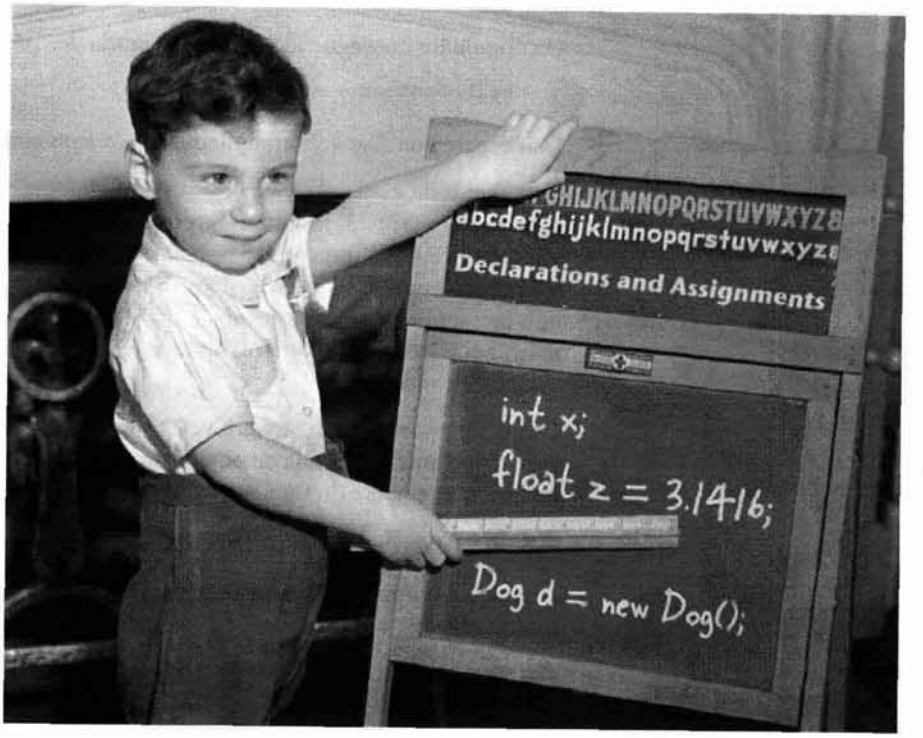
Conheça suas Variáveis



Existem duas versões de variáveis: primitivas e de referência. Até agora você usou variáveis em duas situações — como **estado** do objeto (variáveis de instância) e como variáveis **locais** (variáveis declaradas dentro de um *método*). Posteriormente, usaremos variáveis como **argumentos** (valores enviados para um método pelo código que o chamou) e como **tipos de retorno** (valores retornados ao código que chamou o método). Você viu variáveis declaradas como valores inteiros **primitivos** simples (tipo int). Examinou variáveis declaradas como algo mais **complexo** do tipo string ou matriz. Porém há mais coisas na vida além de inteiros, strings e matrizes. E se você tiver um objeto DonodeAnimal com uma variável de instância Cão? Ou um Carro com um Motor? Neste capítulo desvelaremos os mistérios dos tipos Java e examinaremos o que você pode *declarar* como uma variável, o que pode *inserir* em uma variável e o que pode *fazer* com ela. E, para concluir, discutiremos o que acontece *realmente* na pilha de lixo coletável.

A Java considera o tipo importante. Você não pode inserir uma girafa em uma variável Coelho.

Declarando uma variável

O Java considera o tipo importante. Ele não permitirá que você faça algo bizarro e perigoso como inserir a referência de uma girafa em uma variável Coelho — o que aconteceria quando alguém tentasse pedir ao suposto *coelho* para saltar( )? E não permitirá que insira um número de ponto flutuante em uma variável de tipo inteiro, a menos que você *informe ao compilador* que sabe que pode perder a precisão (o que se encontra após a vírgula decimal).

O compilador consegue identificar a maioria dos problemas:

```
Coelho saltador = new Girafa( );
```

Não espere que isso seja compilado. *Ainda bem que não será.*

Para que toda essa segurança dos tipos funcione, você deve declarar o tipo de sua variável. Ela é um inteiro? Um Cão? Um único caractere? As variáveis vêm em duas versões: ***primitivas*** e ***de referência de objeto***. As primitivas contêm valores básicos (pense em padrões de bits simples) que incluem inteiros, booleanos e números de ponto flutuante. As referências de objeto contêm, bem, *referências* a *objetos*. (Puxa! *Isso* não esclareceu tudo?)

Examinaremos primeiro as variáveis primitivas e, em seguida, passaremos para o que uma referência de objeto significa realmente. Mas independentemente do tipo, você deve seguir duas regras de declaração:

As variáveis devem ter um tipo

Além de um tipo, uma variável precisa de um nome, para que você possa usar esse nome no código.

As variáveis devem ter um nome

```
int count;
```

tipo *nome*

Nota: quando você se deparar com uma instrução como "um objeto de **tipo X**", pense em *tipo* e *classe* como sinônimos. (Detalharemos isso um pouco mais em capítulos posteriores.)

"Gostaria de um café duplo, não traga um do tipo inteiro."

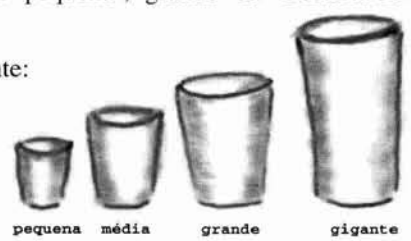
Quando você pensar em variáveis Java, pense em xícaras. Xícaras de café, xícaras de chá, canecas gigantes onde cabe muita cerveja, esses grandes copos em que as pipocas são vendidas no cinema, xícaras com alças curvilíneas e sexy e canecas com acabamento metálico que lhe disseram para nunca colocar em um microondas.

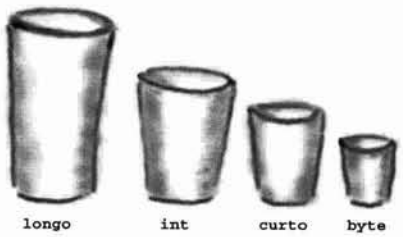
Uma variável é apenas uma xícara. Um contêiner. Ela *contém* algo.

Ela tem um tamanho e um tipo. Neste capítulo, examinaremos primeiro as variáveis (xícaras) que contêm tipos **primitivos** e, um pouco mais adiante, discutiremos as xícaras que contêm *referências a objetos*. Não deixe de acompanhar toda a nossa analogia com as xícaras — tão simples como está sendo agora, ela nos fornecerá uma maneira comum de examinar as coisas quando a discussão ficar mais complexa. E isso ocorrerá em breve.

As variáveis primitivas são como as xícaras que vemos nos cafés. Se você já foi a um Starbucks nos Estados Unidos, sabe sobre o que estamos falando aqui. Elas têm tamanhos diferentes e cada uma tem um nome como 'pequena', 'grande' ou "Gostaria de um 'moca' grande com pouca cafeína e chantilly".

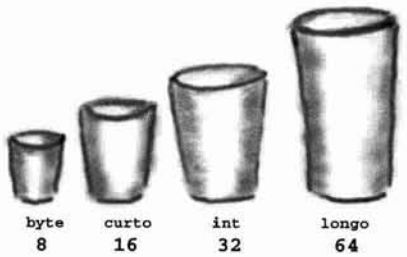
Podemos ver as xícaras dispostas no balcão, portanto, é possível ordená-las corretamente:

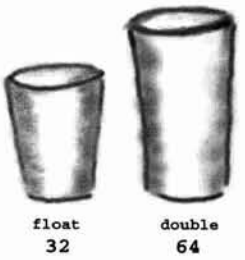




E, em Java, existem tamanhos diferentes para as variáveis primitivas e esses tamanhos têm nomes. Quando você declarar uma variável em Java, deve declará-la com um tipo específico. Os quatro contêineres mostrados aqui são para os quatro tipos primitivos inteiros em Java.

Cada xícara contém uma quantidade, o mesmo ocorrendo com as variáveis primitivas Java, de modo que, em vez de dizer "Quero uma xícara grande de café francês torrado", você diria ao compilador "Quero uma variável int com o número 90, por favor". Exceto por uma pequena diferença... Em Java, você também terá que fornecer um *nome* para sua xícara. Portanto, na verdade diríamos "Quero um int, por favor, com o valor 2.486 e chame a variável de ***tamanho***". Cada variável primitiva possui uma quantidade fixa de bits (tamanho da xícara). Os tamanhos das seis variáveis primitivas numéricas em Java são mostrados a seguir:





Tipos primitivos

Tipo Quantidade de bits Intervalo de valores

Bolleano e char

Booleano (específica da JVM) ***verdadeiro*** ou ***falso***

```
char     16 bits          0 a 65535
```

numéricos (todos têm sinal)

inteiro

```
byte     8 bits           -128 a 127
curto    16 bits          -32768 a 32767
int      32 bits          -2147483648 a 2147483647
longo    64 bits          -enorme a enorme
```

ponto flutuante

```
float    32 bits          varia
double   64 bits          varia
```

Declarações primitivas com atribuições:

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean powerOn;
powerOn = isFun;
long big = 3456789;
float f = 32.5f;
```

Observe o 'f'. É preciso inseri-lo em um tipo float, porque o Java considera tudo que encontra com um ponto flutuante como um tipo double, a menos que 'f' seja usado.

Você não queria derramar isso *realmente*...

Certifique-se de que o valor cabe na variável.

Você não pode desejar uma quantidade grande em uma xícara pequena.

Bem, certo, você pode, mas vai perder uma parte. Você terá o que chamamos de *derramamento*. O compilador tentará ajudar a impedir isso se conseguir perceber que algo em seu código não caberá no contêiner (variável/xícara) que você está usando.

Por exemplo, você não pode despejar muitos inteiros em um contêiner de tamanho byte, como descrito a seguir:

```
int x = 24;
byte b = x;
//não funcionará!!
```

Por que isso não funcionou, você poderia perguntar? Afinal, o valor de *x* é 24, e 24 definitivamente é um valor suficientemente baixo para caber em um tipo byte. *Você* sabe disso, e nós também, mas tudo que importa ao compilador é que houve a tentativa de se inserir algo grande em um recipiente pequeno, e há a *possibilidade* de derramamento. Não espere que o compilador saiba qual é o valor de x, mesmo se por acaso você puder vê-lo literalmente em seu código.

Você pode atribuir um valor a uma variável de várias maneiras dentre elas:

- digitar um valor *literal* depois do sinal de igualdade (x = ***12***, isGod = ***true***, etc).

- atribuir o valor de uma variável a outra (x = y)

- usar uma expressão combinando os dois (x = y + ***43***)

Nos exemplos a seguir, os valores literais estão em itálico e negrito:

```
int size = 32;          declara um int chamado size e atribui a ele o valor 32
char initial = 'j';     declara um char chamado initial e atribui a ele o valor 'j'
double d = 456,709;     declara um double chamado d e atribui a ele o valor 456,709
boolean isCrazy;        declara um booleano chamado isCrazy (sem atribuição)
int y = x + 456;        declara um int chamado y e atribui a ele um valor que é igual à
                        soma do valor atual de x mais 456
```

Aponte seu lápis

O compilador não deixará que você insira a quantidade de uma xícara grande em uma pequena. Mas e quanto à operação inversa — despejar o conteúdo de uma xícara pequena em uma grande? ***Sem problemas***.

Baseado no que você sabe sobre o tamanho e o tipo das variáveis primitivas, veja se consegue descobrir quais dessas linhas são válidas e quais não são. Não abordamos todas as regras ainda, portanto, em algumas das opções, você terá que usar o seu melhor palpite. ***Dica:*** sempre que o compilador erra, é em nome da segurança.

Na lista a seguir, ***circule*** as instruções que seriam válidas se essas linhas estivessem em um único método:

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

Afaste-se dessa palavra-chave!

Você sabe que precisa de um nome e de um tipo para suas variáveis. Você já conhece os tipos primitivos.

Mas o que pode usar como nomes? As regras são simples. Você pode nomear uma classe, método ou variável de acordo com as regras a seguir (as regras reais são um pouco mais flexíveis, mas estas o manterão em segurança):

- Ele deve começar com uma letra, um sublinhado (_) ou o cifrão ($). Você não pode iniciar um nome com um número.

- Depois do primeiro caractere, você também pode usar números. Só não comece com um número.

- O nome pode ser o que você quiser, se obedecer as regras, contanto que não seja uma das palavras reservadas do Java.

As palavras reservadas são palavras-chave (e outras coisas) que o compilador reconhece. Mas se você quiser realmente brincar de confundir o compilador, simplesmente *tente* usar uma palavra reservada como um nome.

Você já viu algumas palavras reservadas quando examinamos a criação de nossa primeira classe principal:

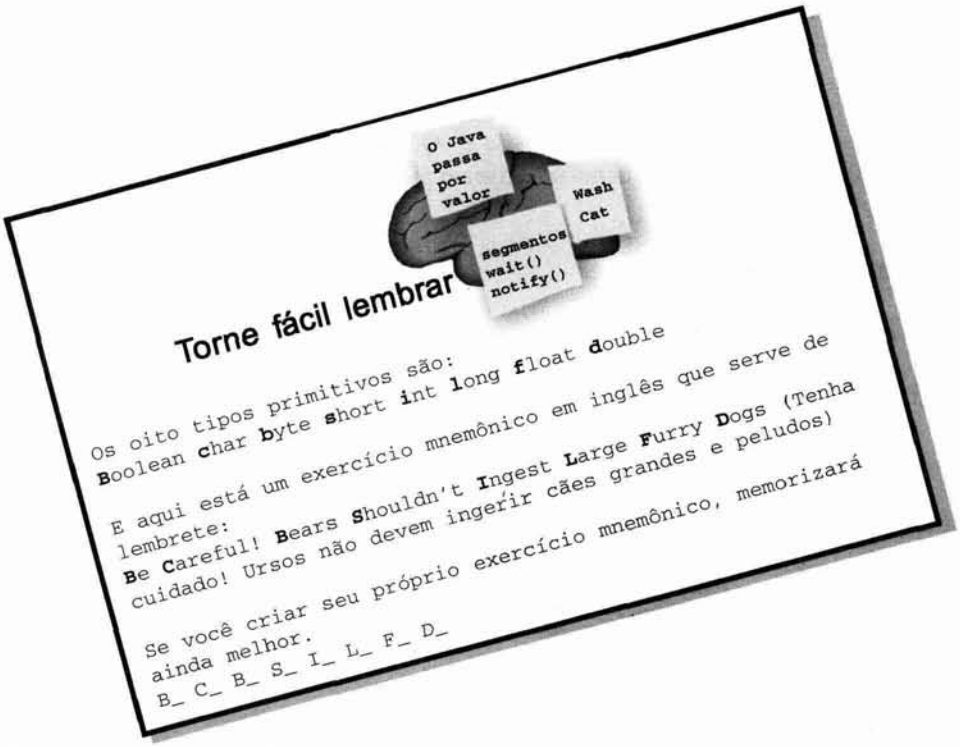
```
public   static   void
```

não use nenhuma dessas palavras nos nomes que criar.

E os tipos primitivos também são reservados:

```
boolean   char   byte   short   int   long   float   double
```

Mas há muitas outras que ainda não discutimos. Mesmo se você não precisar saber o que elas significam, terá que saber que não pode usá-las em suas criações. ***Não tente — de forma alguma — memorizá-las agora***. Para reservar espaço em sua mente, provavelmente você teria que perder alguma outra coisa. Como onde seu carro está estacionado. Não se preocupe, até o fim do livro você terá a maioria delas memorizada.



Essa tabela está reservada

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	stricfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

As palavras-chave Java e outras palavras reservadas (em ordem aleatória). Se você usá-las como nomes, o compilador ficará muito confuso.

Controlando seu objeto Dog

Você sabe como declarar uma variável primitiva e atribuir a ela um valor. Mas e quanto às variáveis não primitivas? Em outras palavras, *e quanto aos objetos?*

- Na verdade não há uma variável de OBJETO.
- Há apenas uma variável de REFERÊNCIA de objeto.
- Uma variável de referência de objeto contém bits que representam uma maneira de acessar um objeto.
- Ela não contém o objeto propriamente dito, mas algo como um ponteiro. Ou um endereço. Em Java, não sabemos realmente *o que* se encontra dentro de uma variável de referência. Sabemos que, o que quer que seja, representará um e somente um objeto. E a JVM sabe como usar a referência para chegar ao objeto.

Você não pode inserir um objeto em uma variável. Geralmente consideramos isso dessa forma... Dizemos coisas como "passei a string para o método System.out.println( )". Ou "o método retorna um objeto Dog" ou ainda "inseri um novo objeto Foo na variável chamada myFoo".

Mas não é isso o que acontece. Não existem xícaras gigantes expansíveis que possam crescer até o tamanho de qualquer objeto. Os objetos residem em um e apenas um local — a pilha de lixo coletável! (Você aprenderá mais sobre isso posteriormente neste capítulo.)

Enquanto uma variável primitiva fica cheia de bits que representam o ***valor*** real da variável, uma variável de referência de objeto fica cheia de bits que representam ***uma maneira de chegar ao objeto***.

Você usará o operador ponto (.) em uma variável de referência para dizer "use o que está *antes* do ponto para me trazer o que está *depois* do ponto". Por exemplo:

```
myDog.bark( );
```

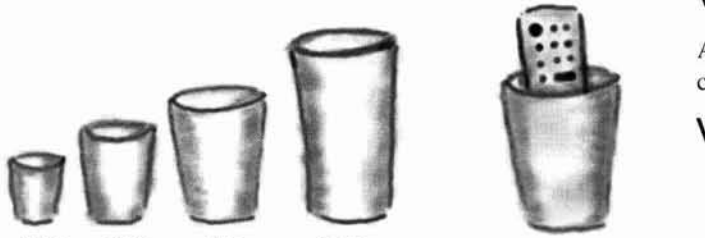
significa "use o objeto referenciado pela variável myDog para chamar o método bark( )". Quando você usar o operador ponto em uma variável de referência de objeto, considere isso como se estivesse pressionando um botão do controle remoto desse objeto.

```
Dog d = new Dog( );
d.bark( );
```

considere isso

como se fosse isso

Pense na variável de referência de Dog como o controle remoto de um objeto Dog.
Você a usará para acessar o objeto e fazer algo (chamar métodos).



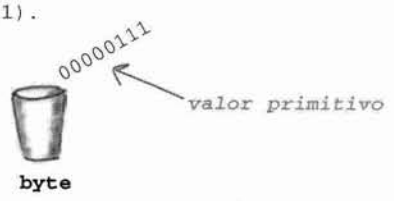
Uma referência de objeto é apenas outro valor da variável.

Algo que é despejado em uma xícara. Só que dessa vez é um controle remoto.

Variável primitva

```
byte x = 7;
```

Os bits que representam 7 estão na variável. (00000111).



Variável de referência

```
Dog myDog = new Dog( );
```

Os bits que representam uma maneira de acessar o objeto Dog ficam dentro da variável.
O objeto Dog propriamente dito não fica na variável!

Nas variáveis primitivas, o valor da variável é... O *valor* literal (5, -26,7, 'a').

Nas variáveis de referência, o valor da variável são... *Bits, que representam uma maneira de chegar a um objeto específico.*

Não sabemos (ou nos importamos) como uma JVM específica implementa as referências de objeto. Certo, elas podem ser um ponteiro que aponte para um ponteiro que aponte para... Mas. mesmo se você souber, não poderá usar os bits para nenhuma outra finalidade que não seja acessar um objeto.

Não nos importamos com quantos algarismos 1 e 0 existem em uma variável de referência. Isso é responsabilidade de cada JVM e da fase da Lua.

As 3 etapas de declaração, criação e atribuição de objetos

```
1        2
Dog myDog = 3 new Dog( );
```

1 Declare uma variável de referência

```
Dog myDog = new Dog( );
```

Solicita à JVM para alocar espaço para uma variável de referência e nomeia essa variável como ***myDog***. A variável de referência será sempre do tipo Dog. Em outras palavras, um controle remoto que tenha botões que controlem um objeto Dog, mas não um objeto Car, Buttton ou Socket.

2 Crie um objeto

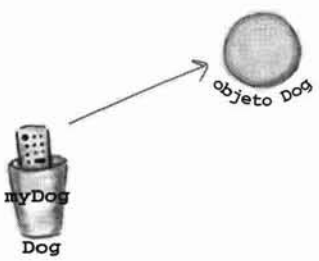
```
Dog myDog = new Dog( );
```

Solicita à JVM para alocar espaço para um novo objeto Dog no heap (aprenderemos muito mais sobre esse processo, principalmente no Capítulo 9). objeto Dog

3 Vincule o objeto e a referência

```
Dog myDog = new Dog( );
```

Atribui o novo objeto Dog à variável de referência myDog. Em outras palavras, ***programa o controle remoto***.



Não existem Perguntas Idiotas

P: Qual o tamanho de uma variável de referência?

R: Você não saberá. A menos que tenha intimidade com alguém da equipe de desenvolvimento da JVM, não saberá como uma referência é representada. Haverá ponteiros em algum local, mas você não poderá acessá-los. Não precisará disso. (Certo, se insistir, não há por que não imaginá-la com um valor de 64 bits.) Mas quando estiver pensando em questões de alocação de memória, sua Grande Preocupação deverá ser com quantos *objetos* (e não *referências* de objeto) está criando e com qual *seu* (dos objetos) verdadeiro tamanho.

P: Mas isso significa que todas as referências de objeto têm o mesmo tamanho, independentemente do tamanho dos objetos reais aos quais elas se referem?

R: Sim. Todas as referências de uma determinada JVM terão o mesmo tamanho, independentemente dos objetos que elas referenciarem, mas cada JVM pode ter uma maneira diferente de representar referências, portanto, as referências de uma JVM podem ser menores ou maiores do que as de outra JVM.

P: Posso fazer cálculos em uma variável de referência, aumentá-la, você sabe — operações próprias da linguagem C?

R: Não. Repita comigo: "O Java não é o C".

Tudo sobre o Java

Entrevista desta semana:
A referência de objeto

Use a Cabeça!: Então, diga-nos, como é vida de uma referência de objeto?

Referência: Bem simples, na verdade. Sou um controle remoto e posso ser programada para controlar diferentes objetos.

Use a Cabeça!: Você quer dizer objetos diferentes mesmo enquanto está sendo executada? Tipo, você pode referenciar um cão e, em seguida, cinco minutos após referenciar um carro?

Referência: É claro que não. Uma vez tendo sido declarada, é isso que sou. Se eu for o controle remoto de um cão, então, nunca poderei apontar (opa — desculpe, não devemos dizer *apontar*), digo, *referenciar* algo que não seja um cão.

Use a Cabeça!: Isso significa que você pode referenciar apenas um cão?

Referência: Não, posso referenciar um cão e, em seguida, cinco minutos após referenciar algum *outro* cão. Contanto que seja um cão, posso ser redirecionada (como na reprogramação de seu controle remoto para uma TV diferente) para ele. A menos que... Deixa pra lá, esquece.

Use a Cabeça!: Não, diga. O que você ia dizer?

Referência: Não acho que você queira entrar nesse assunto agora, mas darei apenas uma explicação rápida — se eu for marcada como final, então, quando me atribuírem um Cão, não poderei ser reprogramada para nada mais exceto *esse* e somente esse cão. Em outras palavras, nenhum outro objeto

poderá ser atribuído a mim.

Use a Cabeça!: Você está certa, não queremos falar sobre isso agora. OK, então a menos que você seja final, pode referenciar um cão e, em seguida, referenciar um cão diferente. Você pode não referenciar *absolutamente nada*? É possível não ser programada para nada?

Referência: Sim, mas me incomoda falar sobre isso.

Use a Cabeça!: Por quê?

Referência: Porque significa que eu seria nula e isso me incomoda.

Use a Cabeça!: Você quer dizer que então não teria valor?

Referência: Oh, nulo *é* um valor. Eu ainda seria um controle remoto, mas é como se você trouxesse para casa um novo controle remoto universal e não tivesse uma TV. Não estarei programada para controlar nada. Vocês poderiam pressionar meus botões o dia inteiro, mas nada de interessante aconteceria. Eu me sentiria tão... Inútil. Um desperdício de bits. Na verdade, nem tantos bits, mas mesmo assim alguns. E essa não é a pior parte. Se eu for a única referência de um objeto específico e, em seguida, for configurada com null (desprogramada), isso significa que agora *ninguém* poderá acessar aquele objeto que eu estava referenciando.

Use a Cabeça!: E isso não é bom porque...

Referência: Precisa *perguntar*? Desenvolvi um relacionamento com esse objeto, uma conexão íntima e, em seguida, o vínculo é repentina e cruelmente rompido. E eu nunca verei esse objeto novamente, porque agora ele estará qualificado para [produtor, deixa para a música trágica] a *coleta de lixo*. Sniff. Mas você acha que os programadores consideram *isso*? Snif. Por que, *por que* não posso ser uma variável primitiva? *Odeio ser uma referência*. A responsabilidade, todos os relacionamentos rompidos...

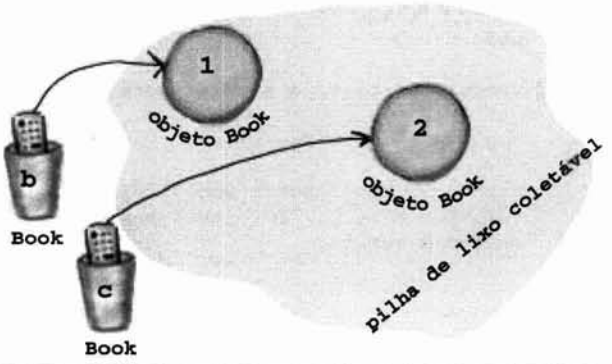
A vida na pilha de lixo coletável

```
Book b = new Book();
Book c = new Book();
```

Declare duas variáveis de referência Book. Crie dois novos objetos Book. Atribua os objetos Book às variáveis de referência.

Agora os dois objetos Book estão residindo na pilha.

Referências: 2
Objetos: 2

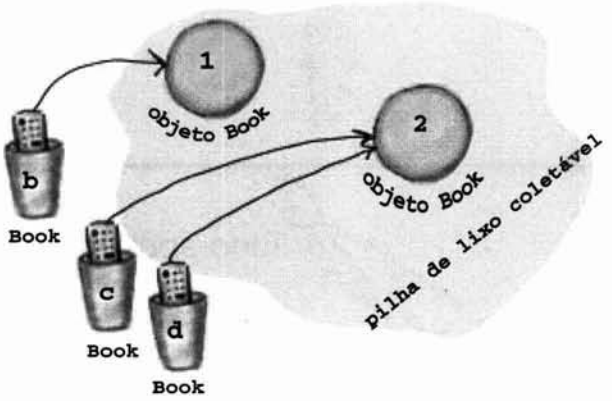


```
Book d = c;
```

Declare uma nova variável de referência Book. Em vez de criar um terceiro objeto Book, atribua o valor da variável **c** à variável **d**. Mas o que isso significa? É como dizer "pegue os bits de **c**, faça uma cópia deles e insira essa cópia em **d**".

Tanto c quanto d referenciam o mesmo objeto.
As variáveis c e d contêm duas cópias diferentes com o mesmo valor. Dois controles remotos programados para uma TV.

Referências: 3
Objetos: 2

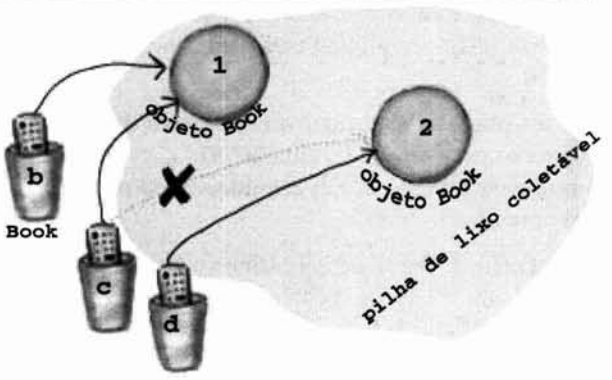


```
c = b;
```

Atribua o valor da variável **b** à variável **c**. Agora você já sabe o que isso significa. Os bits da variável **b** serão copiados e essa nova cópia será inserida na variável **c**.

Tanto b quanto c referenciam o mesmo objeto.

Referências: 3
Objetos: 2



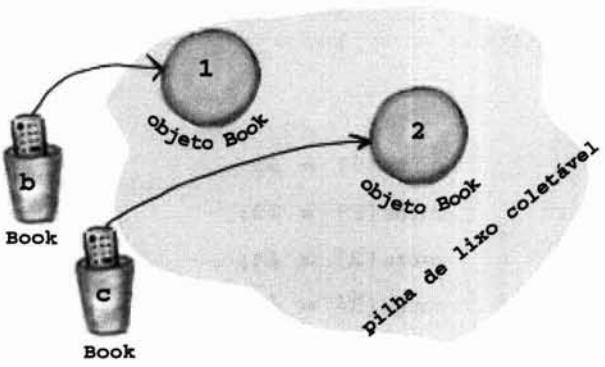
Vida e morte na pilha

```
Book b = new Book( );
Book c = new Book( );
```

Declare duas variáveis de referência. Crie dois novos objetos Book. Atribua os objetos Book às variáveis de referência.

Agora os dois objetos Book estão residindo na pilha.

Referências ativas: 2
Objetos alcançáveis: 2



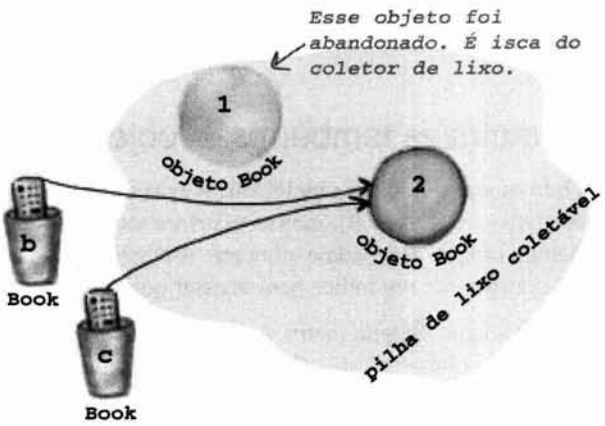
```
b = c;
```

Atribua o valor da variável **c** à variável **b**. Os bits da variável **c** serão copiados, e essa nova cópia será inserida na variável **b**. As duas variáveis contêm valores idênticos.

Tanto b quanto c referenciam o mesmo objeto. O objeto 1 será abandonado e estará qualificado para a Coleta de Lixo (GC, Garbage Collection).

Referências ativas: 2
Objetos alcançáveis: 1
Objetos abandonados: 1

O primeiro objeto que **b** referenciava, o objeto 1, não tem mais referências, se tornou ***inalcançável***.

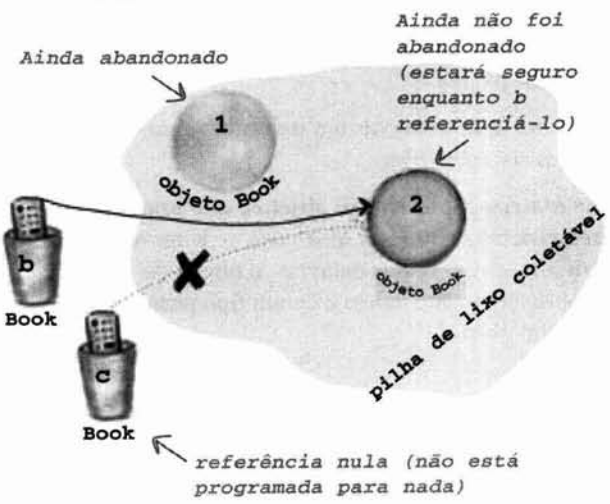


```
c = null;
```

Atribua o valor null à variável **c**. Isso a tornará uma ***referência nula***, o que significa que ela não está referenciando nada. Mas continua a ser uma variável de referência e outro objeto Book pode ser atribuído a ela.

O objeto 2 ainda tem uma referência ativa (b) e, enquanto a tiver, não estará qualificado para a GC.

Referências ativas: 1
Referências nulas: 1
Objetos alcançáveis: 1
Objetos abandonados: 1



Uma matriz é como uma bandeja com xícaras

1. Declare uma variável de matriz int. Uma variável de matriz é o controle remoto de um objeto de matriz.

```
int[ ] nums;
```

2. Crie uma nova matriz int de tamanho 7 e a atribua à variável int[ ] nums já declarada

```
nums = new int[7];
```

3 Forneça para cada elemento da matriz um valor int.
Lembre-se de que os elementos de uma **matriz** int são apenas **variáveis** int.

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```

7 variáveis int

Note que a matriz será um objeto, mesmo se tiver os 7 elementos primitivos.

As matrizes também são objetos

A biblioteca padrão Java inclui várias estruturas de dados sofisticadas incluindo mapas, árvores e conjuntos (consulte o Apêndice B), mas as matrizes servirão bem quando você quiser apenas obter uma lista de coisas de maneira rápida, ordenada e eficiente. As matrizes lhe concederão acesso aleatório rápido, permitindo que você use a posição de um índice para acessar qualquer elemento existente nelas.

Todo elemento de uma matriz é apenas uma variável. Em outras palavras, um dos oito tipos primitivos de variável (lembre-se: Large Furry Dog) ou uma variável de referência. Qualquer coisa que você inserir em uma *variável* desse tipo poderá ser atribuída a um *elemento de matriz* do mesmo tipo. Portanto, em uma matriz de tipo int (int[ ]), cada elemento pode conter um inteiro. Em uma matriz Dog (Dog[ ]) cada elemento pode conter... Um objeto Dog? Não, lembre-se de que uma variável de referência só armazena uma referência (um controle remoto) e não o próprio objeto. Logo, em uma matriz Dog, cada elemento pode conter o controle remoto de um objeto Dog. É claro que ainda teremos que criar os objetos Dog... E você verá tudo isso na próxima página.

Não deixe de observar um item-chave no cenário acima — *a matriz será um objeto, mesmo se tiver variáveis primitivas.*

As matrizes são sempre objetos, não importando se foram declaradas para conter tipos primitivos ou referências de objeto. Mas você pode ter um objeto de matriz que tenha sido declarado para *conter* valores primitivos. Em outras palavras, o objeto de matriz pode ter *elementos* que sejam primitivos, mas a matriz propriamente dita *nunca* é de um tipo primitivo. Independentemente do que a matriz armazenar, ela sempre será um objeto!

Crie uma matriz de objetos Dog

1 Declare uma variável de matriz Dog

```
Dog[] pets;
```

2 Crie uma nova matriz Dog com tamanho igual a 7 e a atribua à variável Dog[] pets já declarada

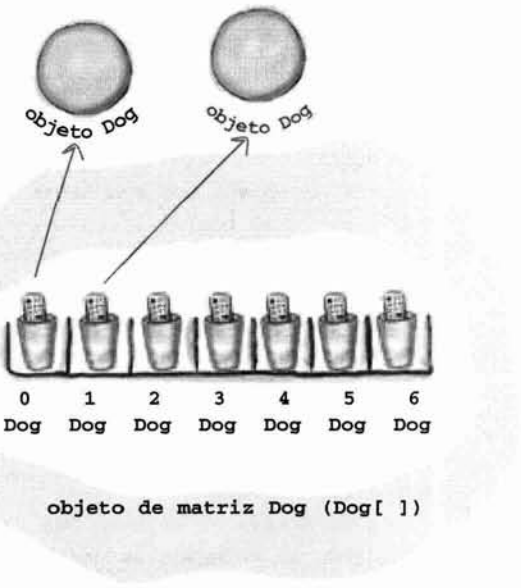
```
pets = new Dog[7];
```

O que está faltando?

Objetos Dog! Temos uma matriz de *referências* Dog, mas nenhum *objeto* Dog real!

3 Crie novos objetos Dog e atribua-os aos elementos da matriz.
Lembre-se de que os elementos de uma ***matriz*** Dog são apenas ***variáveis*** de referência Dog. Ainda precisamos de objetos Dog!

```
pets[0[ = new Dog;
pets[1] = new Dog;
```



Aponte seu lápis

Qual é o valor atual de pets[2]?

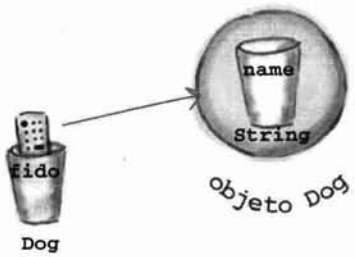
Que código faria pets[3] referenciar um dos dois objetos Dog existentes?

Controle seu objeto Dog
(com uma variável de referência)

```
Dog fido = new Dog();
fido.name = "Fido";
```

Criamos um objeto Dog e usamos o operador ponto na variável de referência ***fido*** para acessar a variável name*.

Podemos usar a referência ***fido*** para fazer o cão latir( ), comer( ) ou perseguirGatos( ).




```
fido.bark();
fido.chaseCat();
```

O que aconteceria se o objeto Dog estivesse em uma matriz Dog?

Sabemos que podemos acessar as variáveis de instância e métodos de Dog usando o operador ponto, mas *onde* usá-lo?

Quando o objeto Dog estiver em uma matriz, não teremos uma variável name real (como ***fido***). Em vez disso usaremos a notação de matriz e apertaremos o botão do controle remoto (operador ponto) do objeto de um índice (posição) específico da matriz:

```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

*Sim, sabemos que não estamos demonstrando o encapsulamento aqui, mas estamos tentando manter o código simples. Por enquanto. Veremos o encapsulamento no Capítulo 4."

O Java acha o tipo importante.

Quando você tiver declarado uma matriz, não poderá inserir nada que não seja do tipo declarado para ela.

Por exemplo, você não pode inserir um objeto Cat em uma matriz Dog (seria muito frustrante se alguém achasse que só há cães na matriz e pedisse a cada um deles que latisse para então com espanto descobrir que há um gato à espreita). E você não pode inserir um tipo double em uma matriz int (derramamento, lembra-se?). No entanto, pode inseri um byte em uma matriz int, porque o tipo byte sempre caberá em uma xícara de tamanho int. Isso é conhecido como alargamento implícito. Entraremos em detalhes posteriormente; por enquanto apenas lembre-se de que o compilador não permitirá que você insira algo errado em uma matriz, com base no tipo declarado para ela.

Um exemplo de Dog

```
class Dog {
   String name;

   public static void main (String[] args) {
      // cria um objeto Dog e o acessa
      Dog dog1 = new Dog();
      dog1.bark();
      dog1.name = "Bart";
      // agora cria uma matriz Dog
      Dog[] myDogs = new Dog[3];
      // and put some dogs in it
      myDogs[0] = new Dog();
      myDogs[1] = new Dog();
      myDogs[2] = dog1;
      // agora acessa os objetos Dog
      // usando as referências da matriz
      myDogs[0].name = "Fred";
      myDogs[1].name = "Marge";
      // Hmmmm... qual é o nemo de myDogs[2]?
      System.out.print("o nome do último cão é ");
      System.out.println(myDogs[2].name);
      // agora executa um loop pela matriz
      // e pede a todos os cães para latirem
      int x = 0;
      while(x < myDogs.length) {
         myDogs[x].bark();
         x = x + 1;
      }
   }

   public void bark() {
      System.out.println(name + " diz Ruff!");
   }
   public void eat() { }
   public void chaseCat() { }
}
```

as matrizes têm uma variável 'length' que lhe fornecerá a quantidade de elementos

Saída

```
Arquivo Editar Janela Ajuda Uivar
%java Dog
null diz Ruff!
o nome do último cão é Bart
Fred diz Ruff!
Marge diz Ruff!
Bart diz Ruff!
```

DISCRIMINAÇÃO DOS PONTOS

- As variáveis vêm em duas versões: primitivas e de referência.
- As variáveis devem sempre ser declaradas com um nome e um tipo.
- O valor de uma variável primitiva são os bits que o representam (5, 'a', verdadeiro, 3.1416, etc.).
- O valor de uma variável de referência são os bits que representam uma maneira de acessar um objeto da pilha.
- A variável de referência é como um controle remoto. Usar o operador ponto (.) em uma variável de referência é como pressionar um botão no controle remoto para acessar um método ou variável de instância.
- Uma variável de referência tem valor nulo quando não está referenciando nenhum objeto.
- Uma matriz é sempre um objeto, mesmo quando é declarada para conter tipos primitivos. Não existe algo como uma matriz primitiva, somente uma matriz que *contenha* tipos primitivos.

Exercício

Seja o compilador

Cada um dos arquivos Java dessa página representa um arquivo-fonte completo. Sua tarefa é personificar o compilador e determinar se cada um deles pode ser compilado. Se não puderem ser compilados, como você os corrigiria?

A

```
class Books {
   String title;
   String author;
}

class BooksTestDrive {
   public static void main(String [] args) {

      Books [] myBooks = new Books[3];
      int x = 0;
      myBooks[0].title = "The Grapes of Java";
      myBooks[1].title = "The Java Gatsby";
      myBooks[2].title = "The Java Cookbook";
      myBooks[0].author = "bob";
      myBooks[1].author = "sue";
      myBooks[2].author = "ian";

      while (x < 3) {
         System.out.print(myBooks[x].title);
         System.out.print(" by ");
         System.out.println(myBooks[x].author);
         x = x + 1;
      }
   }
}
```

B

```
class Hobbits {

   String name;

   public static void main(String [] args) {

      Hobbits [] h = new Hobbits[3];
      int z = 0;

      while (z < 4) {
         z = z + 1;
         h[z] = new Hobbits();
         h[z].name = "bilbo";
         if (z == 1) {
            h[z].name = "frodo";
         }
         if (z == 2) {
            h[z].name = "sam";
         }
         System.out.print(h[z].name + " is a ");
         System.out.println("good Hobbit name");
      }
   }
}
```

Exercício

Imãs com código

Um programa Java está todo misturado sobre a geladeira. Você conseguiria reconstruir os trechos de código para criar um programa Java funcional que produzisse a saída listada a seguir? Algumas das chaves caíram no chão e são muito pequenas para que as recuperemos, portanto, fique à vontade para adicionar quantas delas precisar!

```
File Edit Window Help Bikini
%java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
int y = 0;
ref = index[y];
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
int ref;
while (y < 4) {
System.out.println(islands[ref]);
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
String [] islands = new String[4];
System.out.print("island = ");
int [] index = new int[4];
y = y + 1;
class TestArrays {
public static void main(String [] args) {
```

Quebra-cabeças na Piscina

Sua ***tarefa*** é pegar os trechos de código da piscina e inseri-los nas linhas em branco do código. Você **pode** usar o mesmo trecho mais de uma vez e não precisa empregar todos os trechos. Seu ***objetivo*** é criar uma classe que seja compilada e executada, produzindo a saída listada.

Saída

```
File Edit Window Help Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = _____
y = _____
```

Pergunta adicional!

Para tentar ganhar mais pontos, use os trechos da piscina para preencher o que falta na saída (acima).

```
                                    (Podemos não usar uma classe
                                    de teste separada, por
                                    estarmos tentando economizar
                                    espaço na página.)
class Triangle {
   double area;
   int height;
   int length;
   public static void main(String [] args) {
      ___________
      _________________________
      while ( __________ ) {
         ___________________________
         ________.height = (x + 1) * 2;
         ________.length = x + 4;
         ___________________________
         System.out.print("triangle "+x+", area");
         System.out.println(" = " + ________.area);
         ________________
      }
      _____________
      x = 27;
      Triangle t5 = ta[2];
      ta[2].area = 343;
      System.out.print("y = " + y);
      System.out.println(", t5 area = "+ t5.area);
   }
   void setArea() {
      ___________ = (height * length) / 2;
   }
}
```

Nota: Cada trecho de código da piscina pode ser usado mais de uma vez!

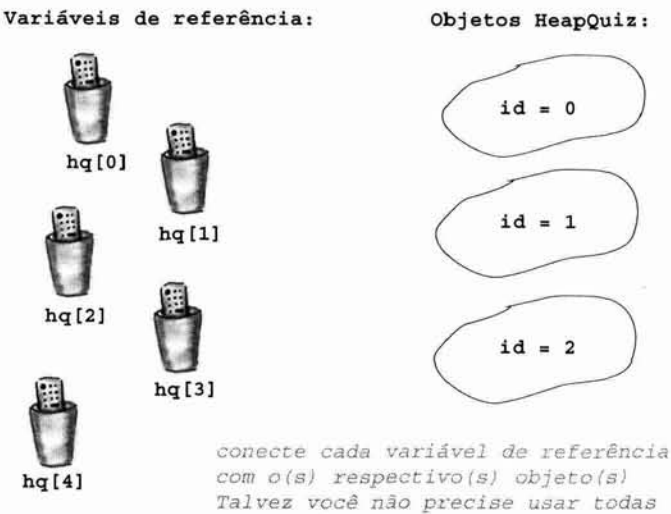
```
area
ta.area      4,   t5 area = 18.0
ta.x.area    4,   t5 area = 343.0
ta[x].area   27,  t5 area = 18.0
             27,  t5 area = 343.0

x                ta[x] = setArea();
y                ta.x = setArea();
                 ta[x].setArea();

Triangle [ ] ta = new Triangle(4);
Triangle ta = new [ ] Triangle[4];
Triangle [ ] ta = new Triangle[4];

int x;
int y;
int x = 0;
int x = 1;
int y = x;

x = x + 1;
x = x + 2;
x = x  - 1;

ta.x
ta(x)
ta[x]

28.0
30.0

x < 4
x < 5

ta = new Triangle();
ta[x] = new Triangle();
ta.x = new Triangle();
```

Uma pilha de problemas

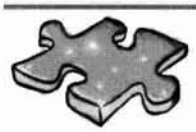
Um programa Java pequeno está listado à direita. Quando a linha '// executa algo' for alcançada, alguns objetos e variáveis de referência terão sido criados. Sua tarefa é determinar que variáveis de referência apontarão para quais objetos. Nem todas as variáveis de referência serão usadas, e alguns objetos podem ser referenciados mais de uma vez. Desenhe linhas conectando as variáveis de referência aos seus respectivos objetos.

Dica: a menos que você seja mais esperto que nós, provavelmente terá que desenhar diagramas como os das páginas 55 e 56 deste capítulo. Use um lápis para poder desenhar e, em seguida, apague os vínculos das referências (as setas que vão do controle remoto da referência para um objeto).

```
class HeapQuiz {
   int id = 0;
   public static void main(String [] args) {
      int x = 0;
      HeapQuiz [ ] hq = new HeapQuiz[5];
      while ( x < 3 ) {
         hq[x] = new HeapQuiz();
         hq[x].id = x;
         x = x + 1;
      }
      hq[3] = hq[1];
      hq[4] = hq[1];
      hq[3] = null;
      hq[4] = hq[0];
      hq[0] = hq[3];
      hq[3] = hq[2];
      hq[2] = hq[0];
      // executa algo
   }
}
```



conecte cada variável de referência com o(s) respectivo(s) objeto(s) Talvez você não precise usar todas as referências.



O caso das referências roubadas

Um pequeno mistério

Era uma noite escura e chuvosa. Tawny caminhava para a cela dos programadores como se fosse proprietária do local. Ela sabia que todos os programadores ainda estariam trabalhando e queria ajuda. Precisava de um novo método adicionado à classe principal, que devia ser carregada no novo celular altamente secreto e habilitado com Java do cliente. O espaço na pilha de memória do celular estava tão apertado quanto o vestido de Tawny, e todo mundo sabia disso. O murmúrio normalmente rouco na cela silenciou quando Tawny se encaminhou para o quadro branco. Ela desenhou uma visão resumida da funcionalidade do novo método e lentamente examinou a sala. "Bem meninos, hora de trabalhar", murmurou. "Quem criar uma versão para esse método que use a memória mais eficientemente irá comigo à festa de lançamento do cliente em Maui amanhã... Para me ajudar a instalar o novo software."

Na manhã seguinte, Tawny entrou na cela usando seu curto vestido Aloha. "Senhores", ela sorriu, "o avião parte em algumas horas, mostrem-me o que vocês tem!" Bob foi o primeiro; quando ele começou a desenhar seu projeto no quadro branco Tawny disse: "Vamos direto ao ponto Bob, mostre-me como você manipulou a atualização da lista de objetos de contato." Bob escreveu rapidamente um fragmento de código no quadro:

```
Contact [] ca = new Contact[10];
while ( x < 10 ) {   // cria 10 objetos de contato
   ca[x] = new Contact();
   x = x + 1;
} // executa complicada atualização da lista de objetos Contact com ca
```

"Tawny, sei que temos pouca memória, mas suas especificações diziam que tínhamos que ser capazes de acessar informações específicas de todos os dez contatos permitidos, e esse foi o melhor esquema que pude criar", disse Bob. Kent foi o próximo, já imaginando coquetéis de coco com Tawny. "Bob", ele disse, "sua solução é um pouco complicada não acha?" Kent sorriu, "Dê uma olhada neste bebezinho":

```
Contact refc;
while ( x < 10 ) {   // make 10 contact objects
   refc = new Contact();
   x = x + 1;
} // executa complicada atualização da lista de objetos Contact com refc
```

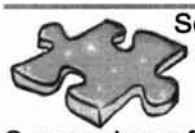
"Economizei muitas variáveis de referência que usariam memória, Bobizinho, portanto, pode guardar seu protetor solar", gozou Kent. "Não tão rápido, Kent!", disse Tawny, "você economizou um pouco de memória, mas é Bob que vem comigo".

Por que Tawny escolheu o método de Bob e não o de Kent, se o de Kent usava menos memória?

Soluções dos Exercícios

Ímãs com código:

```
class TestArrays {
   public static void main(String [] args) {
      int [] index = new int[4];
      index[0] = 1;
      index[1] = 3;
      index[2] = 0;
      index[3] = 2;
      String [] islands = new String[4];
      islands[0] = "Bermuda";
      islands[1] = "Fiji";
      islands[2] = "Azores";
      islands[3] = "Cozumel";
      int y = 0;
      int ref;
      while (y < 4) {
         ref = index[y];
         System.out.print("island = ");
         System.out.println(islands[ref]);
         y = y + 1;
      }
   }
}
```

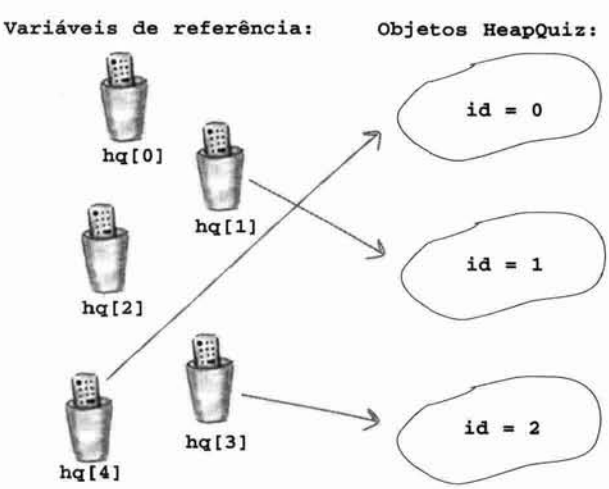


Soluções dos quebra-cabeças

O caso das referências roubadas

Tawny percebeu que o método de Kent tinha uma falha séria. É verdade que ele não usou tantas variáveis de referência quanto Bob, mas não havia como acessar nenhum dos objetos Contact que esse método criava, exceto o último. A cada passagem do loop, ele estava atribuindo um novo objeto à variável de referência, portanto, o objeto referenciado anteriormente era abandonado na pilha - *inalcançável*. Sem acessar nove dos dez objetos criados, o método de Kent era inútil.

(O software foi um grande sucesso, e o cliente deu a Tawny e Bob uma semana a mais no Havaí. Gostaríamos de lhe dizer que, ao terminar este livro, você também conseguirá coisas desse tipo.)



A

```
class Books {
   String title;
   String author;
}
class BooksTestDrive {
   public static void main(String [] args) {
      Books [] myBooks = new Books[3];
      int x = 0;
      myBooks[0] = new Books();
      myBooks[1] = new Books();
      myBooks[2] = new Books();
      myBooks[0].title = "The Grapes of Java";
      myBooks[1].title = "The Java Gatsby";
      myBooks[2].title = "The Java Cookbook";
      myBooks[0].author = "bob";
      myBooks[1].author = "sue";
      myBooks[2].author = "ian";
      while (x < 3) {
         System.out.print(myBooks[x].title);
         System.out.print(" by ");
         System.out.println(myBooks[x].author);
         x = x + 1;
      }
   }
}
```

Lembre-se: temos que criar realmente os objetos Book!

B

```
class Hobbits {
   String name;
   public static void main(String [] args) {
      Hobbits [] h = new Hobbits[3];
      int z = -1;
      while (z < 2) {
         z = z + 1;
         h[z] = new Hobbits();
         h[z].name = "bilbo";
         if (z == 1) {
            h[z].name = "frodo";
         }
         if (z == 2) {
            h[z].name = "sam";
         }
         System.out.print(h[z].name + " is a ");
         System.out.println("good Hobbit name");
      }
   }
}
```

Lembre-se: as matrizes começam com o elemento 0!

```
class Triangle {
   double area;
   int height;
   int length;
   public static void main(String [] args) {
      int x = 0;
      Triangle [ ] ta = new Triangle[4];
      while ( x < 4 ) {
         ta[x] = new Triangle();
         ta[x].height = (x + 1) * 2;
         ta[x].length = x + 4;
         ta[x].setArea();
         System.out.print("triangle "+x+", area");
         System.out.println(" = " + ta[x].area);
         x = x + 1;
      }
      int y = x;
      x = 27;
      Triangle t5 = ta[2];
      ta[2].area = 343;
      System.out.print("y = " + y);
      System.out.println(", t5 area = "+ t5.area);
   }
   void setArea() {
      area = (height * length) / 2;
   }
}
```