos as diferenças entre argumentos de chamada de função e parâmetros de função e como os valores são transferidos de argumentos de chamada para parâmetros de função.

Este capítulo continua em uma linha semelhante, discutindo primeiro algo que escondemos de você no capítulo anterior: os parâmetros de função implícitos `this` e `arguments`. Eles são silenciosamente passados para as funções e podem ser acessados como qualquer outro parâmetro de função explicitamente nomeado dentro do corpo da função.

O `esta` parâmetro representa o contexto da função, o objeto no qual nossa função é chamada, enquanto o `argumentos` parâmetro representa todos os argumentos que são transmitidos por meio de uma chamada de função. Ambos os parâmetros são vitais no código JavaScript. O `esta` parâmetro é um dos ingredientes fundamentais do JavaScript orientado a objetos, e o `argumentos` parâmetro nos permite ser criativos com os argumentos que são aceitos por nossas funções. Por esse motivo, exploraremos algumas das armadilhas comuns relacionadas a esses argumentos implícitos.

Em seguida, continuaremos explorando maneiras de invocar funções em JavaScript. A maneira como invocamos uma função tem uma grande influência em como os parâmetros implícitos da função são determinados.

Finalmente, concluiremos o capítulo aprendendo sobre pegadinhas comuns relacionadas ao contexto da função, o `esta` parâmetro. Sem mais delongas, vamos começar a explorar!

.....

Você sabe? Porque é o `esta` parâmetro conhecido como a função *contexto*?

Qual é a diferença entre uma função e um método? O que aconteceria se um construtor funcionasse explicitamente retornou um objeto?

.....

4,1 *Usando parâmetros de função implícitos*

No capítulo anterior, exploramos as diferenças entre funções *parametros* (variáveis listadas como parte de uma definição de função) e função *argumentos* (valores passados para a função quando a invocamos). Mas não mencionamos que, além dos parâmetros que declaramos explicitamente na definição da função, as invocações da função geralmente recebem dois parâmetros implícitos: `argumentos` e `esta`.

De *implícito*, queremos dizer que esses parâmetros não são listados explicitamente na assinatura da função, mas são passados silenciosamente para a função e acessíveis dentro da função. Eles podem ser referenciados dentro da função como qualquer outro parâmetro explicitamente nomeado. Vamos dar uma olhada em cada um desses parâmetros implícitos.

4.1.1 *O parâmetro de argumentos*

O `argumentos` parâmetro é uma coleção de todos os argumentos passados para uma função. É útil porque nos permite acessar todos os argumentos da função, independentemente de o parâmetro correspondente estar definido explicitamente. Isso nos permite implementar a sobrecarga de função, um recurso que o JavaScript não suporta nativamente, e funções variáveis que aceitam um número variável de argumentos. Para ser honesto, com resto parâmetros, introduzidos no capítulo anterior, a necessidade do `argumentos` parâmetro foi bastante reduzido. Ainda assim, é importante entender como o `argumentos` parâmetro funciona, porque você está fadado a topá-lo ao lidar com código legado.

O argumento `objeto` tem uma propriedade chamada `comprimento` que indica o número exato de argumentos. Os valores de argumentos individuais podem ser obtidos usando a notação de indexação de matriz; por exemplo, `argumentos[2]` buscaria o terceiro parâmetro. Dê uma olhada na lista a seguir.

Listagem 4.1 Usando o argumento parâmetro

função qualquer (a, b, c) {	← Declara uma função com três parâmetros: a, b e c
<pre> assert (a === 1, 'O valor de a é 1'); assert (b === 2, 'O valor de b é 2'); assert (c === 3, 'O valor de c é 3'); </pre>	Testa os valores corretos
<pre> assert (arguments.length === 5, 'Passamos 5 parâmetros'); </pre>	Ao todo, a função recebe cinco argumentos.
<pre> assert (argumentos [0] === a, 'O primeiro argumento é atribuído a a'); assert (argumentos [1] === b, 'O segundo argumento é atribuído a b'); assert (argumentos [2] === c, 'O terceiro argumento é atribuído a c'); </pre>	Verifica se os três primeiros argumentos correspondem os parâmetros da função
<pre> assert (argumentos [3] === 4, 'Podemos acessar o quarto argumento'); assert (argumentos [4] === 5, 'Podemos acessar o quinto argumento'); } </pre>	Verifica se os argumentos em excesso podem ser acessados por meio do parâmetro <code>argumentos</code>
qualquer que seja (1,2,3,4,5);	← Chama uma função com cinco argumentos

Aqui temos um `qualquer que seja` função que é chamada com cinco argumentos, seja o que for (1,2,3,4,5), embora tenha apenas três parâmetros declarados, `a`, `b`, `c`:

```

função qualquer (a, b, c) {
  ...
}

```

Podemos acessar os três primeiros argumentos por meio de seus respectivos parâmetros de função, `a`, `b`, e `c`:

```

assert (a === 1, 'O valor de a é 1'); assert (b === 2, 'O valor de b é 2');
assert (c === 3, 'O valor de c é 3');

```

Também podemos verificar quantos argumentos no total foram passados para a função usando `argumentos.length` propriedade.

O argumento `parâmetro` também pode ser usado para acessar cada argumento individual por meio de notação de matriz. É importante observar que isso também inclui os argumentos em excesso que não estão associados a nenhum parâmetro de função:

```
assert (argumentos [0] === a, 'O primeiro argumento é atribuído a'); assert (argumentos [1] === b, 'O segundo argumento é
atribuído a b'); assert (argumentos [2] === c, 'O terceiro argumento é atribuído a c'); afirmar( argumentos [3] === 4, ' Podemos
acessar o quarto argumento '); afirmar( argumentos [4] === 5, ' Podemos acessar o quinto argumento ');
```

Ao longo desta seção, fazemos o possível para evitar chamar o argumentos parâmetro um *variedade*. Você pode se enganar pensando que é um array; afinal, tem um comprimento parâmetro e suas entradas podem ser obtidos usando notação de matriz. Mas é *não* um array JavaScript, e se você tentar usar métodos de array em argumentos (por exemplo, o organizar método usado no capítulo anterior), você não encontrará nada além de tristeza e decepção. Apenas pense em argumentos como um *como matriz* construir e exibir contenção em seu uso.

Como já mencionamos, o ponto principal do argumentos objeto é permitir que acessemos todos os argumentos que foram passados para a função, independentemente de um argumento específico estar associado a um parâmetro de função. Vamos ver como fazer isso implementando uma função que pode calcular a soma de um número arbitrário de argumentos.

Listagem 4.2 Usando o argumentos objeto para executar operações em todos os argumentos de função

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

← Uma função sem parâmetros definidos explicitamente

Itera por meio de todos os argumentos passados e acessa itens individuais por meio de notação de índice

```
assert (sum (1, 2) === 3, "Podemos somar dois números"); assert (sum (1, 2, 3) === 6, "Podemos somar três
números"); assert (sum (1, 2, 3, 4) === 10, "Podemos adicionar quatro números");
```

Chama a função com qualquer número de argumentos

Aqui, primeiro definimos um soma função que não lista explicitamente nenhum parâmetro. Independentemente disso, ainda podemos acessar todos os argumentos da função por meio do argumentos objeto. Nós iteramos todos os argumentos e calculamos sua soma.

Agora vem a recompensa. Podemos chamar a função com qualquer número de argumentos, então testamos alguns casos para ver se tudo funciona. Este é o verdadeiro poder do argumentos objeto. Ele nos permite escrever funções mais versáteis e flexíveis que podem lidar facilmente com diferentes situações.

NOTA Mencionamos anteriormente que, em muitos casos, podemos usar o resto parâmetro em vez do argumentos parâmetro. O resto parâmetro é um array real, o que significa que podemos usar todos os nossos métodos de array favoritos nele. Isso lhe dá uma certa vantagem sobre o argumentos objeto. Como exercício, reescreva a lista 4.2 para usar o resto parâmetro em vez do argumentos parâmetro.

Agora que entendemos como o argumentos objeto funciona, vamos explorar algumas de suas pegadinhas.

ARGUMENTOS OBJETOS COMO ALIAS PARA PARÂMETROS DE FUNÇÃO

O argumentos parâmetro tem uma característica curiosa: ele cria um apelido para parâmetros de função. Se definirmos um novo valor para, por exemplo, argumentos [0], o valor do primeiro parâmetro também será alterado. Dê uma olhada na lista a seguir.

Listagem 4.3 O argumentos parâmetros de função de aliases de objeto

<pre>função infiltrada (pessoa) { afirmar (pessoa === 'jardineiro', 'A pessoa é jardineiro'); assert (argumentos [0] === 'jardineiro', 'O primeiro argumento é um jardineiro'); argumentos [0] = 'ninja'; afirmar (pessoa === 'ninja', 'A pessoa é um ninja agora'); assert (argumentos [0] === 'ninja', 'O primeiro argumento é um ninja'); pessoa = 'jardineiro'; afirmar (pessoa === 'jardineiro', 'A pessoa é jardineiro mais uma vez'); assert (argumentos [0] === 'jardineiro', 'O primeiro argumento é um jardineiro novamente'); }</pre>	<p>O parâmetro pessoa tem o valor "jardineiro" enviado como primeiro argumento.</p>
	<p>Mudando os argumentos objeto também mudará o parâmetro correspondente.</p>
<pre>infiltrar ("jardineiro");</pre>	<p>O alias funciona em ambos os sentidos.</p>

Você pode ver como o argumentos object é um alias para os parâmetros da função. Nós definimos uma função, infiltrar, que leva um único parâmetro, pessoa, e nós o invocamos com o argumento jardineiro. Podemos acessar o valor jardineiro através do parâmetro de função pessoa e através do argumentos objeto:

```
afirme (pessoa === 'jardineiro', 'A pessoa é um jardineiro'); assert (argumentos [0] === 'jardineiro', 'O primeiro argumento é um jardineiro');
```

Porque o argumentos objeto é um alias para os parâmetros da função, se mudarmos o argumentos objeto, a mudança também se reflete no parâmetro de função correspondente:

```
argumentos [0] = 'ninja';  
  
afirme (pessoa === 'ninja', 'A pessoa é um ninja agora'); assert (argumentos [0] === 'ninja', 'O primeiro argumento é um ninja');
```

O mesmo se aplica à outra direção. Se mudarmos um parâmetro, a mudança pode ser observada tanto no parâmetro quanto no argumentos objeto:

```

pessoa = 'jardineiro';

afirmar (pessoa === 'jardineiro',
  'A pessoa é jardineiro mais uma vez'); assert (argumentos [0] ===
'jardineiro',
  'O primeiro argumento é um jardineiro novamente');

```

EVITANDO ALIASES

O conceito de aliasing de parâmetros de função por meio do argumentos objeto pode ser confuso, portanto, o JavaScript fornece uma maneira de recusar usando *modo estrito*.

Modo estrito

O modo estrito é uma adição ES5 ao JavaScript que altera o comportamento dos mecanismos JavaScript para que os erros sejam lançados em vez de detectados silenciosamente. O comportamento de alguns recursos de linguagem é alterado, e alguns recursos de linguagem insegura são completamente banidos (mais sobre isso mais tarde). Uma das coisas que o modo estrito muda é que ele desativa argumentos aliasing.

Como sempre, vamos dar uma olhada em um exemplo simples.

Listagem 4.4 Usando o modo estrito para evitar argumentos aliasing

```

"use estrito";
                                  <----- Ativa o modo estrito

função infiltrada (pessoa) {
  afirmar (pessoa === 'jardineiro',
    'A pessoa é jardineiro'); assert (argumentos [0] === 'jardineiro',
    'O primeiro argumento é um jardineiro');

  argumentos [0] = 'ninja';
                                  <----- Muda o primeiro argumento

  assert (argumentos [0] === 'ninja',
    'O primeiro argumento agora é um ninja');

  afirmar (pessoa === 'jardineiro',
    'A pessoa ainda é um jardineiro');
}

infiltrar ("jardineiro");

```

O argumento da pessoa e o primeiro argumento começam com o mesmo valor.

O primeiro argumento Mudou.

O valor do parâmetro pessoa não mudou.

Aqui começamos colocando a string simples use estrito como a primeira linha de código. Isso informa ao mecanismo JavaScript que desejamos executar o código a seguir no modo estrito. Neste exemplo, o modo estrito altera a semântica do nosso programa de forma que o pessoa parâmetro e o primeiro argumento comece com o mesmo valor:

```
afirme (pessoa === 'jardineiro', 'A pessoa é um jardineiro'); assert (argumentos [0] === 'jardineiro', 'O primeiro argumento é um jardineiro');
```


Mas, ao contrário do modo não estrito, desta vez em torno do argumentos objeto não alias aos parâmetros. Se mudarmos o valor do primeiro argumento, argumentos [0] = 'ninja', o primeiro argumento é alterado, mas o pessoa parâmetro não é:

```
assert (argumentos [0] === 'ninja', 'O primeiro argumento agora é um ninja'); assert (pessoa === 'jardineiro', 'A pessoa ainda é um jardineiro');
```

Vamos revisitar o argumentos objeto posteriormente neste livro, mas por enquanto, vamos nos concentrar em outro parâmetro implícito: esta, o que é de certa forma ainda mais interessante.

4.1.2 O parâmetro this: introduzindo o contexto da função

Quando uma função é chamada, além dos parâmetros que representam os argumentos explícitos fornecidos na chamada da função, um parâmetro implícito denominado esta é passado para a função. O esta parâmetro, um ingrediente vital em JavaScript orientado a objetos, refere-se a um objeto que está associado à invocação da função. Por este motivo, é frequentemente denominado de *contexto da função*.

O contexto da função é uma noção que aqueles que vêm de linguagens orientadas a objetos, como Java, podem pensar que entendem. Em tais linguagens, esta geralmente aponta para uma instância da classe dentro da qual o método é definido.

Mas cuidado! Como veremos em breve, em JavaScript, invocar uma função como um *método* é apenas uma maneira pela qual uma função pode ser chamada. E ao que parece, o que esta parâmetro aponta para não (como em Java ou C #) definido apenas por como e onde a função é definida; também pode ser fortemente influenciado por como a função é *invocada*. Porque compreender a natureza exata do esta parâmetro é um dos pilares mais importantes do JavaScript orientado a objetos, estamos prestes a examinar várias maneiras de invocar funções. Você verá que uma das principais diferenças entre eles é como o valor de esta está determinado. E então daremos uma olhada longa e detalhada nos contextos de função novamente nos próximos capítulos, então não se preocupe se as coisas não se encaixarem imediatamente.

Agora vamos ver, em grande detalhe, como as funções podem ser chamadas.

4,2 Invocando funções

Todos nós chamamos funções JavaScript, mas você já parou para se perguntar o que realmente acontece quando uma função é chamada? Acontece que a maneira como uma função é chamada tem um grande impacto em como o código dentro dela opera, principalmente em como o esta parâmetro, o contexto da função, é estabelecido. Essa diferença é muito mais importante do que pode parecer à primeira vista. Vamos examiná-lo nesta seção e explorá-lo no restante deste livro para ajudar a elevar nosso código ao nível ninja.

Podemos invocar uma função de quatro maneiras, cada uma com suas próprias nuances:

- *Como uma função* - skulk (), em que a função é chamada de maneira direta
- *Como método* - ninja.skulk (), que liga a invocação a um objeto, permitindo a programação orientada a objetos

- *Como construtor* - novo Ninja (), em que um novo objeto é trazido à existência
- *Através da função Aplique ou chamar métodos* - skulk.call (ninja) ou skulk.apply (ninja)

Aqui estão alguns exemplos:

```
função skulk (nome) {}
```

```
função Ninja (nome) {}
```

```
skulk ('Hattori');
```

```
(função (quem) {retornar quem;}) ('Hattori');
```

Chamado como uma função

```
var ninja = {
```

```
  skulk: function () {}
```

```
};
```

```
ninja.skulk ('Hattori');
```

← Invocado como um método de ninja

```
ninja = novo Ninja ('Hattori');
```

← Chamado como um construtor

```
skulk.call (ninja, 'Hattori');
```

← Chamado por meio do método call

```
skulk.apply (ninja, ['Hattori']);
```

← Chamado por meio do método apply

Para todos, exceto o chamar e Aplique abordagens, o operador de invocação de função é um conjunto de parênteses após qualquer expressão avaliada para uma referência de função.

Vamos começar nossa exploração com a forma mais simples, invocando funções como funções.

4.2.1 *Invocação como uma função*

Invocação como função? Bem, é claro que as funções são chamadas como *funções*. Que tolice pensar de outra forma. Mas, na realidade, dizemos que uma função é invocada "como uma função" para distingui-la dos outros mecanismos de invocação: métodos, construtores e aplicar / ligar. Se uma função não for chamada como um método, como um construtor ou via Aplique ou chamar, é invocado como uma função.

Esse tipo de chamada ocorre quando uma função é chamada usando o operador () e a expressão à qual o operador () é aplicado não faz referência à função como uma propriedade de um objeto. (Nesse caso, teríamos uma invocação de método, mas discutiremos isso a seguir.) Aqui estão alguns exemplos simples:

```
função ninja () {};
```

```
ninja ();
```

Declaração de função

invocado como uma função

Imediatamente invocado
expressão de função,
invocado como uma função

```
var samurai = função () {}; samurai();
```

```
(função(){}())()
```

Expressão de função

invocado como uma função

Quando chamado desta maneira, o contexto da função (o valor do esta palavra-chave) pode ser duas coisas: No modo não estrito, será o contexto global (o janela objeto), enquanto no modo estrito, será Indefinido.

A lista a seguir ilustra a diferença de comportamento entre os modos estrito e não estrito.

Listagem 4.5 Invocação como uma função

<pre>function ninja () { devolva isso; }</pre>	<p>Uma função em modo não estrito</p>
<pre>function samurai () { "use estrito"; devolva isso; }</pre>	<p>Uma função em modo estrito</p>
<pre>janela assert (ninja () ===, "Em uma função ninja 'não estrita'," + "o contexto é o objeto de janela global");</pre>	<p>Como esperado, uma função não estrita tem janela como contexto de função.</p>
<pre>assert (samurai () === indefinido, "Em uma função de samurai 'estrита'," + "o contexto é indefinido");</pre>	<p>A função estrita, por outro lado, possui um contexto indefinido.</p>

NOTA Como você pode ver, o modo estrito é, na maioria dos casos, muito mais simples do que o modo não estrito. Por exemplo, quando a listagem 4.5 invoca uma função como uma função (em oposição a um método), ela não especificou um objeto no qual a função deve ser invocada. Então, em nossa opinião, faz mais sentido que o esta palavra-chave deve ser definida para Indefinido (como no modo estrito), em oposição ao global janela objeto (como no modo não estrito). Em geral, o modo estrito corrige muitas dessas pequenas esquisitices do JavaScript. (Lembra-se dos argumentos com alias do início do capítulo?)

Você provavelmente já escreveu um código como este muitas vezes sem se preocupar muito. Agora vamos subir um degrau, vendo como as funções são chamadas como *métodos*.

4.2.2 Invocação como método

Quando uma função é atribuída a uma propriedade de um objeto e a invocação ocorre referenciando a função usando essa propriedade, então a função é invocada como um *método* desse objeto. Aqui está um exemplo:

```
var ninja = {};
ninja.skulk = função () {};
ninja.skulk ();
```

OK; E daí? A função é chamada de *método* neste caso, mas o que torna isso interessante ou útil? Bem, se você vem de um plano de fundo orientado a objetos, você se lembrará de que o objeto ao qual um método pertence está disponível dentro do corpo do método como esta. A mesma coisa acontece aqui. Quando invocamos uma função como um *método* de um objeto, esse objeto se torna o contexto da função e está disponível dentro

a função através do este parâmetro. Este é um dos principais meios pelos quais o JavaScript permite que o código orientado a objetos seja escrito. (Construtores são outra, e vamos chegar a eles em breve.)

Vamos considerar alguns códigos de teste na próxima listagem para ilustrar as diferenças e semelhanças entre a chamada como uma função e a chamada como um método.

Listagem 4.6 As diferenças entre invocações de função e método

getMyThis obtém um referência ao whatsMyContext função.	<pre>function whatsMyContext () { devolva isso; }</pre> <pre>assert (whatsMyContext () === window, "Chamada de função na janela");</pre>	Retorna o contexto da função que nos permitirá examinar o contexto de fora
Um objeto ninja1 é criado com um getMyThis propriedade que faz referência a whatsMyContext função.	<pre>var getMyThis = whatsMyContext;</pre> <pre>assert (getMyThis () === window, "Outra chamada de função na janela");</pre> <pre>var ninja1 = { getMyThis: whatsMyContext };</pre> <pre>assert (ninja1.getMyThis () === ninja1, "Trabalhando com o 1º ninja");</pre>	Invoca a função usando a variável getMyThis. Até embora usemos uma variável, a função ainda é chamada como uma função e o contexto da função é o objeto da janela.
Outro objeto, ninja2, também tem uma propriedade getMyThis referenciando whatsMyContext.	<pre>var ninja2 = { getMyThis: whatsMyContext };</pre> <pre>assert (ninja2.getMyThis () === ninja2, "Trabalhando com o 2º ninja");</pre>	Invocar as funções por meio de getMyThis o chama como um método de ninja1. O contexto da função agora é ninja1. Isso é orientação a objetos!
		Invocar a função como um método de ninja2 mostra que o contexto da função agora é ninja2.

Este teste configura uma função chamada `whatsMyContext` que usaremos no restante da lista. A única coisa que essa função faz é retornar seu contexto de função para que possamos ver, de fora da função, qual é o contexto da função para a chamada. (Caso contrário, teríamos dificuldade em saber.)

```
function whatsMyContext () {
    devolva isso;
}
```

Quando chamamos a função diretamente pelo nome, este é um caso de invocar a função como uma função, então esperamos que o contexto da função seja o contexto global (`janela`), porque estamos no modo não estrito. Afirmamos que é assim:

```
assert (whatsMyContext () === window, ...)
```

Em seguida, criamos uma referência para a função `whatsMyContext` em uma variável chamada `getMyThis`: `var getMyThis = whatsMyContext`. Isso não cria uma segunda instância da função; apenas cria uma referência para a mesma função (você sabe, objeto de primeira classe e tudo).

Quando invocamos a função por meio da variável - algo que podemos fazer porque o operador de invocação de função pode ser aplicado a qualquer expressão avaliada como uma função - estamos novamente invocando a função como uma função. Como tal, esperamos novamente que o contexto da função seja janela, e isso é:

```
assert (getMyThis () === window,
        "Outra chamada de função na janela");
```

Agora, ficamos um pouco mais complicados e definimos um objeto em variável `ninja1` com uma propriedade chamada `getMyThis` que recebe uma referência ao `whatsMyContext` função. Ao fazer isso, dizemos que criamos um *método* nomeado `getMyThis` no objeto. Nós não dizemos isso `whatsMyContext` tem *tornar-se* um método de `ninja1`; não tem. Você já viu isso `whatsMyContext` é sua própria função independente que pode ser chamada de várias maneiras:

```
var ninja1 = {
  getMyThis: whatsMyContext
};
```

De acordo com o que afirmamos anteriormente, quando invocamos a função por meio de uma referência de método, esperamos que o contexto da função seja o objeto do método (neste caso, `ninja1`) e afirmamos tanto:

```
assert (ninja1.getMyThis () === ninja1,
        "Trabalhando com o 1º ninja");
```

NOTA Invocar funções como métodos é crucial para escrever JavaScript de uma maneira orientada a objetos. Isso permite que você use esta dentro de qualquer método para fazer referência ao objeto "proprietário" do método - um conceito fundamental na programação orientada a objetos.

Para esclarecer esse ponto, continuamos testando criando outro objeto, `ninja2`, também com uma propriedade chamada `getMyThis` que faz referência a `whatsMyContext` função. Ao invocar esta função por meio do `ninja2` objeto, afirmamos corretamente que seu contexto de função é `ninja2`:

```
var ninja2 = {
  getMyThis: whatsMyContext
};

assert (ninja2.getMyThis () === ninja2,
        "Trabalhando com o 2º ninja");
```

Mesmo que a *mesma* função- `whatsMyContext` —É usado em todo o exemplo, o contexto da função retornado por esta muda dependendo de como `whatsMyContext` é invocado. Por exemplo, a mesma função exata é compartilhada por ambos `ninja1` e `ninja2`,

no entanto, quando é executada, a função tem acesso e pode realizar operações no objeto por meio do qual o método foi invocado. Não precisamos criar cópias separadas de uma função para realizar exatamente o mesmo processamento em objetos diferentes. Este é um princípio da programação orientada a objetos.

Embora seja um recurso poderoso, a maneira como é usado neste exemplo tem limitações. Acima de tudo, quando criamos os dois objetos ninja, podemos compartilhar a mesma função a ser usada como método em cada um, mas temos que usar um pouco de código repetido para configurar os objetos separados e seus métodos.

Mas não há nada para se desesperar - o JavaScript fornece mecanismos para tornar a criação de objetos a partir de um único padrão muito mais fácil do que neste exemplo. Exploraremos esses recursos em profundidade no capítulo 7. Mas, por agora, vamos considerar uma parte desse mecanismo que se relaciona às invocações de função: o *construtor*.

4.2.3 *Invocação como um construtor*

Não há nada de especial em uma função que será usada como construtor. *Funções de construtor* são declaradas como quaisquer outras funções, e podemos usar facilmente declarações e expressões de função para construir novos objetos. A única exceção é a função de seta, que, como você verá mais adiante neste capítulo, funciona de maneira um pouco diferente. Mas, em qualquer caso, a principal diferença está em como a função é chamada.

Para invocar a função como um construtor, precedemos a invocação da função com a palavra-chave `new`. Por exemplo, lembre-se do `whatsMyContext` função da seção anterior:

```
função whatsMyContext () {retornar isto; }
```

Se quisermos invocar o `whatsMyContext` funcionar como um construtor, escrevemos isto:

```
new whatsMyContext ();
```

Mas mesmo que possamos invocar `whatsMyContext` como um construtor, essa função não é um construtor particularmente útil. Vamos descobrir o porquê discutindo o que torna os construtores especiais.

NOTA Lembre-se do capítulo 3, quando discutimos maneiras de definir funções? Entre as declarações de função, expressões de função, funções de seta e funções de gerador, também mencionamos *construtores de função*, que nos permitem construir novas funções a partir de strings. Por exemplo: nova função ('a', 'b', 'retornar a + b') cria uma nova função com dois parâmetros,

uma e b, que retorna sua soma. Tenha cuidado para não confundir estes *construtores de função* com *funções de construtor*! A diferença é sutil, mas significativa. Um construtor de função nos permite criar funções a partir de strings criadas dinamicamente. Por outro lado, *funções construtoras*, o tópico desta seção, são funções que usamos para criar e inicializar instâncias de objeto.

AS SUPERPOTÊNCIAS DOS CONSTRUTORES

Invocar uma função como um construtor é um recurso poderoso do JavaScript que exploraremos na listagem a seguir.

Listagem 4.7

Usando um construtor para configurar objetos comuns

Cria dois objetos invocando o construtor com novo. O novo objetos criados são referenciados por ninja1 e ninja2.

```
function Ninja () {
  this.skulk = function () {
    devolva isso;
  };
}

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

assert (ninja1.skulk () === ninja1,
  "O primeiro ninja está se escondendo"); afirmar
(ninja2.skulk () === ninja2,
  "O segundo ninja está se escondendo");
```

Um construtor que cria uma propriedade skulk em qualquer objeto que seja o contexto da função. O método mais uma vez retorna o contexto da função para que possamos testá-lo externamente.

Testa o método skulk dos objetos construídos.
Cada um deve retornar seu próprio objeto construído.

Neste exemplo, criamos uma função chamada Ninja que usaremos para construir, bem, ninjas. Quando invocado com a palavra-chave novo, uma instância de objeto vazio é criada e passada para a função como seu contexto de função, o esta parâmetro. O construtor cria uma propriedade chamada esconder-se neste objeto, ao qual é atribuída uma função, tornando essa função um método do objeto recém-criado.

Em geral, quando um construtor é invocado, algumas ações especiais ocorrem, conforme mostrado na figura 4.1.

Chamando uma função com a palavra-chave novo aciona as seguintes etapas:

- 1 Um novo objeto vazio é criado.
- 2 Este objeto é passado para o construtor como o esta parâmetro e, portanto, torna-se o contexto da função do construtor.
- 3 O objeto recém-construído é retornado como o novo valor do operador (com uma exceção que veremos em breve).

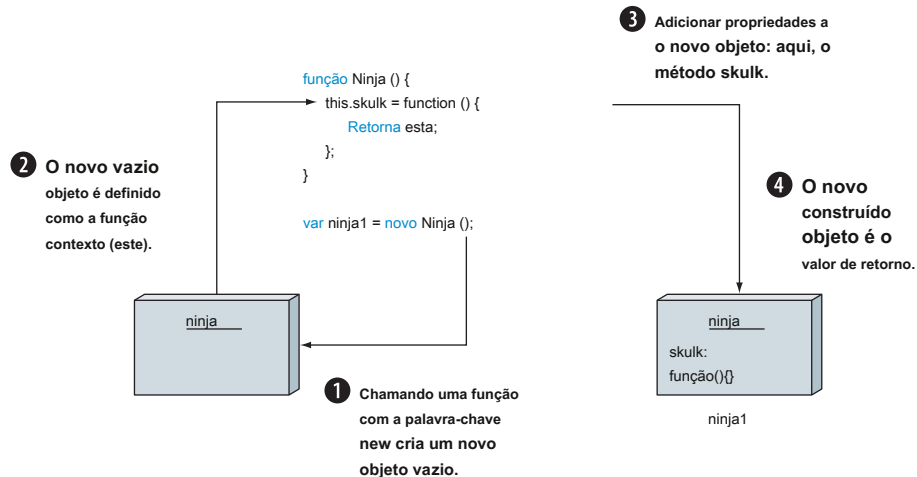


Figura 4.1 Ao chamar uma função com uma palavra-chave **novo**, um novo objeto vazio é criado e definido como o contexto da função do construtor, o **esta** parâmetro.

Os dois últimos pontos tocam no porquê `whatsMyContext` no novo `whatsMyContext ()` faz para um péssimo construtor. O objetivo de um construtor é fazer com que um novo objeto seja criado, configurá-lo e retorná-lo como o valor do construtor. Qualquer coisa que interfira com essa intenção não é apropriada para construtores.

Vamos considerar um construtor mais apropriado, `Ninja`, que configura ninjas furtivos, conforme mostrado na listagem 4.7:

```
function Ninja () {
  this.skulk = function () {
    devolva isso;
  };
}
```

O esconder-se método executa a mesma operação que `whatsMyContext` nas seções anteriores, retornando o contexto da função para que possamos testá-la externamente.

Com o construtor definido, criamos dois novos `Ninja` objetos invocando o construtor duas vezes. Observe que os valores retornados das invocações são armazenados em variáveis que se tornam referências ao recém-criado `Ninja` s:

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

Em seguida, executamos os testes que garantem que cada invocação do método opere no objeto esperado:

```
assert (ninja1.skulk () === ninja1,
  "O primeiro ninja está se escondendo"); afirmar
(ninja2.skulk () === ninja2,
  "O segundo ninja está se escondendo");
```

É isso! Agora você sabe como criar e inicializar novos objetos com funções construtoras. Chamar uma função com a palavra-chave `novo` retorna o objeto recém-criado. Mas vamos verificar se isso é sempre exatamente verdade.

VALORES DE RETORNO DO CONSTRUTOR

Mencionamos anteriormente que os construtores se destinam a inicializar objetos recém-criados e que o objeto recém-construído é retornado como resultado de uma invocação do construtor (por meio do `novo` operador). Mas o que acontece quando o construtor retorna um valor próprio? Vamos explorar essa situação na lista a seguir.

Listagem 4.8 Construtores retornando valores primitivos

```
function Ninja () {
  this.skulk = function () {
    return true;
  };
  return 1;
}
```

Define uma função construtora chamada `Ninja`

O construtor retorna um valor primitivo específico, o número 1.


```

    }
    afirmar (Ninja () === 1,
        "Valor de retorno honrado quando não chamado como construtor");
    var ninja = novo Ninja ();
    assert (typeof ninja === "objeto",
        "Objeto retornado quando chamado como construtor"); assert (typeof ninja.skulk ===
        "função",
        "objeto ninja tem um método skulk");

```

A função é chamado como um função e seu valor de retorno é 1, conforme esperado.

A função é chamada como um construtor por meio do novo operador.

Os testes verificam se o valor de retorno de 1 é ignorado e se um novo valor inicializado objeto foi voltou de novo.

Se executarmos esta lista, veremos que está tudo bem e bem. O fato de que isso Ninja função retorna um número simples 1 não tem influência significativa sobre como o código se comporta. Se chamarmos o Ninja função como uma função, ele retorna 1 (tal como esperávamos); e se o chamarmos de construtor, com a palavra-chave novo, uma nova ninja objeto é construído e retornado. Até agora tudo bem.

Mas agora vamos tentar algo diferente, uma função construtora que retorna outro objeto, conforme mostrado na lista a seguir.

Listagem 4.9 Construtores que retornam explicitamente os valores do objeto

```

var puppet = {
    regras: falso
};

function Imperador () {
    this.rules = true;
    boneco de retorno;
}

imperador var = novo imperador ();

afirmar (imperador === fantoche,
    "O imperador é apenas um fantoche!"); assert (emperador.rules ===
false,
    "O fantoche não sabe governar!");

```

Cria o nosso objeto com um propriedade conhecida

Retorna aquele objeto apesar de inicializar o objeto passado como este

Invoca a função como um construtor

Os testes mostram que o objeto retornado pelo construtor é atribuído à variável imperador (e não o objeto criado pela nova expressão).

Esta lista tem uma abordagem ligeiramente diferente. Começamos criando um fantoche objeto com a propriedade as regras definido como falso:

```

var puppet = {
    regras: falso
};

```

Então, definimos um Imperador função que adiciona um as regras propriedade para o objeto recém-construído e a define como verdade. Além disso, o Imperador função tem uma peculiaridade; retorna o global fantoche objeto:

```
function Emperor () {
  this.rules = true;
  boneco de retorno;
}
```

Mais tarde, chamamos o Imperador funcionar como um construtor, com a palavra-chave novo:

```
imperador var = novo imperador ();
```

Com isso, criamos uma situação ambígua: obtemos um objeto passado para o construtor como o contexto da função em esta, que inicializamos, mas depois retornamos um completamente diferente fantoche objeto. Qual objeto reinará supremo?

Vamos testar:

```
afirmam (imperador === fantoche, "O imperador é apenas um fantoche!"); assert (emperor.rules === false,
  "O fantoche não sabe governar!");
```

Acontece que nossos testes indicam que o fantoche objeto é retornado como o valor da invocação do construtor, e que a inicialização que realizamos no contexto da função no construtor foi em vão. O fantoche foi exposto!

Agora que passamos por alguns testes, vamos resumir nossas descobertas:

- Se o construtor retorna um objeto, esse objeto é retornado como o valor de todo novo expressão, e o objeto recém-construído passado como esta para o construtor é descartado.
- Se, no entanto, um não-objeto for retornado do construtor, o valor retornado será ignorado e o objeto recém-criado será retornado.

Por causa dessas peculiaridades, as funções destinadas ao uso como construtores são geralmente codificadas de maneira diferente de outras funções. Vamos explorar isso com mais detalhes.

CONSIDERAÇÕES DE CODIFICAÇÃO PARA CONSTRUTORES

A intenção dos construtores é inicializar o novo objeto que será criado pela invocação da função para as condições iniciais. E embora tais funções *posso* ser chamados como funções “normais”, ou mesmo atribuídos a propriedades de objeto para serem chamados como métodos, eles geralmente não são úteis como tal. Por exemplo

```
function Ninja () {
  this.skulk = function () {
    devolva isso;
  };
}
var qualquer = Ninja ();
```

Podemos ligar Ninja como uma função simples, mas o esconder-se propriedade seria criada em janela no modo não estrito - não é uma operação particularmente útil. As coisas ficam ainda mais complicadas no modo estrito, como esta seria indefinido e nosso aplicativo JavaScript travaria. Mas isso é uma coisa boa; se cometermos esse erro no modo não estrito, pode