Como os Objetos se Comportam

O estado afeta o comportamento, o comportamento afeta o estado. Sabemos que os objetos têm **estado** e **comportamento**, representados pelas **variáveis de instância** e **métodos**. Mas, até agora, não examinamos como o estado e o comportamento estão *relacionados*. Já sabemos que cada instância de uma classe (cada objeto de um tipo específico) pode ter seus próprios valores exclusivos para suas variáveis de instância. O cão A pode ter o *nome* "Fido" e *peso* de 37 quilos. O cão B se chama "Killer" e pesa 5 quilos. E se a classe Cão tiver um método emitirSom( ), bem, você não acha que um cão de 37 quilos latirá um pouco mais alto do que o de 5 quilos? (Supondo que o som incômodo de um ganido possa ser considerado um *latido*.) Felizmente, isso é o que há de importante em um objeto — ele tem um *comportamento* que atua sobre seu *estado*. Em outras palavras, os ***métodos usam os valores das variáveis de instância***. Por exemplo: "Se o cão pesar menos de 8 quilos, emita um ganido, caso contrário..." ou "aumente o peso em 3 quilos". ***Alteremos alguns estados!***

Lembre-se: uma classe descreve o que um objeto conhece e o que ele faz

Uma classe é o projeto de um objeto. Quando você criar uma classe, estará descrevendo como a JVM deve criar um objeto desse tipo. Você já sabe que todo objeto desse tipo pode ter diferentes valores para as *variáveis de instância*. Mas e quanto aos métodos?

Cada objeto desse tipo pode ter um método com comportamento diferente?

Bem... ***Mais ou menos.****

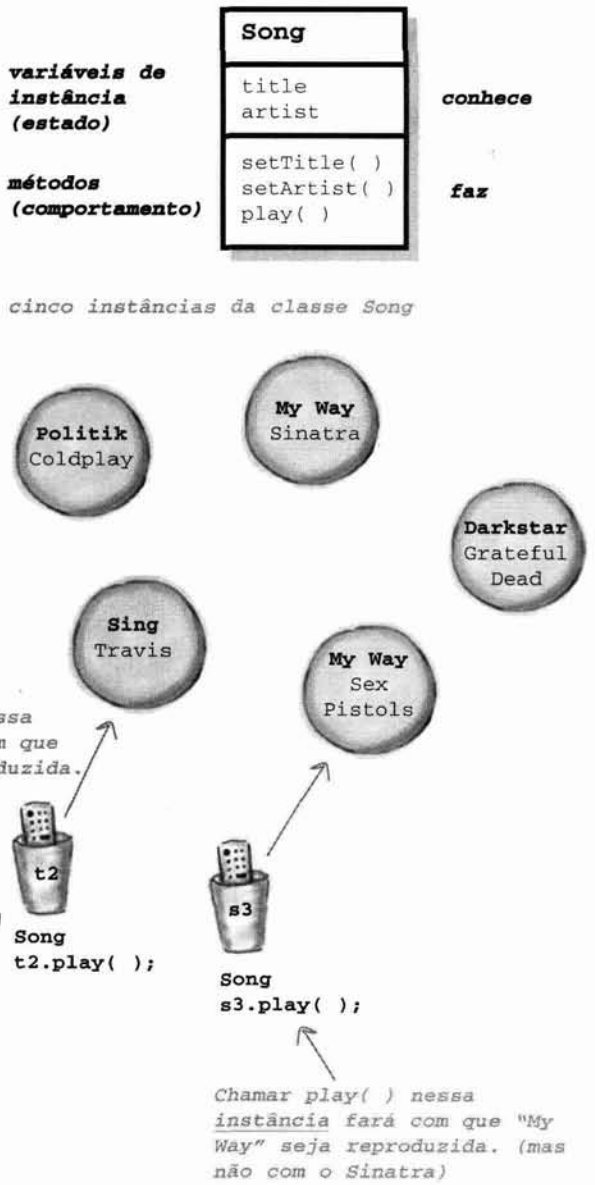
Todas as instâncias de uma classe específica têm os mesmos métodos, mas eles podem *se comportar* diferentemente com base no valor das variáveis de instância.

A classe Song tem duas variáveis de instância, *title* e *artist*. O método play( ) reproduz uma canção, mas a instância em que você o chamar reproduzirá a canção representada pelo valor da variável de instância *title* (título) dessa instância. Portanto, se você chamar o método play( ) em uma instância, reproduzirá a canção "politik", enquanto outra instância reproduzirá "Darkstar". O código do método, porém, é o mesmo.

```
void play() {
   soundPlayer.playSound(title);
}
```

```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
Song s3 = new Song();
s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```

*Sim, outra resposta surpreendentemente clara!



O tamanho afeta o latido

O latido de um cão pequeno é diferente do de um cão grande

A classe Dog tem uma variável de instância *size* (tamanho), que o método *bark( )* usa para decidir que tipo de som emitir para o latido.

```
class Dog {
   int size;
   String name;

   void bark() {
      if (size > 60) {
         System.out.println("Wooof! Wooof!");
      } else if (size > 14) {
         System.out.println("Ruff!  Ruff!");
      } else {
         System.out.println("Yip! Yip!");
      }
   }
}
```

Lata Diferente

```
class DogTestDrive {

   public static void main (String[] args) {
      Dog one = new Dog();
      one.size = 70;
      Dog two = new Dog();
      two.size = 8;
      Dog three = new Dog();
      three.size = 35;

      one.bark();
      two.bark();
      three.bark();
   }
}
```

```
File Edit Window Help Playdead
%java DogTestDrive
Woof! Woof!
Yip! Yip!
Ruff! Ruff!
```

Você pode enviar valores para um método

Como é de se esperar de qualquer linguagem de programação, você pode passar valores para seu método. Pode, por exemplo, querer informar a um objeto Dog quantas vezes latir chamando:

```
d.bark(3);
```

Dependendo de sua experiência em programação e preferências pessoais, *você* pode usar o termo *argumentos* ou talvez *parâmetros* para os valores passados para um método. Embora essas sejam distinções formais na ciência da computação que pessoas que usam jalecos e que quase certamente não lerão este livro fazem, temos coisas mais importantes com que nos preocupar aqui. Portanto, *você* pode chamá-los como quiser (argumentos, *donuts*, bolas de pêlo, etc.), mas usaremos essa convenção:

Um método usa parâmetros. Um chamador passa argumentos.

Os argumentos são os valores que você passará para os métodos. Um ***argumento*** (um valor como 2, "Foo" ou uma referência que aponte para um objeto Dog) será inserido diretamente em um... Adivinhe... ***Parâmetro***. E um parâmetro nada mais é do que uma variável local. Uma variável com um tipo e um nome, que poderá ser usada dentro do corpo do método.

Mas aqui está a parte importante: **se um método usar um parâmetro, você *terá* que passar algo para ele**. E esse algo deve ser um valor do tipo apropriado.

1. Chame o método bark na variável de referência Dog e passe o valor 3 (como o argumento do método).
2. Os bits que representam o valor int 3 serão distribuídos para o método bark.


```
void bark(int numOfBarks) {
   while (numOfBarks > 0) {
      System.out.println("ruff");
      numOfBarks = numOfBarks - 1;
   }
}
```

3. Os bits serão inseridos no parâmetro numOfBarks (uma variável de tamanho int).
4. Use o parâmetro numOfBarks como uma variável no código do método.

Você pode fazer valores serem retornados por um método

Os métodos retornam valores. Todos os métodos são declarados com um tipo de retorno, mas até agora criamos todos os nossos métodos com o tipo de retorno **void**, o que significa que eles não retornam nada.

```
void go() {
}
```

Mas podemos declarar um método que retorne um tipo específico de valor para o chamador, como em:

```
int giveSecret() {
   return 42;
}
```

Se você declarar um método que retorne um valor, *terá* que retornar um valor do tipo declarado! (Ou um valor que seja *compatível* com o tipo declarado. Nós nos aprofundaremos mais nisso quando falarmos sobre polimorfismo nos capítulos 7 e 8.)

O compilador não permitirá que você retorne o tipo errado de coisa.

O que você *disser* que retornará é *bom* que seja mesmo retornado!

Os bits que representam 42 são retornados pelo método giveSecret( ) e inseridos na variável chamada theSecret.

Você pode enviar mais de um valor para um método

Os métodos podem ter vários parâmetros. Separe-os com vírgulas ao declará-los e separe os argumentos com vírgulas ao passá-los. O mais importante é que se um método tiver parâmetros, você *deve* passar argumentos do tipo e na ordem corretos.

Chamando um método de dois parâmetros e enviando dois argumentos para ele.

```
void go() {
   TestStuff t = new TestStuff();
   t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

Os argumentos serão inseridos na mesma ordem em que você os passar. O primeiro argumento será inserido no primeiro parâmetro, o segundo argumento no segundo parâmetro e assim por diante.

Você pode passar variáveis para um método, contanto que o tipo da variável seja igual ao tipo do parâmetro.

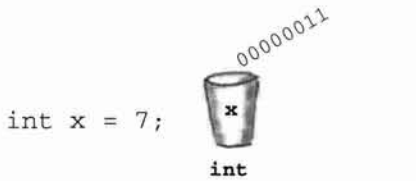
```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

Os valores de foo e bar serão inseridos nos parâmetros x e y. Portanto, agora os bits de x serão idênticos aos de foo (o padrão de bits do inteiro '7') e os bits de y serão idênticos aos de bar.

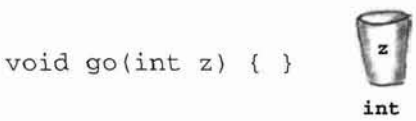
Qual é o valor de z? O resultado será o mesmo que você obteria se somasse foo + bar ao passá-los para o método takeTwo

O Java passa por valor.

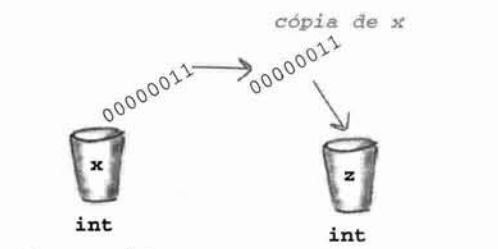
Isso significa passar por cópia.



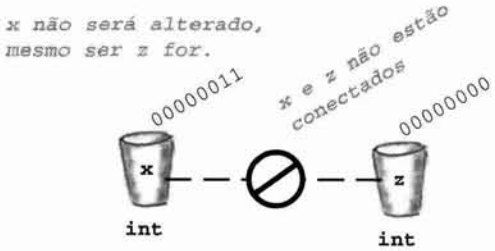
1. Declare uma variável int e atribua a ela o valor '7'. O padrão de bits do número 7 será inserido na variável chamada x.



2. Declare um método com um parâmetro int chamado z.



3. Chame o método go( ), passando a variável x como argumento. Os bits de x serão copiados, e a cópia será inserida em z.



4. Altere o valor de z dentro do método. O valor de x não será alterado! O argumento passado para o parâmetro z era apenas uma cópia de x.
O método não pode alterar os bits que estavam na variável x que o chamou.

Não existem Perguntas Idiotas

P: O que aconteceria se o argumento que você quisesse passar fosse um objeto em vez de uma variável primitiva?

R: Você aprenderá mais sobre isso em capítulos posteriores, mas já *sabe* a resposta. A Java passa *tudo* por valor. ***Tudo***. Porém... *Valor* significa os *bits existentes na variável*. E lembre-se de que você não pode inserir objetos em variáveis; a variável é um controle remoto — *uma referência a um objeto*. Portanto, se você passar a referência de um objeto para um método, estará passando uma *cópia do controle remoto*. Mas fique atento, temos muito mais há dizer sobre isso.

P: Um método pode declarar diversos valores de retorno? Ou há alguma maneira de retornar mais de um valor?

R: Mais ou menos. Um método pode declarar somente um valor de retorno. MAS... Se você quiser retornar, digamos, três valores int, então, o tipo de retorno declarado pode ser uma *matriz* int. Insira esses tipos int dentro da

matriz e retorne-a. É um pouco mais complicado retornar diversos valores com tipos diferentes; falaremos sobre isso em um capítulo posterior quando discutirmos o objeto ArrayList.

P: Tenho que retornar o tipo exato que declarei?

R: Você poderá retornar qualquer coisa que possa ser *implicitamente* elevada a esse tipo. Portanto, pode passar um byte onde um int for esperado. O chamador não se importará, porque o byte caberá perfeitamente no int que ele usará para atribuir o resultado. Você deve usar uma conversão *explícita* quando o tipo declarado for *menor* do que o que você estiver tentando retornar.

P: Tenho que fazer algo com o valor de retorno de um método? Posso apenas ignorá-lo?

R: O Java não exige que o valor de retorno seja usado. Você pode querer chamar um método com um tipo de retorno que não seja nulo, ainda que não se importe com o valor de retorno. Nesse caso, estará chamando o método pelo que ele executa *internamente*, em vez de pelo que *retorna*. Em Java, você não precisa atribuir ou usar o valor de retorno.

Lembrete: O Java acha o tipo importante!

Você não pode retornar uma Girafa quando o tipo de retorno for declarado como um Coelho. O mesmo ocorre com os parâmetros. Você não pode passar uma Girafa para um método que use um Coelho.

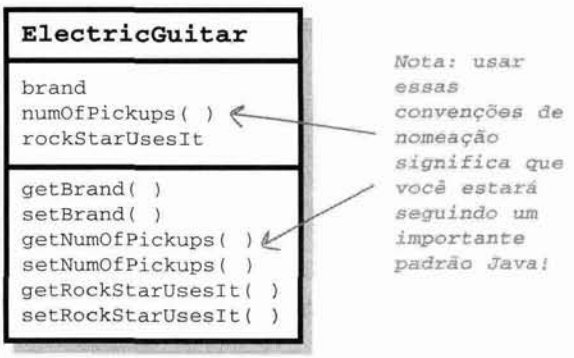
DISCRIMINAÇÃO DOS PONTOS

- As classes definem o que um objeto conhece e o que ele faz.
- As coisas que um objeto conhece são suas **variáveis de instância** (estado).
- As coisas que um objeto faz são seus **métodos** (comportamento).
- Os métodos podem usar variáveis de instância para que objetos do mesmo tipo possam se comportar diferentemente.
- Um método pode ter parâmetros, o que significa que você pode passar um ou mais valores para ele.
- A quantidade e o tipo dos valores que você passar devem corresponder à ordem e tipo dos parâmetros declarados pelo método.
- Os valores passados para dentro e fora dos métodos podem ser elevados implicitamente a um tipo maior ou convertidos explicitamente para um tipo menor.
- O valor que você passar como argumento para um método pode ser literal (2, 'c', etc.) ou uma variável com o tipo de parâmetro declarado (por exemplo, *x* onde *x* for uma variável int). (Há outras coisas que você pode passar como argumentos, mas ainda não chegamos lá.)
- Um método *deve* declarar um tipo de retorno. Um tipo de retorno void significa que o método não retorna nada.
- Se um método declarar um tipo de retorno que não seja void, *deve* retornar um valor compatível com o tipo declarado.

Coisas interessantes que você pode fazer com os parâmetros e tipos de retorno

Agora que vimos como os parâmetros e tipos de retorno funcionam, é hora de lhes darmos alguma utilidade prática: os métodos **Getter** e **Setter**. Se você quiser ser formal com relação a isso, pode preferir chamá-los de *acessadores* e *modificadores*. Mas desperdiçaria sílabas. Além do que, Getter e Setter se enquadram na convenção de nomeação Java, portanto, é assim que os chamaremos.

Os método Getter (de captura) e Setter (de configuração) permitirão que você, bem, *capture e configure coisas*. Geralmente variáveis de instância. A única finalidade de um método de captura é enviar, como valor de retorno, o valor do que quer que esse método de captura específico capture. Portanto, não é surpresa que um método de configuração só exista para esperar a chance de receber o valor de um argumento e usá-lo para *configurar* uma variável de instância.



```
class ElectricGuitar {

   String brand;
   int numOfPickups;
   boolean rockStarUsesIt;

   String getBrand() {
      return brand;
   }

   void setBrand(String aBrand) {
      brand = aBrand;
   }

   int getNumOfPickups() {
      return numOfPickups;
   }

   void setNumOfPickups(int num) {
      numOfPickups = num;
   }

   boolean getRockStarUsesIt() {
      return rockStarUsesIt;
   }

   void setRockStarUsesIt(boolean yesOrNo) {
      rockStarUsesIt = yesOrNo;
   }
}
```

Encapsulamento

Use-o ou arrisque-se a ser humilhado e ridicularizado.

Até esse momento tão importante, cometemos uma das piores falhas na OO (e não estamos falando de violações menores como não usar 'B maiúsculo' em BYOB). Não, estamos falando de uma Falha com 'F' maiúsculo.

Qual foi nossa transgressão vergonhosa?

Expor nossos dados!

Aqui estamos nós, apenas seguindo em frente sem sequer nos importarmos em deixar nossos dados expostos para que *todos* vejam e até mesmo mexam.

Talvez você já tenha experimentado esse sentimento vagamente inquietante que surge quando deixamos nossas variáveis de instância expostas.

Exposto significa alcançável através do operador ponto, como em:

```
theCat.height = 27;
```

Pense nessa idéia do uso de nosso controle remoto para fazermos uma alteração direta na variável de instância do tamanho do objeto Cat. Nas mãos da pessoa errada, uma variável de referência (controle remoto) seria uma arma bem perigosa. Porque nada impediria isso:

```
theCat.height = 0;
```
Opa! Não podemos deixar isso acontecer!

Seria péssimo. Precisamos construir métodos de configuração para todas as variáveis de instância e encontrar uma maneira de forçar os outros códigos a chamarem esses métodos em vez de acessar os dados diretamente.

```
public void setHeight(int ht) {
   if (ht > 9) {
      height = ht;
   }
}
```
Inserimos verificações para garantir uma altura mínima para o objeto Cat.

Ao forçar todos os códigos a chamarem um método de configuração, podemos proteger o objeto Cat de alterações inaceitáveis no tamanho.

Oculte os dados

Sim, é muito simples passar de uma implementação que esteja pedindo para receber dados inválidos para uma que proteja seus dados *e* seu direito de alterá-la posteriormente.

Certo, então como exatamente *ocultar* os dados? Com os modificadores de acesso **public** e **private**. Você está familiarizado com **public** — ele foi usado em todos os métodos main.

Aqui está uma regra prática *inicial* para o encapsulamento (todas as isenções de responsabilidade referentes às regras práticas são aplicáveis): marque suas variáveis de instância com ***private*** e forneça métodos de captura e configuração ***públicos***, para ter controle sobre o acesso. Quando você conhecer melhor o projeto e a codificação em Java, provavelmente fará as coisas de uma forma um pouco diferente, mas, por enquanto, essa abordagem o manterá seguro.

Marque as variáveis de instância com private.

Marque os métodos de configuração e captura com public.

"Infelizmente, Bill se esqueceu de encapsular sua classe Cat e acabou com um gato gordo."
(Ouvido no bebedouro.)

Tudo sobre o Java

Entrevista desta semana:
Um objeto abre o jogo sobre o encapsulamento.

Use a Cabeça!: Qual é o grande problema do encapsulamento?

Objeto: Certo, você conhece aquele sonho em que está fazendo uma palestra para 500 pessoas quando subitamente percebe estar *nu*?

Use a Cabeça!: Sim, já aconteceu conosco. É semelhante àquele da máquina Pilates e... Bem, isso não importa. Certo, então você se sente nu. Mas além de estar um pouco exposto, há algum perigo?

Objeto: Se há algum perigo? Algum *perigo*? [começa a rir] Ei, instâncias, ouviram isso, "*Há algum perigo?*", ele pergunta? [rola de tanto rir]

Use a Cabeça!: Qual é a graça? Me parece uma pergunta sensata.

Objeto: Certo, vou explicar. É que [não consegue controlar o riso novamente]

Use a Cabeça!: Posso pegar algo para você? Água?

Objeto: Uau! Nossa. Não, estou bem. Falarei a sério. Vou respirar fundo. Certo. Prossiga.

Use a Cabeça!: Então, de que o encapsulamento o protege?

Objeto: O encapsulamento cria um campo de força ao redor de minhas variáveis de instância, portanto, ninguém pode configurá-las com, digamos, algo *inapropriado*.

Use a Cabeça!: Você pode citar um exemplo?

Objeto: Não é preciso ser um PhD. A maioria dos valores das variáveis de instância é codificada com certas definições sobre seus limites. Por exemplo, pense em todas as coisas que não funcionariam se números negativos fossem permitidos. A quantidade de banheiros de um escritório. A velocidade de um avião. Aniversários. O peso de halteres. Números de celular. A potência de fornos de microondas.

Use a Cabeça!: Entendo o que você quer dizer. E como o encapsulamanto permite a definição de limites?

Objeto: Ao forçar os outros códigos a passarem por métodos de configuração. Dessa forma, o método de configuração pode validar o parâmetro e decidir se é viável. Ele poderá rejeitá-lo e não fazer nada ou lançar uma exceção (por exemplo, se houver um número de CPF nulo em um aplicativo de cartões de crédito) ou ainda arredondar o parâmetro enviado para o valor mais próximo aceitável. O importante é o seguinte: você pode fazer o que quiser no método de configuração, contanto que não faça *nada* se suas variáveis de instância forem públicas.

Use a Cabeça!: Mas às vezes vejo métodos de configuração que simplesmente configuram o valor sem verificar nada. Se você tiver uma variável de instância que não tenha um limite, esse método de configuração não geraria uma sobrecarga desnecessária? Um impacto no desempenho?

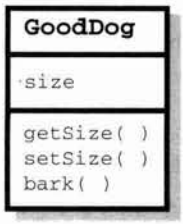
Objeto: O importante nos métodos de configuração (e também nos de captura) é que ***você pode mudar de idéia posteriormente, sem travar o código de ninguém!*** Imagine se metade das pessoas da empresa usasse sua classe com variáveis de instância públicas e um dia você percebesse, de repente, que "opa — há algo que não planejei para esse valor, terei que passar para um método de configuração". Você travaria o código de todo mundo. O interessante no encapsulamento é que *você pode mudar de idéia.* E ninguém é prejudicado. O ganho no desempenho pelo uso das variáveis diretamente é tão pequeno que raramente vale a pena, *se é que vale*.

Encapsulando a classe GoodDog

```
class GoodDog {

   private int size;

   public int getSize() {
      return size;
   }

   public void setSize(int s) {
      size = s;
   }

   void bark() {
      if (size > 60) {
         System.out.println("Wooof! Wooof!");
      } else if (size > 14) {
         System.out.println("Ruff!  Ruff!");
      } else {
         System.out.println("Yip! Yip!");
      }
   }
}

class GoodDogTestDrive {

   public static void main (String[] args) {
      GoodDog one = new GoodDog();
      one.setSize(70);
      GoodDog two = new GoodDog();
      two.setSize(8);
      System.out.println("Dog one: " + one.getSize());
      System.out.println("Dog two: " + two.getSize());
      one.bark();
      two.bark();
   }
}
```

Torna a variável de instância privada.

Torna os métodos de captura e configuração públicos.



Ainda que os métodos não adicionem realmente uma nova funcionalidade, o interessante é que você pode mudar de idéia posteriormente. Pode voltar e tornar um método mais seguro, mais rápido e melhor.

Qualquer local onde um valor específico puder ser usado, uma *chamada de método que retorne esse tipo poderá ser empregada.*

em vez de:

```
int x = 3 + 24;
```

você pode usar:

```
int x = 3 + one.getSize( );
```

Como os objetos de uma matriz se comportam?

Exatamente como qualquer outro objeto. A única diferença é como você os *capturará*. Em outras palavras, como será o controle remoto. Tentaremos chamar métodos nos objetos Dog de uma matriz.

1 Declare e crie uma matriz Dog, que tenha 7 referências Dog.

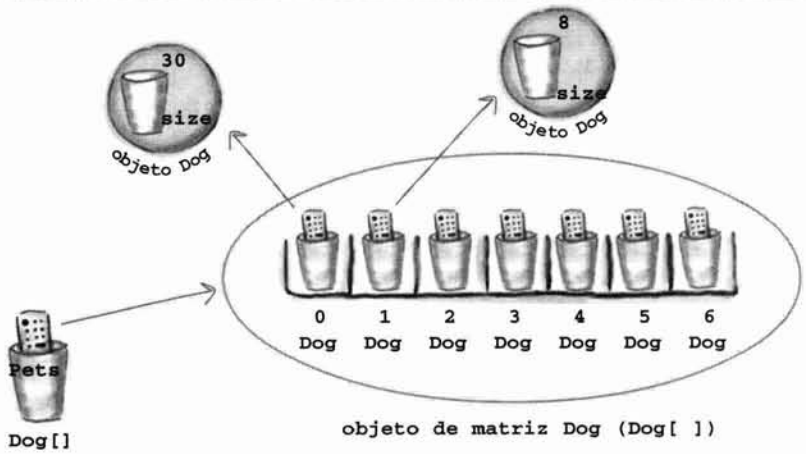
```
Dog[] pets;
pets = new Dog[7];
```

2

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

3

```
pets[0].setSize(30);
int x =
pets[0].getSize();
pets[1].setSize(8);
```



Declarando e inicializando variáveis de instância

Você já sabe que uma declaração de variável precisa de pelo menos um nome e um tipo:

```
int size;
String name;
```

E sabe que pode inicializar (atribuir um valor) a variável ao mesmo tempo:

```
int size = 420;
String name = "Donny";
```

Mas se você não inicializar uma variável de instância, o que acontecerá quando chamar um método de captura? Em outras palavras, qual será o *valor* de uma variável de instância *antes* de você a inicializar?

As variáveis de instância sempre recebem um valor padrão. Se você não atribuir explicitamente um valor a uma variável de instância, ou não chamar um método de configuração, mesmo assim ela terá um valor!

inteiros	0
pontos flutuantes	0,0
booleanos	falso
referências	nulo

```
class PoorDog {

   private int size;
   private String name;

   public int getSize() {
      return size;
   }
   public String getName() {
      return name;
   }
}

public class PoorDogTestDrive {
   public static void main (String[] args) {
      PoorDog one = new PoorDog();
      System.out.println("O tamahho do cão é " + one.getSize());
      System.out.println("O nome do cão é " + one.getName());
   }
}
```

Declara duas variáveis de instância, mas não atribui um valor

O que esses métodos retornarão?

O que você acha? Isso será compilado?

```
Arquivo Editar Janela Ajuda ChamarVet
%java PoorDogTestDrive
O tamanho do cão é 0
O nome do cão é nulo
```

Não é preciso inicializar variáveis de instância, porque elas sempre têm um valor padrão. Números primitivos (inclusive do tipo char) recebem o valor 0, booleanos recebem falso e variáveis de referência de objeto ficam com valor nulo.

(Lembre-se de que nulo significa apenas um controle remoto que não está controlando/programado para nada. É uma referência, porém sem um objeto real.)

A diferença entre variáveis de instância e locais

As variáveis **de instância** são declaradas dentro de uma classe, mas não dentro de um método.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // mais código...
}
```

As variáveis **locais** são declaradas dentro de um método.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

As variáveis locais NÃO recebem um valor padrão! O compilador reclamará se você tentar usar uma variável local antes dela ser inicializada.

As variáveis **locais** DEVEM ser inicializadas antes de ser usadas!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Não será compilado! Você pode declarar x sem um valor, mas, assim que tentar USÁ-LO, o compilador ficará confuso.

```
File Edit Window Help Yikes
%javac Foo.java
Foo.java:4: variable x might not have
been initialized
      int z = x + 3;
1 error       ^
```

Não existem Perguntas Idiotas

P: E quanto aos parâmetros do método? Como as regras sobre variáveis locais se aplicam a eles?

R: Os parâmetros dos métodos são praticamente iguais às variáveis locais — são declarados *dentro* do método (bem, tecnicamente eles são declarados na *lista de argumentos* do método em vez de dentro do *corpo* dele, mas ainda são variáveis locais e não variáveis de instância). Porém nunca serão inicializados, e você nunca verá uma mensagem de erro do compilador informando que uma variável de parâmetro não foi inicializada.

Mas isso ocorre porque o compilador exibirá uma mensagem de erro se você tentar chamar um método sem enviar os argumentos de que ele precisa. Portanto, os parâmetros são SEMPRE inicializados, já que o compilador garante que esses métodos sejam sempre chamados com argumentos que correspondam aos parâmetros declarados para eles, e os argumentos são atribuídos (automaticamente) aos parâmetros.

Comparando variáveis (primitivas ou de referência)

Haverá situações em que você pode querer saber se duas variáveis *primitivas* são iguais. Isso é muito fácil, basta que use o operador = =. Em outras, pode querer saber se duas variáveis de referência apontam para o mesmo objeto da pilha. Também é fácil, use o operador = =. Mas você pode querer saber se dois *objetos* são iguais. E para fazer isso, precisará do método .equals( ). A idéia de igualdade no que diz respeito a objetos depende do tipo de objeto. Por exemplo, se dois objetos String diferentes tiverem os mesmos caracteres (digamos, "ativo"), serão significativamente equivalentes, independentemente de serem dois objetos distintos da pilha. Mas e quanto a um Cão? Você vai querer tratar dois Cães como se fossem iguais se, por acaso, tiverem o mesmo tamanho e peso? Provavelmente não. Portanto, o fato de dois objetos diferentes serem tratados como se fossem iguais dependerá do que for relevante para esse tipo de objeto específico. Examinaremos a noção de igualdade entre objetos novamente em capítulos posteriores (e no Apêndice B), mas, por enquanto, temos que saber que o operador = = é usado *apenas* para comparar os bits de duas variáveis. *O que* esses bits representam não interessa. Eles são iguais ou não.

Use = = para comparar duas variáveis primitivas ou saber se duas referências apontam para o mesmo objeto.

Use o método equals( ) para saber se dois objetos *diferentes* são iguais.

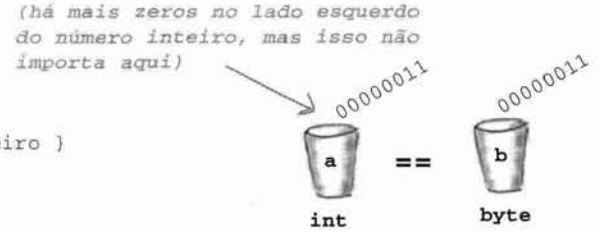
(Como dois objetos String diferentes representando os caracteres de "Fred".)

Para comparar duas variáveis primitivas, use o operador = =

O operador = pode ser usado para comparar duas variáveis de qualquer tipo, e ele comparará apenas os bits.

A instrução if (a= =b) {...} examinará os bits de a e b e retornará verdadeiro se o padrão de bits for o mesmo (porém ela não verificará o tamanho da variável, logo, os zeros do lado esquerdo são irrelevantes).

```
int a = 3;
byte b = 3;
if (a = = b) { // verdadeiro }
```



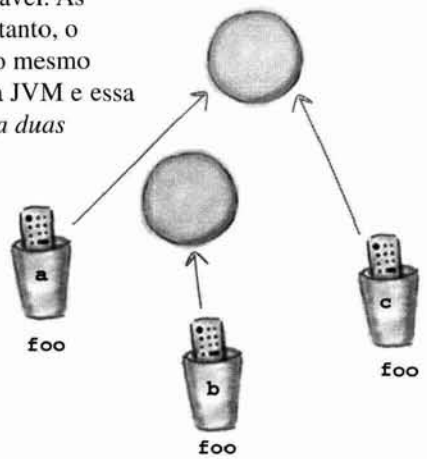
os padrões de bits são os mesmos, portanto, essas duas variáveis serão iguais se usarmos = =

Para saber se duas referências são iguais (o que significa que elas referenciam o mesmo objeto da pilha) use o operador = =

Lembre-se de que o operador = = só leva em consideração o padrão de bits da variável. As regras serão as mesmas, sendo a variável uma referência ou um tipo primitivo. Portanto, o operador = = retornará verdadeiro se duas variáveis de referência apontarem para o mesmo objeto! Nesse caso, não saberemos qual é o padrão de bits (porque não depende da JVM e essa informação estará oculta), mas *sabemos* que, qualquer que seja, *será o mesmo para duas referências que apontem para o mesmo objeto.*

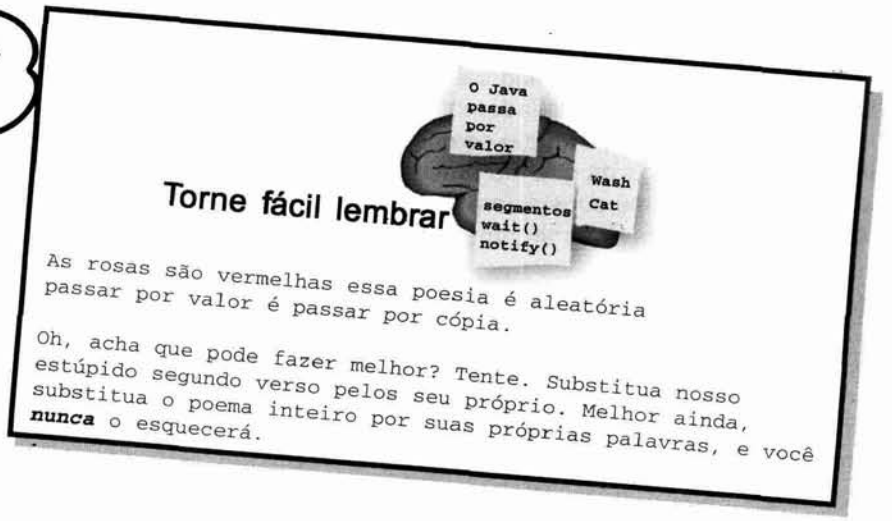
```
Foo a =  new Foo();
Foo b = new Foo();
Foo c = a;

if (a = = b) { // falso}
if (a = = c) { // verdadeiro}
if (b = = c) { // falso}
```

Os padrões de bits são os mesmos para a e c, portanto, serão iguais se usarmos = =



a = = c é verdadeiro

a = = b é falso



Aponte seu lápis

O que é válido?

Dado o método a seguir, qual das chamadas listadas à direita são válidas?

Insira uma marca de seleção próxima às chamadas que forem válidas. (Algumas instruções só foram incluídas para atribuir os valores usados nas chamadas do método.)

```
int calcArea(int height, int width) {
   return height * width;
}
```

Mantenha-se à direita

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c,15);
int d = calcArea(57);
calcArea(2,3);
long t = 42;
int f = calcArea(t,17);
int g = calcArea();
calcArea();
byte h = calcArea(4,20);
int j = calcArea(2,3,5);
```

Exercício

Seja o compilador

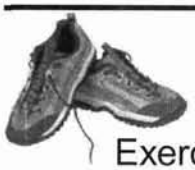
Cada um dos arquivos Java dessa página representa um arquivo-fonte completo. Sua tarefa é personificar o compilador e determinar se cada um deles pode ser compilado. Se não puderem ser compilados, como você os corrigiria, e, se eles forem compilados, qual seria sua saída?

A

```
class XCopy {

   public static void main(String [] args) {

      int orig = 42;

      XCopy x = new XCopy();

      int y = x.go(orig);

      System.out.println(orig + " " + y);
   }

   int go(int arg) {

      arg = arg * 2;

      return arg;
   }
}
```

B

```
class Clock {
   String time;

   void setTime(String t) {
      time = t;
   }

   void getTime() {
      return time;
   }
}

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

      Clock c = new Clock();

      c.setTime("1245");
      String tod = c.getTime();
      System.out.println("time: " + tod);
   }
}
```



Exercício

Quem sou eu?

Um grupo de componentes Java, vestidos a rigor, está participado do jogo, "Quem sou eu?" Eles lhe darão uma pista e você tentará adivinhar quem são, baseado no que disserem. Suponha que eles sempre digam a verdade quando falam de si mesmos. Se por acaso disserem algo que possa ser verdadeiro para mais de um deles, anote todos aos quais a frase possa ser aplicada. Preencha as linhas em branco próximas à frase com os nomes de um ou mais candidatos.

Candidatos desta noite:

Variável de instância, argumento, retorno, método de captura, método de configuração, encapsulamento, public, private, passar por valor, método

Uma classe pode ter quantos quiser. _____

O método só pode ter um. _____

Pode ser elevado implicitamente. _____

Prefiro minhas variáveis de instância privadas. _____

Na verdade significa 'fazer uma cópia'. _____

Só os métodos de configuração devem atualizá-los. _____

Um método pode ter muitos deles. _____

Retorno algo por definição. _____

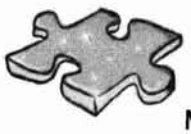
Não devo ser usado com variáveis de instância. _____

Posso ter muitos argumentos. _____

Por definição, uso um argumento. _____

Ajudam a criar o encapsulamento. _____

Estou sempre sozinho. _____



Mensagens misturadas

Um programa Java curto está listado à sua direita. Dois blocos do programa estão faltando. Seu desafio é **comparar os blocos de código candidatos** (a seguir) **com a saída** que você veria se eles fossem inseridos.

Nem todas as linhas de saída serão usadas, e algumas delas podem ser usadas mais de uma vez. Desenhe linhas conectando os blocos de código candidatos à saída de linha de comando correspondente.

Candidatos:

```
x < 9
index < 5
```

```
x < 20
index < 5
```

```
x < 7
index < 7
```

```
x < 19
index < 1
```

Saídas possíveis:

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

```
20 5
```

```
public class Mix4 {
   int counter = 0;
   public static void main(String [] args) {
      int count = 0;
      Mix4 [] m4a  =new Mix4[20];
      int x = 0;

      while (          ) {

         m4a[x] = new Mix4();
         m4a[x].counter = m4a[x].counter + 1;
         count = count + 1;
         count = count + m4a[x].maybeNew(x);
         x = x + 1;
      }
      System.out.println(count + " " + m4a[1].counter);
   }

   public int maybeNew(int index) {

      if (          ) {

         Mix4 m4 = new Mix4();
         m4.counter = m4.counter + 1;
         return 1;
      }
      return 0;
   }
}
```

Quebra-cabeças na Piscina

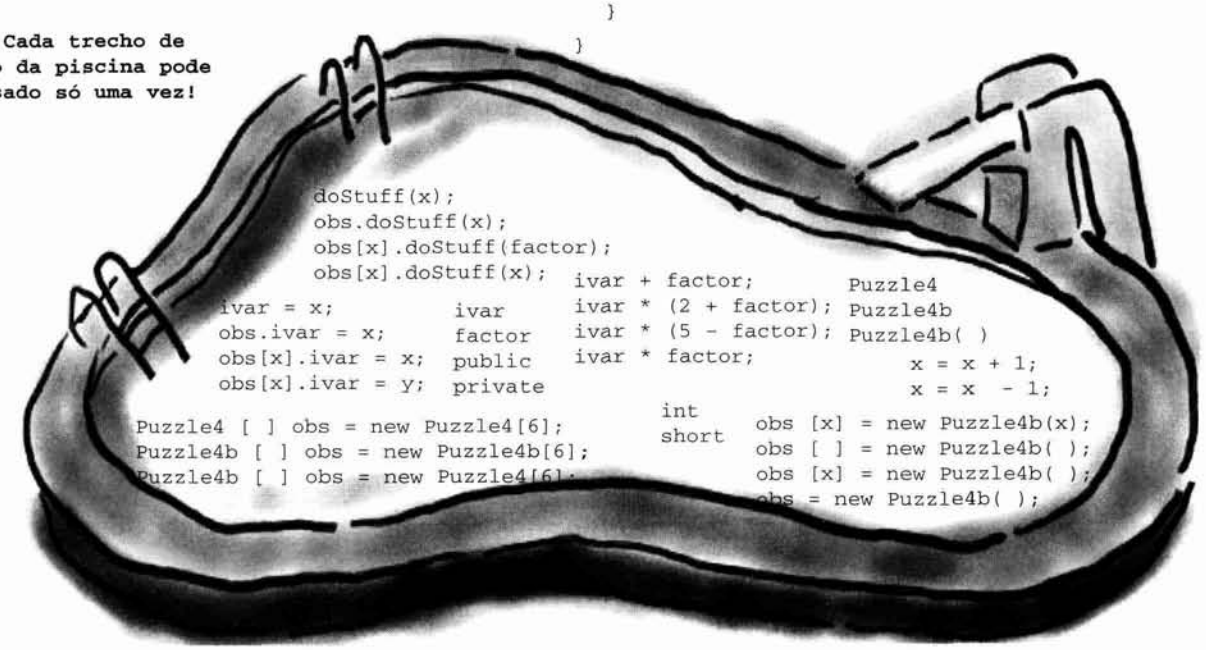
Sua *tarefa* é pegar os trechos de código da piscina e inseri-los nas linhas em branco do código. Você pode **não** usar o mesmo trecho mais de uma vez e não precisa empregar todos os trechos. Seu *objetivo* é criar uma classe que seja compilada e executada produzindo a saída listada.

```
public class Puzzle4 {
   public static void main(String [] args) {
      ____________________________________
      int y = 1;
      int x = 0;
      int result = 0;
      while (x < 6) {
         ____________________________
         ____________________________
         y = y * 10;
         ________________
      }
      x = 6;
      while (x > 0) {
         ________________
         result = result + ____________________
      }
      System.out.println("result " + result);
   }
}

class ___________ {
   int ivar;
   ________  ______ doStuff(int _________) {
      if (ivar > 100) {
         return ____________________________
      } else {
         return ____________________________
      }
   }
}
```

Saída

```
File Edit Window Help BellyFlop
%java Puzzle4

result 543345
```

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



```
doStuff(x);
obs.doStuff(x);
obs[x].doStuff(factor);
obs[x].doStuff(x);

ivar = x;
obs.ivar = x;
obs[x].ivar = x;
obs[x].ivar = y;

ivar
factor
public
private

ivar + factor;
ivar * (2 + factor);
ivar * (5 - factor);
ivar * factor;

int
short

Puzzle4
Puzzle4b
Puzzle4b( )
    x = x + 1;
    x = x  - 1;

Puzzle4 [ ] obs = new Puzzle4[6];
Puzzle4b [ ] obs = new Puzzle4b[6];
Puzzle4b [ ] obs = new Puzzle4[6];

obs [x] = new Puzzle4b(x);
obs [ ] = new Puzzle4b( );
obs [x] = new Puzzle4b( );
obs = new Puzzle4b( );
```

Um pequeno mistério

Tempos difíceis em Stim-City

Quando Buchanan encostou sua arma em Jai, ele congelou. Jai sabia que Buchanan era tão estúpido quanto feio e não queria assustar o grandalhão. Buchanan ordenou que ele entrasse no escritório de seu chefe, mas como Jai não tinha feito nada de errado (ultimamente), pensou que conversar com Leveler, o chefe de Buchanan, não seria tão ruim. Ele vinha movimentando muitos estimulantes neurais no lado oeste nos últimos tempos e achava que Leveler ficaria satisfeito. Estimulantes do mercado negro não geravam as maiores quantias que circulavam, mas eram inofensivos. A maioria dos viciados em estimulantes que ele conheceu desistiu após algum tempo e voltou à vida normal, talvez um pouco menos concentrada do que antes.

O 'escritório' de Leveler era um esconderijo de má aparência, mas quando Buchanan o empurrou para dentro, Jai pôde ver que tinha sido modificado para fornecer toda a velocidade e segurança extra que um chefe local como Leveler poderia esperar. "Jai meu caro", sussurrou Leveler, "bom te ver novamente". "A recíproca é verdadeira...", disse Jai, sentindo a malícia por trás do cumprimento de Leveler, "devíamos estar quites, Leveler, há algo que eu não saiba?" "Ah! Você está fazendo tudo direitinho, Jai, sua carga foi preenchida, mas tenho sentido, digamos, algumas 'falhas' ultimamente..." disse Leveler.

Jai recuou involuntariamente, ele tinha sido um ótimo hacker no auge de sua carreira. Sempre que alguém descobria como burlar a segurança de um sistema, uma atenção indesejada se voltava para Jai. "Não fui eu meu caro", disse Jai, "não vale a pena. Eu me aposentei das invasões, agora cuido dos meus próprios negócios". "Sim, sim", sorriu Leveler, "tenho certeza de que dessa vez não foi você, mas vou perder grandes margens de lucro até que esse novo hacker seja eliminado!" "Bem, boa sorte Leveler, me deixe aqui e eu conseguirei mais algumas 'unidades' para você antes de terminar por hoje", disse Jai.

"Receio que não seja tão fácil, Jai, Buchanan disse que agora você está trabalhando com a J37NE", insinuou Leveler. "A Edição Neural? É verdade que me divirto um pouco com ela, e daí?", Jai respondeu sentindo-se um pouco preocupado. "A edição neural é como deixo os viciados em estimulantes saberem onde poderão encontrar a próxima dose", explicou Leveler. "O problema é que algum viciado ficou sem o estimulante por tempo suficiente para descobrir como invadir meu banco de dados WareHousing". "Preciso de alguém que pense rápido como você, Jai, para examinar minha classe J37NE StimDrop; os métodos, as variáveis de instância, o conjunto todo, e descobrir como estão invadindo. Você terá...". "Ei!", exclamou Buchanan, "não quero nenhum hacker como Jai examinando meu código!". "Calma grandão", Jai percebeu uma chance, "Tenho certeza de que você fez um ótimo trabalho com seu modificador de aces...". "Você acha mesmo, embaralhador de bits!", gritou Buchanan, "Deixei públicos todos os métodos usados pelos viciados, para que eles pudessem acessar os dados do site, mas marquei todos os métodos críticos do WareHousing como privados. Ninguém do ambiente externo pode acessar esses métodos meu caro, ninguém!"

"Acho que posso identificar a falha, Leveler; o que diz de deixarmos Buchanan aqui e darmos uma volta pelo quarteirão", sugeriu Jai. Buchanan procurou sua arma, mas a mão de Leveler já estava em seu pescoço, "Deixe estar, Buchanan", sorriu Leveler, "Largue a arma e saia, acho que Jai e eu temos alguns planos a pôr em prática".

De que Jai suspeitou?

Ele conseguirá sair do esconderijo de Leveler com todos os ossos no lugar?

Soluções dos Exercícios

A

A classe 'XCopy' será compilada e executada na forma em que se encontra! A saída será: '42 84'. Lembre-se de que a Java passa por valor (o que significa passar por cópia), a variável 'orig' não será alterada pelo método go( ).

B

```
class Clock {
   String time;
   void setTime(String t) {
      time = t;
   }
   String getTime() {
      return time;
   }
}

class ClockTestDrive {
   public static void main(String [] args) {
      Clock c = new Clock();
      c.setTime("1245");
      String tod = c.getTime();
      System.out.println("time: " + tod);
   }
}
```

Nota: os métodos 'de captura' têm um tipo de retorno por definição.

Uma classe pode ter quantos quiser.	**variáveis de instância, métodos de captura e configuração**
O método só pode ter um.	**retorno**
Pode ser elevado implicitamente.	**retorno, argumento**
Prefiro minhas variáveis de instância privadas.	**encapsulamento**
Na verdade significa 'fazer uma cópia'.	**passar por valor**
Só os métodos de configuração devem atualizá-las.	**variáveis de instância**
Um método pode ter muitos deles.	**argumento**
Retorno algo por definição.	**método de captura**
Não devo ser usado com variáveis de instância.	**public**
Posso ter muitos argumentos.	**método**
Por definição, uso um argumento.	**método de configuração**
Ajudam a criar o encapsulamento.	**métodos de captura e configuração, public, private**
Estou sempre sozinho.	**retorno**

Soluções dos Quebra-cabeças

```
public class Puzzle4 {
   public static void main(String [] args) {
      Puzzle4b [ ] obs = new Puzzle4b[6];
      int y = 1;
      int x = 0;
      int result = 0;
      while (x < 6) {
         obs[x] = new Puzzle4b( );
         obs[x] . ivar = y;
         y = y * 10;
         x = x + 1;
      }
      x = 6;
      while (x > 0) {
         x = x - 1;
         result = result + obs[x].doStuff(x);
      }
      System.out.println("result " + result);
   }
}

class Puzzle4b {
   int ivar;
   public int doStuff(int factor) {
      if (ivar > 100) {
         return ivar * factor;
      } else {
         return ivar * (5 - factor);
      }
   }
}
```

Resposta do pequeno mistério...

Jai sabia que Buchanan não era muito inteligente. Quando falou sobre seu código, Buchanan não mencionou as variáveis de instâncias. Jai suspeitou que embora Buchanan tivesse realmente manipulado seus métodos corretamente, não tinha marcado suas variáveis de instância com private. Esse deslize pode facilmente ter custado milhões a Leveler.