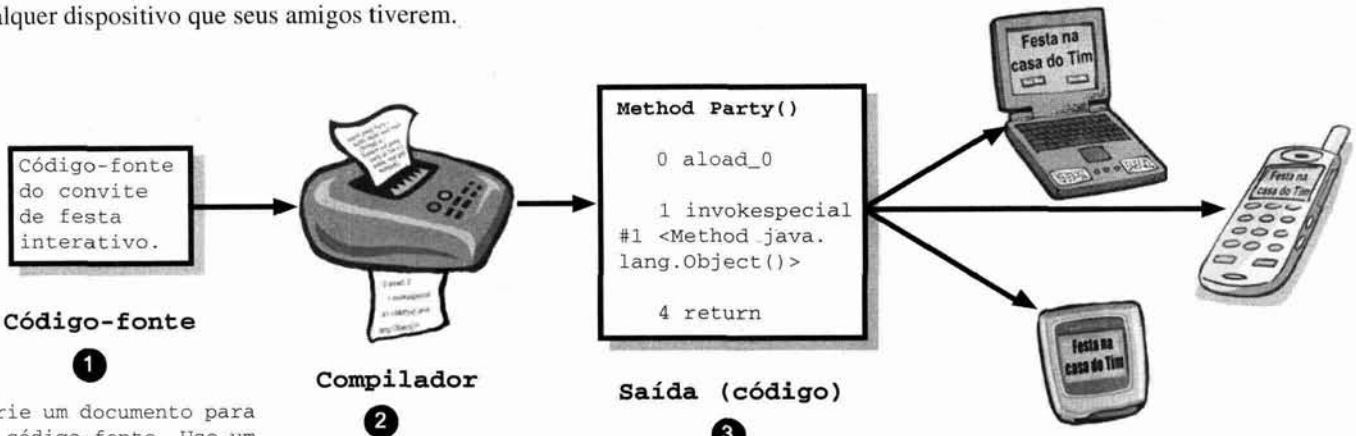
Aprofundando-se

O Java o levará a novas fronteiras. No humilde lançamento para o público como a (suposta) versão 1.02, o Java seduziu os programadores com sua sintaxe amigável, recursos orientados a objetos, gerenciamento de memória e, o melhor de tudo — a promessa de portabilidade. A possibilidade de **escrever uma vez/ executar em qualquer local** exerce uma atração muito forte. Seguidores devotados surgiram, enquanto os programadores combatiam os erros, limitações e, ah sim, o fato de ela ser muito lenta. Mas isso foi há muito tempo. Se você for iniciante em Java, **tem sorte**. Alguns de nós tiveram que passar por algo como andar quase dez quilômetros na neve e subir montanhas pelos dois lados (descalços), para fazer até mesmo o applet mais simples funcionar. Mas *você* pode manipular o **mais fácil, rápido e muito mais poderoso** Java atual.

Como o Java funciona

O objetivo é escrever um aplicativo (neste exemplo, um convite de festa interativo) e fazê-lo funcionar em qualquer dispositivo que seus amigos tiverem.



Código-fonte

1

Crie um documento para o código-fonte. Use um protocolo estabelecido (nesse caso, a linguagem Java).

Compilador

2

Execute seu documento em um compilador de código-fonte. O compilador procurará erros e não deixará você compilar até ter certeza de que tudo será executado corretamente.

Saída (código)

3

O compilador criará um novo documento, codificado em ***bytecode*** Java. Qualquer dispositivo capaz de executar Java conseguirá interpretar/converter esse arquivo em algo que possa processar. O bytecode compilado é dependente da plataforma.

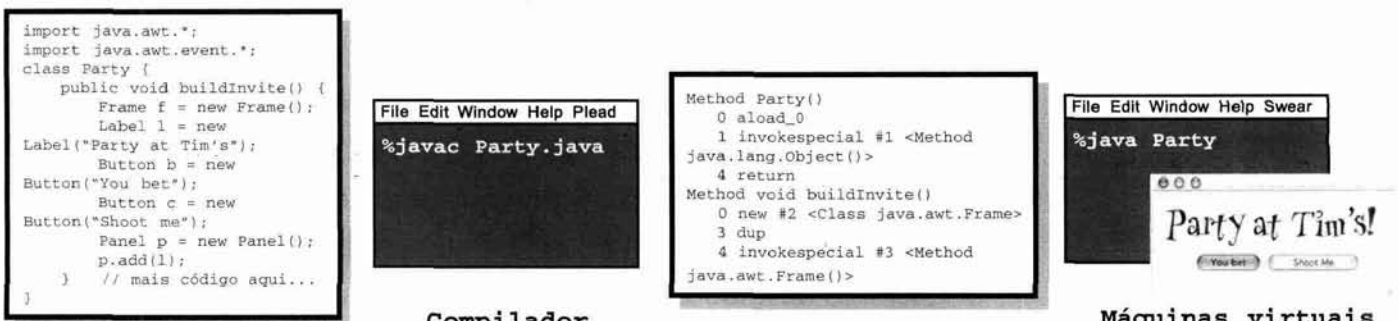
Máquinas virtuais

4

Seus amigos não têm uma máquina Java física, mas todos têm uma máquina Java ***virtual*** (implementada em software) sendo executada dentro de seus aparelhos eletrônicos. A máquina virtual lerá e ***executará*** o bytecode.

O que você fará em Java

Você criará um arquivo de código-fonte, compilará usando o compilador javac e, em seguida, executará o bytecode compilado em uma máquina virtual Java.



Código-fonte

1

Digite seu código-fonte. Salve como: ***Party.java***

Compilador

2

Compile o arquivo ***Party.java*** executando o javac (o aplicativo do compilador). Se não houver erros, você terá um segundo documento chamado ***Party.class*** O arquivo Party.class gerado pelo compilador é composto de ***bytecodes***.

Saída (código)

3

Código compilado: ***Party.class***

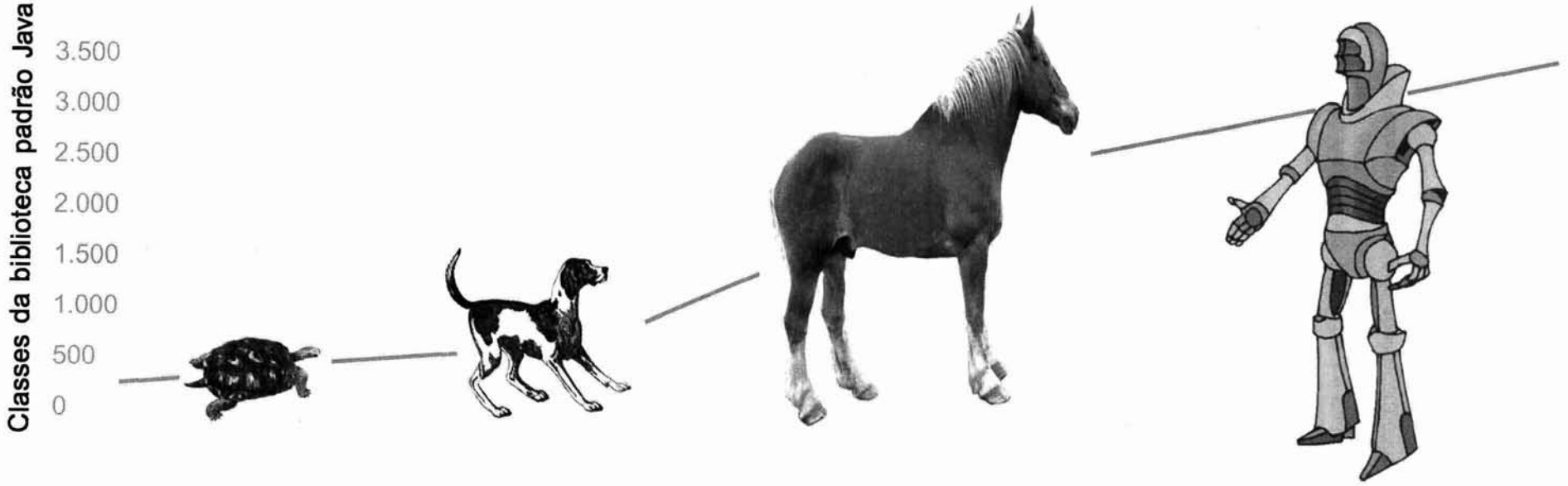
Máquinas virtuais

4

Execute o programa iniciando a Java Virtual Machine (JVM) com o arquivo ***Party.class***. A JVM converterá o ***bytecode*** em algo que a plataforma subjacente entenda e executará seu programa.

(Nota: não pretendemos que essas instruções sejam um tutorial... Você vai escrever código real em breve, mas, por enquanto, queremos apenas que tenha uma idéia de como tudo se encaixa.)

Um histórico bem resumido do Java



Java 1.02

250 classes

Lenta.

Nome e logotipo interessantes. Divertida de usar. Muitos erros. Os ***applets*** são o destaque.

Java 1.1

500 classes

Um ***pouco*** mais rápida.

Mais recursos, ***mais amigável***. Começando a se tornar muito ***popular***. Código de GUI mais adequado.

Java 2 (versões 1.2 — 1.4)

2.300 classes

Muito mais rápida.

Pode (em algumas situações) ser executada em velocidades condizentes. Profissional, ***poderosa***. Vem em três versões: Micro Edition (J2ME), Standard Edition (J2SE) e Enterprise Edition (J2EE). Torna-se a ***linguagem preferida*** para novos aplicativos empresariais (principalmente os baseados na Web) e móveis.

Java 5.0 (versões 1.5 e posteriores)

3.500 classes

Mais recursos, mais fácil de desenvolver.

Além de adicionar mais de mil classes complementares, a Java 5.0 (conhecida como "Tiger") acrescentou alterações significativas à própria linguagem, tornando-a mais fácil (pelo menos em teoria) para os programadores e fornecendo novos recursos que eram populares em outras linguagens.

Aponte seu lápis

Tente adivinhar o que cada linha de código está fazendo...
(As respostas estão na próxima página.)

Veja como é fácil escrever código Java.

```
int size = 27;'
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
   readTheFile("myFile.txt");
}
catch(FileNotFoundException ex) {
System.out.print("File not found.");
}
```

declara uma variável de tipo inteiro chamada 'size' e lhe atribui o valor 27

P: Sei que existem o Java 2 e o Java 5.0, mas existiram o Java 3 e 4? E por que Java 5.0 e não Java 2.0?

R: As brincadeiras do marketing... Quando a versão do Java passou de 1.1 para 1.2, as alterações foram tão significativas, que os anunciantes decidiram que precisavam de um "nome" totalmente novo, portanto .começaram a chamá-la de ***Java 2***, ainda que a versão fosse realmente a 1.2. Porém, as versões 1.3 e 1.4 continuaram a ser consideradas como ***Java 2***. Nunca *houve* o Java 3 ou 4. Começando pelo Java versão 1.5, os anunciantes decidiram novamente que as alterações eram tão significativas, que um novo nome era necessário (e a maioria dos desenvolvedores concordou), logo, eles avaliaram as opções. O próximo número na seqüência do nome seria "3", mas chamar o Java 1.5 de ***Java 3*** parecia mais confuso, portanto, decidiram nomeá-lo ***Java 5.0*** para usar o "5" da versão "1.5".

Logo, o Java original compreendeu as versões que iam da 1.02 (o primeiro lançamento oficial) às conhecidas simplesmente como "Java". As versões 1.2, 1.3 e 1.4 consistiram no "Java 2". E começando na versão 1.5, ele passou a se chamar "Java 5.0". Mas você também o verá sendo chamado de "Java 5" (sem o ".0") e "Tiger" (seu codinome original). Não temos idéia do que acontecerá com a *próxima* versão...

Respostas do exercício **Aponte seu lápis**

Ainda não é preciso se preocupar em entender tudo isso!

Tudo que se encontra aqui é explicado com maiores detalhes no livro, grande parte nas primeiras 40 páginas. Se o Java lembrar uma linguagem que você usou no passado, alguns desses itens parecerão simples. Caso contrário, não se preocupe com isso. *Chegaremos lá...*

Veja como é fácil escrever código Java.

`int size = 27;`	*declara uma variável de tipo inteiro chamada 'size' e lhe atribui o valor 27*
`String name = "Fido";`	*declara uma variável de string de caracteres chamada 'name' e lhe atribui o valor "Fido"*
`Dog myDog = new Dog(name, size);`	*declara a nova variável de tipo Dog chamada 'myDog' e cria o novo objeto Dog usando 'name' e 'size'*
`x = size - 5;`	*subtrai 5 de 27 (valor de 'size') e atribui o valor a uma variável chamada 'x'*
`if (x < 15) myDog.bark(8);`	*se x (valor = 22) for menor do que 15, informa ao cão (dog) para latir (bark) 8 vezes*
`while (x > 3) {`	*mantém o loop até x ser maior que 3...*
`myDog.play();`	*pede ao cão que brinque (independentemente do que ISSO signifique para um cão...)*
`}`	*aqui parece ser o fim do loop — tudo que estiver entre { } será feito no loop*
`int[] numList = {2,4,6,8};`	*declara a lista de variáveis de tipo inteiro 'numList' e insere 2, 4, 6, 8 nela*
`System.out.print("Hello");`	*exibe "Hello"... provavelmente na linha de comando*
`System.out.print("Dog: " + name);`	*exibe "Hello Fido" (o valor de 'name' é "Fido") na linha de comando*
`String num = "8";`	*declara a variável de string de caracteres 'num' e lhe atribui o valor "8"*
`int z = Integer.parseInt(num);`	*converte a string de caracteres "8" no valor numérico real 8*
`try {`	*tenta fazer algo... Pode ser que o que estamos tentando não funcione...*
`readTheFile("myFile.txt");`	*lê um arquivo de texto chamado "myFile.txt" (ou pelo menos TENTA ler o arquivo...)*
`}`	*deve ser o fim das "tentativas", portanto, acho que é possível tentar fazer muitas coisas...*
`catch(FileNotFoundException ex) {`	*aqui deve ser onde você saberá se o que tentou não funcionou...*
`System.out.print("File not found.");`	*se o que tentamos não deu certo, exibiremos "File not found" na linha de comando*
`}`	*parece que tudo que se encontra entre { } é o que deve ser feito se a 'tentativa' não funcionar...*

Estrutura do código em Java

O que existe em um arquivo-FONTE?

Um arquivo de código-fonte (com a extensão *.java*) contém uma definição de ***classe***. A classe representa uma *parte* de seu programa, embora um aplicativo muito pequeno possa precisar apenas de uma classe. A classe deve ficar dentro de uma par de chaves.

```
public class Dog{



}
```
classe

O que existe em uma CLASSE?

Uma classe tem um ou mais ***métodos***. Na classe Dog, o método ***bark*** conterá instruções de como o cão deve latir. Seus métodos devem ser declarados *dentro* de uma classe (em outras palavras, dentro das chaves da classe).

```
public class Dog {
   void bark( ) {




}
}
```
método

O que existe em um MÉTODO?

Dentro das chaves de um método, escreva as instruções de como ele deve ser executado. O *código* do método é basicamente um conjunto de instruções, e por enquanto você pode considerar o método como se fosse uma função ou procedimento.

```
public class Dog {
   void bark( ) {

      instrução1;
      instrução2;


   }
}
```

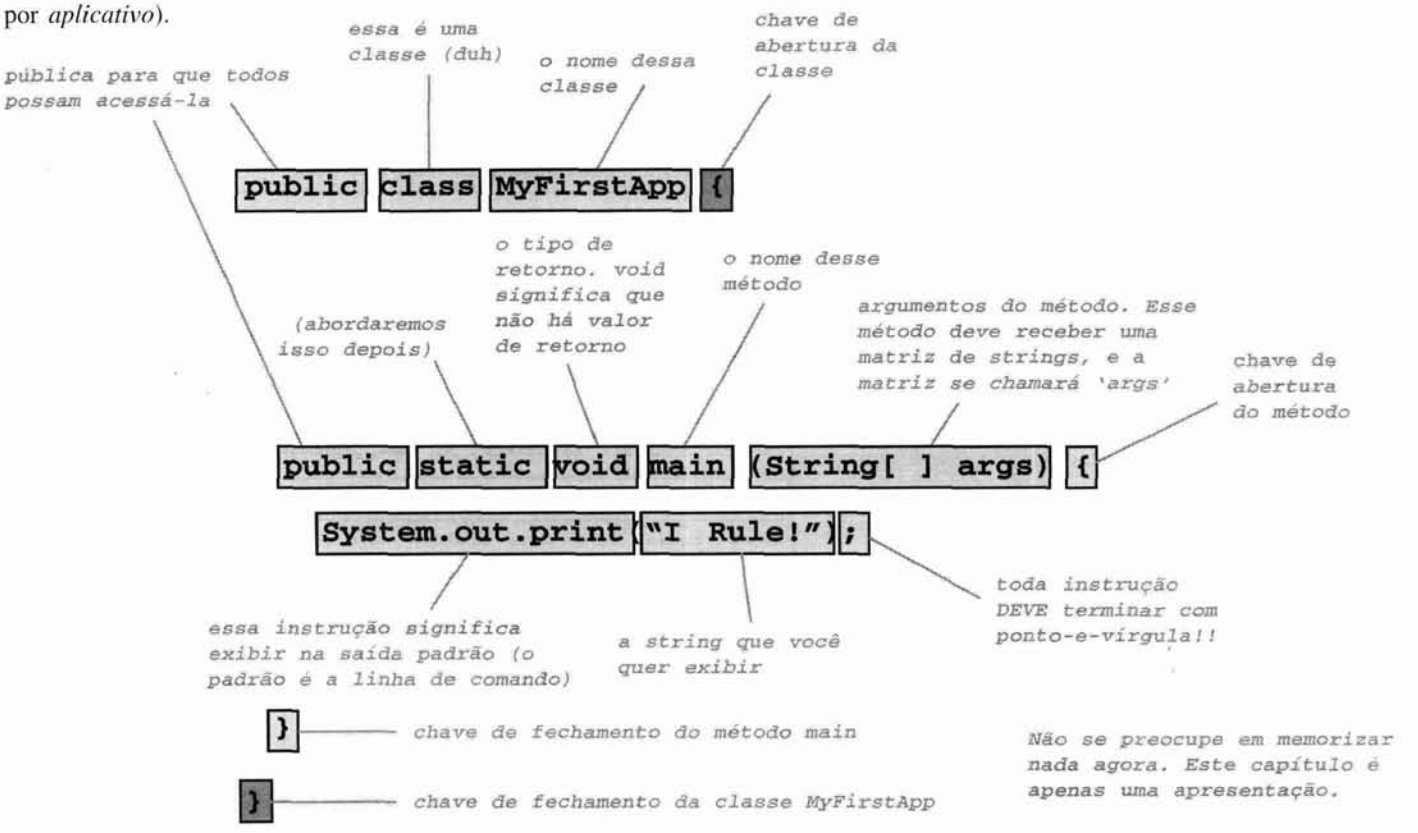
instruções

Anatomia de uma classe

Quando a JVM começar a ser executada, procurará a classe que você forneceu na linha de comando. Em seguida, começará a procurar um método especialmente escrito que se pareça exatamente com:

```
public static void main (String[] args) {
   // seu código entra aqui
}
```

Depois a JVM executará tudo que estiver entre as chaves { } de seu método principal. Todo aplicativo Java precisa ter pelo menos uma **classe** e um método **main** (não um método main por *classe*, apenas um por *aplicativo*).



Criando uma classe com um método main

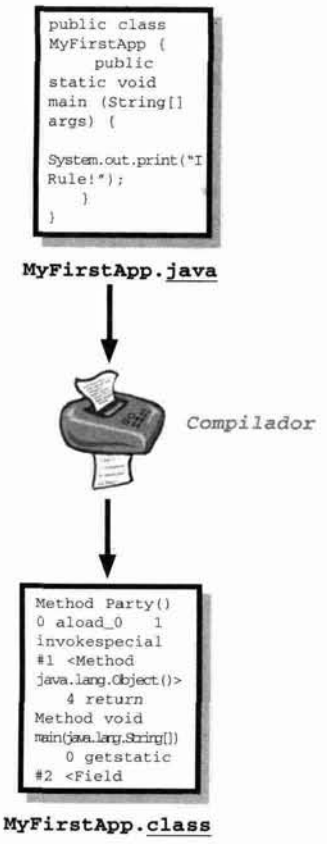
Em Java, tudo é inserido em uma **classe**. Você criará seu arquivo de código-fonte (com extensão *.java*) e, em seguida, o converterá em um novo arquivo de classe (com extensão *.class*). Quando executar seu programa, na verdade estará executando uma *classe*.

Executar um programa significa informar à Java Virtual Machine (JVM) para "carregar a classe **Hello** e, em seguida, execute seu método **main**( ). Continue executando até todo o código de main ter terminado".

No Capítulo 2, nos aprofundaremos no assunto das *classes*, mas, por enquanto, você só precisa pensar nisto: ***como escrever um código Java de modo que ele seja executado?*** E tudo começa com **main**( ).

O método **main** ( ) é onde seu programa começará a ser executado.

Independentemente do tamanho de seu programa (em outras palavras, não importa quantas *classes* o seu programa vai usar), é preciso que haja um método **main**( ) que dê início ao processo.




```
public class MyFirstApp {

   public static void main (String[] args) {
      System.out.println("I Rule!");
      System.out.println("The World");
   }

}
```

1. **Salve**
   MyFirstApp.java
2. **Compile**
   javac MyFirstApp.java
3. **Execute**

```
File Edit Window Help Scream
%java MyFirstApp

I Rule!

The World
```

O que você pode inserir no método main?

Quando você estiver dentro de main (ou de *qualquer* método), a diversão começará. Você pode inserir todas as coisas que costumam ser usadas na maioria das linguagens de programação para ***fazer o computador executar algo***.

Seu código pode instruir a JVM a:

1 fazer algo

Instruções: declarações, atribuições, chamadas de método, etc.

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x 'is " + x);
double d = Math.random();
// isto é um comentário
```

2 fazer algo repetidamente

Loops: *for* e *while*

```
while (x' > 12) {
   x = x -1;
}

for (int x = 0; x < 10; x = x + 1) {
   System.out.print("x is now " + x);
}
```

3 fazer algo sob essa condição

Ramificação: testes ***if/else***

```
if (x == 10) {
   System.out.print("x must be 10");
} else {
   System.out.print("x isn't 10");
}
if ((x < 3) & (name.equals("Dirk"))) {
   System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```

A brincadeira da sintaxe

- Cada instrução deve terminar com ponto-e-vírgula.

```
x = x + 1;
```

- Um comentário de linha única começa com duas barras.

```
x = 22;
// esta linha me incomoda
```

- A maioria dos espaços em branco não é importante.

```
x        =        3     ;
```

- As variáveis são declaradas com um **nome** e um **tipo** (você aprenderá todos os ***tipos*** Java no Capítulo 3).

```
Int weight;
// tipo: int, nome: weight
```

- As classes e métodos devem ser definidos dentro de um par de chaves.

```
public void go( ) {
// o código entra aqui
}
```

```
while maisBolas = = verdadeiro) {
   continueJogando( );
}
```

Iterando e iterando e...

O Java tem três estruturas de loop padrão: *while, do-while* e *for*. Você verá todos os loops posteriormente no livro, mas não nesse momento; dessa forma, usemos *while* por enquanto.

A sintaxe (para não mencionar a lógica) é tão simples, que provavelmente você já deve ter adormecido. Contanto que alguma condição seja verdadeira, você fará algo dentro do *bloco* de loop. O bloco de loop é delimitado por um par de chaves; portanto, o que você quiser repetir terá que estar dentro desse bloco.

A principal parte de um loop é o *teste condicional*. Em Java, o teste condicional é uma expressão que resulta em um valor *booleano* — em outras palavras, algo que é ***verdadeiro*** ou ***falso***.

Se você disser algo como "Enquanto (while) *sorveteNoPote* for *verdadeiro*, continue a tirar", terá claramente um teste booleano. *Há* sorvete no pote ou *não há*. Mas se você dissesse "Enquanto *Bob* continuar a tirar", não teria realmente um teste. Para fazer isso funcionar, você teria que alterar para algo como "Enquanto Bob estiver roncando..." ou "Enquanto Bob *não* estiver usando xadrez...".

Testes booleanos simples

Você pode criar um teste booleano simples para verificar o valor de uma variável, usando um *operador de comparação* como:

< (menor que)

> (maior que)

= = (igualdade) (sim, são *dois* sinais de igualdade)

Observe a diferença entre o operador de *atribuição* (apenas *um* sinal de igualdade) e o operador *igual a* (*dois* sinais de igualdade). Muitos programadores digitam acidentalmente = quando querem dizer = =. (Mas não você.)

```
int x = 4; // atribui 4 a x
while (x > 3) {
   // o código do loop será executado porque
   // x é maior que 3
   x = x - 1; // ou ficaríamos eternamente no loop
}
int z = 27; //
while (z == 17) {
   // o código do loop não será executado porque
   // z não é igual a 17
}
```

Não existem Perguntas Idiotas

P: Por que temos que inserir tudo em uma classe?

R: O Java é uma linguagem orientada a objetos (OO). Não é como antigamente, quando tínhamos compiladores antiquados e criávamos um arquivo-fonte monolítico com uma pilha de procedimentos. No Capítulo 2, você aprenderá que uma classe consiste no projeto de um objeto e que quase tudo em Java é um objeto.

P: Tenho que inserir um método main em toda classe que criar?

R: Não. Um programa em Java pode usar várias classes (até mesmo centenas), mas você pode ter só *uma* com um método main — que fará o programa começar a ser executado. Você pode criar classes de teste, no entanto, que tenham métodos main para testar suas *outras* classes.

P: Em minha outra linguagem, posso fazer um teste booleano com um tipo inteiro. Em Java, posso dizer algo como

```
int x = 1;
while (x){ }
```

R: Não. Um *booleano* e um *inteiro* não são tipos compatíveis em Java. Como o resultado de um teste condicional *deve* ser um booleano, a única variável que você pode testar diretamente (sem usar um operador de comparação) é um ***booleano***. Por exemplo, você pode dizer:

```
boolean isHot = true;
while(isHot) { }
```

Exemplo de um loop while

```
public class Loopy {
   public static void main (String[] args) {
      int x = 1;
      System.out.println("Antes do Loop");
      while (x < 4) {
         System.out.println("No loop");
         System.out.println("O valor de x é " + x);
         x = x + 1;
      }
      System.out.println("Esse é o fim do loop");
   }
}
```

```
% java Loopy
Antes do Loop
No loop
O valor de x é 1
No loop                    <-- Esta é a saída
O valor de x é 2
No loop
O valor de x é 3
Esse é o fim do loop
```

DISCRIMINAÇÃO DOS PONTOS

- As instruções terminam em ponto-e-vírgula;
- Os blocos de código são definidos por um par de chaves { }
- Declare uma variável *int* com um nome e um tipo: int x;
- O operador de **atribuição** é *um* sinal de igualdade =
- O operador **igual a** são *dois* sinais de igualdade = =
- O loop *while* executará tudo que estiver dentro de seu bloco (definido por chaves), contanto que o *teste condicional* seja ***verdadeiro***.
- Se o teste condicional for ***falso***, o bloco de código do loop *while* não será executado, e o processamento passará para baixo até o código imediatamente posterior ao bloco do loop.
- Coloque um teste booleano entre parênteses:

while **(x = = 4)** { }

Ramificação condicional

Em Java, um teste *if* é basicamente o mesmo que o teste booleano de um loop *while* — só que em vez de dizer "***while*** ainda houver cerveja...", você dirá "***if*** (se) ainda houver cerveja..."

```
class IfTest {
   public static void main (String[] args) {
      int x = 3;
      if (x == 3) {
         System.out.println("x deve ser igual a 3");
      }
      System.out.println("Isso será executado de qualquer forma");
   }
}
```

```
% java IfTest
x deve ser igual a 3                <-- saída do código
Isso será executado de qualquer forma
```

O código anterior executará a linha que exibe "x deve ser igual a 3" somente se a condição (x é igual a 3) for atendida. Independentemente de ser verdadeira, no entanto, a linha que exibe "Isto será executado de qualquer forma" será executada. Portanto, dependendo do valor de x, uma ou duas instruções serão exibidas.

Mas podemos adicionar uma instrução *else* à condição, para podermos dizer algo como "*If* ainda houver cerveja, continue a codificar, *else* (caso contrário) pegue mais cerveja e então continue..."

```
class IfTest2 {
   public static void  main (String[] args) {
      int x = 2;
      if (x == 3) {
         System.out.println("x deve ser igual a 3");
      } else {
         System.out.println("x NÃO é igual a 3");
      }
      System.out.println("Isso será executado de qualquer forma");
   }
}
```

```
% java IfTest2
x NÃO é igual a 3
Isso será executado de qualquer forma
```

nova saída

System.out.PRINT versus System.out.printLN

Se você prestou atenção (é claro que prestou), deve ter notado que nos alternamos entre **print** e **println**.

Você entendeu a diferença?

System.out.***println*** insere uma nova linha (pense em ***println*** como **print*new*line**), enquanto System.out.***print*** continua exibindo na *mesma* linha. Se você quiser que cada coisa que exibir esteja em sua própria linha, use **println**. Se quiser que tudo fique junto em uma linha, use **print**.

Aponte seu lápis

Dada a saída:

```
% java DooBee
DooBeeDooBeeDo
```

Preencha as lacunas do código:

```
public class DooBee {
   public static void main (String[] args) {
      int x = 1;
      while (x < _____ ) {
         System.out._________("Doo");
         System.out._________("Bee");
         x = x + 1;
      }
      if (x == _____ ) {
         System.out.print("Do");
      }
   }
}
```

Codificando um aplicativo empresarial real

Colocaremos todas as suas novas aptidões em Java em uso com algo prático. Precisamos de uma classe com um método *main( )*, um tipo *int* e uma variável *String*, um loop *while* e um teste *if*. Mais alguns retoques e você estará construindo esse back-end empresarial sem demora. Mas *antes* examinará o código dessa página e pensará por um instante em como *você* codificaria esse grande clássico, "99 garrafas de cerveja".

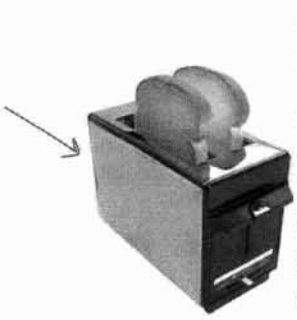
```
public class BeerSong {
   public static void main (String[] args) {
      int beerNum = 99;
      String word = "bottles";

      while (beerNum > 0) {

         if (beerNum == 1) {
            word = "bottle"; // no singular, como em UMA garrafa.
         }

         System.out.println(beerNum + " " + word + " of beer on the wall");
         System.out.println(beerNum + " " + word + " of beer.");
         System.out.println("Take one down.");
         System.out.println("Pass it around.");
         beerNum = beerNum - 1;

         if (beerNum > 0) {
            System.out.println(beerNum + " " + word + " of beer on the wall");
         } else {
            System.out.println("No more bottles of beer on the wall");
         } // fim de else
      } // fim do loop while
   } // fim do método main
} // fim da classe
```

Ainda há uma pequena falha em nosso código. Ele será compilado e executado, mas a saída não está 100% perfeita. Veja se consegue identificar a falha e corrija.

Segunda de manhã na casa de Bob

O despertador de Bob toca às 8:30 da manhã de segunda, como em todos os outros dias da semana. Mas Bob teve um fim de semana cansativo e procura o botão SNOOZE. E é aí que a ação começa e os aparelhos habilitados com Java despertam.

Primeiro, o despertador envia uma mensagem para a cafeteira:* "Ei, o espertinho está dormindo de novo, atrase o café em 12 minutos."



A cafeteira envia uma mensagem para a torradeira Motorola™: "Segure a torrada, Bob está tirando uma soneca."

Em seguida, o despertador envia uma mensagem para o celular Nokia Navigator™ de Bob: "Chame Bob às 9 horas e diga para ele que estamos ficando um pouco atrasados."

Para concluir, o despertador envia uma mensagem para a coleira sem fio de Sam (Sam é o cachorro), com um sinal bastante familiar que significa: "Pegue o jornal, mas não espere ser levado para um passeio."

Alguns minutos depois, o despertador toca novamente. E *mais uma vez* Bob aperta o botão SNOOZE, e os aparelhos começam a se comunicar. O despertador toca uma terceira vez. Mas, assim que Bob alcança o botão SNOOZE, o despertador envia o sinal "salte e lata" para a coleira de Sam. Trazido à consciência por esse choque, Bob se levanta, grato por suas aptidões em Java e um pequeno passeio à Radio Shack™ terem melhorado a rotina diária de sua vida.

Sua torrada está pronta.

Seu café está quente.

Seu jornal está esperando.

Apenas mais uma maravilhosa manhã na ***Casa Habilitada com Java***.

***Você* também pode ter um lar habilitado com Java.** Adote uma solução sensata que use Java, a Ethernet e a tecnologia Jini. Cuidado com imitações ao usar outras plataformas "plug and play" (que na verdade significa "conecte e perca os próximos três dias tentando fazer funcionar") ou "portáveis". A irmã de Bob, Betty, testou uma dessas *outras* plataformas e os resultados foram, digamos, não muito atraentes, ou seguros. Também não deu certo com seu cachorro...

Essa história poderia ser verdadeira? Sim e não. Embora *haja* versões de Java sendo executadas em dispositivos, dentre os quais os PDAs, telefones celulares (*principalmente nos telefones celulares)*, pagers, alarmes, cartões inteligentes e outros — talvez você não encontre uma torradeira ou coleira com Java. Mas mesmo se você não conseguir encontrar uma versão habilitada para Java de seus aparelhos favoritos, ainda poderá executá-los como se *fossem um* dispositivo Java, controlando-os através de alguma outra interface (como seu laptop) que *esteja* executando a Java. Isso é conhecido como *arquitetura substituta* Jini. Sim, você *pode* ter essa casa dos sonhos de um nerd.

* *IP multicast* se você for sistemático com relação a protocolos

Certo, quer dizer que a canção da cerveja não era *na verdade* um aplicativo empresarial profissional. Ainda precisa de algo prático para mostrar ao chefe? Veja o código da Paráfrase.

Nota: quando você digitar isso em um editor, deixe o código criar sua própria quebra de palavra/linha! Nunca pressione a tecla Return quando estiver digitando uma string (algo entre "aspas"), ou ela não será compilada. Portanto, os hifens vistos nessa página são reais e você pode digitá-los, mas não pressione a tecla Enter antes de terminar uma string.

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

1    // crie três conjuntos de palavras onde será feita
a seleção. Adicione o que quiser!

      String[] wordListOne = {"24/7", "várias camadas",
"30.000 pés", "B-to-B", "todos ganham", "front-end",
"baseado na Web", "difundido", "inteligente", "seis
sigma", "caminho crítico", "dinâmico"};


      String[] wordListTwo = {"habilitado", "adesivo",
"valor agregado", "orientado", "central", "distribuído",
agrupado", "solidificado", "independente da máquina",
"posicionado", "em rede", "dedicado", "alavancado",
"alinhado", "destinado", "compartilhado", "cooperativo",
acelerado"};

      String[] wordListThree = {"processo", "ponto
máximo", "solução", "arquitetura", "habilitação no
núcleo", "estratégia", "mindshare", "portal", "espaço",
"visão", "paradigma", "missão"};

2    // descubra quantas palavras existem em cada lista
      int oneLength = wordListOne.length;
      int twoLength = wordListTwo.length;
      int threeLength = wordListThree.length;

3    // gere três números aleatórios
      int rand1 = (int) (Math.random() * oneLength);
      int rand2 = (int) (Math.random() * twoLength);
      int 'rand3 = (int) (Math.random() * threeLength);

4    // agora construa uma frase
      String phrase = wordListOne[rand1] + " " +
wordListTwo[rand2] + " " + wordListThree[rand3];

5    // exiba a frase
      System.out.println("Precisamos de " + phrase);
   }
}
```

Código da paráfrase

Como funciona

Em resumo, o programa cria três listas de palavras e, em seguida, seleciona aleatoriamente uma palavra de cada uma das três listas e exibe o resultado. Não se preocupe se você não entender *exatamente* o que está acontecendo em cada linha. Ora, temos o livro inteiro à frente, portanto, relaxe. Isso é apenas um rápido paradigma alavancado e destinado a exibir 10.000 metros de independência de máquina.

1. A primeira etapa é criar três matrizes de strings — os contêineres que armazenarão todas as palavras. Declarar e criar uma matriz é fácil; a seguir temos uma pequena:

```
String[] pets = {"Fido", "Zeus", "Bin"};
```

Todas as palavras estão entre aspas (como toda string precisa estar) e foram separadas por vírgulas.

2. Em cada uma das três listas (matrizes), o objetivo é selecionar uma palavra aleatória, portanto, temos que saber quantas palavras existem em cada lista. Se houver 14 palavras em uma lista, precisaremos de um número aleatório entre 0 e 13 (as matrizes Java começam em zero, logo, a primeira palavra estará na posição 0, a segunda na posição 1 e a última na posição 13 em uma matriz de 14 elementos). Uma matriz Java não fará objeções em exibir seu tamanho imediatamente. Você só precisa perguntar. Na matriz de animais de estimação, diríamos:

```
int x = pets.length;
```

e agora **x** teria o valor 3.

3. Precisamos de três números aleatórios. O Java vem empacotada, independente, predefinida e habilitada na memória central com um conjunto de métodos de cálculo (por enquanto, considere-os como funções). O método **random( )** retorna um número aleatório entre 0 e quase 1, portanto, temos que multiplicá-lo pela quantidade de elementos (o tamanho da matriz) da lista que estivermos usando. Temos que forçar para que o resultado seja um inteiro (decimais não são permitidos!), logo, vamos inserir uma conversão (você verá os detalhes no Capítulo 4). É o mesmo que se tivéssemos um número de ponto flutuante que quiséssemos converter em um inteiro:

```
int x = (int) 24.6;
```

4. Agora construiremos a frase, selecionando uma palavra em cada uma das três listas e unindo-as (além de inserir espaços entre elas). Usaremos o operador "+", que *concatenará* (preferimos a palavra mais técnica '*unirá*') os objetos String. Para selecionar um elemento da matriz, você fornecerá o número do índice (posição) do item que quer usar:

```
String s = pets[0]; // agora s é a string "Fido"
S = s +" " + "é um cão"; // agora s é "Fido é um cão"
```

5. Para concluir, exibiremos a frase na linha de comando e... *voila*! *Somos do marketing.*

O que precisamos aqui é um...

Processo destinado e difundido

Ponto máximo dinâmico independente da máquina

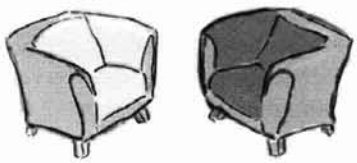
Habilitação no núcleo distribuída e inteligente

Mindshare habilitado em 24/7

Visão de 30.00 pés em que todos ganham

Portal em rede seis sigma

Bate-papo na fogueira



Conversa de hoje: **O compilador e a JVM discutem a questão "Quem é mais importante?**

A Máquina Virtual Java

O que, você está brincando? *Olá. Sou* o Java. Sou eu quem efetivamente faz um programa *ser executado*. O compilador apenas lhe fornece um *arquivo*. É só. Apenas um arquivo. Você pode imprimir e usá-lo como papel de parede, para acender fogo, forrar a gaiola de pássaros ou *seja lá o que for*, mas o arquivo não *fará* nada a menos que eu esteja lá para executá-lo.

E esse é outro problema, o compilador não tem senso de humor. Lógico, se *você* tivesse que passar o dia inteiro verificando pequenas violações na sintaxe minuciosamente...

Não estou falando que você é, digamos, *completamente* inútil. Mas convenhamos, o que você faz? Sério. Não faço idéia. Um programador poderia apenas escrever bytecode manualmente, e eu o usaria. Você pode ficar sem trabalho em breve, amigo.

O compilador

Não aprecio esse tom.

Desculpe, mas sem a *minha* presença, o que exatamente você executaria? Há uma *razão* para o Java ter sido projetado para usar um compilador de bytecode, se você não sabe. Se ele fosse uma linguagem puramente interpretada, onde - no tempo de execução - a máquina virtual tivesse que converter código-fonte diretamente de um editor de texto, o programa Java seria executado a uma velocidade comicamente lenta. O Java já demorou tempo suficiente para convencer as pessoas de que é rápido e poderoso o bastante para a maioria dos trabalhos.

Desculpe, mas esse é um ponto de vista bem displicente (para não dizer *arrogante*). Embora *seja* verdade que — *teoricamente* — você possa executar qualquer bytecode formatado apropriadamente mesmo se ele não vier de um compilador Java, na prática isso é um absurdo. Um

programador escrevendo bytecode manualmente é como se você executasse seu processamento de palavras usando PostScript pura. E eu apreciaria se *não* se dirigisse a mim como "amigo".

(Vou continuar insistindo na veia irônica.) Mas você ainda não respondeu minha pergunta, o *que* faz realmente?

Lembre-se de que o Java é uma linguagem fortemente tipificada, o que significa que não posso permitir que as variáveis armazenem dados com o tipo errado. Esse é um recurso de segurança crucial e posso bloquear a grande maioria das violações antes que elas cheguem até você. Além disso -

Mas algumas ainda passam! Posso lançar exceções ClassCastException e às vezes vejo pessoas tentando inserir o tipo errado de coisa em uma matriz que foi declarada como contendo algo diferente e -

Desculpe, mas não terminei. E sim, *há* algumas exceções de tipo de dado que podem surgir no tempo de execução, mas algumas delas têm que ser permitidas, para que outro recurso Java importante tenha suporte - a vinculação dinâmica. No tempo de execução, um programa Java pode incluir novos objetos que não eram *conhecidos* nem mesmo pelo programador original, portanto, tenho que permitir um certo nível de flexibilidade. Mas meu trabalho é bloquear qualquer coisa que nunca seria - *poderia* ser - bem sucedida no tempo de execução. Geralmente consigo saber quando algo não vai funcionar, por exemplo, se um programador tentasse acidentalmente usar um objeto Button como uma conexão de soquete, eu detectaria isso e evitaria que ele causasse danos no tempo de execução.

OK. Certo. Mas e quanto à *segurança*? Veja tudo que eu faço com relação à segurança e você fica, digamos, procurando sinais de *ponto-e-vírgula*? Oooooh, mas que grande risco à segurança! Muito obrigado!

Desculpe, mas sou a primeira linha de defesa, como dizem. As violações de tipo de dado que descrevi anteriormente poderiam danificar um programa se fosse permitido que elas se manifestassem. Também sou eu que impeço as violações de acesso, como, por exemplo, um código que tentasse chamar um método privado, ou alterar um método que - por razões de segurança - não pudesse nunca ser alterado. Impeço as pessoas de mexer em códigos que não tenham permissão para ver, inclusive códigos que tentem acessar dados críticos de outra classe. Demoraria horas, talvez dias, para eu conseguir descrever a importância de meu trabalho.

Não importa. Acabo tendo que fazer a mesma coisa só para me certificar se alguém obteve acesso depois de você e alterou o bytecode antes de executá-lo.

É claro, mas como descrevi anteriormente, se eu não impedisse o que talvez chegue a 99% dos problemas potenciais, você acabaria travando. E parece que não temos mais tempo, portanto, teremos que voltar a isso em um bate-papo posterior.

Oh, pode contar com isso. *Amigo*.

Exercício

Ímãs com código

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

```
File Edit Window Help Sleep
% java Shuffle1
a-b c-d
```

```
if (x == 1) {
System.out.print("d");
  x = x - 1;
}
```

```
if (x == 2) {
   System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args
{
```

```
if (x > 2) {
   System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Exercício

Seja o compilador

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

A

```
class Exercise1b {
   public static void main(String []
args) {
      int x = 1;
      while ( x < 10 ) {
         if ( x > 3) {
            System.out.println("big x");
         }
      }
   }
}
```

B

```
public static void main(String []
args) {
   int x = 5;
   while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
         System.out.println("small x");
      }
   }
}
```

C

```
class Exercise1b {
   int x = 5;
   while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
         System.out.println("small x");
      }
   }
}
```

Cruzadas Java

Agora daremos algo para o lado direito de seu cérebro fazer.

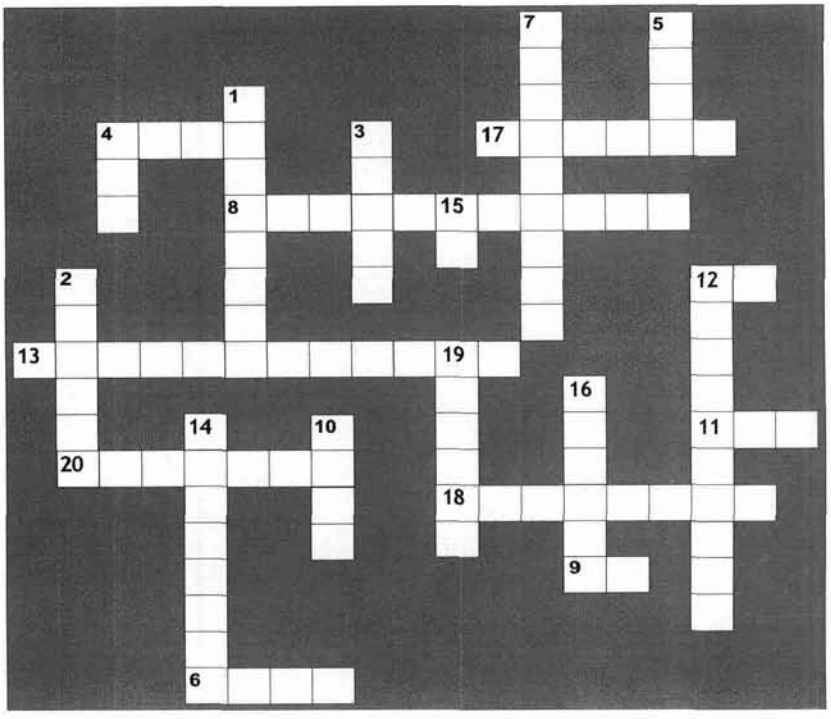
É uma palavra cruzada padrão, mas quase todas as palavras da solução vêm do Capítulo 1. Apenas para que você fique alerta, também incluímos algumas palavras (não relacionadas à Java) do universo tecnológico.

Horizontais

4. Código de linha de comando
6. Mais uma vez?
8. Não pode seguir dois caminhos
9. Acrônimo do tipo de energia de seu laptop
11. Tipo numérico de variável
12. Acrônimo de um chip
13. Exibir algo
17. Um conjunto de caracteres
18. Anunciar uma nova classe ou método
20. Para que serve um prompt?

Verticais

1. Não é um inteiro (ou seu barco é um objeto ____)
2. Voltou de mãos vazias
3. As portas estão abertas
4. Depto. de manipuladores de LAN
5. Contêineres de 'itens'
7. Até que as atitudes melhorem
10. Consumidor de código-fonte
12. Não é possível fixá-la
14. *Modificador inesperado*
15. É preciso ter um
16. Como fazer algo
19. Consumidor de bytecode



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. (As respostas estão no final do capítulo.)

```
class Test {
   public static void main(String [] args) {
      int x = 0;
      int y = 0;
      while ( x < 5 ) {

         System.out.print(x + "" + y +" ");
         x = x + 1;
      }
   }
}
```

O código candidato entra aqui

Candidatos:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
   y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
   x = x + 1;
   if ( y < 3 ) {
      x = x - 1;
   }
}
y = y + 2;
```

Saídas possíveis:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Compare cada candidato com uma das saídas possíveis

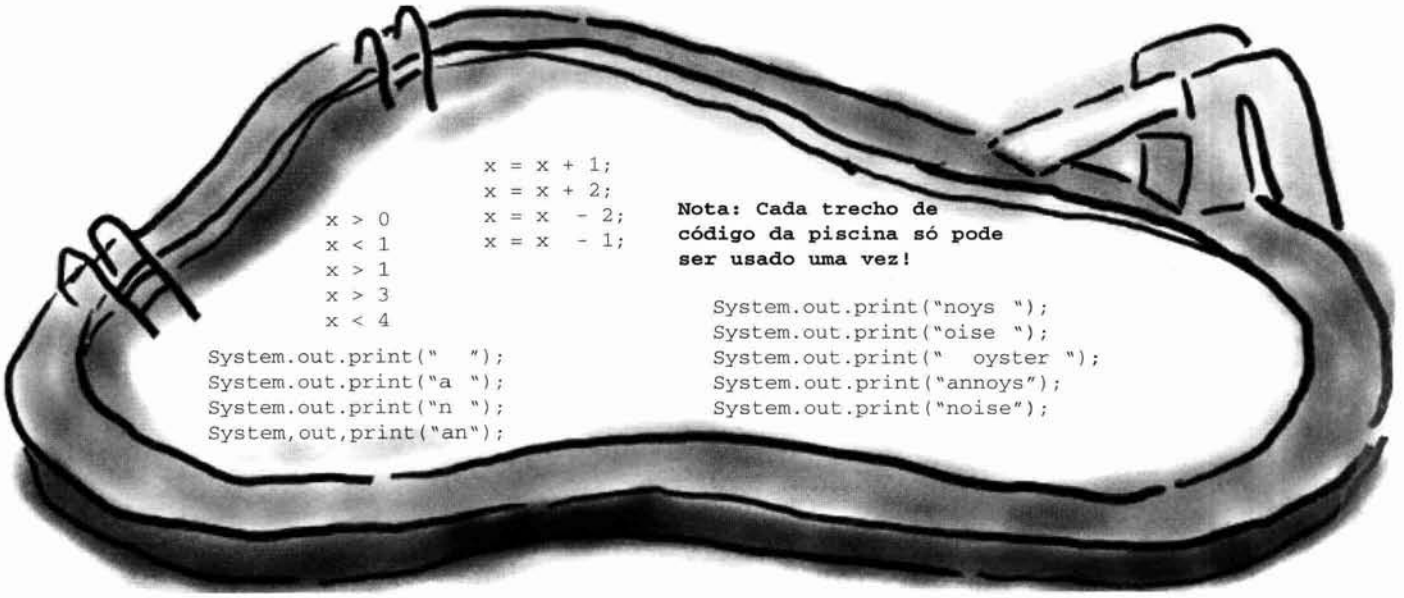
Quebra-cabeças na Piscina

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

```
class PoolPuzzleOne {
   public static void main(String [] args) {
      int x = 0;

      while ( __________ ) {

         _____________________________
         if ( x < 1 ) {
            _____________________________
         }
         _____________________________

         if ( __________ ) {

            _____________________________

            ___________
         }
         if ( x == 1 ) {

            _____________________________
         }
         if ( ___________ ) {

            _____________________________
         }
         System.out.println("");

         ___________
      }
   }
}
```

Saída

```
File Edit Window Help Cheat
%java PollPuzzleOne
a noise
annoys
an oyster
```



```
x > 0
x < 1
x > 1
x > 3
x < 4

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

System.out.print("   ");
System.out.print("a  ");
System.out.print("n  ");
System,out,print("an");
```

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

```
System.out.print("noys  ");
System.out.print("oise  ");
System.out.print("   oyster  ");
System.out.print("annoys");
System.out.print("noise");
```

Soluções dos Exercícios

Ímãs com código:

```
class Shuffle1 {
   public static void main(String [] args) {
      int x = 3;
      while (x > 0) {
         if (x > 2) {
            System.out.print("a");
         }
         x = x - 1;
         System.out.print("-");
         if (x == 2) {
            System.out.print("b c");
         }
         if (x == 1) {
            System.out.print("d");
            x = x - 1;
         }
      }
   }
}
```

A

```
class Exercise1b {
   public static void main(String [] args) {
      int x = 1;
      while ( x < 10 ) {
         x = x + 1;
         if ( x > 3) {
            System.out.println("big x");
         }
      }
   }
}
```

Esse código será compilado e executado (sem saída), mas sem uma linha adicionada ao programa, ele seria processado indefinidamente em um loop 'while' infinito!

B

```
class foo {
   public static void main(String [] args) {
      int x = 5;
      while ( x > 1 ) {
         x = x - 1;
         if ( x < 3) {
            System.out.println("small x");
         }
      }
   }
}
```

Esse arquivo não será compilado sem uma declaração de classe, e não se esqueça da chave correspondente!

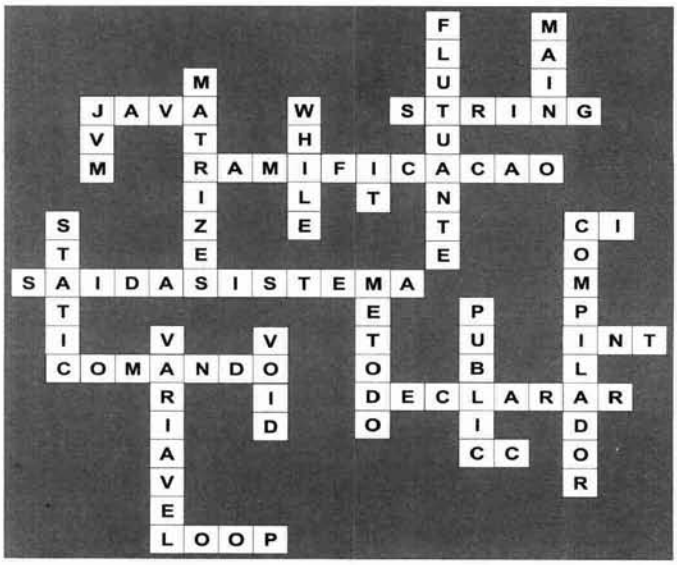
C

```
class Exercise1b {
   public static void main(String[]args) {
      int x = 5;
      while ( x > 1 ) {
         x = x - 1;
         if ( x < 3) {
            System.out.println("small x");
         }
      }
   }
}
```

O código do loop 'while' deve ficar dentro de um método. Não pode ficar simplesmente isolado fora da classe.

Soluções dos quebra-cabeças

```
class PoolPuzzleOne {
   public static void main(String [] args) {
      int x = 0;
      while ( X < 4 ) {
         System.out.print("a");
         if ( x < 1 ) {
            System.out.print(" ");
         }
         System.out.print("n");
         if ( X > 1 ) {
            System.out.print(" oyster");
            x = x + 2;
         }
         if ( x == 1 ) {
            System.out.print("noys");
         }
         if ( X < 1 ) {
            System.out.print("oise");
         }
         System.out.println("");
         X = X + 1;
      }
   }
}
```



Candidatos:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
   y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
   x = x + 1;
   if ( y < 3 ) {
      x = x - 1;
   }
}
y = y + 2;
```

Saídas possíveis:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```