Uma Viagem até Objetópolis

Ouvi dizer que haveria objetos. No Capítulo 1, colocamos todo o código no método main( ). Essa não é exatamente uma abordagem orientada a objetos. Na verdade ela *definitivamente* não é orientada a objetos. Bem, *usamos* alguns objetos, como as matrizes de strings no código da paráfrase, mas não desenvolvemos nenhum *tipo* de objeto por nossa própria conta. Portanto, agora temos que deixar esse universo procedimental para trás, sair de main( ) e começar a criar alguns objetos por nossa própria conta. Examinaremos o que torna o desenvolvimento orientado a objetos (OO, *object-oriented*) em Java tão divertido. Discutiremos a diferença entre uma *classe* e um *objeto*. Examinaremos como os objetos podem melhorar sua vida (pelo menos a parte dela dedicada à programação. Não podemos fazer muito com relação à moda). Uma vez chegando em Objetópolis, você pode não voltar mais. Envie-nos um cartão-postal.

Guerra nas Cadeiras

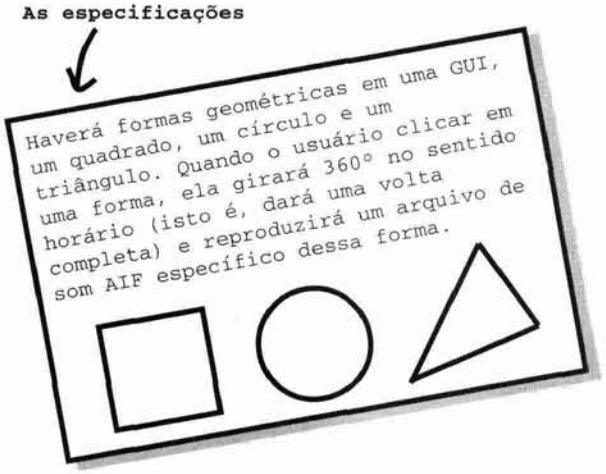
(ou como os objetos podem mudar sua vida)

Era uma vez em uma loja de softwares, dois programadores que receberam as mesmas especificações e a ordem "construam". O Gerente de Projetos Muito Chato forçou os dois codificadores a competirem, prometendo que quem acabasse primeiro ganharia uma daquelas modernas cadeiras Aeron™ que todo mundo no Vale de Santa Clara tem. Tanto Larry, o programador de procedimentos, quanto Brad, o adepto da OO, sabiam que isso seria fácil.

Sentado em sua baia, Larry pensou: "O que esse programa precisa fazer? de que ***procedimentos*** precisamos?". E respondeu "**girar e emitir som**". Portanto, ele começou a construir os procedimentos. Afinal, o que *é* um programa além de uma pilha de procedimentos?

Enquanto isso, Brad voltou ao restaurante e pensou: "Que ***itens*** existiriam nesse programa... Quem são os principais *envolvidos*?" Primeiro ele pensou nas **Formas Geométricas**. É claro que ele considerou outros objetos como o Usuário, o Som e o evento de Clicar. Mas já tinha uma biblioteca de códigos para esses itens, portanto, se dedicou à construção das Formas. Continue a ler para saber como Brad e Larry construíram seus programas e para conhecer a resposta à inquietante pergunta *"**Mas quem ganhou a Aeron?**"*

As especificações



A cadeira

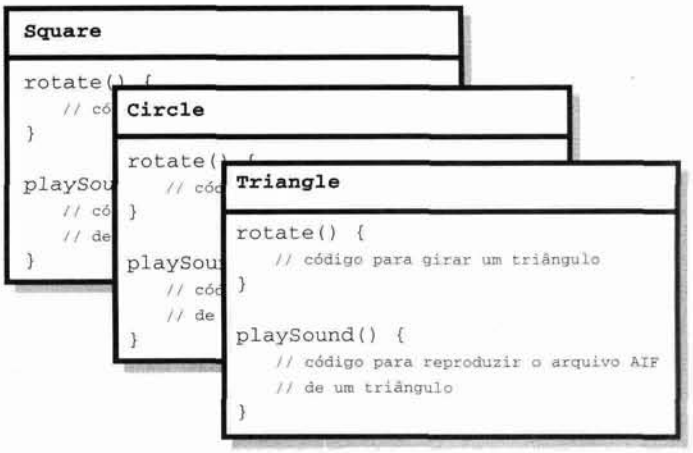
Na baia de Larry

Como já tinha feito milhares de vezes, Larry começou a escrever seus **Procedimentos Importantes**. Ele criou **rotate** e **playSound** sem demora.

```
rotate(shapeNum) {
   // faz a forma girar 360º
}

playSound(shapeNum) {
   // usa shapeNum para pesquisar
   // que som AIF reproduzir e executá-lo
}
```

No laptop de Brad dentro do restaurante

Brad criou uma ***classe*** para cada uma das três formas

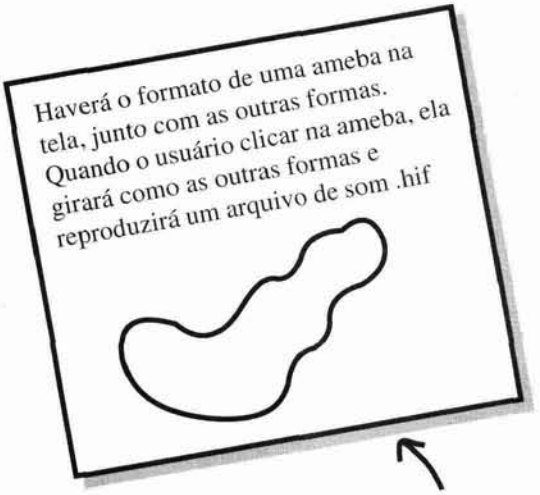


Larry achou que tinha conseguido. Podia quase sentir as rodas de aço da Aeron rolando embaixo de seu...

Mas espere! Houve uma alteração nas especificações.

"Certo, tecnicamente você venceu. Larry", disse o Gerente, "mas temos que adicionar apenas mais um pequeno item ao programa. Não será problema para programadores avançados como vocês dois."

"Se eu ganhasse uma moeda sempre que ouvisse isso", pensou Larry, sabendo que alterações nas especificações sem problemas era ilusão. *"E mesmo assim Brad parece estranhamento tranqüilo. O que estará acontecendo?"* Larry continuou mantendo sua crença de que fazer da maneira orientada a objetos, embora avançado, era lento. E que, se alguém quisesse fazê-lo mudar de idéia, teria que fazer isso à força.



O que foi adicionado às especificações

De volta à baia de Larry

O procedimento de rotação ainda funcionaria; o código usava uma tabela de pesquisa para comparar o argumento shapeNum com a figura de uma forma real. Mas ***playSound teria que mudar***. E o que diabos é um arquivo .hif?

```
playSound(shapeNum) {
   // se a forma não for uma ameba,
      // use shapeNum para pesquisar que
      // som AIF reproduzir e execute-o
   // ou
      // reproduza o som .hif da ameba
}
```

Não pareceu ser uma grande idéia, mas ***ele se sentia desconfortável em alterar código já testado.*** Entre *todas* as pessoas, *ele* sabia que, independentemente do que o gerente de projetos dissesse, ***as especificações sempre seriam alteradas***.

Usando o laptop de Brad na praia

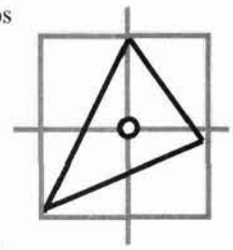
Brad sorriu, tomou um gole de sua marguerita e *criou uma nova classe*. Às vezes o que ele mais adorava na OO era não ser preciso mexer em código que já tivesse sido testado e distribuído. "Flexibilidade, extensibilidade..." ele pensou, refletindo sobre os benefícios da OO.

Amoeba

```
rotate() {
   // código para girar a ameba
}

playSound() {
   // código para reproduzir o novo
   // arquivo .hif de uma ameba
}
```

Larry acabou alguns minutos na frente de Brad.

(Ah! Tanto barulho por aquela besteira de OO.) Mas o sorriso de Larry desapareceu quando o Gerente de Projetos Muito Chato disse (com esse tom de desapontamento): "Oh, não, não é *assim* que a ameba deve girar...!"



Os dois programadores acabaram escrevendo seu código de rotação dessa forma:

1) determine o retângulo que circula a forma

2) calcule o centro desse retângulo e gire a forma ao redor desse ponto.

Mas a forma de ameba devia girar ao redor de um ponto em uma *extremidade*, como um ponteiro de relógio.

"Estou frito" pensou Larry, visualizando um Wonderbread™ chamuscado. "Porém, hmmm. Eu poderia apenas adicionar outra instrução if/else ao procedimento de rotação e, em seguida, embutir o código do ponto de rotação da ameba. Provavelmente isso não atrapalhará nada." Mas uma voz longínqua em sua mente dizia: *"É um grande erro. Você acha honestamente que as especificações não mudarão novamente?"*

O que as especificações esqueceram de mencionar adequadamente

De volta à baia de Larry

Ele achou que seria melhor adicionar os pontos de rotação como argumentos do procedimento de rotação. **Grande parte do código foi afetada.** Teste, compilação, todo o trabalho teve que ser feito novamente. O que funcionava deixou de funcionar.

```
rotate(shapeNum, xPt, yPt) {
   // se a forma não for uma ameba,
      // calcule o ponto central
      // baseado em um retângulo,
      // e, em seguida, gire
   // ou
      // use xPt e yPt como
      // o deslocamento do ponto de rotação
      // e, em seguida, gire
}
```

Usando o laptop de Brad em sua espreguiçadeira no Festival de Bluegrass de Telluride

Sem perder nada, Brad modificou o **método** de rotação, mas só na classe Amoeba. ***Ele não tocou no código funcional já testado e compilado*** das outras partes do programa. Para fornecer à classe Amoeba um ponto de rotação, ele adicionou um **atributo** que todos os objetos Amoeba teriam. Ele modificou, testou e distribuiu (com tecnologia sem fio) o programa revisto apenas durante o show de Bela Fleck.

Amoeba

```
int x point;
int y point;
rotate() {
    // código para girar a ameba
    // usando os pontos x e y
}

playSound() {
    // código para reproduzir o novo
    // arquivo .hif de uma ameba
}
```

Então, Brad, o adepto da OO ganhou a cadeira, certo?

Não tão rápido. Larry encontrou uma falha na abordagem de Brad. E, já que tinha certeza de que, se ganhasse a cadeira, também se daria bem com a Lucy da contabilidade, tinha que reverter a situação.

Larry: Você tem código duplicado! O procedimento de rotação aparece em todos os quatro itens Shape.

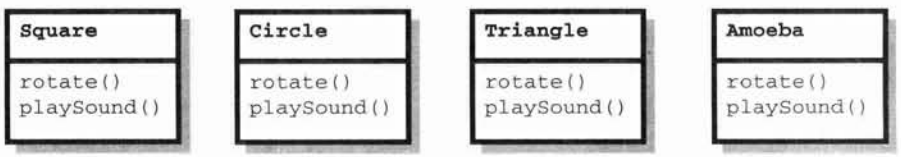
Brad: Trata-se de um **método** e não um *procedimento*. E essas são ***classes*** e não *itens*.

Larry: Não importa, É um projeto estúpido. Você tem que manter *quatro* "métodos" de rotação diferentes. Em que isso poderia ser bom?

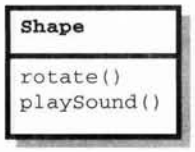
Brad: Oh, acho que você não viu o projeto final. Deixe que eu lhe mostre como a **herança** da OO funciona, Larry.

O que Larry queria (ele achava que a cadeira a impressionaria)

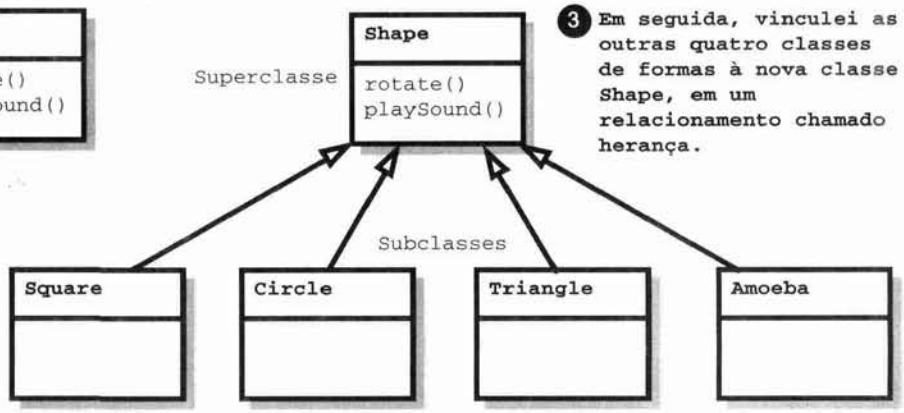
1 Procurei o que as quatro classes tinham em comum.



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



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



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, automaticamente, as subclasses terão essa mesma funcionalidade.*

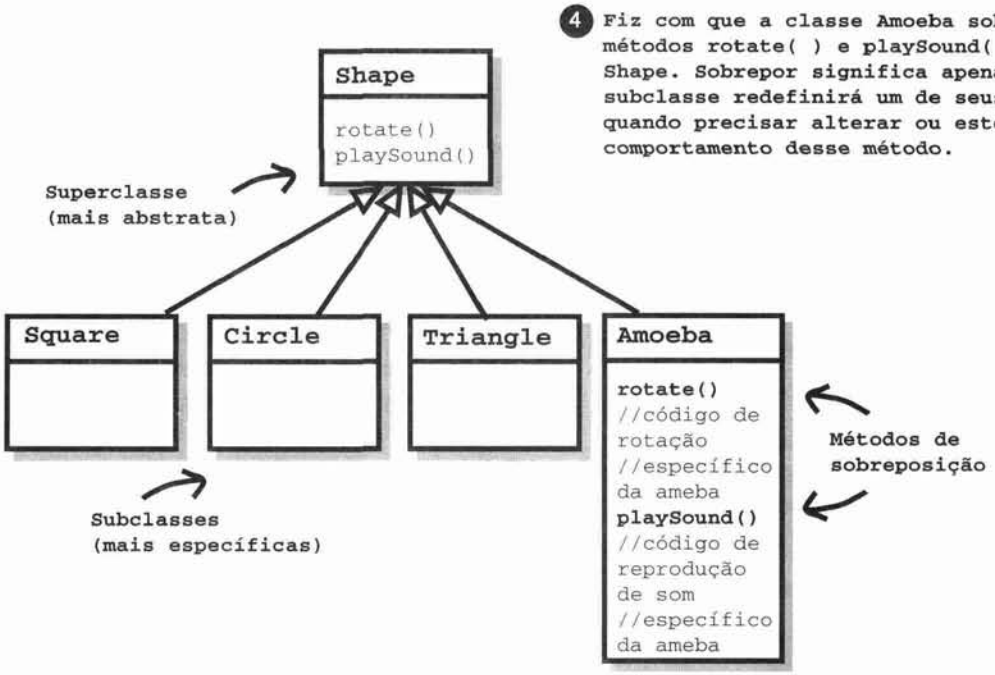
E quanto ao método rotate( ) de Amoeba?

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

Brad: Método.

Larry: Não importa. 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.



Larry: Como você "diria" a um objeto Amoeba para fazer algo? Não é preciso chamar o procedimento, desculpe – *método*, e, em seguida, lhe informar *que* item girar?

Brad: Isso é o que há de mais interessante na OO. Quando for hora, digamos, de o triângulo girar, o código do programa referenciará (chamará) o método rotate( ) *no objeto Triangle*. O resto do programa não saberá ou se importará realmente em *como* o triângulo o fará. E quando você precisar adicionar algo novo ao programa, apenas criará uma nova classe para o novo tipo de objeto, para que os **novos objetos tenham seu próprio comportamento**.

O suspense está me matando. Quem ganhou a cadeira?

Amy, que trabalha no segundo andar.

(Sem que ninguém soubesse, o Gerente de Projetos tinha dado as especificações para *três* programadores.)

O que você gosta na OO?

"Ajuda a projetar de um modo mais natural. As coisas têm uma maneira de evoluir."

- Joy, 27, engenheira de software

"Não preciso mexer em código que já testei , só para adicionar um novo recurso."

- Brad, 32, programador

"Gosto do fato de que os dados e os métodos que os utilizam ficam juntos em uma classe."

- Josh, 22, bebedor de cerveja

"A reutilização do código em outros aplicativos. Quando crio uma nova classe, posso torná-la flexível o suficiente para que seja usada em algo novo posteriormente."

- Chris, 39, gerente de projetos

"Não posso acreditar que Chris disse isso. Ele não escreve uma linha de código há 5 anos."

- Daryl, 44, trabalha para Chris

"Além da cadeira?"

-Amy, 34, programadora

O poder do cérebro

Hora de ativar alguns neurônios

Você acabou de ler uma história sobre um programador de procedimentos competindo com um programador orientado a objetos. Tivemos uma breve visão geral de alguns conceitos-chave da OO, que incluiu as classes, métodos e atributos. Passaremos o resto do capítulo examinando as classes e objetos (retornaremos à herança e à sobreposição em capítulos posteriores).

Baseado no que você viu até agora (e no que deve saber de alguma linguagem orientada a objetos com a qual já trabalhou), faça uma pausa para pensar nestas perguntas:

Quais são os itens fundamentais que você terá que considerar quando projetar uma classe Java? Que perguntas terá que fazer para você mesmo? Se pudesse projetar uma lista de conferência para usar quando estiver projetando uma classe, o que incluiria nela?

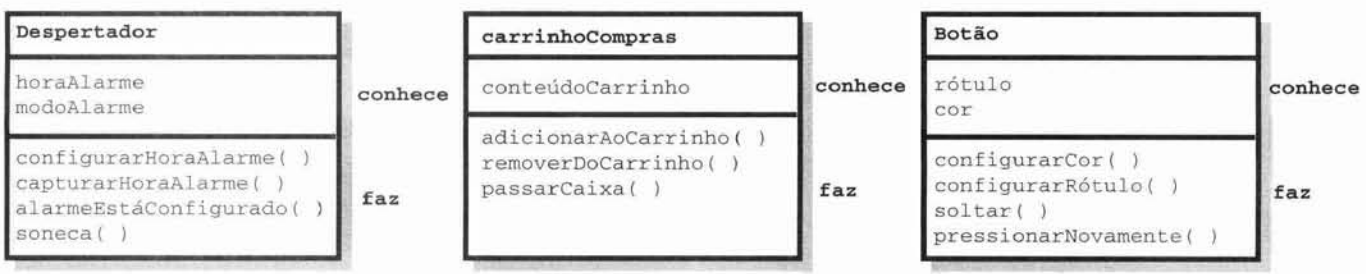
dica metacognitiva

Se você empacou em um exercício, tente falar sobre ele em voz alta. Falar (e ouvir) ativará uma parte diferente de seu cérebro. Embora isso funcione melhor quando temos outra pessoa com quem discutir, também funciona com animais de estimação. Foi assim que nosso cão aprendeu polimorfismo.

Quando você projetar uma classe, pense nos objetos que serão criados com esse tipo de classe. Considere:

- as coisas que o objeto **conhece**

- as coisas que o objeto **faz**

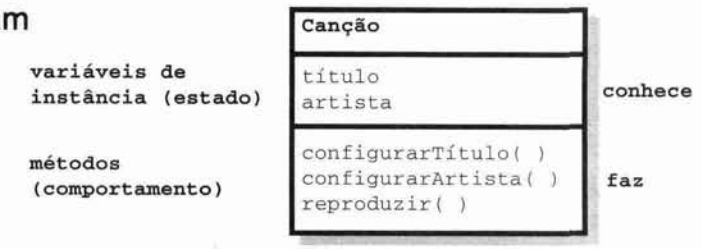


As coisas que um objeto *conhece* sobre si mesmo se chamam

- variáveis de instância

As coisas que um objeto pode *fazer* se chamam

- métodos



As coisas que um objeto ***conhece*** sobre ele são chamadas de **variáveis de instância.** Elas representam o estado de um objeto (os dados) e podem ter valores exclusivos para cada objeto desse tipo.

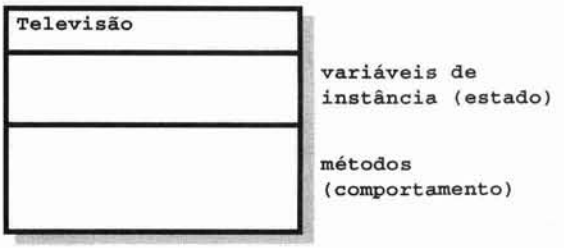
Considere **instância** como outra maneira de dizer **objeto**.

As coisas que um objeto ***faz*** são chamadas de **métodos**. Quando projetar uma classe, você pensará nos dados que um objeto terá que conhecer sobre si mesmo e também projetará os métodos que operarão sobre esses dados. É comum um objeto ter métodos que leiam ou gravem os valores das variáveis de instância. Por exemplo, os objetos Despertador têm uma variável de instância que armazena a hora de despertar e dois métodos que capturam e configuram essa hora.

Portanto, os objetos têm variáveis de instância e métodos, mas essas variáveis de instância e métodos são projetadas como parte da classe.

Aponte seu lápis

Preencha com o que um objeto televisão pode ter que saber e fazer.



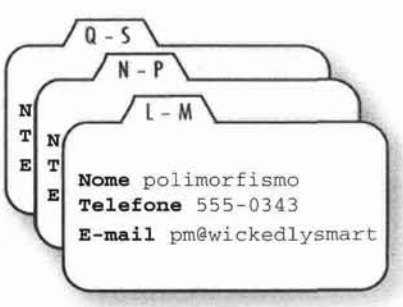
Qual é a diferença entre uma classe e um objeto?

Uma classe não é um objeto.

(Mas é usada para construí-los.)

Uma classe é o *projeto* de um objeto. Ela informa à máquina virtual *como* criar um objeto desse tipo específico. Cada objeto criado a partir dessa classe terá seus próprios valores para as variáveis de instância da classe. Por exemplo, você pode usar a classe Button para criar vários botões diferentes, e cada botão poderá ter sua própria cor, tamanho, forma, rótulo e assim por diante.

Olhe dessa forma...



Um objeto seria como um registro de sua agenda de endereços.

Uma analogia que poderíamos usar para os objetos seria um conjunto de fichas Rolodex™ não utilizadas. Todas as fichas tem os mesmos campos em branco (as variáveis de instância). Quando você preencher uma ficha, estará criando uma instância (objeto), e as entradas que criar nessa ficha representarão seu estado.

Os métodos da classe são as coisas que você pode fazer com uma ficha específica; obterNome( ), alterarNome( ), configurarNome( ), todos poderiam ser métodos da classe Rolodex.

Portanto, todas as fichas *fazem* as mesmas coisas (obterNome( ), alterarNome( ), etc.), mas cada uma *conhece* coisas exclusivas sobre si mesma.

Criando seu primeiro objeto

Mas o que é necessário para a criação e uso de um objeto? Você precisa de *duas* classes. Uma para o tipo de objeto que deseja usar (Dog, AlarmClock, Television, etc.) e outra para *testar* sua nova classe. É na classe *testadora* que você inserirá o método principal e nesse método main( ) criará e acessará objetos de seu novo tipo de classe. A classe testadora terá apenas uma tarefa: *testar* os método e variáveis de seu novo tipo de classe de objetos.

Desse ponto do livro em diante, você verá duas classes em muitos de nossos exemplos. Uma será a classe *real* – a classe cujos objetos realmente queremos usar, e a outra será a classe *testadora*, que chamaremos de *<qualquerQueSejaNomeSuaClasse>***TestDrive**. Por exemplo, se criarmos uma classe **Bungee**, também precisaremos de uma classe **BungeeTestDrive**. Só a classe *<nomeAlgumaClasse>***TestDrive** terá um método main( ), e sua única finalidade será criar objetos de seu novo tipo (a classe que não for a de teste) para em seguida usar o operador ponto (.) para acessar os métodos e variáveis dos novos objetos. Faremos tudo isso muito claramente nos exemplos a seguir.

O operador ponto (.)

O operador ponto (.) lhe dará acesso ao estado e comportamento (variáveis de instância e métodos) de um objeto.

```
//cria um novo objeto
Dog d = new Dog( );

// solicita que ele lata usando o
// operador ponto na
// variável d para chamar bark( )
d.bark( );

// configure seu tamanho usando
// o operador ponto
d.size=40;
```

1 Crie sua classe

```
class Dog {

   int size;
   String breed;          <-- variáveis de instância
   String name;

   void bark() {          <-- um método
      System.out.println("Ruff! Ruff!");
   }
}
```

2 Crie uma classe testadora (TestDrive)

```
class DogTestDrive {
   public static void main (String[] args) {
      // o código de teste de Dog entra aqui
   }
}
```

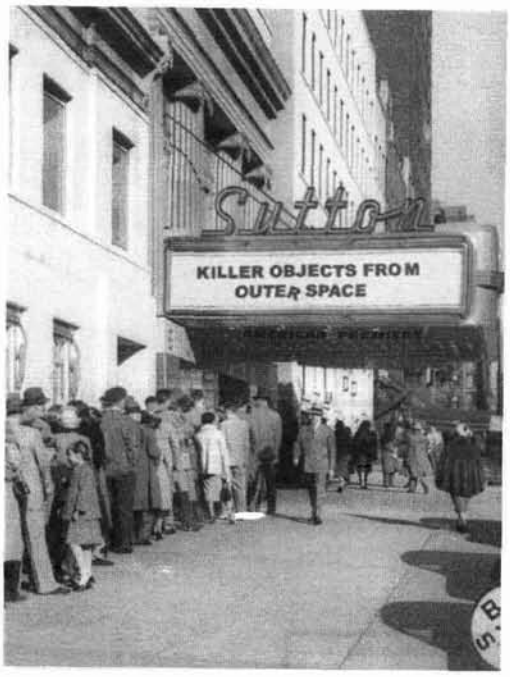
Apenas um método main (forneceremos um código para ele na próxima etapa)

3 Em sua classe testadora, crie um objeto e acesse suas variáveis e métodos

```
class DogTestDrive {
   public static void main (String[] args) {
      Dog d = new Dog();          Crie um objeto Dog

      d.size = 40;                Use o operador ponto (.) para configurar o
                                  tamanho do objeto Dog
Operador
ponto
      d.bark();                   <-- E para chamar seu método bark( )
   }
}
```

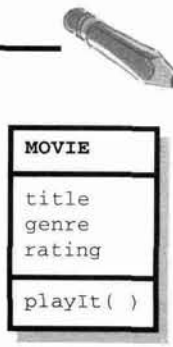
Se você já tem algum código OO pronto, sabe que não estamos usando encapsulamento. Abordaremos esse assunto no Capítulo 4.

Criando e testando objetos Movie




```
class Movie {
   String title;
   String genre;
   int rating;

   void playIt() {
      System.out.println("Playing the movie");
   }
}

public class MovieTestDrive {
   public static void main(String[] args) {
      Movie one = new Movie();
      one.title = "Gone with the Stock";
      one.genre = "Tragic";
      one.rating = -2;
      Movie two = new Movie();
      two.title = "Lost in Cubicle Space";
      two.genre = "Comedy";
      two.rating = 5;
      two.playIt();
      Movie three = new Movie();
      three.title = "Byte Club";
      three.genre = "Tragic but ultimately uplifting";
      three.rating = 127;
   }
}
```

Aponte seu lápis



A classe MovieTestDrive cria objetos (instâncias) da classe Movie e usa o operador ponto (.) para configurar as variáveis de instância com um valor específico. Ela também referencia (chama) um método em um dos objetos. Preencha a figura à direita com os valores que os três objetos apresentam no fim de main( ).

Objeto 1

title

genre

rating

Objeto 2

title

genre

rating

Objeto 3

title

genre

rating

Rápido! Saia de main!

Se você estiver em main( ), não estará realmente em Objetópolis. É adequado um programa de teste ser executado dentro do método main, mas, em um aplicativo OO real, você precisará de objetos que se comuniquem com outros objetos e não de um método main () estático criando e testando objetos.

As duas finalidades de main:

- **testar** sua classe real
- **acionar/iniciar** seu **aplicativo** Java

Um aplicativo Java real nada mais é do que objetos se comunicando com outros objetos. Nesse caso, *comunicar-se* significa os objetos chamando os métodos uns dos outros. Na página anterior, e no Capítulo 4,

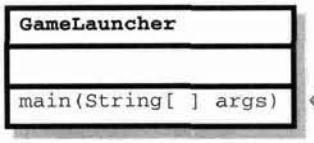
examinamos o uso de um método main( ) em uma classe TestDrive separada para criar e testar os métodos e variáveis de outra classe. No Capítulo 6 examinaremos o uso de uma classe com um método main( ) para iniciar um aplicativo Java *real* (criando objetos e, em seguida, deixando-os livres para interagir com outros objetos, etc.)

No entanto, como uma prévia de como um aplicativo Java real pode se comportar, aqui está um pequeno exemplo. Já que ainda estamos nos estágios iniciais do aprendizado de Java, trabalharemos com um pequeno kit de ferramentas , portanto, você achará esse programa um pouco complicado e ineficiente. Talvez pense no que poderia fazer para aperfeiçoá-lo, e em capítulos posteriores é exatamente isso que faremos. Não se preocupe se parte do código for confusa; o ponto-chave desse exemplo é que os objetos se comunicam entre si.

O jogo de adivinhação

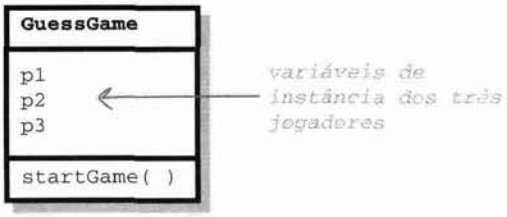
Resumo:

O jogo de adivinhação envolve um objeto 'game' e três objetos 'player'. O jogo gera um número aleatório entre 0 e 9 e os três objetos player tentam adivinhá-lo. (Não dissemos que seria um jogo *divertido.*)



Classes:

GuessGame.class Player.class GameLauncher.class

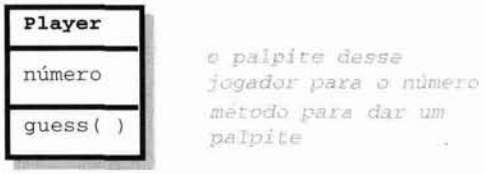


A lógica:

1) É na classe GameLauncher que o aplicativo é iniciado; ela tem o método main( ).

2) No método main( ), um objeto GuessGame é criado e seu método *startGame( ) é chamado.*

3) É no método startGame( ) do objeto GuessGame que o jogo inteiro se desenrola. Ele cria três jogadores e, em seguida, "pensa" em um número aleatório (aquele que os jogadores têm que adivinhar). Depois solicita a cada jogador que adivinhe, verifica o resultado e exibe informações sobre o(s) jogador(es) vencedor(es) ou pede que adivinhem novamente.




```
public class GuessGame {
   Player p1;
   Player p2;
   Player p3;

   public void startGame() {
      p1 = new Player();
      p2 = new Player();
      p3 = new Player();

      int guessp1 = 0;
      int guessp2 = 0;
      int guessp3 = 0;

      boolean p1isRight = false;
      boolean p2isRight = false;
      boolean p3isRight = false;

      int targetNumber = (int) (Math.random() * 10);
      System.out.println("Estou pensando em um número entre 0 e 9...");

      while(true) {
         System.out.println("O número a adivinhar é " + targetNumber);

         p1.guess();
         p2.guess();
         p3.guess();
```

GuessGame tem três variáveis de instância para os três objetos Player

cria três objetos Player e atribui a eles as três variáveis de instância Player

declara três variáveis para armazenar os três palpites que os jogadores fornecerão

declara três variáveis para armazenar um valor verdadeiro ou falso baseado na resposta do jogador

cria um número "alvo" que os jogadores terão que adivinhar

chama o método guess( ) de cada jogador

```
            guessp1 = p1.number;
            System.out.println("O jogador um forneceu o palpite " + guessp1);
            guessp2 = p2.number;
            System.out.println("O jogador dois forneceu o palpite " + guessp2);
            guessp3 = p3.number;
            System.out.println("O jogador três forneceu o palpite " + guessp3);

            if (guessp1 == targetNumber) {
               p1isRight = true;
            }
            if (guessp2 == targetNumber) {
               p2isRight = true;
            }
            if (guessp3 == targetNumber) {
               p3isRight = true;
            }

            if (p1isRight || p2isRight || p3isRight) {

               System.out.println("Temos um vencedor!");
               System.out.println("O jogador um acertou? " + p1isRight);
               System.out.println("O jogador dois acertou? " + p2isRight);
               System.out.println("O jogador três acertou? " + p3isRight);
               System.out.println("Fim do jogo.");
               break; // fim do jogo, portanto saia do loop
            } else {
               // devemos continuar porque ninguém acertou!
               System.out.println("Os jogadores terão que tentar novamente.");
            } // fim de if/else
         } // fim do loop
      } // fim do método
   } // fim da classe
```

obtem o palpite de cada jogador (o resultado da execução de seu método guess( )) acessando a variável numérica de cada um

verifica o palpite de cada jogador para ver se é igual ao número-alvo. Se um jogador acertar, sua variável será configurada com true (lembre-se de que configuramos false como o padrão)

se o jogador um OU o jogador dois OU o jogador três acertar... (O operador || significa OU)

caso contrário, fique no loop e peça aos jogadores outro palpite.

Executando o jogo de adivinhação

```
public class Player {
   int number = 0;  // onde entra o palpite

   public void guess() {
      number = (int) (Math.random() * 10);
      System.out.println("Estou pensando em " + number);
   }
}

public class GameLauncher {
   public static void main (String[] args) {
      GuessGame game = new GuessGame();
      game.startGame();
   }
}
```

Saída (será diferente a cada vez que você executar)

```
File Edit Window Help Explode
%java GameLauncher
Estou pensando em um número entre 0 e 9...
O número a adivinhar é 7
Estou pensando em 1
Estou pensando em 9
Estou pensando em 9
O jogador um forneceu o palpite 1
O jogador dois forneceu o palpite 9
O jogador três forneceu o palpite 9
Os jogadores terão que tentar novamente.
O número a adivinhar é 7
Estou pensando em 3
Estou pensando em 0
Estou pensando em 9
O jogador um forneceu o palpite 3
O jogador dois forneceu o palpite 0
O jogador três forneceu o palpite 9
Os jogadores terão que tentar novamente.
O número a adivinhar é 7
Estou pensando em 7
Estou pensando em 5
Estou pensando em 0
O jogador um forneceu o palpite 7
O jogador dois forneceu o palpite 5
O jogador três forneceu o palpite 0
Temos um vencedor!
O jogador um acertou? verdadeiro
O jogador dois acertou? falso
O jogador três acertou? falso
Fim do jogo.
```

O Java coleta o lixo

Sempre que um objeto é criado em Java, ele vai para uma área da memória conhecida como **Heap**. Todos os objetos - independentemente de quando, onde ou como sejam criados - residem no heap. Mas não se trata simplesmente de qualquer memória heap como as antigas; na verdade a memória heap Java se chama **Pilha de Lixo Coletável**. Quando você criar um objeto, a Java alocará espaço na memória heap de acordo com quanto esse objeto específico vai precisar. Um objeto com, digamos, 15 variáveis de instância, provavelmente precisará de mais espaço do que um objeto com apenas duas variáveis de instância. Mas o que acontecerá quando você precisar reclamar esse espaço? Como você tirará um objeto do heap quando não precisar mais dele? A Java gerenciará essa memória para você! Quando a JVM 'perceber' que um objeto pode nunca mais ser usado, ele se tornará *qualificado para a coleta de lixo*. E se você estiver ficando com pouco espaço na memória, o Coletor de Lixo será executado, eliminará os objetos inalcançáveis e liberará espaço, para que esse possa ser reutilizado. Em capítulos posteriores você aprenderá mais sobre como isso funciona.

Não existem Perguntas Idiotas

P: E se eu precisar de variáveis e métodos globais? Como conseguirei isso, se tudo precisa estar em uma classe?

R: Não há um conceito de variáveis e métodos 'globais' em um programa Java orientado a objetos. Na prática, entretanto, haverá situações em que você pode querer que um método (ou uma constante) esteja disponível para qualquer código que for executado em qualquer parte de seu programa. Considere o método random( ) do aplicativo da paráfrase; é um método que tem que poder ser chamado de qualquer local. E quanto a uma constante como *pi*? Você aprenderá no Capítulo 10 que marcar um método como public e static faz com que ele se comporte de maneira semelhante a um método 'global'. Qualquer código, de qualquer classe de seu aplicativo, poderá acessar um método estático público. E se você marcar uma variável como public, static e final – terá essencialmente criado uma *constante* disponível globalmente.

P: Mas como poderia chamar isso de orientado a objetos se ainda é possível tornar globais as funções e dados?

R: Em primeiro lugar, tudo em Java reside em uma classe. Portanto, a constante *pi* e o método random( ), embora públicos e estáticos, são definidos dentro da classe Math. E você deve se lembrar que esses itens estáticos (semelhantes aos globais) são a exceção em vez da regra em Java. Eles representam um caso muito especial, em que não se tem várias instâncias/objetos.

P: O que é um programa Java? O que é realmente *distribuído*?

R: Um programa Java consiste em uma pilha de classes (ou, pelo menos, *uma* classe). Em um aplicativo Java, *uma* das classes deve ter um método main, usado para iniciar o programa. Portanto, como programador, você escreve uma ou mais classes. E essas classes são que você distribuirá. Se o usuário final não tiver uma JVM, você também precisará incluir nas classes de seu aplicativo, para que eles possam executar seu programa. Há vários programas de instalação que permitem incluir nas classes diversos JVMs (digamos, para diferentes plataformas) e inserir tudo em um CD-ROM. Assim o usuário final poderá instalar a versão correta da JVM (supondo que eles já não a tenham em suas máquinas).

P: E se eu tiver uma centena de classes? Ou mil? Não seria complicado distribuir todos esses arquivos? Posso empacotá-los em um *Kit Aplicativo*?

R: Sim, seria complicado distribuir uma grande quantidade de arquivos para seus usuários finais, mas você não precisa fazer isso. você pode inserir todos os arquivos de seu aplicativo em um Java Archive – *um arquivo .jar* – que usa o formato pkzip. No arquivo jar, você poderá incluir um arquivo de texto simples formatado como algo chamado *mainfesto*, que definirá que classe desse arquivo contém o método main() que deve ser executado.

DISCRIMINAÇÃO DOS PONTOS

- A programação orientada a objetos lhe permitirá estender um programa sem ser preciso mexer em código funcional já testado.

- Todo código Java é definido em uma **classe**.

- Uma classe descreve como criar um objeto desse tipo de classe. **Uma classe é como um projeto.**

- Um objeto pode cuidar de si próprio; você não precisa conhecer ou se importar com a *maneira* de ele agir.

- Um objeto **conhece** coisas e **faz** coisas.

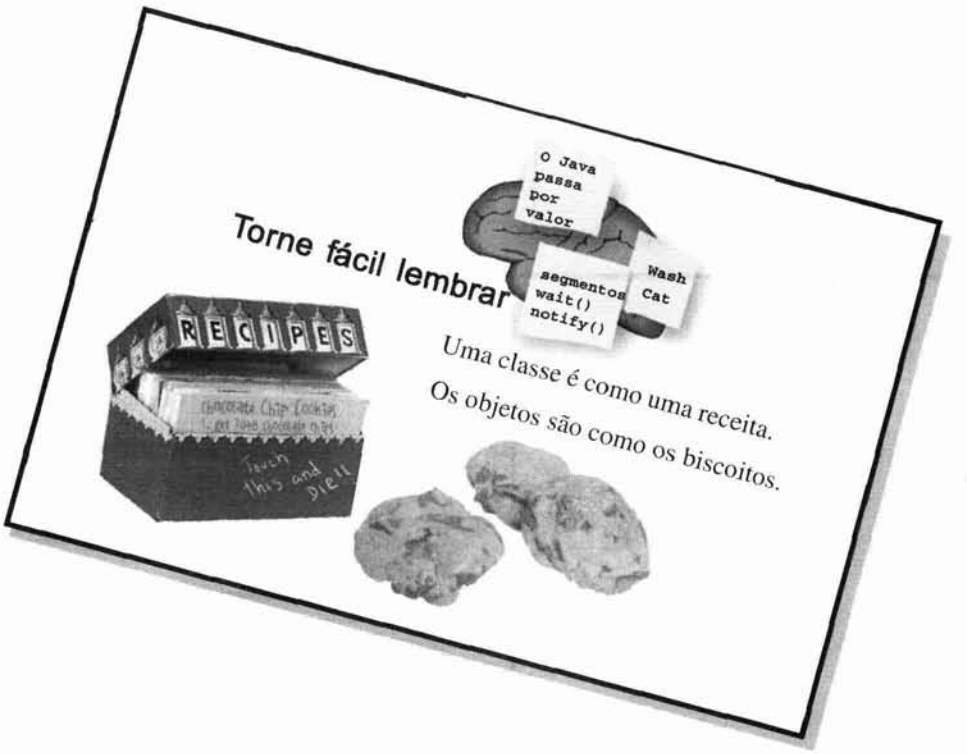
- As coisas que um objeto conhece sobre si próprio se chamam **variáveis de instância**. Elas representam o *estado* de um objeto.

- As coisas que um objeto faz são chamadas de **métodos**. Eles representam o *comportamento* de um objeto.

- Quando você criar uma classe, talvez queira criar uma classe de teste separada, que usará para gerar objetos de seu novo tipo de classe.

- Uma classe pode **herdar** variáveis de instância e métodos de uma **superclasse** mais abstrata.

- No tempo de execução, um programa Java nada mais é do que objetos 'comunicando-se' com outros objetos.



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?

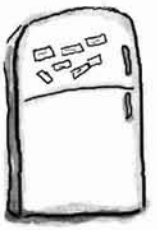
A

```
class TapeDeck {

   boolean canRecord = false;

   void playTape() {
      System.out.println("tape playing");
   }

   void recordTape() {
      System.out.println("tape recording");
   }
}

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

      t.canRecord = true;
      t.playTape();

      if (t.canRecord == true) {
         t.recordTape();
      }
   }
}
```

B

```
class DVDPlayer {

   boolean canRecord = false;

   void recordDVD() {
      System.out.println("DVD recording");
   }
}

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

      DVDPlayer d = new DVDPlayer();
      d.canRecord = true;
      d.playDVD();

      if (d.canRecord == true) {
         d.recordDVD();
      }
   }
}
```

Exercício



Ímãs com código

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

```
File Edit Window Help Dance
%java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```

```
d.playSnare();
```

```
DrumKit d = new DrumKit();
```

```
boolean topHat = true;
boolean snare = true;
```

```
void playSnare() {
   System.out.println("bang bang ba-
bang");
}
```

```
public static void main(String [] args)
{
```

```
if (d.snare == true) {
   d.playSnare();
}
```

```
d.snare = false;
```

```
class DrumKitTestDrive {
```

```
d.playTopHat();
```

```
class DrumKit
```

```
void playTopHat () {
   System.out.println("ding ding da-ding");
}
```

Quem sou eu?

Um grupo de componentes Java, vestido a rigor, está participado do jogo, "Quem sou eu?" em uma festa. Eles lhe darão uma pista e você tentará adivinhar quem são, baseado no que disserem. Se por acaso disserem algo que possa ser verdadeiro para mais de um deles, selecione todos aos quais a frase possa ser aplicada. Preencha as linhas em branco próximas à frase com os nomes de um ou mais candidatos. A primeira é por nossa conta.

Candidatos desta noite: Classe Método Objeto Variável de instância

Sou compilado em um arquivo .java. classe _____

Os valores de minha variável de instância podem ser diferentes dos de meu colega. _____

Comporto-me como um modelo. _____

Gosto de fazer coisas. _____

Posso ter muitos métodos. _____

Represento o 'estado'. _____

Odeio comportamentos. _____

Estou situado nos objetos. _____

Vivo no heap. _____

Costumo criar instâncias de objeto. _____

Meu estado pode se alterar. _____

Declaro métodos. _____

Posso mudar no tempo de execução. _____

Quebra-cabeças na Piscina

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

Saída

```
File Edit Window Help Implode
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

Pergunta adicional!

Se a última linha da saída fosse **24** em vez de **10**, como você concluiria o quebra-cabeça?

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    _____________________________
    int x = 0;
    while ( _____________ ) {
      e1.hello();
      _____________________________
      if ( _____________ ) {
        e2.count = e2.count + 1;
      }
      if ( _____________ ) {
        e2.count = e2.count + e1.count;
      }
        x = x + 1;
      }
      System.out.println(e2.count);
    }
  }
```

```
  class _____________ {
    int _____________ = 0;
    void _____________ {
      System.out.println("helloooo... ");
  }
}
```

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

```
x < 4
x < 5
x > 0
x > 1
x
y
e2
count
e1 = e1 + 1;
e1 = count + 1;
e1.count = count + 1;
e1.count = e1.count + 1;
Echo
Tester
echo( )
count( )
hello( )
x == 3
x == 4
e2 = e1;
Echo e2;
Echo e2 = e1;
Echo e2 = new Echo( );
```

Soluções dos Exercícios

Imãs com código:

```
class DrumKit {
   boolean topHat = true;
   boolean snare = true;
   void playTopHat() {
      System.out.println("ding ding da-ding");
   }
   void playSnare() {
      System.out.println("bang bang ba-bang");
   }
}

class DrumKitTestDrive {
   public static void main(String [] args) {
      DrumKit d = new DrumKit();
      d.playSnare();
      d.snare = false;
      d.playTopHat();
      if (d.snare == true) {
         d.playSnare();
      }
   }
}
```

Seja o compilador:

A

```
class TapeDeck {
   boolean canRecord = false;
   void playTape() {
      System.out.println("tape playing");
   }
   void recordTape() {
      System.out.println("tape recording");
   }
}

class TapeDeckTestDrive {
   public static void main(String [] args) {
      TapeDeck t = new TapeDeck( );
      t.canRecord = true;
      t.playTape();
      if (t.canRecord == true) {
         t.recordTape();
      }
   }
}
```

Temos o modelo, agora temos que criar um objeto!

B

```
class DVDPlayer {
   boolean canRecord = false;
   void recordDVD() {
      System.out.println("DVD recording");
   }
   void playDVD ( ) {
      System.out.println("DVD playing");
   }
}

class DVDPlayerTestDrive {
   public static void main(String [] args) {
      DVDPlayer d = new DVDPlayer();
      d.canRecord = true;
      d.playDVD();
      if (d.canRecord == true) {
         d.recordDVD();
      }
   }
}
```

A linha: d.playDVD( ); não seria compilada sem um método!

Soluções dos quebra-cabeças

Quem sou eu?

Sou compilado em um arquivo .java. **classe**

Os valores de minha variável de instância podem ser diferentes dos de meu colega. **objeto**

Comporto-me como um modelo. **classe**

Gosto de fazer coisas. **objeto, método**

Posso ter muitos métodos. **classe, objeto**

Represento o 'estado'. **variável de instância**

Odeio comportamentos. **objeto, classe**

Estou situado nos objetos. **método, variável de instância**

Vivo no heap. **objeto**

Costumo criar instâncias de objeto. **classe**

Meu estado pode se alterar. **objeto, variável de instância**

Declaro métodos. **classe**

Posso mudar no tempo de execução. **objeto, variável de instância**

Nota: diz-se que tanto as classes quanto os objetos possuem estado e comportamento. Eles são definidos na classe, mas também são considerados parte do objeto. Por enquanto, não vamos nos preocupar com a questão técnica de onde eles residem.

Quebra-cabeça da piscina

```
public class EchoTestDrive {
   public static void main(String [] args) {
      Echo e1 = new Echo();
      Echo e2 = new Echo( );  // a resposta correta
                  - ou -
      Echo e2 = e1;   // a da pergunta adicional!
      int x = 0;
      while (x < 4) {
         e1.hello();
         e1.count = e1.count + 1;
         if (x == 3) {
            e2.count = e2.count + 1;
         }
         if (x > 0) {
            e2.count = e2.count + e1.count;
         }
         x = x + 1;
      }
      System.out.println(e2.count);
   }
}


class Echo {
   int count = 0;
   void hello( ) {
      System.out.println("helloooo... ");
   }
}
```