Melhor Viver em Objetópolis

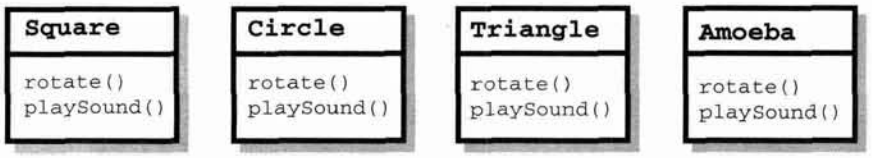
Planeje seus programas com o futuro em mente. Se houvesse uma maneira de escrever código Java de tal modo que se pudesse tirar mais férias, o quanto isso seria bom para você? E se pudesse escrever códigos que *outra* pessoa conseguisse estender **facilmente**? E se pudesse escrever códigos que fossem flexíveis, para aquelas irritantes alterações de último minuto nas especificações, isso seria algo no qual estaria interessado? Então este é seu dia de sorte. Por apenas três pagamentos facilitados de 60 minutos, você poderá ter tudo isso. Quando chegar ao Planejamento do Polimorfismo, você aprenderá as 5 etapas para a obtenção de um projeto de classes mais adequado, os 3 truques do polimorfismo, as 8 maneiras de criar um código flexível e, se agir agora — uma lição bônus sobre as 4 dicas para a exploração da herança. Não demore, uma oferta dessa grandeza lhe fornecerá a liberdade para projetar e a flexibilidade para programar que você merece. É rápido, é fácil e já está disponível. Comece hoje e forneceremos um nível extra de abstração!

A guerra das cadeiras revisitada...

Lembra-se do Capítulo 4, quando Larry (o profissional dos procedimentos) e Brad (o sujeito da OO) estavam competindo pela cadeira Aeron? Vejamos alguns trechos dessa história para examinarmos os aspectos básicos da herança.

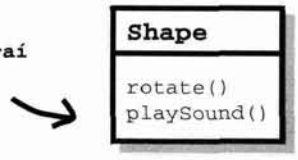
Larry: Você tem código duplicado! O procedimento de rotação aparece em todos os quatro itens Shape. É um projeto estúpido. Você tem que manter *quatro* "métodos" de rotação diferentes. Em que isso poderia ser bom?

Brad: Oh, acho que você não viu o projeto final. Deixe que eu lhe mostre como a **herança** da OO funciona, Larry.

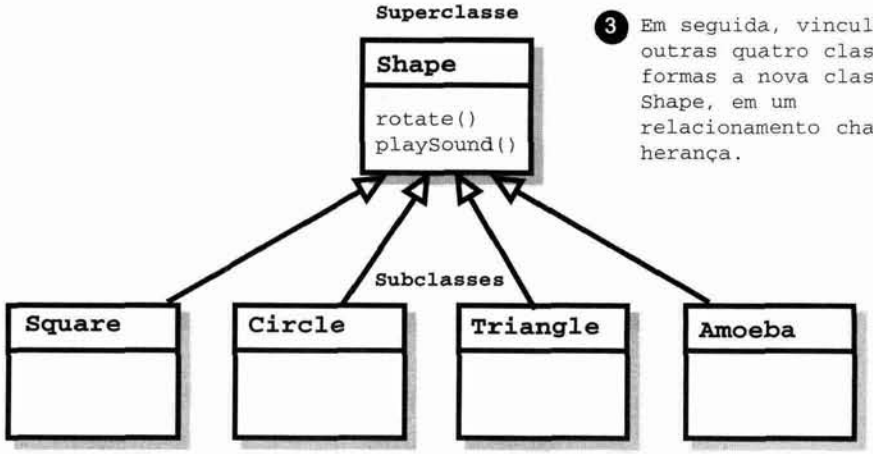


1 **Procurei o que as quatro classes tinham em comum.**

2 **Elas são formas e todas giram e reproduzem som. Portanto Portanto extraí os recursos comuns e os inseri em uma nova classe chamada Shape.**



Você pode ler isso como "**Square herda de Shape**", "**Circle herda de Shape**" e assim por diante. Removi rotate( ) e playSound( ) das outras formas, portanto agora há apenas uma cópia a manter. Diz-se que a classe Shape é a **superclasse** das outras quatro classes. As outras quatro são as **subclasses** de Shape. As subclasses herdam os métodos da superclasse. Em outras palavras, ***se a classe Shape tiver uma funcionalidade, então, as subclasses automaticamente terão essa mesma funcionalidade.***



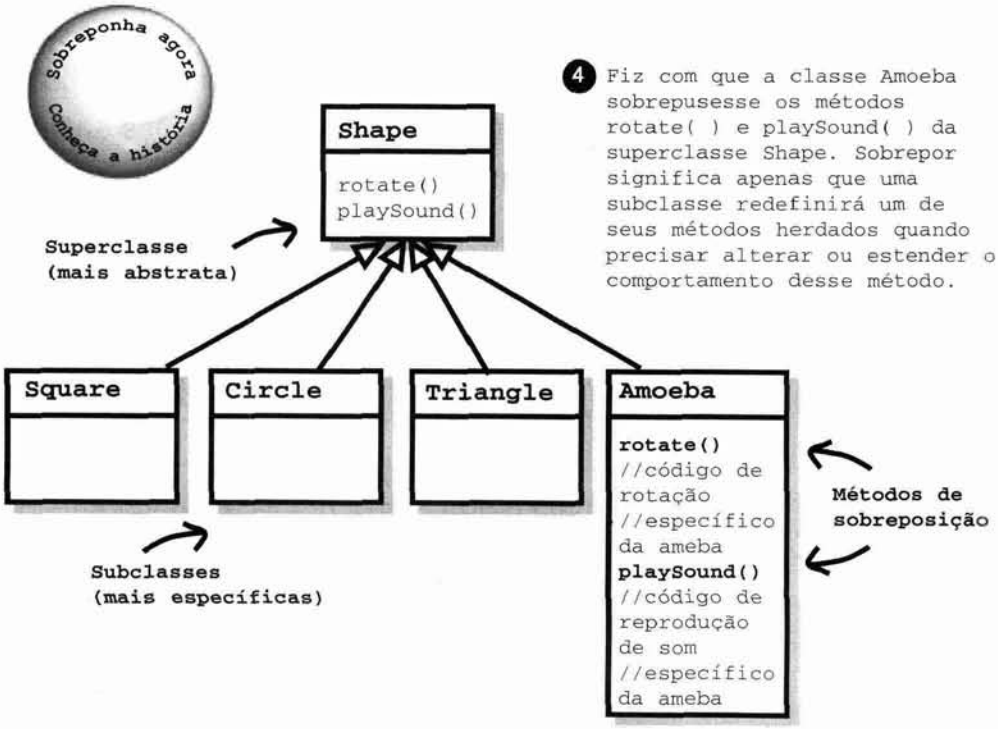
3 Em seguida, vinculei as outras quatro classes de formas a nova classe Shape, em um relacionamento chamado herança.

E quanto ao método rotate( ) de Amoeba?

Larry: Não é esse o problema aqui — que a forma de ameba tinha um procedimento de rotação e reprodução de som totalmente diferentes?

Como a ameba pode fazer algo diferente se ela *herda* sua funcionalidade da classe Shape?

Brad: Essa é a última etapa. A classe Amoeba ***sobrepõe*** os métodos da classe Shape. Portanto, no tempo de execução, a JVM saberá exatamente que método rotate( ) executar quando alguém solicitar que o objeto Amoeba gire.



4 Fiz com que a classe Amoeba sobrepusesse os métodos rotate( ) e playSound( ) da superclasse Shape. Sobrepor significa apenas que uma subclasse redefinirá um de seus métodos herdados quando precisar alterar ou estender o comportamento desse método.

O poder do cérebro

Como você representaria um gato doméstico e um tigre, em uma estrutura de herança? Um gato doméstico seria a versão especializada de um tigre? Qual seria a subclasse e quem seria a superclasse? Ou os dois são subclasses de alguma *outra* classe?

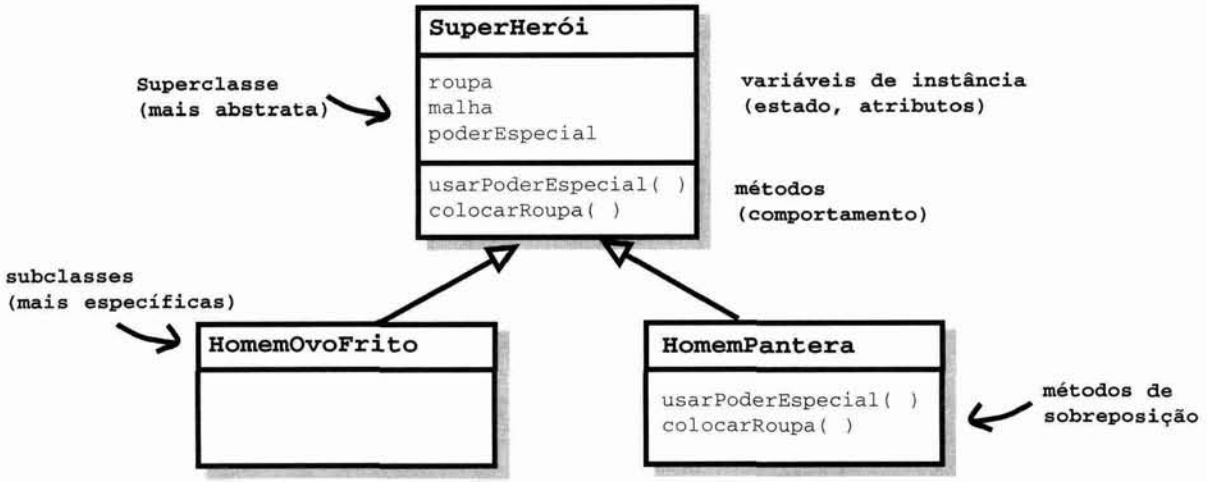
Como você projetaria uma estrutura de herança? Que métodos seriam sobrepostos?

Pense nisso. *Antes* de continuar lendo.

Entendendo a herança

Quando você projetar usando herança, inserirá código comum em uma classe e, em seguida, informará a outras classes mais específicas que a classe comum (mais abstrata) é sua superclasse. Quando uma classe herda de outra, *a subclasse herda da superclasse.*

Em Java, dizemos que a **subclasse *estende* a superclasse**. Um relacionamento de herança significa que a subclasse herdará os **membros** da superclasse. Com o termo "membros de uma classe" queremos nos referir às variáveis de instância e métodos. Por exemplo, se HomemPantera for uma subclasse de SuperHerói, a classe HomemPantera herdará automaticamente as variáveis de instância e métodos comuns a todos os super-heróis inclusive roupa, malha, poderEspecial, usarPoderEspecial( ) e assim por diante. Mas a **subclasse HomemPantera poderá adicionar novos métodos e variáveis de instância** exclusivos e **sobrepor os métodos que herdar da superclasse** SuperHerói.



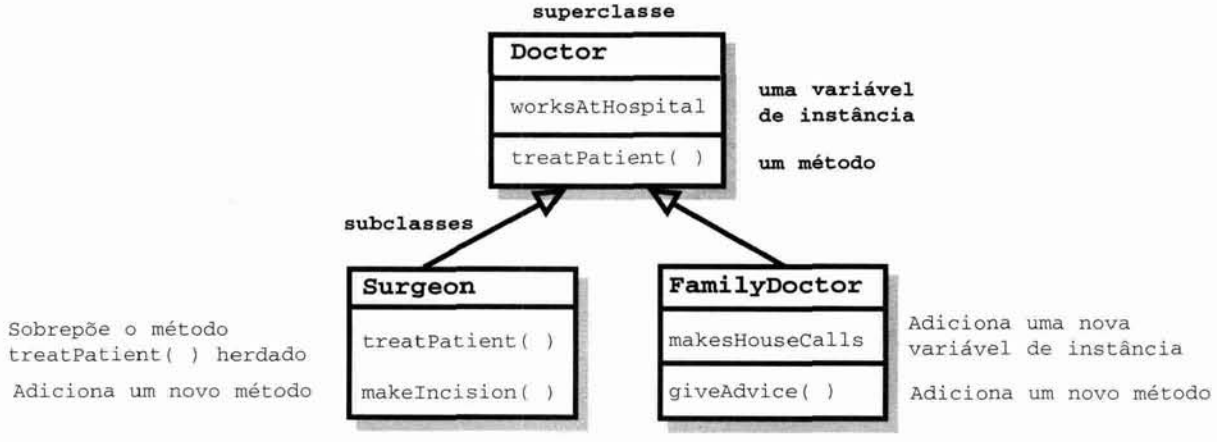
O HomemOvoFrito não precisa de nenhum comportamento que seja exclusivo, portanto ele não sobrepõe nenhum método. Os métodos e variáveis de instância de SuperHerói são suficientes. No entanto, o HomemPantera apresenta requisitos específicos quanto à roupa e aos poderes especiais, portanto usarPoderEspecial( ) e colocarRoupa( ) são sobrepostos na classe HomemPantera.

As variáveis de instância não são sobrepostas porque não precisam ser. Elas não definem nenhum comportamento especial, logo, uma subclasse pode fornecer a uma variável de instância herdada o valor que quiser. O HomemPantera pode configurar a malha que herdou com roxo, enquanto o HomemOvoFrito configurará a sua com branco.

Um exemplo de herança

```
public class Doctor {

   boolean worksAtHospital;

   void treatPatient() {
      // faz um check-up

   }
}

public class FamilyDoctor extends Doctor {

   boolean makesHouseCalls;
   void giveAdvice() {
      // dá conselhos
   }

}

public class Surgeon extends 'Doctor{

   void treatPatient() {
      // executa cirurgia
   }

   void makeIncision() {
      // faz incisão (eca!)
   }
}
```

Aponte seu lápis



Quantas variáveis de instância tem Surgeon?

Quantas variáveis de instância tem FamilyDoctor?

Quantos métodos tem Doctor?

Quantos métodos tem Surgeon?

Quantos métodos tem FamilyDoctor?

Uma classe FamilyDoctor pode usar o método treatPatient( )?

Uma classe FamilyDoctor pode usar o método makeIncision( )?

Projetemos a árvore de herança de um programa de simulação de animais

Suponhamos que você fosse solicitado a projetar um programa de simulação que permitisse ao usuário reunir vários animais diferentes em um ambiente para ver o que acontece. Não temos que codificá-lo agora, estamos mais interessados no projeto.

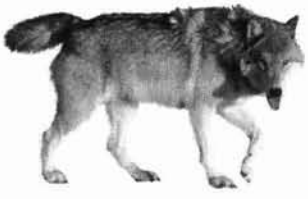
Recebemos uma lista com *alguns* dos animais que estarão no programa, mas não todos. Sabemos que cada animal será representado por um objeto e que os objetos se moverão dentro de um ambiente, fazendo o que cada tipo específico for programado para fazer.

E queremos que outros programadores possam adicionar novos tipos de animais ao programa a qualquer momento.

Primeiro temos que descobrir as características comuns abstratas que todos os animais apresentam e inseri-las dentro de uma classe que todas as classes de animais possam estender.

1 Procure objetos que possuam atributos e comportamentos em comum.

O que esses seis tipos têm em comum? Isso o ajudará a abstrair os comportamentos (etapa 2).
Como esses tipos estão relacionados? Isso o ajudará a definir os relacionamentos da árvore de herança (etapas 4-5)



Usando a herança para evitar a duplicação de código em subclasses

Temos cinco ***variáveis de instância:***

picture - o nome do arquivo que representa a figura JPEG desse animal

food -o tipo de alimento que esse animal come. No momento, só podemos ter dois valores: *meat* ou *grass*.

hunger -um inteiro que representa o nível de fome do animal. Sua alteração depende de quando (e quanto) o animal come.

boundaries - valores que representam a altura e largura do 'espaço' (por exemplo, 640 X 480) em que os animais circularão.

location - as coordenadas X e Y de onde o animal se encontra no espaço.

Temos quatro ***métodos***:

makeNoise( ) - comportamento para quando o animal tiver que fazer algum ruído.

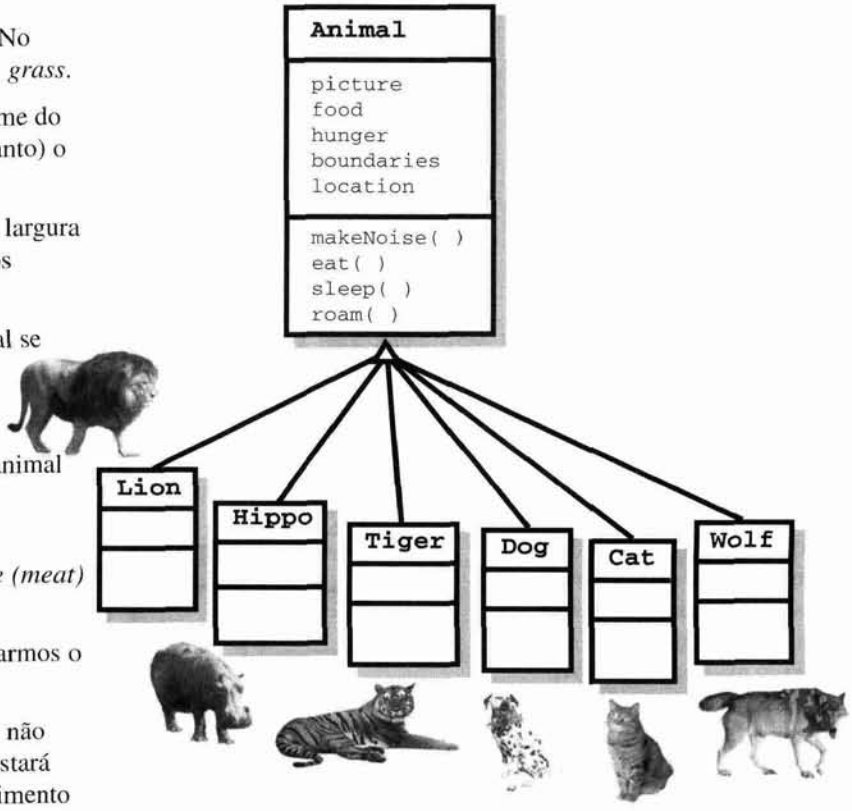
eat( ) - comportamento para quando o animal encontrar sua fonte de comida preferida, *carne (meat)* ou *capim (grass)*.

sleep( ) - comportamento para quando considerarmos o animal sonolento.

roam( ) - comportamento para quando o animal não estiver comendo ou dormindo (provavelmente estará apenas circulando até encontrar uma fonte de alimento ou chegar ao limite do espaço).

2 Projete uma classe que represente o estado e o comportamento comuns.

Esses objetos são todos animais, portanto criaremos uma superclasse comum chamada Animal. Inseriremos métodos e variáveis de instância que todos os animais podem precisar.



Todos os animais comem da mesma maneira?

Suponhamos que concordássemos com uma coisa: as variáveis de instância de *todos* os tipos de animais serão iguais. Um leão terá seu próprio valor para a figura, comida (estamos pensando em *carne*), fome, limites e local. Um hipopótamo terá *valores* diferentes para suas variáveis de instância, mas ele ainda terá as mesmas variáveis que os outros tipos de animais tiverem. O mesmo ocorrerá com o cão, o tigre e assim por diante. Mas e quanto ao *comportamento*?

Que métodos devemos sobrepor?

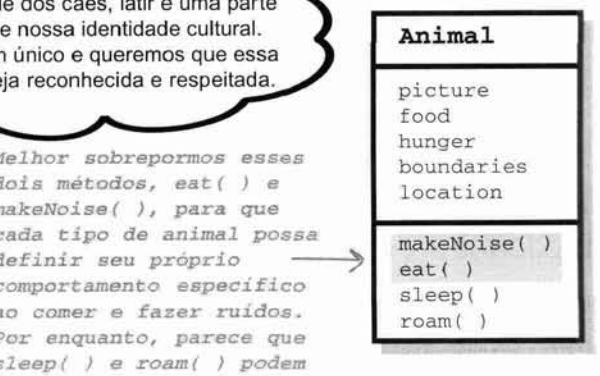
Um leão faz o mesmo **ruído** de um cão? Um gato **come** como um hipopótamo? Talvez em *sua* versão, mas, na nossa, comer e fazer ruídos são específicos do tipo de animal. Não sabemos como codificar esses métodos de uma maneira que eles sejam adequados a qualquer animal. Certo, isso não é verdade. Poderíamos escrever o método makeNoise( ), por exemplo, de um modo que tudo que teria que fazer seria reproduzir um arquivo de som definido em uma variável de instância desse tipo, mas isso não seria muito especializado. Alguns animais podem fazer ruídos diferentes em situações distintas (como um ruído para comer e outro quando ele salta sobre um inimigo, etc.).

Portanto exatamente como a Ameba sobrepôs o método rotate( ) da classe Shape, para ter um comportamento mais específico (em outras palavras, *exclusivo*), teremos que fazer o mesmo em nossas subclasses de Animal.

3 Defina se uma subclasse precisa de comportamentos (implementações de métodos) que sejam específicos desse tipo de subclasse em particular.

Examinando a classe Animal, definimos que eat( ) e makeNoise( ) devem ser sobrepostos pelas subclasses individuais.

Melhor sobrepormos esses dois métodos, eat( ) e makeNoise( ), para que cada tipo de animal possa definir seu próprio comportamento específico ao comer e fazer ruídos. Por enquanto, parece que sleep( ) e roam( ) podem permanecer genéricos.



Procurando mais oportunidades de usar a herança

A hierarquia de classes está começando a se compor. Cada classe está sobrepondo os métodos *makeNoise( )* e *eat( )*, para que não haja confusão entre o latido de um cão e o miado de um gato (o que seria um insulto para os dois grupos). E um hipopótamo não comerá como um leão.

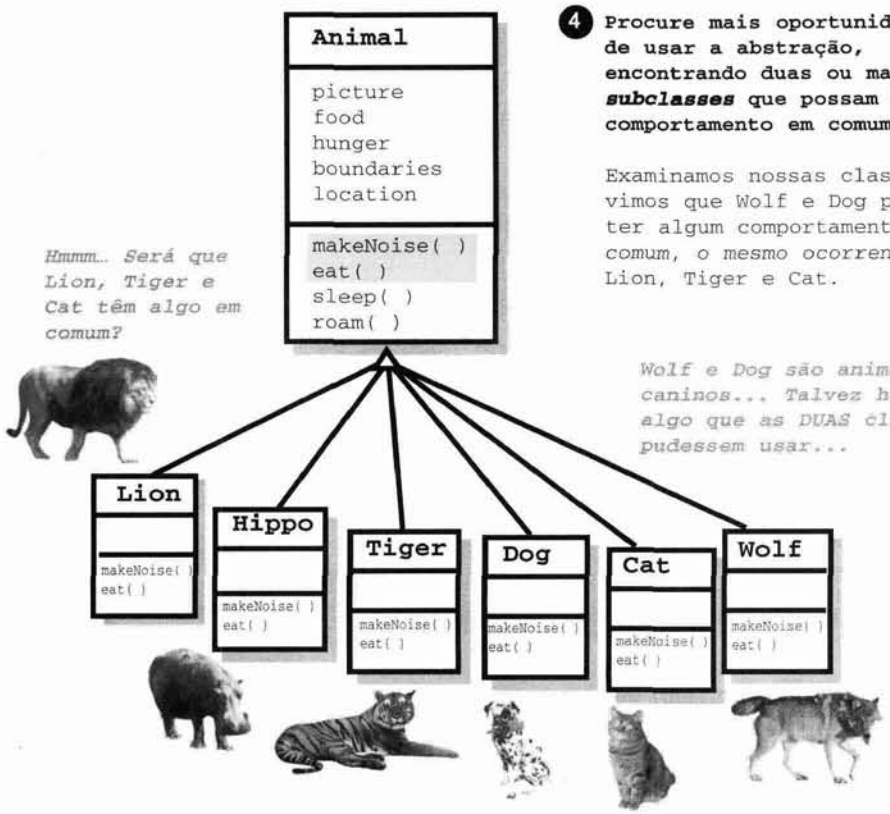
Mas talvez possamos fazer mais. Temos que examinar as subclasses de Animal e ver se duas ou mais podem ser agrupadas de alguma maneira, recebendo um código que seja comum apenas a *esse* novo grupo. O lobo e o cão apresentam semelhanças. Assim como o leão, o tigre e o gato.

4 Procure mais oportunidades de usar a abstração, encontrando duas ou mais *subclasses* que possam ter um comportamento em comum.

Examinamos nossas classes e vimos que Wolf e Dog podem ter algum comportamento em comum, o mesmo ocorrendo com Lion, Tiger e Cat.

Hmmm... Será que Lion, Tiger e Cat têm algo em comum?

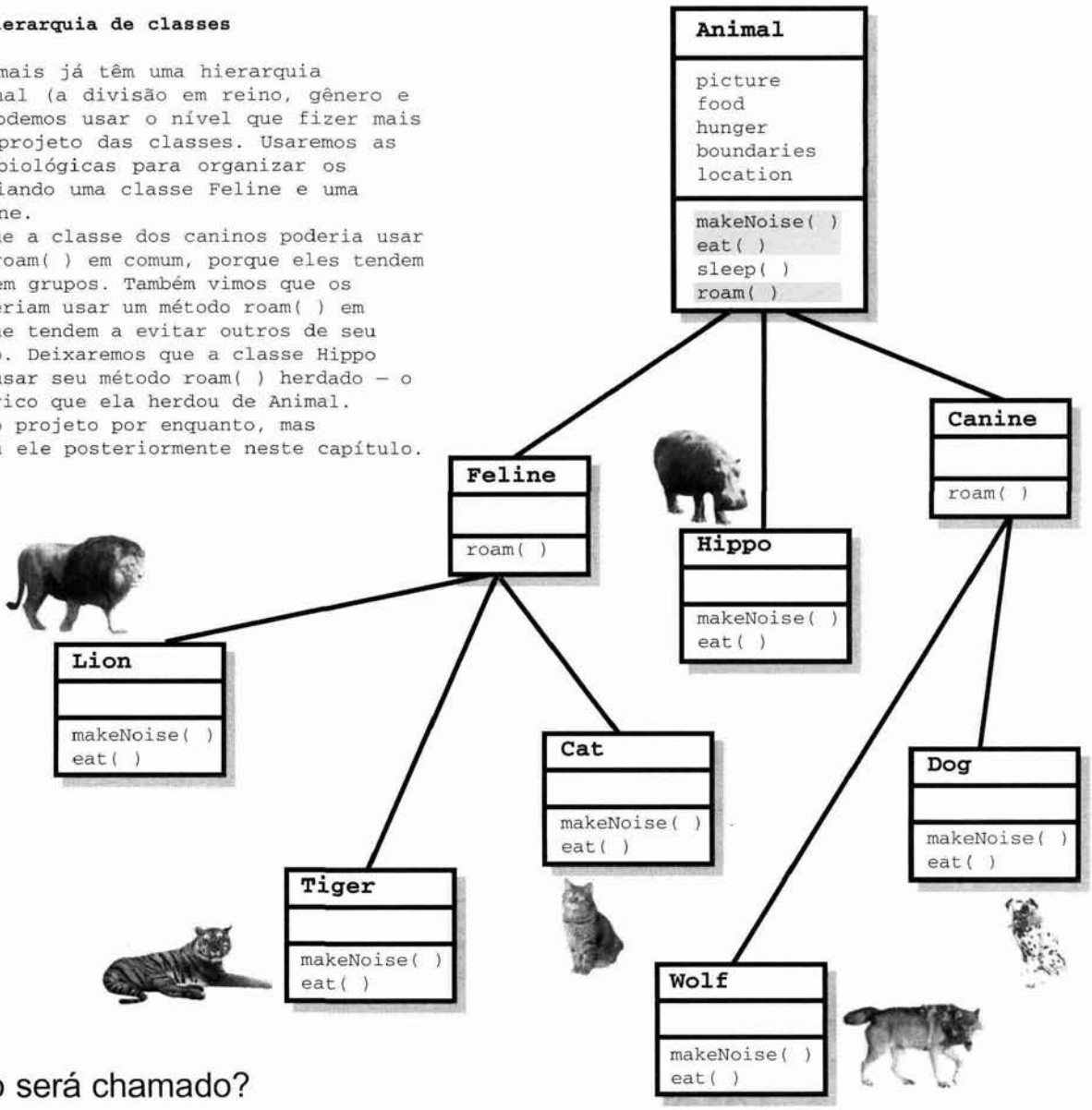
Wolf e Dog são animais caninos... Talvez haja algo que as DUAS classes pudessem usar...



5 Termine a hierarquia de classes

Como os animais já têm uma hierarquia organizacional (a divisão em reino, gênero e família), podemos usar o nível que fizer mais sentido ao projeto das classes. Usaremos as "famílias" biológicas para organizar os animais, criando uma classe Feline e uma classe Canine.

Decidimos que a classe dos caninos poderia usar um métodos roam( ) em comum, porque eles tendem a se mover em grupos. Também vimos que os felinos poderiam usar um método roam( ) em comum, porque tendem a evitar outros de seu próprio tipo. Deixaremos que a classe Hippo continue a usar seu método roam( ) herdado — o método genérico que ela herdou de Animal. Terminamos o projeto por enquanto, mas voltaremos a ele posteriormente neste capítulo.



Que método será chamado?

A classe Wolf tem quatro métodos. Um herdado de Animal, um herdado de Canine (que na verdade é a versão sobreposta de um método da classe Animal) e dois sobrepostos pela própria classe Wolf. Quando você criar um objeto Wolf e atribuí-lo a uma variável, poderá usar o operador ponto nessa variável de referência para chamar todos os quatro métodos. Mas que *versão* desses métodos será chamada?

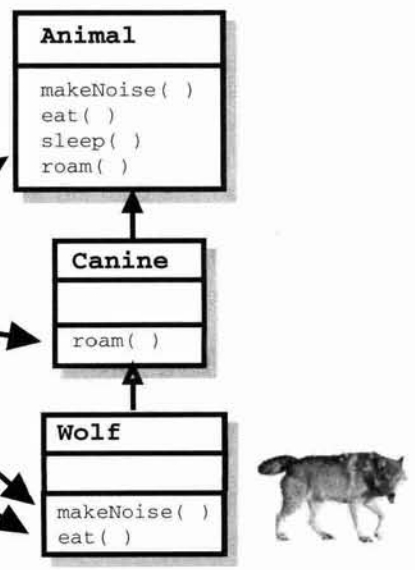
cria um novo objeto Wolf `Wolf w = new Wolf( );`

chama a versão de Wolf `w.makeNoise( );`

chama a versão de Canine `w.roam( );`

chama a versão de Wolf `w.eat( );`

chama a versão de Animal `w.sleep( );`



Quando você chamar um método em uma referência de objeto, estará chamando a versão mais específica do método para esse tipo de objeto.

Em outras palavras, ***o inferior vence!***

"Inferior" significando o mais baixo na árvore de herança. Canine é inferior a Animal e Wolf é inferior a Canine, portanto chamar um método na referência a um objeto Wolf significa que a JVM examinará primeiro a classe Wolf. Se não encontrar uma versão do método nessa classe, começará a retroceder na hierarquia de herança até achar algo que atenda.

Projetando uma árvore de herança

Classe	Superclasse	Subclasse
Roupas	—	Short, Camisa
Short	Roupas	
Camisa	Roupas	

Tabela de herança

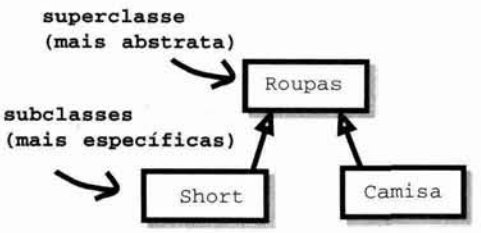


Diagrama de classes da herança

Aponte seu lápis

Defina os relacionamentos que fazem sentido. Preencha as duas últimas colunas.

Classe	Superclasse	Subclasse
Músico		
Cantor de rock		
Fã		
Baixista		
Pianista clássico		

Desenhe um digrama de herança aqui.

Dica: nem todas as classes podem ser conectadas a alguma outra classe.
Dica: você pode aumentar ou alterar as classes listadas.

Não existem Perguntas Idiotas

P: Você disse que a JVM iniciará subindo a árvore de herança, a partir do tipo de classe em que o método foi chamado (como no exemplo de Wolf na página anterior). Mas o que acontecerá se ela não conseguir encontrar algo correspondente?

R: Boa pergunta! Mas você não tem que se preocupar com isso. O compilador garante que um método específico possa ser chamado para um determinado tipo de referência, mas não informa (ou verifica) a *classe* de onde esse método veio realmente no tempo de execução. No exemplo de Wolf, o compilador procurará um método sleep( ), mas não se importará com o fato de ele ter sido definido (e herdado) na classe Animal. Lembre-se de que, se uma classe *herdar* um método, ela *terá* o método. *Onde* o método herdado foi definido (em outras palavras, em que superclasse ele foi definido) não faz diferença para o compilador. Mas no tempo de execução, **a JVM sempre usará o método correto.** E por correto queremos dizer ***a versão mais específica desse objeto em particular.***

Usando É-UM e TEM-UM

Lembre-se de que, quando uma classe herda de outras, dizemos que a subclasse *estende* a superclasse. Quando você quiser saber se uma coisa deve estender outra, aplique o teste É-UM.

O triângulo É-UMA Forma, sim, isso faz sentido.

O Gato é É-UM Felino, isso também faz sentido

O Cirurgião É-UM Médico, continua fazendo sentido

Banheira estende Banheiro, soa sensato.

Até você aplicar o teste É-UM.

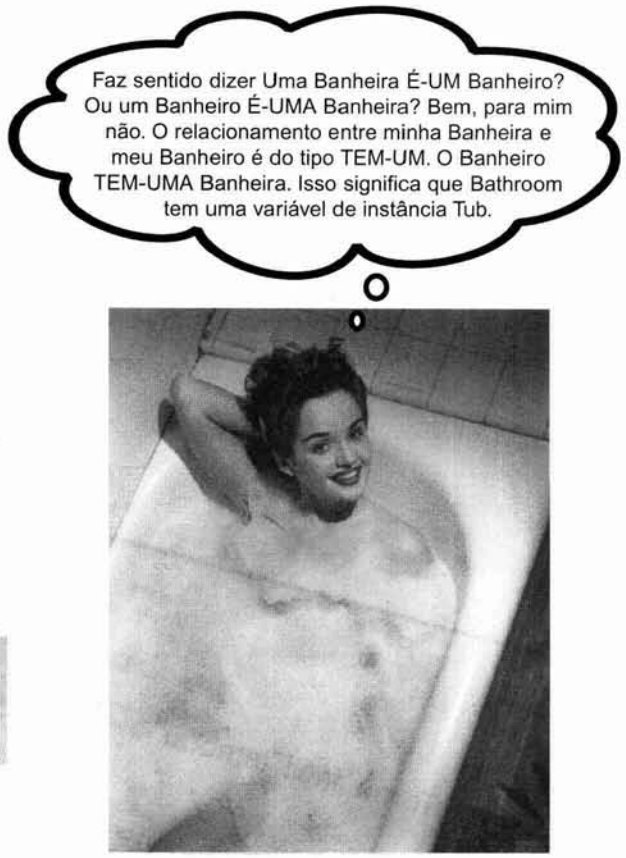
Para saber se você projetou seus tipos corretamente, pergunte "Faz sentido dizer X É-UM tipo Y?". Se não fizer, você saberá que algo está errado no projeto, portanto, se aplicarmos o teste É-UM, Banheira É-UM Banheiro é definitivamente falso.

E se invertermos para Banheiro estende Banheira? Isso ainda não faz sentido, Banheiro É-UMA Banheira não funciona.

Banheira e Banheiro *estão* relacionados, mas não através da herança. Estão associados por um relacionamento TEM-UM. Faz sentido dizermos "O Banheiro TEM-UMA Banheira?". Se fizer, isso significa que Banheiro (Bathroom) tem uma variável de instância Banheira (Tub). Em outras palavras, Bathroom tem uma *referência* a Tub, mas não estende Tub e vice-versa.

Bathroom TEM-UMA variável de instancia Tub e Tub TEM-UMA variável Bubbles.

Mas ninguém herda (estende) nada de ninguém.



Mas espere! Há mais!

O teste É-UM funciona em *qualquer local* da árvore de herança. Se sua árvore de herança tiver sido bem projetada, o teste É-UM deve fazer sentido quando você perguntar a *qualquer* subclasse se ela É-UM de seus supertipos.

Se a classe B estende a classe A, ela É-UMA classe A.

Isso será verdadeiro em qualquer local da árvore de herança. Se a classe C estender a classe B, ela passará no teste É-UM tanto com a classe B *quanto* com a classe A.

Canine estende Animal

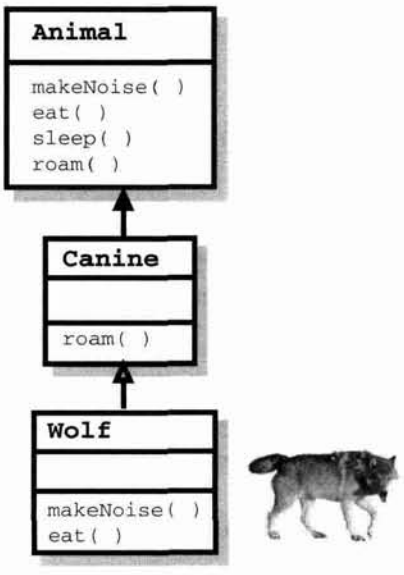
Wolf estende Canine

Wolf estende Animal

O Canino É-UM Animal

O Lobo É-UM Canino

O Lobo É-UM Animal



Em uma árvore de herança como a mostrada aqui, você *sempre* poderá dizer "**Wolf estende Animal**" ou "**O Lobo É-UM Animal**". Não faz diferença se Animal é a superclasse da superclasse de Wolf. Na verdade, **contanto que Animal esteja em *algum local* da hierarquia de herança acima de Wolf, Lobo É-UM Animal sempre será verdadeiro.**

A estrutura da árvore de herança de Animal mostra claramente que:

"o Lobo É-UM Canino, portanto pode fazer qualquer coisa que um Canino faria. E o Lobo É-UM Animal, logo, pode fazer qualquer coisa que um Animal faria."

Não faz diferença se Wolf sobrepõe alguns dos métodos de Animal ou Canine. No escopo geral (dos códigos), Wolf pode executar esses quatro métodos. *Como* os executa ou *em que classe eles são sobrepostos* não faz diferença. Wolf pode executar makeNoise( ), eat( ), sleep( ) e roam( ) porque estende a classe Animal.

Como saber se você construiu sua herança corretamente?

É claro que há mais coisas envolvidas do que o que foi abordado até agora, mas examinaremos outras questões referentes à OO no próximo capítulo (onde acabaremos detalhando e aperfeiçoando parte do projeto que construímos *neste* capítulo).

Por enquanto, porém, uma boa diretriz é usar o teste É-UM. Se "X É-UM Y" fizer sentido, é provável que as duas classes (X e Y) residam na mesma hierarquia de herança. Há chances de terem comportamentos iguais ou sobrepostos.

Lembre-se de que o relacionamento É-UM da herança funciona somente em uma direção!

O Triângulo É-UMA Forma faz sentido, portanto Triângulo pode estender Forma.

Mas o inverso — a Forma É-UM Triângulo — *não* faz sentido , portanto portanto a Forma não deve estender o Triângulo. Lembre-se de que o relacionamento É-UM implica que, se X É-UM Y, logo, X pode fazer qualquer coisa que Y faria (e possivelmente mais).

Aponte seu lápis

Insira uma marca de seleção ao lado dos relacionamentos que fazem sentido.

- ☐ Forno estende Cozinha
- ☐ Guitarra estende Instrumento
- ☐ Pessoa estende Funcionário
- ☐ Ferrari estende Motor
- ☐ OvoFrito estende Alimento
- ☐ Beagle estende AnimalEstimação
- ☐ Contêiner estende Jarro
- ☐ Metal estende Titânio
- ☐ GratefulDead estende Banda
- ☐ Loura estende Inteligente
- ☐ Bebida estende Martini

Dica: aplique o teste É-UM

Não existem Perguntas Idiotas

P: Vimos como uma subclasse faz para herdar um método da superclasse, mas e se a superclasse quiser usar a versão do método da subclasse?

R: Uma superclasse não *conhece* necessariamente todas suas subclasses. Você pode criar uma classe e muito tempo depois outra pessoa pode estendê-la. Mas mesmo se o criador da superclasse não conhecer (e quiser usar) a versão de um método da subclasse, não existe algo do tipo herança *inversa* ou *regressiva*. Pense bem, as crianças herdam dos pais e não o contrário.

P: Em uma subclasse, digamos que eu quisesse usar um método tanto com a versão da superclasse quanto com minha versão de sobreposição existente na subclasse? Em outras palavras, não quero *substituir* totalmente a versão da superclasse, apenas adicionar algo mais a ela.

R: Você pode fazer isso! E é um recurso importante do projeto. Pense na palavra "estende" significando "quero *estender* a funcionalidade da superclasse".

```
public void roam() {
   super.roam(); <-- esse bloco chama a versão herdada de roam( ) e, em seguida,
   // meu próprio método roam <-- retorna à execução do código específico da subclasse
}
```

Você pode atribuir os métodos de sua superclasse de uma maneira que eles contenham implementações de métodos que funcionarão para qualquer subclasse, mesmo se as subclasses tiverem que 'adicionar' mais código. No método de sobreposição de sua subclasse, você pode chamar a versão da superclasse usando a palavra-chave **super**. É como dizer "primeiro execute a versão da superclasse e, em seguida, volte e termine com meu próprio código...".

Quem fica com o Porsche e quem fica com a porcelana?

(como saber o que uma subclasse pode herdar de sua superclasse)

Uma subclasse herda membros da superclasse. Os membros incluem as variáveis de instância e métodos, porém mais adiante neste livro examinarmos outros membros herdados. Uma superclasse pode selecionar se quer ou não que uma subclasse herde um membro específico pelo nível de acesso que esse membro receber.

Há quatro níveis de acesso que abordaremos neste livro. Indo do maior ao menor nível de restrição, os quatro níveis de acesso são:

privado **padrão** **protegido** **público**

Os níveis de acesso controlam *quem vê o quê*, e são essenciais a um código Java robusto e bem-projetado. Por enquanto enfocaremos apenas os acessos público e privado. As regras desses dois são simples:

membros **públicos** *são* herdados

membros **privados** *não* são herdados

Quando uma subclasse herda um membro, é ***como se ela própria o definisse***. No exemplo de Shape, Square herdou os métodos rotate( ) e playSound( ) e para o ambiente externo (outros códigos) essa classe simplesmente *possui* os dois métodos.

Os membros de uma classe incluem as variáveis e métodos definidos na classe mais qualquer coisa herdada de uma superclasse.

Nota: veja mais detalhes sobre os acessos padrão e protegido no Capítulo 16 (implantação) e no Apêndice B.

Ao projetar empregando a herança, você está *usando* ou *abusando*?

Já que algumas das razões para a existência dessas regras só serão reveladas posteriormente neste livro, por enquanto, simplesmente *conhecer* algumas regras o ajudará a construir um projeto de herança mais adequado.

USE a herança quando uma classe for o tipo mais específico de uma superclasse. Exemplo: Salgueiro *é* um tipo mais específico de Árvore, portanto Salgueiro estender Árvore faz sentido.

CONSIDERE a herança quando tiver um comportamento (código implementado) que deva ser compartilhado entre várias classes do mesmo tipo geral. Exemplo: Square, Circle e Triangle precisam girar e reproduzir som, portanto inserir essa funcionalidade em uma superclasse Shape pode fazer sentido e proporcionará mais facilidade na manutenção e extensibilidade. Lembre-se, no entanto, de que, embora a herança seja um dos recursos-chave da programação orientada a objetos, não é necessariamente a melhor maneira de conseguirmos reutilizar comportamentos. Ela lhe ajudará a começar, e geralmente é a melhor opção de projeto, mas os padrões de projeto o ajudarão a conhecer outras opções mais sutis e flexíveis. Caso não conheça os padrões de projeto, um bom acompanhamento a este livro seria *Use a Cabeça! Design Patterns*.

NÃO use a herança apenas para poder reutilizar o código de outra classe, se o relacionamento entre a superclasse e a subclasse violar uma das duas regras acima. Por exemplo, suponhamos que você escrevesse um código especial de impressão na classe Alarme e agora tivesse que imprimir o código da classe Piano, portanto precisaria que Piano estendesse Alarme para que herdasse o código de impressão. Isso não faz sentido! Um Piano *não* é um tipo mais específico de Alarme. (Logo, o código de exibição deveria estar em uma classe Impressora, da qual todos os objetos imprimíveis pudessem se beneficiar através de um relacionamento TEM-UM.)

Não use a herança se a subclasse e a superclasse não passarem no teste É-UM. Pergunte sempre se a subclasse É-UM tipo mais específico da superclasse. Exemplo: Chá É-UMA Bebida faz sentido. Bebida É-UM Chá não.

DISCRIMINAÇÃO DOS PONTOS

- A subclasse *estende* a superclasse.

- Uma subclasse herda todas as variáveis de instância e métodos *públicos* da superclasse, mas não suas variáveis de instância e métodos *privados*.

- Os métodos herdados podem ser sobrepostos; as variáveis de instância *não* (embora possam ser *redefinidas* na subclasse, mas isso não é a mesma coisa, e quase nunca é preciso fazê-lo).

- Use o teste É-UM para verificar se sua hierarquia de herança é válida. Se X *estende* Y, então, X *É-UM* Y deve fazer sentido.

- O relacionamento É-UM funciona em uma única direção. Um hipopótamo é uma animal, mas nem todos os animais são hipopótamos.

- Quando um método é sobreposto em uma subclasse e é chamado em uma instância dela, a versão sobreposta do método é que é chamada. (*O inferior vence.*)

- Se a classe B estende A e C estende B, a classe B É-UMA classe A e a classe C É-UMA classe B, portanto a classe C também É-UMA classe A.

Mas o que toda essa herança lhe proporcionará?

Você se beneficiará muito da OO projetando com a herança. Poderá eliminar código duplicado generalizando o comportamento comum a um grupo de classes e inserindo esse código em uma superclasse. Assim, quando precisar alterá-lo, terá apenas um local a atualizar, e *a alteração repercutirá instantaneamente em todas as classes que herdarem esse comportamento.* Bem, não se trata de mágica, na verdade *é* muito simples: faça a alteração e compile a classe novamente. Apenas isso. **Você não terá que mexer nas subclasses!**

Basta distribuir a superclasse recém-alterada, e todas as classes que a estenderem usarão automaticamente a nova versão.

Um programa Java nada mais é do que uma pilha de classes, portanto as subclasses não precisam ser recompiladas para usar a nova versão da superclasse. Contanto que a superclasse não *trave* nada na subclasse, tudo estará bem. (Discutiremos o que a palavra 'travar' significa nesse contexto posteriormente no livro. Por enquanto, pense nela como a modificadora de algo na superclasse de que a subclasse dependa, como os argumentos ou o tipo de retorno de um método específico ou o nome do método, etc.)

① Você evitará código duplicado.
Insira o código comum em um local e deixe as subclasses herdarem esse código de uma superclasse. Quando quiser alterar esse comportamento, só precisará fazê-lo em um local e todo mundo (isto é, todas as subclasses) ficará sabendo da alteração.

② Você definirá um protocolo comum para um grupo de classes.

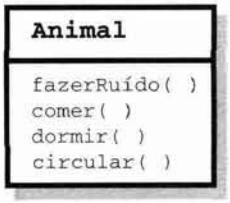
A herança lhe permitirá garantir que todas as classes agrupadas sob um certo supertipo tenham todos os métodos que o supertipo tem.*

Em outras palavras, você definirá um protocolo comum para um conjunto de classes relacionadas através da herança.

Quando você definir métodos em uma superclasse, que possam ser herdados por subclasses, estará anunciando um tipo de protocolo para outros códigos que diz "todos os meus subtipos (isto é, as subclasses) poderão fazer essas coisas, com esses métodos que têm essa assinatura..."

Em outras palavras, você estabelecerá um *contrato.*

A classe Animal estabelece um protocolo comum para todos os seus subtipos:



Você está dizendo para o resto do mundo que qualquer animal pode fazer essas quatro coisas. Isso inclui os argumentos e tipos de retorno do método.

E lembre-se de que, quando dizemos *qualquer animal*, estamos nos referindo à classe Animal e *qualquer outra classe que a estenda*. O que significa também, *qualquer classe que tenha a classe Animal em algum local acima dela na hierarquia de herança.*

Mas ainda nem chegamos na parte realmente interessante, porque deixamos o melhor - *o polimorfismo* - por último.

Quando você definir um supertipo para um grupo de classes, *qualquer subclasse desse supertipo poderá ser substituída onde o supertipo for esperado.*

Como é mesmo?

Não se preocupe, ainda não acabamos de explicar. Após avançarmos duas páginas, você será um especialista.

*Quando dizemos "todos os métodos" estamos nos referindo a "todos os métodos que *podem ser herdados*", o que por enquanto significa "todos os métodos *públicos*", porém essa definição será melhorada posteriormente.

E me preocupo porque...

Porque você se beneficiará do polimorfismo.

O que importa para mim porque...

Porque você poderá chamar o objeto de uma subclasse usando uma referência declarada como o supertipo.

E isso significa para mim que...

Você poderá escrever códigos realmente flexíveis. Códigos que serão mais claros (mais eficientes e simples). Códigos que não serão apenas mais fáceis de *desenvolver*, mas também muito mais fáceis de *estender*, de maneiras que você nunca imaginou no momento em que originalmente os escreveu.

Isso significa que você poderá tirar aquelas férias tropicais enquanto seus colaboradores atualizam o programa e talvez eles nem precisem de seu código-fonte.

Você verá como isso funciona logo abaixo.

Não o conhecemos, mas pessoalmente, achamos as férias tropicais particularmente motivadoras.

Para ver como o polimorfismo funciona, temos que voltar para examinar a maneira como *normalmente* declaramos uma referência e criamos um objeto...

As 3 etapas de declaração e atribuição de objetos

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

1 Declare uma variável de referência

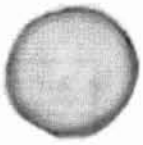
```
Dog myDog = new Dog( );
```

Solicita à JVM para alocar espaço para uma variável de referência. 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( );
```

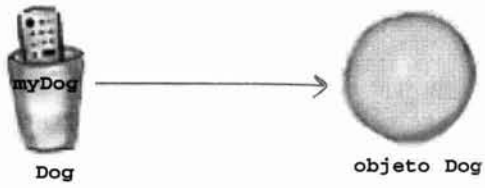
Solicita à JVM para alocar espaço para um novo objeto Dog na pilha de lixo coletável.



3 Vincule o objeto e a referência

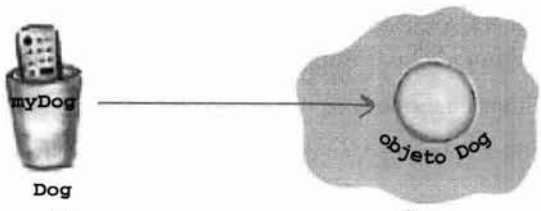
```
Dog myDog = new Dog( );
```

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



O importante é que o tipo da referência E o tipo do objeto sejam iguais.

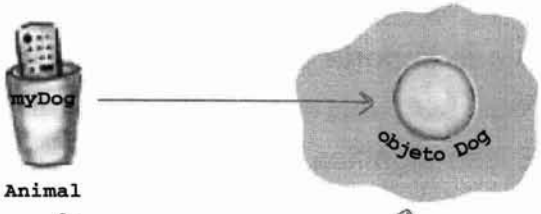
Nesse exemplo, os dois são do tipo Dog.



Esses dois tipos são iguais. O tipo da variável de referência foi declarado como Dog e o objeto foi criado como novo objeto Dog( ).

Mas com o polimorfismo, a referência e o objeto podem ser *diferentes*.

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



Esses dois NÃO são tipos iguais. O tipo da variável de referência foi declarado como Animal, mas o objeto foi declarado como novo objeto Dog( ).

No polimorfismo, o tipo da referência pode ser uma superclasse com o tipo do objeto real.

Quando você declarar uma variável de referência, qualquer objeto que passar no teste É-UM quanto ao tipo declarado para ela poderá ser atribuído a essa referência. Em outras palavras, qualquer coisa que *estender* o tipo declarado para a variável de referência poderá ser *atribuída* a ela. ***Isso permitirá que você faça coisas como criar matrizes polimórficas.***

Certo, talvez um exemplo ajude.

```
Animal[] animals = new Animal[5];

animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {

   animals[i].eat();

   animals[i].roam();
}
```

Declara uma matriz de tipo Animal. Em outras palavras, uma matriz que conterá objetos de tipo Animal.

Mas olhe o que você pode fazer... Pode inserir QUALQUER subclasse de Animal na matriz Animal!

E aqui está a melhor parte do polimorfismo (o objetivo do exemplo): você pode percorrer a matriz e chamar um dos métodos da classe Animal, e todos os objetos se comportarão da forma correta!

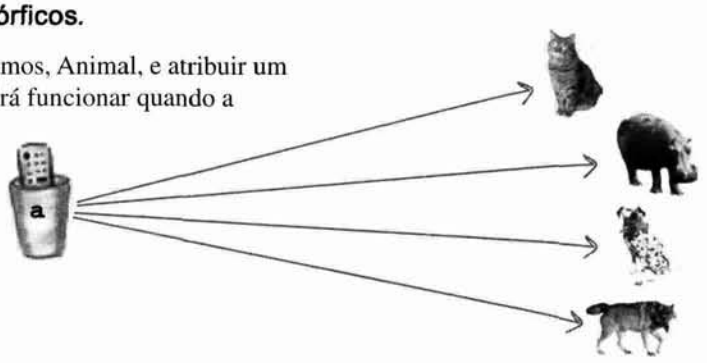
Quando 'i' for igual a 0, um objeto Dog estará no índice 0 da matriz, portanto você acionará o método eat( ) de Dog. Quando 'i' for igual a 1, você acionará o método eat( ) de Cat

O mesmo acontecerá com roam( ).

Mas espere! Há mais!

Você pode ter argumentos e tipos de retorno polimórficos.

Se você declarar a variável de referência de um supertipo, digamos, Animal, e atribuir um objeto da subclasse a ela, digamos, Dog, pense em como isso irá funcionar quando a referência for o argumento de um método...




```
class Vet {
   public void giveShot(Animal a) {
      // faz coisas horríveis com o animal na
      // outra extremidade do parâmetro 'a'
      a.makeNoise();
   }
}
```

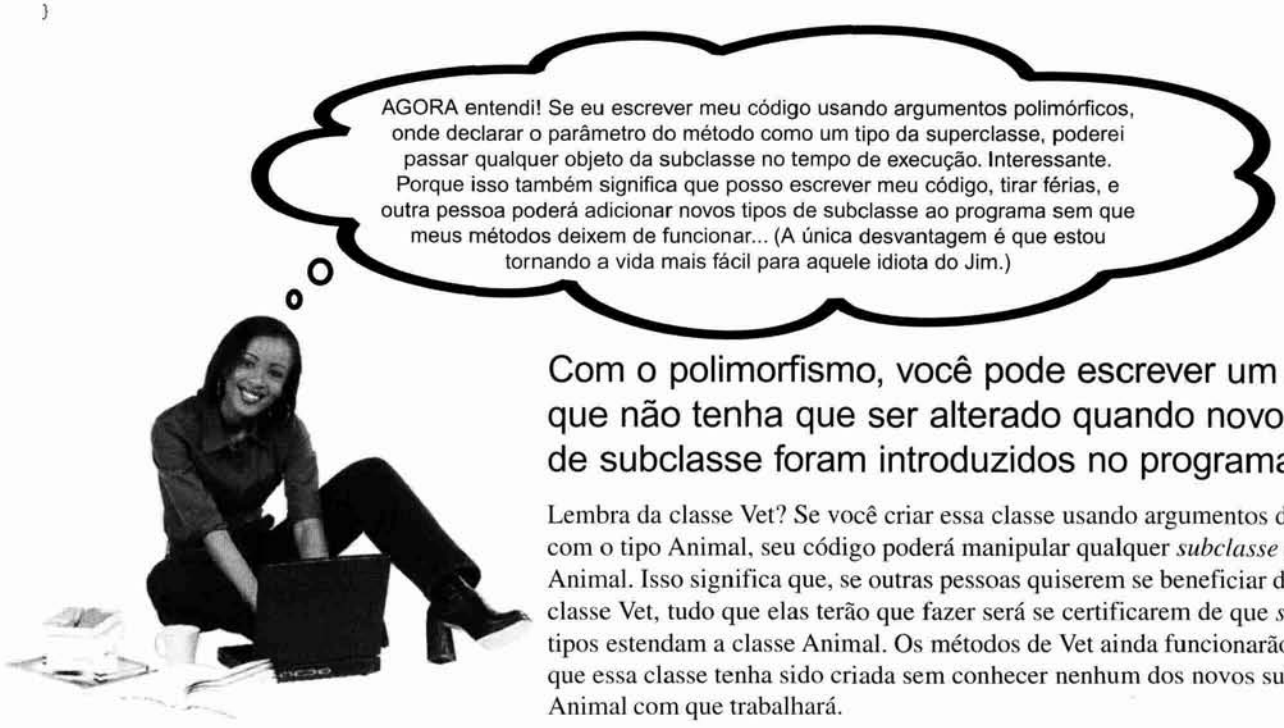
O parâmetro de Animal pode usar QUALQUER tipo de animal como argumento. Quando o veterinário tiver aplicado a injeção, o parâmetro solicitará ao animal que façaRuídos( ) e, independentemente do animal que estiver na pilha, seu método makeNoise( ) será executado.

```
class PetOwner {
   public void start() {
      Vet v = new Vet();

      Dog d = new Dog();
      Hippo h = new Hippo();

      v.giveShot(d);

      v.giveShot(h);
   }
}
```

O método giveShot( ) de Vet pode usar qualquer objeto Animal que você fornecer para ele. Contanto que o objeto que você passar como argumento seja uma subclasse de Animal, ele funcionará.

O método makeNoise( ) de Dog será executado

O método makeNoise( ) de Hippo será executado



Com o polimorfismo, você pode escrever um código que não tenha que ser alterado quando novos tipos de subclasse foram introduzidos no programa.

Lembra da classe Vet? Se você criar essa classe usando argumentos declarados com o tipo Animal, seu código poderá manipular qualquer *subclasse* de Animal. Isso significa que, se outras pessoas quiserem se beneficiar de sua classe Vet, tudo que elas terão que fazer será se certificarem de que *seus* novos tipos estendam a classe Animal. Os métodos de Vet ainda funcionarão, ainda que essa classe tenha sido criada sem conhecer nenhum dos novos subtipos de Animal com que trabalhará.

O poder do cérebro

Por que podemos ter certeza de que o polimorfismo funcionará dessa maneira? Por que é sempre seguro supor que qualquer tipo de *subclasse* terá os métodos que achamos estar chamando no tipo da *superclasse* (o tipo da referência da superclasse no qual estamos usando o operador ponto)?

Não existem Perguntas Idiotas

P: Há algum limite prático quanto à criação de níveis de subclasses? Até onde podemos ir?

R: Se você examinar o API Java, verá que a maioria das hierarquias de herança é ampla horizontalmente, mas não verticalmente. A maioria não passa de um ou dois níveis verticais, embora haja exceções (principalmente nas classes de GUI). Você perceberá que geralmente faz mais sentido manter as árvores de herança achatadas, mas não se trata de um limite rígido (bem, não de um que você tenha que obedecer).

P: Acabei de pensar em algo... Se você não tiver acesso ao código-fonte de uma classe, mas quiser alterar a maneira como um método dessa classe funciona, poderia usar subclasses para fazer isso? Para estender a classe "inválida" e sobrepor o método com seu próprio código aprimorado?

R: Sim. Esse é um recurso interessante da OO e às vezes evita a necessidade de reescrever a classe a partir do zero ou de procurar o programador que ocultou o código-fonte.

P: Podemos estender *qualquer* classe? Ou é como ocorre com os membros que quando a classe é privada não é possível herdá-los...

R: Não há classes privadas, exceto em um caso muito especial chamado classe *interna*, que ainda não examinamos. Mas *há* três coisas que podem impedir uma classe de gerar subclasses.

A primeira é o controle de acesso. Ainda que uma classe *não possa* ser marcada com private, ela *pode* não ser pública (o que ocorre quando não declaramos a classe como public). Uma classe não-pública pode gerar subclasses somente a partir de classes do mesmo pacote que ela. As classes de um pacote diferente não poderão criar subclasses (ou mesmo *usar*) da classe não-pública.

A segunda coisa que impede uma classe de gerar subclasses é o modificador cuja palavra-chave é final. Uma classe final significa que ela está no fim da linha de herança. Ninguém, em hipótese alguma, pode estender uma classe final.

O terceiro problema é que, se uma classe tiver somente construtores private (examinaremos os construtores no Capítulo 9), ela não poderá gerar subclasses.

P: Qual o interesse em criar uma classe final? Qual a vantagem de impedir que uma classe gere subclasses?

R: Normalmente, não marcamos nossas classes como finais. Mas se você precisar de segurança - a segurança de saber que os métodos sempre funcionarão da maneira que você os criou (por não poderem ser sobrepostos) -, uma classe final proporcionará isso. Várias classes do API Java são finais por essa razão. A classe String, por exemplo, é final porque, bem, imagine a confusão se alguém alterasse a maneira como as Strings se comportam!

P: Podemos fazer com que um método seja final, sem que a *classe* inteira tenha que ser?

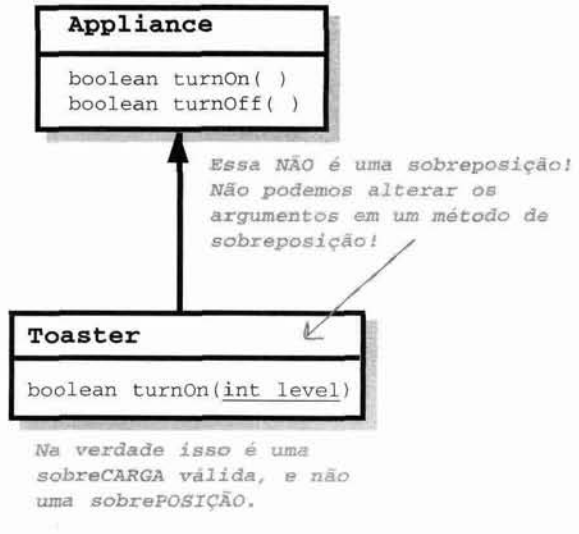
R: Se você quiser evitar que um método específico seja sobreposto, marque-o com o modificador final. Marque a *classe* inteira com final se quiser garantir que *nenhum* dos métodos dessa classe seja sobreposto.

Mantendo o contrato: regras para a sobreposição

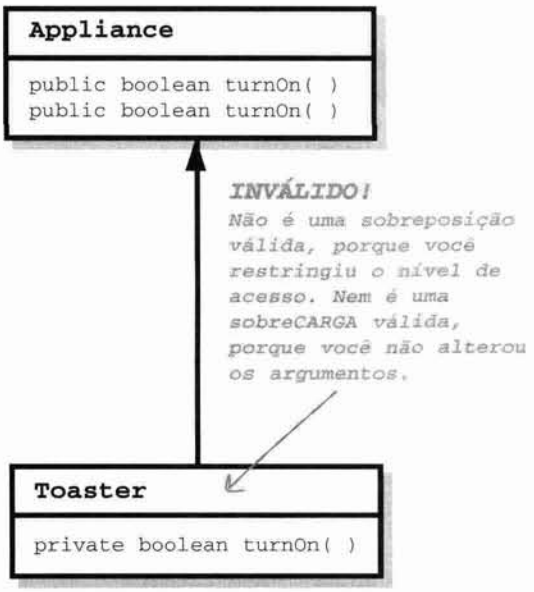
Quando você sobrepuser o método de uma superclasse, estará concordando em obedecer o contrato. O contrato que diz, por exemplo, "não uso argumentos e retorno um booleano". Em outras palavras, os argumentos e tipos de retorno de seu método de sobreposição devem parecer para o ambiente externo *exatamente* como o método sobreposto da superclasse.

Os métodos *são* o contrato.

Para o polimorfismo ser eficaz, a versão de Toaster para o método sobreposto de Appliance tem que funcionar no tempo de execução. Lembre-se de que o compilador examinará o tipo da referência para decidir se você poderá chamar um método específico nela. No caso da referência de Appliance a um objeto Toaster, o compilador só se preocupará em saber se a classe *Appliance* tem o método que você está chamando na referência. Mas no tempo de execução, a JVM não examinará o tipo da *referência* (Appliance), mas o objeto *Toaster* real na pilha. Portanto, se o compilador já tiver *aprovado* a chamada do método,



a única maneira dele funcionar será se o método de sobreposição tiver os mesmos argumentos e tipos de retorno. Caso contrário, alguém com uma referência de Appliance poderia chamar turnOn( ) como um método sem argumentos, ainda que haja uma versão de Toaster que use um int. Qual será chamada no tempo de execução? A de Appliance. Em outras palavras, ***o método turnOn(int level) de Toaster não é uma sobreposição!***



① Os argumentos devem ser iguais e os tipos de retorno devem ser compatíveis.

O contrato da superclasse define como outros códigos podem usar um método. Independentemente do argumento que a superclasse usar, a subclasse que sobrepuser o método deve usar esse mesmo argumento. E independentemente do tipo de retorno que a superclasse declarar, o método de sobreposição deve declarar o mesmo tipo ou um tipo da subclasse. Lembre-se de que um objeto da subclasse tem que poder fazer tudo que sua superclasse declarar, portanto é seguro retornar uma subclasse onde a superclasse for esperada.

② O método não pode ser menos acessível.

Isso significa que o nível de acesso deve ser o mesmo, ou mais amigável. Ou seja, você não pode, por exemplo, sobrepor um método público e torná-lo privado. Que surpresa seria para um código que chamasse o que pensava (no tempo de compilação) ser um método público, se repentinamente a JVM se opusesse porque a versão de sobreposição chamada no tempo de execução é privada!

Até agora falamos sobre dois níveis de acesso: privado e público. Os outros dois estão no capítulo sobre implantação (Lance seu código) e no Apêndice B. Também há outra regra a respeito de sobreposição relacionada à manipulação de exceções, mas esperaremos o capítulo sobre exceções (Comportamento arriscado) para abordar esse assunto.

Sobrecarregando um método

A sobrecarga de métodos nada mais é do que termos dois métodos com o mesmo nome, porém listas de argumentos diferentes. Ponto. Não há polimorfismo envolvido com métodos sobrecarregados!

A sobrecarga lhe permitirá criar diversas versões de um método, com listas de argumentos diferentes, para a conveniência dos chamadores. Por exemplo, se você tiver um método que use somente um int, o código que o chamar terá que converter, digamos, um double em um int antes de chamar seu método. Mas, se você sobrecarregou o método com outra versão que usa um double, tornou tudo mais fácil para o chamador. Veremos mais sobre isso quando examinarmos os construtores no capítulo sobre o ciclo de vida dos objetos.

Já que um método de sobrecarga não tem que obedecer ao contrato de polimorfismo definido por sua superclasse, os métodos sobrecarregados têm muito mais flexibilidade.

Um método sobrecarregado é apenas um método diferente que por acaso tem o mesmo nome. Não há nenhuma relação com a herança e o polimorfismo. Um método sobrecarregado NÃO é o mesmo que um método sobreposto.

① Os tipos de retorno podem ser diferentes.

Você poderá alterar os tipos de retorno de métodos sobrecarregados, contanto que as listas de argumentos sejam diferentes.

② Você não pode alterar SOMENTE o tipo de retorno.

Se somente o tipo de retorno for diferente, essa não será uma ***sobrecarga*** válida – o compilador presumirá que você está tentando sobre***por*** o método. E nem mesmo ***isso*** será válido, a menos que o tipo de retorno seja um subtipo do tipo de retorno declarado na

superclasse. Para sobrecarregar um método, você DEVE alterar a lista de argumentos, embora ***possa*** alterar o tipo de retorno para qualquer coisa.

③ Você *pode* variar os níveis de acesso em qualquer direção.

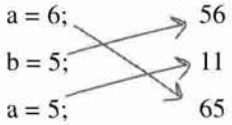
Você poderá sobrecarregar um método com outro que seja mais restritivo. Não haverá problema, já que o novo método não é obrigado a obedecer ao contrato do método sobrecarregado.

Exemplos válidos de sobrecarga de métodos:

```
public class Overloads {

   String uniqueID;

   public int addNums(int a, int b) {
      return a + b;
   }

   public double addNums(double a, double b) {
      return a + b;
   }

   public void setUniqueID(String theID) {
      // um extenso código de validação e então:
      uniqueID = theID;
   }

   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```

Exercício

Mensagens misturadas



Um programa Java curto é listado a seguir. Um bloco do programa está faltando! Seu desafio é comparar o bloco de código candidato (à esquerda) com a saída que você veria se ele fosse inserido. 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.

o programa:

```
class A {
   int ivar = 7;
   void m1() {
      System.out.print("A's m1, ");
   }
   void m2() {
      System.out.print("A's m2, ");
   }
   void m3() {
      System.out.print("A's m3, ");
   }
}

class B extends A {
   void  m1() {
      System.out.print("B's m1, ");
   }
}
```

```
class C extends B {
   void m3() {
      System.out.print("C's m3, "+(ivar + 6));
   }
}

public class Mixed2 {
   public static void main(String [] args) {
      A a = new A();
      B b = new B();
      C c = new C();
      A a2 = new C();
      // o código candidato entra aqui (três linhas)
   }
}
```

códigos candidatos:

```
b.m1();
c.m2();
a.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

saída:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```

Exercício

Seja o compilador

Qual dos pares de métodos A-B listados à direita, quando inseridos nas classes à esquerda, seriam compilados e produziriam a saída mostrada? (O método A é inserido na classe Monster e o método B é inserido na classe Vampire.)

```
public class MonsterTestDrive {
   public static void main(String [] args) {
      Monster [] ma = new Monster[3];
      ma[0] = new Vampire();
      ma[1] = new Dragon();
      ma[2] = new Monster();
      for(int x = 0; x < 3; x++) {
         ma[x].frighten(x);
      }
   }
}

class Monster {

    A

}

class Vampire extends Monster {

    B

}

class Dragon extends Monster {
   boolean frighten(int degree) {
      System.out.println("breath fire");
      return true;
   }
}
```

```
File Edit Window Help SaveYourself
% java MonsterTestDrive
a bite?
breath fire
arrrgh
```

1

A
```
boolean frighten(int d) {
   System.out.println("arrrgh");
   return true;
}
```

B
```
boolean frighten(int x) {
   System.out.println("a bite?");
   return false;
}
```

2

A
```
boolean frighten(int x) {
   System.out.println("arrrgh");
   return true;
}
```

B
```
int frighten(int f) {
   System.out.println("a bite?");
   return 1;
}
```

3

A
```
boolean frighten(int x) {
   System.out.println("arrrgh");
   return false;
}
```

B
```
boolean scare(int x) {
   System.out.println("a bite?");
   return true;
}
```

4

A
```
boolean frighten(int z) {
   System.out.println("arrrgh");
   return true;
}
```

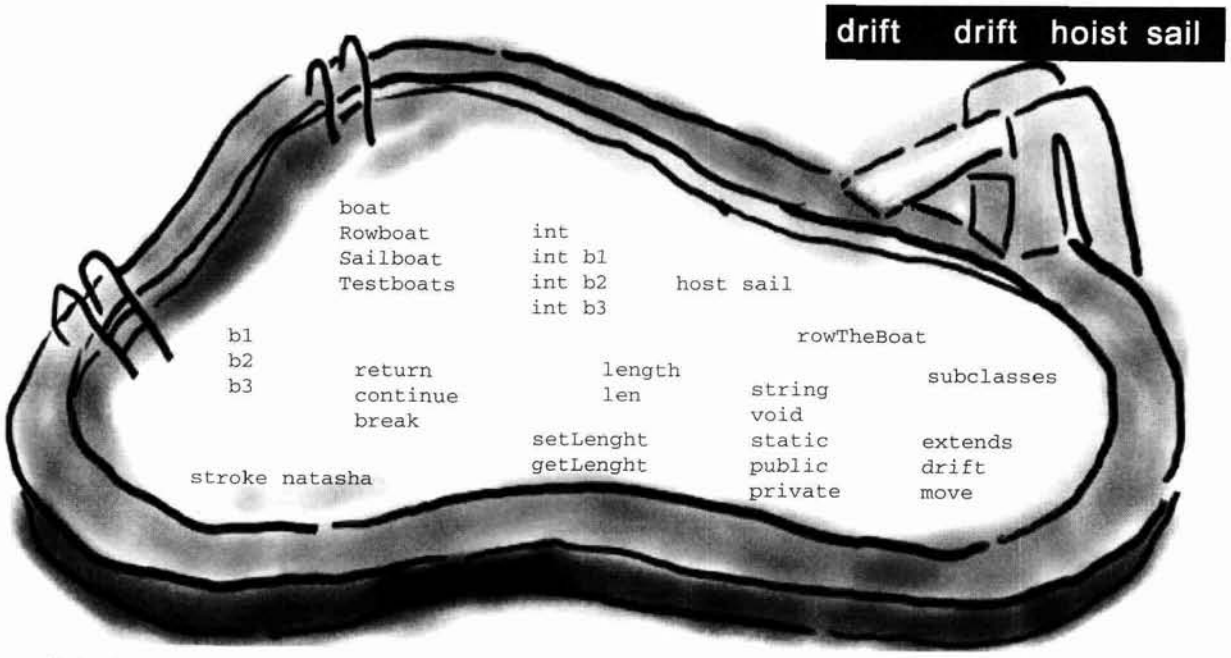
B
```
boolean frighten(byte b) {
   System.out.println("a bite?");
   return true;
}
```

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 talvez não precise empregar todos os trechos. Seu ***objetivo*** é criar um conjunto de classes que sejam compiladas e executadas juntas como um programa. Não se iluda - é mais difícil do que parece.

```
public class Rowboat _________ _________ {
   public ____________ rowTheBoat() {
      System.out.print("stroke natasha");
   }
}

public class _________ {
   private int ____________ ;
   _________ void __________ ( _______ ) {
      length = len;
   }
   public int getLength() {
      _________ __________ ;
   }
   public ____________ move() {
      System.out.print("_____________");
   }
}

public class TestBoats {
   _______ _______ ________ main('String[] args){
      __________ b1 = new Boat();
      Sailboat b2 = new ____________();
      Rowboat _________ = new Rowboat();
      b2.setLength(32);
      b1.____________();
      b3.____________();
      _________.move();
   }
}

public class ____________ _________ Boat {
   public ________ ___________() {
      System.out.print("_____________");
   }
}
```

Saída



Seja o Compilador

O conjunto 1 funcionará.

O conjunto 2 não será compilado por causa do tipo de retorno (int) de Vampire.

O método frighten( ) de Vampire (B) não é uma sobreposição OU sobrecarga válida do método frighten( ) de Monster. Alterar SOMENTE o tipo de retorno não é suficiente para a criação de uma sobrecarga válida e, já que um int não é compatível com um booleano, o método não é uma sobreposição válida. (Lembre-se de que se você alterar SOMENTE o tipo de retorno, terá que ser para um tipo que seja compatível com o tipo de retorno da versão da superclasse, então teríamos uma sobre*posição*.)

Os conjuntos 3 e 4 serão compilados, mas produzirão:

```
arrrgh

breath fire

arrrgh
```

Lembre-se de que a classe Vampire não sobre*pôs* o método frighten( ) da classe Monster. (O método frighten( ) do conjunto 4 de Vampire usa um byte e não um int.)

Soluções dos exercícios

Mensagens misturadas

códigos candidatos:

```
b.m1();
c.m2();
a.m3();

c.m1();
c.m2();
c.m3();

a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

saída:

```
A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13
```

Saída

```
drift   drift   hoist sail
```

```
public class Rowboat extends Boat {
   public void rowTheBoat() {
      System.out.print("stroke natasha");
   }
}

public class Boat {
   private int length ;
   public void setLength (int len) {
      length = len;
   }
   public int getLength() {
      return length ;
   }
   public void move() {
      System.out.print("drift   ");
   }
}
```

```
public class TestBoats {
   public static void main(String[] args){
      Boat b1 = new Boat();
      Sailboat b2 = new Sailboat();
      Rowboat b3 = new Rowboat();
      b2.setLength(32);
      b1.move();
      b3.move();
      b2.move();
   }
}

public class Sailboat extends Boat {
   public void move() {
      System.out.print("hoist sail ");
   }
}
```