Apêndice B

Os dez principais tópicos que quase entraram no livro...

Cobrimos um terreno vasto e você está quase terminando este livro. Sentiremos sua falta, mas antes de deixar você ir, não nos sentiríamos bem em enviá-lo para o universo Java sem um pouco mais de preparação. Não conseguiríamos fornecer tudo que você precisa saber nesse apêndice relativamente pequeno. Na verdade, originalmente incluímos tudo que você precisa saber sobre Java (ainda não abordado nos outros capítulos), reduzindo o tamanho da letra para ,00003. Coube, mas ninguém conseguia ler. Logo, retiramos grande parte, porém guardamos os melhores trechos para esse apêndice com os Dez Mais.

Esse *é* realmente o fim do livro. Exceto pelo índice (uma leitura e tanto!).

#10 Manipulação de bits

Por que se importar?

Falamos sobre o fato de que há 8 bits em um byte, 16 bits em um tipo curto e assim por diante. Você pode ativar ou desativar bits individuais. Por exemplo, se estiver escrevendo um código para sua nova torradeira habilitada com Java e perceber que, devido a graves limitações de memória, certas configurações da torradeira serão controladas no nível de bits. Para tornar a leitura mais fácil, estamos mostrando somente os últimos 8 bits nos comentários em vez dos 32 completos de um tipo int).

Operador bit a bit NÃO: ~

Esse operador 'troca todos os bits' de um tipo primitivo.

```
int x = 10;     // os bits são 00001010
x = ~x;         // agora os bits são 11110101
```

Os três próximos operadores comparam dois tipos primitivos bit a bit e retornam um resultado baseado na comparação desses bits. Usaremos o exemplo a seguir para esses três operadores:

```
int x = 10;     // os bits são 00001010
int y = 6;      // agora os bits são 00000110
```

Operador bit a bit E: &

Esse operador retorna um valor cujos bits só serão ativados se ***os dois*** bits originais estiverem ativados:

```
int a = x & y;   // os bits são 00000010
```

Operador bit a bit OU: |

Esse operador retorna um valor cujos bits só serão ativados se ***um dos*** bits originais estiver ativado:

```
int a = x | y;   // os bits são 00001110
```

Operador bit a bit XOR (OU Exclusivo): ^

Esse operador retorna um valor cujos bits só serão ativados se ***exatamente um*** dos bits originais estiver ativado:

```
int a = x ^ y;   // os bits são 00001100
```

Os operadores de deslocamento

Esses operadores pegam um único tipo inteiro primitivo e deslocam (ou empurram) todos os seus bits para uma direção ou outra. Se você lembrar o que aprendeu sobre cálculo binário, vai perceber que o deslocamento de bits *à esquerda* efetivamente *multiplica* um número por uma potência de dois e o deslocamento de bits *à direita divide* um número por uma potência de dois.

Usaremos o exemplo a seguir para os próximos três operadores:

```
int x = -11;    // os bits são 11110101
```

Certo, certo, estivemos adiando isso, aqui está a explicação mais curta do mundo sobre o armazenamento de números negativos e o *complemento de dois*. Lembre-se de que o bit da esquerda de um número inteiro é chamado de ***bit de sinal***. Um número inteiro negativo em Java *sempre* está com seu bit de sinal *ativado* (isto é, configurado com 1). Um número inteiro positivo fica com seu bit de sinal *desativado* (0). O Java usa a fórmula do complemento de dois para armazenar números negativos. Para mudar o sinal de um número usando o complemento de dois, troque todos os bits e, em seguida, adicione 1 (com um byte, por exemplo, isso significaria adicionar 00000001 ao valor invertido).

Operador de deslocamento à direita: >>

Esse operador desloca todos os bits de um número à direita por um certo número de casas e preenche todos os bits do lado esquerdo com qualquer que fosse o bit original da estrema esquerda. **O bit de sinal *não* é alterado**:

```
int y = x >> 2;   // os bits são 11111101
```

Operador de deslocamento à direita sem sinal: >>>

Igual ao operador de deslocamento à direita PORÉM sempre preenche os bits da extrema esquerda com zeros. **O bit de sinal *pode* ser alterado:**

```
int y = x >>> 2;   // os bits são 00111101
```

Operador de deslocamento à esquerda: <<

Igual ao operador de deslocamento à direita sem sinal, mas na outra direção; os bits da extrema direita são preenchidos com zeros. **O bit de sinal *pode* ser alterado:**

```
int y = x << 2;   // os bits são 11010100
```

#9 Imutabilidade

Por que se importar com o fato de que as Strings são imutáveis?

Quando seus programas Java começarem a ficar grandes, você inevitavelmente acabará com vários objetos String. Por razões de segurança, e com a finalidade de conservar espaço na memória (lembre-se de que seus programas Java podem ser executados em pequenos celulares habilitados com Java) as Strings em Java são imutáveis. Isso significa que quando você escrever:

```
String s = "0";
for (int x = 1; x < 10; x++) {
  s = s + x;
}
```

Na verdade estará criando dez objetos String (com os valores "0", "01", "012" até "0123456789"). A variável *s* acabará referenciando a String de valor "0123456789", mas nesse momento teremos criado *dez* Strings!

Sempre que você criar uma nova String, a JVM a inserirá em uma parte especial da memória chamada 'Reservatório de

Strings' (parece refrescante não?). Se já houver uma String no Reservatório de Strings com o mesmo valor, a JVM não criará uma duplicata, simplesmente apontará sua variável de referência para a entrada existente. A JVM pode fazer isso porque as Strings são imutáveis; uma variável de referência não pode alterar o valor da String de outra variável de referência que aponte para a mesma String.

O outro problema do Reservatório de Strings é que o Coletor de Lixo *não chega até lá*. Portanto, em nosso exemplo, a menos que por coincidência posteriormente você crie uma String chamada, digamos, "01234", as primeiras nove Strings criadas em nosso loop *for* ficarão apenas aguardando desperdiçando memória.

Como isso economiza memória?

Bem, se você não for cuidadoso, *não economizará!* Mas se entender como a imutabilidade de Strings funciona, então, em algumas situações poderá se beneficiar dela para economizar memória. No entanto, se você tiver que executar muitas manipulações com Strings (como concatenações, etc.), há uma classe StringBuilder, mais adequada para essa finalidade. Falaremos mais sobre StringBuilder algumas páginas adiante.

Por que se importar com o fato de que os Empacotadores são imutáveis?

No capítulo sobre Math falamos sobre as duas principais utilidades das classes empacotadoras:

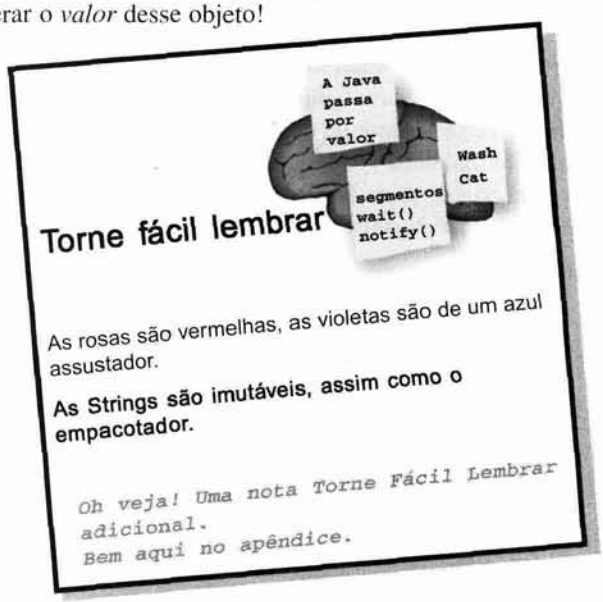
- Empacotar um tipo primitivo para que ele possa simular ser um objeto.

- Usar os métodos estáticos utilitários [por exemplo, Integer.parseInt()].

É importante lembrar que quando você criar um objeto empacotador como:

```
Integer iWrap = new Integer(42);
```

Esse objeto empacotador estará definido. Seu valor *sempre* será 42. ***Não há método de configuração para um objeto empacotador***. É claro que você pode apontar *iWrap* para um objeto empacotador *diferente*, mas então terá *dois* objetos. Quando você criar um objeto empacotador, não terá como alterar o *valor* desse objeto!



#8 Asserções

Não falamos muito sobre como depurar seu programa Java enquanto você estiver desenvolvendo-o. Acreditamos que você deve aprender Java na linha de comando, como fizemos no decorrer do livro. Quando for um profissional em Java, se decidir usar um IDE,* pode ter outras ferramentas de depuração para usar. Antigamente, quando um programador Java queria depurar seu código, inseria várias instruções System.out.println() no decorrer do programa, exibindo os valores atuais das variáveis e mensagens "Estou aqui", para ver se o controle do fluxo estava funcionando apropriadamente. (O código predefinido do capítulo 6 continua com algumas instruções 'print' de depuração.) Em seguida, se o programa estivesse funcionando corretamente, ele o percorria e removia novamente todas essas instruções System.out.println(). Era tedioso e propenso a erros. Mas, a partir da Java 1.4 (e 5.0), a depuração ficou muito mais fácil. A resposta?

Asserções

As asserções são instruções System.out.prinln() reforçadas. Adicione-as a seu código como faria com as instruções de exibição. O compilador do Java 5.0 presumirá que você está compilando arquivos de código-fonte compatíveis com a versão 5.0, portanto a partir do Java 5.0, a compilação com asserções vem ativada por padrão.

No tempo de execução, se você não fizer nada, as instruções assert que adicionou a seu código serão ignoradas pela JVM e não retardarão seu programa. Mas se você solicitar à JVM que *ative* suas asserções, elas o ajudarão a fazer a depuração, sem alterar uma linha de código!

Algumas pessoas têm reclamado por terem que deixar instruções assert em seu código de produção, mas deixá-las pode ser muito útil quando seu código já tiver sido distribuído. Se seu cliente estiver com problemas, você poderá instruí-lo a executar o programa com as asserções ativadas e solicitar que lhe envie a saída. Se as asserções fossem removidas do código implantado, você não teria essa opção. E quase não há desvantagens; quando as asserções não estão ativadas, são totalmente ignoradas pela JVM, portanto, não é preciso se preocupar com perdas no desempenho.

Como fazer as asserções funcionarem

Adicione as instruções de asserção a seu código onde achar que algo *tenha que ser verdadeiro*. Por exemplo:

```
assert (height > 0);

// se verdadeiro, o programa continuará
normalmente
// se falso, lance um AssertionError
```

Você pode adicionar um pouco mais de informações ao rastreamento da pilha escrevendo:

```
assert (height > 0) : "height = " +
height + " weight = " + weight;
```

A expressão após o sinal de dois-pontos pode ser qualquer expressão Java válida ***que resulte em um valor que não seja nulo***. Mas o que quer que você faça, ***não crie asserções que alterem o estado de um objeto!*** Se o fizer, a ativação das asserções no tempo de execução pode alterar a maneira como seu programa será executado.

Compilando e executando com asserções

Para *compilar* com asserções:

```
javac TestDriveGame.java
```

(Observe que não foi necessária nenhuma opção de linha de comando.)

Para *executar* com asserções:

```
java -ea TestDriveGame
```

*IDE significa Integrated Development Environment e inclui ferramentas como o Eclipse, o JBuilder da Borland ou a ferramenta de fonte aberta NetBeans (netbeans.org).

#7 Escopo de bloco

No Capítulo 9, falamos sobre como as variáveis locais só existem enquanto o método em que foram declaradas se encontra na pilha. Mas algumas variáveis podem ter existências ainda *mais curtas*. Dentro dos métodos, geralmente criamos *blocos* de código. Fizemos isso o tempo todo, mas não *falamos* explicitamente em termos de *blocos*. Normalmente, os blocos de codigo ocorrem dentro de métodos e são delimitados por chaves { }. Alguns exemplos comuns de blocos de código que você reconhecerá incluem os loops (*for, while*) e as expressões condicionais (como as instruções *if*).

Examinemos um exemplo:

```
void doStuff() {                          // Início do bloco do método.

  int x = 0;                              // Variável local com escopo no método inteiro.

  for(int y = 0; y < 5; y++) {            // Começo do bloco de um loop for com o escopo de y
                                          // sendo somente o loop!
    x = x + y;                            // Sem problemas, tanto x quanto y estão no escopo.
  }                                       // Fim do bloco do loop for.

  x = x * y;                              // Opa! Não será compilado! y está fora de escopo aqui!
                                          // (não é assim que funciona em outras linguagens,
                                          // portanto, cuidado!)
}                                         // Fim do bloco do método, agora x também está fora
                                          // de escopo.
```

No exemplo anterior, y era uma variável de bloco, ou seja, declarada dentro de um bloco, mas saiu de escopo assim que o loop for terminou. Seus programas Java serão mais depuráveis e expansíveis se você usar variáveis locais em vez de variáveis de instância e variáveis de bloco em vez de variáveis locais, sempre que possível. O compilador verificará se você não está tentando usar uma variável que esteja fora de escopo, portanto não é preciso se preocupar com problemas no tempo de execução.

#6 Chamadas encadeadas

Embora você tenha visto muitas delas neste livro, tentamos manter nossa sintaxe o mais possível organizada e legível. Há, no entanto, muitos atalhos válidos em Java, aos quais sem dúvida você será exposto, principalmente se tiver que ler muitos códigos que não escreveu. Uma das estruturas mais comuns que encontrará é conhecida como *chamadas encadeadas*. Por exemplo:

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2,"umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// o resultado é sb = summer
```

Mas o que está acontecendo na segunda linha de código? É claro que esse é um exemplo planejado, mas você precisa aprender a decifrar.

1 – Trabalhe da esquerda para a direita.

2 – Encontre o resultado do método mais à esquerda, nesse caso, sb.delete(3,6). Se você procurar StringBuffer nos documentos do API, verá que o método delete() retorna um objeto StringBuffer. O resultado da execução do método delete() será um objeto StringBuffer com o valor "spr".

3 – O próximo método mais à esquerda (insert()) foi chamado no recém-criado objeto StringBuffer "spr". O resultado dessa chamada de método [do método insert()], *também* será um objeto StringBuffer (embora não

precisasse ter o mesmo tipo de retorno do método anterior) e assim por diante, ou seja, o objeto retornado será usado para chamar o próximo método à direita. Teoricamente, você pode encadear quantos métodos quiser em uma única instrução (embora seja raro ocorrerem mais de três métodos encadeados na mesma instrução). Sem o encadeamento, a segunda linha do código anterior ficaria mais legível e teria uma aparência semelhante a esta:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"umme");
sb = sb.deleteCharAt(1);
```

Mas aqui está um exemplo mais comum e útil, que você nos viu usar, entretanto sabíamos que iríamos voltar a ele aqui. Será empregado quando seu método main() tiver que chamar o método de uma instância da classe main, porém você não precisar manter uma *referência* da instância da classe. Em outras palavras, o método main() terá que criar a instância *só* para poder chamar um dos *métodos* dela.

```
class Foo {
   public static void main(String [] args) {
      new Foo().go();
   }
   void go() {
      // nesse local estará o que REALMENTE queremos...
   }
}
```

Queremos chamar go(), mas não nos importa a instância de Foo, portanto não haverá problema em atribuirmos o novo objeto Foo a uma referência.

#5 Classes anônimas e estáticas aninhadas

As classes aninhadas vêm em muitas versões

Na seção de manipulação de eventos de GUI do livro, começamos usando classes internas (aninhadas) como uma solução para a implementação de interfaces de escuta. Essa é a forma mais comum, prática e legível de uma classe interna - em que a classe é simplesmente aninhada dentro das chaves de outra classe encapsuladora. E lembre-se de que isso significa que você precisa de uma instância da classe externa para capturar uma instância da classe interna, porque a classe interna é um *membro* da classe externa/encapsuladora.

Mas há outros tipos de classes internas inclusive *estáticas* e *anônimas*. Não entraremos em detalhes aqui, mas não queremos que você fique confuso com uma sintaxe estranha quando encontrá-la no código de alguém. Porque de praticamente tudo que você possa fazer com a linguagem Java, talvez nada produzirá um código de aparência mais bizarra do que as classes internas anônimas. Mas começaremos com algo mais simples - classes estáticas aninhadas.

Classes estáticas aninhadas

Você já sabe o que estático significa - algo vinculado à classe e não a uma instância específica. Uma classe estática aninhada tem a mesma aparência das classes não-estáticas que usamos para as interfaces de escuta, exceto por serem marcadas com a palavra-chave **static**.

```
public class FooOuter {

   static class BarInner {
      void sayIt() {
         System.out.println("method of a static inner class");
      }
   }
}

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

      FooOuter.BarInner foo = new FooOuter.BarInner();

      foo.sayIt();
   }
}
```

Uma classe estática aninhada é apenas isso – uma classe inserida em outra e marcada com o modificador static.

Já que uma classe estática aninhada é... Estática, você não usará uma instância da classe externa. Usará apenas o nome da classe, da mesma maneira que chamaria métodos estáticos ou acessaria variáveis estáticas.

As classes estáticas aninhadas são mais como classes comuns não-aninhadas por não gozarem de um relacionamento especial com um objeto encapsulador externo. Mas já que mesmo assim são consideradas *membros* da classe externa/encapsuladora, terão acesso a qualquer membro privado da classe externa... Mas *só os que também forem estáticos*. Já que a classe estática aninhada não está conectada a uma instância da classe externa, não tem nenhuma maneira especial de acessar as variáveis e métodos não estáticos (de instância).

Classes anônimas e estáticas aninhadas (continuação)

A diferença entre *aninhado* e *interno*

Qualquer classe Java definida dentro do escopo de outra classe é chamada de classe ***aninhada***. Não importa se é anônima, estática, comum, o que for. Se estiver dentro de outra classe, será tecnicamente considerada uma classe *aninhada*. Mas classes aninhadas *não-estáticas* geralmente são chamadas de classes *internas*, que é como as chamamos anteriormente no livro. A conclusão: todas as classes internas são classes aninhadas, mas nem todas as classes aninhadas são classes internas.

Classes internas anônimas

Suponhamos que você estivesse escrevendo algum código de GUI e de repente percebesse que precisa da instância de uma classe que implemente ActionListener. Mas constatou que não *tem* uma instância de ActionListener. Em seguida, percebeu que também não criou uma *classe* para esse ouvinte. Você tem duas alternativas nesse momento:

1) Crie uma classe interna em seu código, como fizemos em nosso código de GUI e, em seguida, instancie-a e passe essa instância para o método de registro de eventos do botão [addActionListener()].

OU

2) Crie uma classe interna *anônima* e instancie-a, nesse local, exatamente agora. ***Literalmente bem no local onde você precisa do objeto de ouvinte.*** É isso mesmo, você criará a classe e a instância no local onde normalmente estaria fornecendo apenas a instância. Pense nisso por um momento - significa que você passará a *classe* inteira onde normalmente passaria apenas uma *instância* como o argumento de um método!

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
   public static void main (String[] args) {

      JFrame frame = new JFrame();
      JButton button = new JButton("click");
      frame.getContentPane().add(button);
      // button.addActionListener(quitListener);
```

Criamos uma moldura e adicionamos um botão, portanto agora precisamos registrar um ouvinte de ações no botão. Porém não criamos uma classe que implemente a interface ActionListener...

Normalmente faríamos algo assim – passaríamos uma referência da instância de uma classe interna... Uma classe interna que implementasse ActionListener [e o método actionPerformed()].

Esta instrução:

```
      button.addActionListener(new ActionListener() {

         public void actionPerformed(ActionEvent ev) {
            System.exit(0);
         }
      });
```

Termina aqui embaixo!

Mas agora em vez de passar uma referência de objeto, passaremos... Toda a definição da classe nova! Em outras palavras, criaremos a classe que implementa ActionListener EXATAMENTE AQUI ONDE PRECISAREMOS DELA. A sintaxe também criará uma instância da classe automaticamente.

Observe que inserimos "new ActionListener" ainda que ActionListener seja uma interface e, portanto, você não pode CRIAR uma instância dela! Mas o que essa sintaxe quer dizer realmente é, "crie uma nova classe (sem nome) que implemente a interface ActionListener e, a propósito, aqui está a implementação dos métodos da interface: actionPerformed()".

```
   }
}
```

#4 Níveis de acesso e modificadores de acesso (quem vê o que)

Java tem *quatro níveis* de acesso e *três modificadores* de acesso. Há somente *três* modificadores porque o *padrão* (o que você empregará quando não usar nenhum modificador de acesso) *é* um dos quatro níveis de acesso.

Níveis de acesso (em ordem de quanto são restritivos, do menos restritivo ao mais restritivo)

público ← *Acesso público significa que qualquer código de qualquer local poderá acessar o item público (com 'item' queremos dizer classe, variável, método, construtor, etc.).*

protegido ← *O protegido funciona como o padrão (códigos do mesmo pacote têm acesso), EXCETO por também permitir que subclasses de fora do pacote herdem o item protegido.*

padrão ← *O acesso padrão significa que somente códigos pertencentes ao mesmo pacote da classe que tem o item padrão poderão acessá-lo.*

privado ← *Privado significa que somente códigos dentro da mesma classe poderão acessar o item privado. Lembre-se de que isso significa privado para a classe e não para o objeto. Um objeto Dog pode ver o item privado de outro objeto Dog, mas um objeto Cat não pode ver os itens privados de Dog.*

Modificadores de acesso

```
public

protected

private
```

Na maioria das situações você usará somente os níveis público e privado.

public

Use public para classes, constantes (variáveis finais estáticas) e métodos que você estiver expondo para outros códigos (por exemplo, métodos de captura e configuração) e para a maioria dos construtores.

private

Use private para praticamente todas as variáveis de instância e para métodos que você não quiser que códigos externos chamem (em outras palavras, métodos *usados* pelos métodos públicos de sua classe).

Mas embora você possa não usar os outros dois (protegido e padrão), ainda terá que saber o que fazem, porque os verá em outros códigos.

padrão e protegido

padrão

Tanto o nível de acesso protegido quanto o padrão estão vinculados a pacotes. O acesso padrão é simples - significa que só códigos *pertencentes ao mesmo pacote* poderão acessar os códigos com nível de acesso padrão. Portanto, uma classe padrão, por exemplo (que significa uma classe que não foi explicitamente declarada como *pública*), pode ser acessada somente por classes pertencentes ao mesmo pacote que o seu.

Mas o que significa realmente *acessar* uma classe? Códigos que não têm acesso a uma classe não podem nem mesmo *pensar* nela. E por pensar queremos dizer *usar* a classe em código. Por exemplo, se você não tiver acesso a uma classe, por causa da restrição de acesso, não poderá instanciá-la e nem mesmo declará-la como o tipo de uma variável, argumento ou valor de retorno. Simplesmente não poderá de forma alguma digitá-la em seu código! Se o fizer, o compilador reclamará.

Pense nas implicações - uma classe padrão com métodos públicos significa que esses métodos não são realmente públicos. Você não poderá acessar um método se não puder *ver* a classe.

Por que alguém poderia querer restringir acesso a códigos pertencentes ao mesmo pacote? Normalmente, os pacotes são projetados como um grupo de classes que funcionam juntas como um conjunto relacionado. Portanto, pode fazer sentido que classes do mesmo pacote precisem acessar o código uma da outra, embora como um pacote, só uma pequena quantidade de classes e métodos sejam expostos para o ambiente externo (isto é, códigos fora desse pacote).

Certo, esse é o padrão. É simples - se algo tiver acesso padrão (que, lembre-se, significa nenhum modificador de acesso explícito!), só códigos do mesmo pacote do *item* padrão (classe, variável, método, classe interna) poderão acessar esse *item*.

Mas para que serve o acesso *protegido*?

protegido

O acesso protegido é quase idêntico ao acesso padrão, com uma exceção: permite que subclasses *herdem* o item protegido, *mesmo se estiverem fora do pacote da superclasse que estendem.* É isso. Isso é *tudo* que o acesso protegido lhe fornecerá - a possibilidade de permitir que suas classes fiquem fora do pacote de sua superclasse e ainda assim *herdem* partes da classe, inclusive métodos e construtores.

Vários desenvolvedores não vêem muitas razões para usar o acesso protegido, mas ele é usado em alguns projetos e quem sabe um dia você pode achar que é exatamente isso de que precisa. Uma das coisas interessantes com relação ao acesso protegido é que - diferente dos outros níveis de acesso - ele só é aplicável à *herança*. Se uma subclasse não pertencente a um pacote tiver a *referência* de uma instância da superclasse (a superclasse que tenha, digamos, um método protegido), ela não poderá acessar o método protegido usando essa referência da superclasse! A única maneira de a subclasse poder acessar esse método será *herdando-o*. Em outras palavras, a subclasse não pertencente ao pacote não terá *acesso* ao método protegido, mas *terá* o método, através da herança.

#3 Métodos de String e StringBuffer/StringBuilder

Duas das classes mais usadas do API Java são String e StringBuilder (lembre-se de que vimos no #9 algumas páginas atrás que as Strings são imutáveis, portanto um StringBuffer/StringBuilder pode ser muito mais eficiente se você estiver manipulando uma String). A partir da Java 5.0 você deve usar a classe String***Builder*** em vez de String***Buffer***, a menos que suas manipulações de Strings precisem ser à prova de segmentos, o que não é comum. Aqui está uma visão geral dos métodos-**chave** dessas classes:

Tanto a classe String quanto StringBuffer/StringBuilder têm:

```
char charAt(int index);                          // que caractere ficará em uma posição específica
int length();                                    // qual o tamanho dessa string
String substring(int start, int end);            // captura uma parte dessa string
String toString();                               // qual é o valor desse objeto String
```

Para concatenar Strings:

```
String concat(string);                           // para a classe String
String append(String);                           // para StringBuffer e StringBuilder
```

A classe String tem:

```
String replace(char old, char new);              // substitui todas as ocorrências de um caractere
String substring(int begin, int end);            // captura parte de uma String
char [] toCharArray();                           // converte em uma matriz de caracteres
String toLowerCase();                            // converte todos os caracteres para minúsculas
String toUpperCase();                            // converte todos os caracteres para maiúsculas
String trim();                                   // remove o espaço em branco das extremidades
String valueOf(char [ ])                         // remove uma String de uma matriz de caracteres
String valueOf(int i)                            // cria uma String a partir de um tipo primitivo
                                                 // outros tipos primitivos também têm suporte
```

As classes StringBuffer e StringBuilder têm:

```
StringBxxxx delete(int start, int end);          // exclui uma parte
StringBxxxx insert(int offset, qualquer tipo
primitivo de uma matriz char [ ]);               // insere algo
StringBxxx replace(int start, int end, String s); // substitui essa parte por essa String
StringBxxx reverse();                            // inverte o SB de trás para frente
Void setCharAt(int index, char ch);              // substitui um caractere específico
```

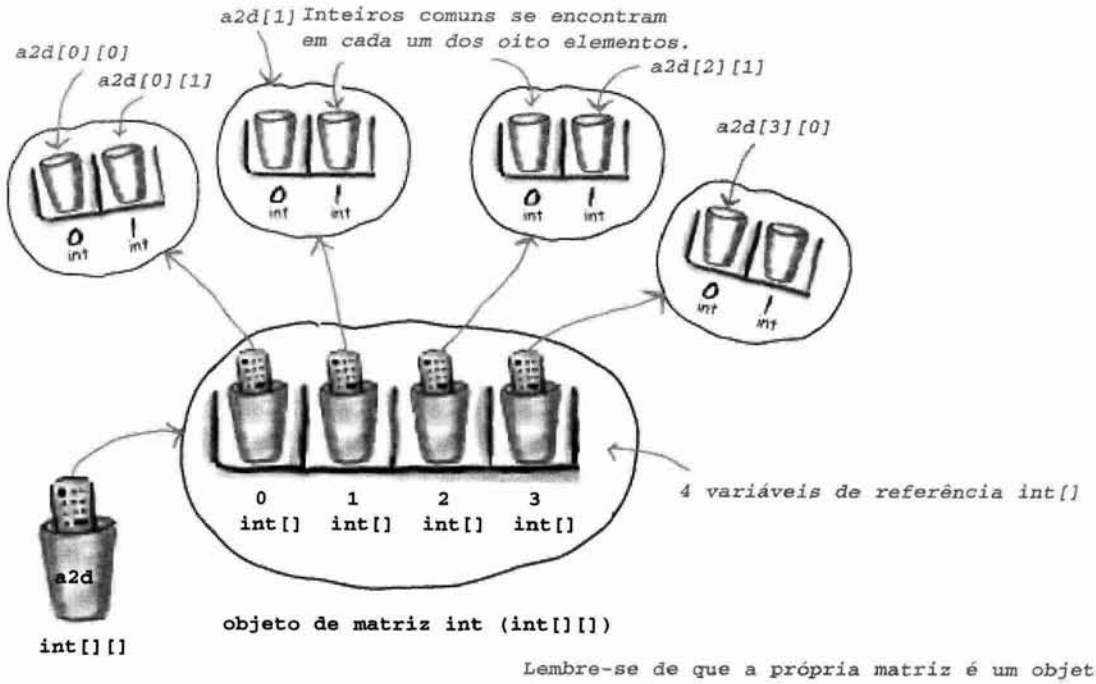
Nota: StringBxxx significará String*Buffer* ou String*Builder*, conforme apropriado.

#2 Matrizes multidimensionais

Na maioria das linguagens, se você criar, digamos, uma matriz bidimensional 4 x 2, visualizará um retângulo de 4 elementos por 2 elementos com um total de 8 elementos. Mas, em Java, uma matriz desse tipo na verdade corresponderia a 5 matrizes encadeadas! Em Java, uma matriz bidimensional é simplesmente *uma matriz composta por matrizes*. (Uma matriz tridimensional é uma matriz composta por matrizes que por sua vez são compostas por outras matrizes, mas deixaremos isso para você descobrir.) Veja como funciona

```
int[][] a2d  = new int [4][2];
```

A JVM criará uma matriz com 4 elementos. *Cada um* desses quatro elementos será na verdade uma variável de referência apontando para uma matriz int (recém-criada) com 2 elementos.



Trabalhando com matrizes multidimensionais

- Para acessar o segundo elemento da terceira matriz: int x = a2d[2][1]; // lembre-se, começando em 0!

- Para criar uma referência unidimensional de uma das submatrizes: int[] copy = a2d[1];

- Atalho para a inicialização de uma matriz 2 x 3: int[][] x = { { 2,3,4 }, { 7,8,9 } };

- Para criar uma matriz 2d com dimensões irregulares:

```
int[][] y = new int [2][];      // cria somente a primeira matriz, com tamanho igual a 2
y[0] = new int [3];     // faz a primeira submatriz ter 3 elementos de dimensão
y[1] = new int [5];     // faz a segunda submatriz ter 5 elementos de dimensão
```

E o tópico número 1 que quase entrou...

#1 Enumerações (também chamadas de Tipos enumerados ou Enums)

Já falamos sobre as constantes que foram definidas no API, por exemplo, **JFrame.EXIT_ON_CLOSE**. Você também pode criar suas próprias constantes marcando uma variável como **estática final**. Mas em algumas situações pode querer criar um conjunto de valores constantes para representar os ***únicos*** valores válidos de uma variável. Esse conjunto de valores válidos normalmente é chamado de *enumeração*. Antes do Java 5.0 só podíamos executar metade da tarefa de criar uma enumeração em Java. A partir do Java 5.0 podemos criar enumerações totalmente desenvolvidas que serão invejadas por todos os seus amigos usuários de versões anteriores à 5.0.

Quem está na banda?

Suponhamos que você estivesse criando um site Web para sua banda favorita e quisesse se certificar de que todos os comentários fossem direcionados a um membro específico dela.

A maneira antiga de simular uma "enumeração":

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;

// posteriormente no código

if (selectedBandMember == JERRY) {
   // faz algo relativo a Jerry
}
```

Esperamos que, quando chegarmos aqui, "selectedBandMember" tenha um valor válido!

A boa notícia sobre essa técnica é que ela REALMENTE torna o código mais fácil de ler. A outra boa notícia é que você nunca poderá alterar o valor das enumerações fictícias que tiver criado; JERRY sempre será 1. A má notícia é que não há uma maneira fácil ou adequada de assegurar que o valor de selectedBandMember sempre seja 1, 2 ou 3. Se algum trecho de código difícil de encontrar configurar selectedBandMember igual a 812, é bem provável que seu código seja interrompido...

A mesma situação com o uso de uma enumeração legítima do Java 5.0. Embora essa seja uma enumeração muito básica, geralmente a maioria das enumerações *é* simples assim.

Uma "enumeração" nova e oficial:

```
public enum Members { JERRY, BOBBY, PHIL };
public Members selectedBandMember;

// posteriormente no código

if (selectedBandMember == Members.JERRY) {

  // faz algo relativo a Jerry
}
```

Isso parece com uma simples definição de classe não? Por acaso as enumerações SÃO um tipo especial de classe. Aqui criamos um novo tipo enumerado chamado "Members".

A variável "selectedBandMember" é de tipo "Members" e SÓ pode ter um valor igual a "JERRY", "BOBBY" ou "PHIL".

A sintaxe para referenciarmos a "instância" de uma enumeração.

Não precisamos nos preocupar com o valor dessa variável!

Sua enumeração estenderá java.lang.Enum

Quando você criar uma enumeração, estará criando uma nova classe e ***estendendo implicitamente java.lang.Enum***. Poderá declarar uma enumeração como sua própria classe autônoma, em seu próprio arquivo de código-fonte, ou como um membro de outra classe.

Usando "if" e "switch" com enumerações

Usando a enumeração que acabamos de criar, podemos gerar ramificações em nosso código empregando a instrução if ou switch. Observe também que podemos comparar as instâncias da enumeração usando o operador = = ou o método equals(). Geralmente o operador = = é considerado um estilo melhor.

```
Members n = Members.BOBBY;
if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");
if (n == Members.BOBBY) System.out.println("Rat Dog");


Members ifName = Members.PHIL;
switch (ifName) {
   case JERRY: System.out.print("make it sing ");
   case PHIL: System.out.print("go deep ");
   case BOBBY: System.out.println("Cassidy! ");
}
```

Atribuindo um valor da enumeração a uma variável

Essas duas instruções funcionarão bem!

"Rat Dog" será exibido.

Quiz pop! Qual é a saída?

Resposta:

```
¡ʎpıssɐƆ deəp oƃ
```

Uma versão realmente complicada de uma enumeração semelhante

Você pode adicionar várias coisas a sua enumeração como um construtor, métodos, variáveis e algo chamado corpo de classe específico de constantes. Elas não são comuns, mas você pode encontrá-las:

```
public class HfjEnum {

   enum Names {
      JERRY("lead guitar") { public String sings() {
                             return "plaintively"; }
                           },
      BOBBY("rhythm guitar") { public String sings() {
                             return "hoarsely"; }
                           },
      PHIL("bass");

      private String instrument;
```

Esse é um argumento passado para o construtor declarado a seguir.

Esses são os chamados "corpos de classes específicos de constantes". Considere-os como uma sobreposição do método básico da enumeração (nesse caso o método "sing()"), se sing() for chamado em uma variável com o valor JERRY ou BOBBY.

```
    Names(String instrument) {
        this.instrument = instrument;
    }
    public String getInstrument() {
        return this.instrument;
    }
    public String sings() {
        return "occasionally";
    }
}

public static void main(String [] args) {
    for (Names n : Names.values()) {
        System.out.print(n);
        System.out.print(", instrument: "+ n.getInstrument());
        System.out.println(", sings: " + n.sings());
    }
}
}
```

Esse é o construtor da enumeração. Ele será executado uma vez para cada valor da enumeração que for declarado (nesse caso ele será executado três vezes).

Você verá esses métodos sendo chamados a partir de "main()".

Toda enumeração vem com um método "values()" embutido que normalmente é usado em um loop "for" como mostrado.

```
File Edit Window Help Bootleg
%java HfjEnum
JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

Observe que o método básico "sing()" só é chamado quando o valor da enumeração não tem um corpo de classe específico de constantes.

Uma longa viagem para casa

Um pequeno mistério

O capitão Byte da nave estelar "Traverser" de Flatland recebeu uma transmissão secreta urgente do quartel general. A mensagem continha 30 códigos de navegação fortemente criptografados que a Traverser precisaria para definir com sucesso um caminho para casa através de setores inimigos. Os inimigos Hackarianos, de uma galáxia vizinha, tinham inventado um diabólico raio misturador de código capaz de criar objetos falsos no heap do único computador de navegação da Traverser. Além disso, o raio alienígena conseguia alterar variáveis de referência válidas para que apontassem para esses objetos falsos. A única defesa que a tripulação da Traverser tinha contra esse maligno raio Hackariano era executar um verificador de vírus embutido que poderia ter sido incorporado ao excepcional código Java 1.4 da nave.

O capitão Byte deu ao cabo Smith as seguintes instruções de programação para o processamento dos códigos críticos de navegação:

"Insira os primeiros cinco códigos em uma matriz de tipo ParsecKey. Insira os últimos 25 códigos em duas matrizes dimensionais cinco por cinco de tipo QuadrantKey. Passe essas duas matrizes para o método plotCourse() da classe pública final ShipNavigation. Quando o objeto de trajeto for retornado, execute o verificador de vírus em todas as variáveis de referência dos programas e, em seguida, execute o programa NavSim e me traga os resultados."

Alguns minutos depois o cabo Smith retornou com a saída do NavSim. "Saída do NavSim pronta para análise, senhor", declarou o cabo Smith. "Ótimo", respondeu o capitão, "Por favor, descreva o trabalho". "Sim senhor!", respondeu o cabo, "Primeiro declarei e construí uma matriz de tipo ParsecKey com o código a seguir; ParsecKey[] p = new ParsecKey[5];, em seguida, declarei e construí uma matriz de tipo QuadrantKey com o código a seguir: QuadrantKey [] [] q = new QuadrantKey [5] [5];. Depois, carreguei os 5 primeiros códigos da matriz ParsecKey usando um loop 'for' e então carreguei os últimos 25 códigos da matriz QuadrantKey usando loops 'for' aninhados. Em seguida, executei o verificador de vírus em todas as 32 variáveis de referência, 1 vez para a matriz ParsecKey e 5 para seus elementos, 1 vez para a matriz QuadrantKey e 25 para seus elementos. Quando a verificação retornou que não tinha detectado nenhum vírus, executei o programa NavSim e re-executei o verificador de vírus, apenas por segurança... Senhor!"

O capitão Byte fitou o cabo fria e longamente e disse com calma, "Cabo, você ficará preso em seu alojamento por colocar em perigo a segurança dessa nave, não quero ver sua cara nessa ponte novamente até que tenha aprendido adequadamente o código Java! Imediato Boolean, assuma pelo cabo e execute esse trabalho corretamente!"

Por que o capitão prendeu o cabo em seu alojamento?

Solução do pequeno mistério

Uma longa viagem para casa

O capitão Byte sabia que em Java, as matrizes multidimensionais na verdade são matrizes composta por matrizes. A matriz cinco por cinco QuadrantKey 'q', precisaria de um total de 31 variáveis de referência para poder acessar todos os seus componentes:

1 – variável de referência de 'q'

5 – variáveis de referência de q[0] – q[4]

25 – variáveis de referência de q[0][0] – q[4][4]

O cabo esqueceu das variáveis de referência das cinco matrizes dimensionais embutidas na matriz 'q'. Qualquer uma dessas cinco variáveis de referência poderia ser corrompida pelo raio Hackariano, e o teste do cabo nunca revelaria o problema.